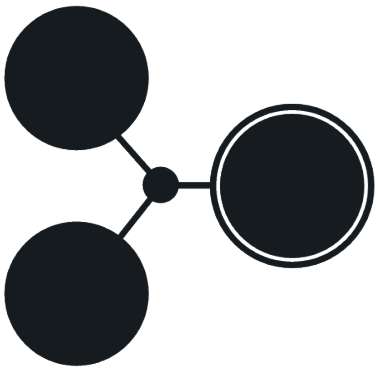


Beginner's Guide to the MOEA Framework

David Hadka

Version 2.8

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

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.

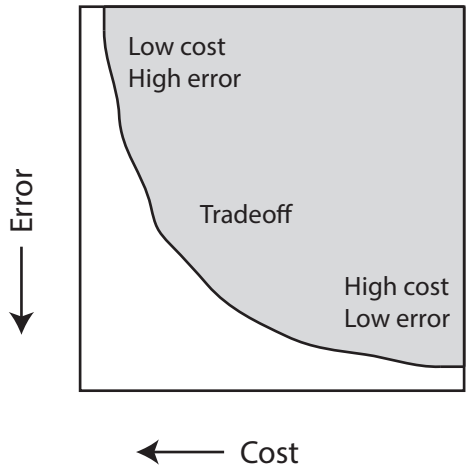


Figure 1.1: Example of the tradeoff between two objectives: (1) cost and (2) error. A tradeoff is formed between these two conflicting objectives where increases in cost lead to reduced error. All figures in this dissertation showing objectives include arrows pointing towards the ideal optimum.

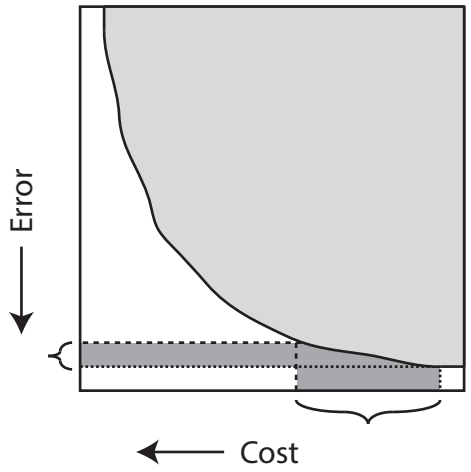


Figure 1.2: Example showing the effect of diminishing returns, where a large increase in cost is necessary to impart a marginal reduction in error.

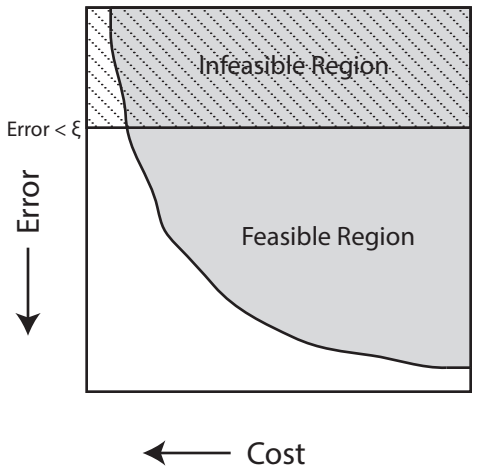


Figure 1.3: Example showing how constraints define an infeasible region (the hashed region). Valid solutions to the optimization problem are only found in the feasible region.

1.1 Multiobjective Problem

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

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

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:** `00110001001010000101111010110111 0011`
- **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

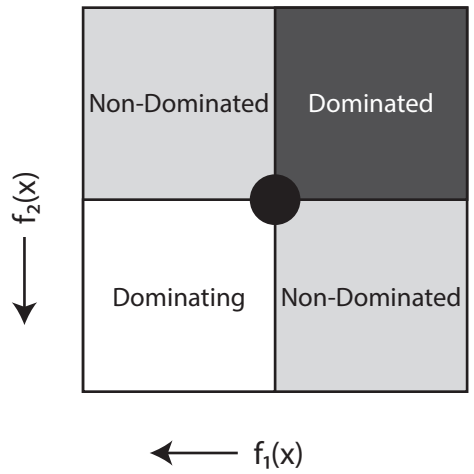


Figure 1.4: Depiction of the various Pareto dominance regions. These regions are relative to each solution, which is centered in the figure. The *dominated* region is inferior in all objectives, the *dominating* region is superior in all objectives and the *non-dominated* region is superior in one objective 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* $\mathbf{u} = (u_1, u_2, \ldots, u_M)$ ***Pareto dominates*** *another vector* $\mathbf{v} = (v_1, v_2, \ldots, v_M)$ *if and only if* $\forall i \in \{1, 2, \ldots, M\}, u_i \leq v_i$ *and* $\exists j \in \{1, 2, \ldots, M\}, u_j < v_j$. *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

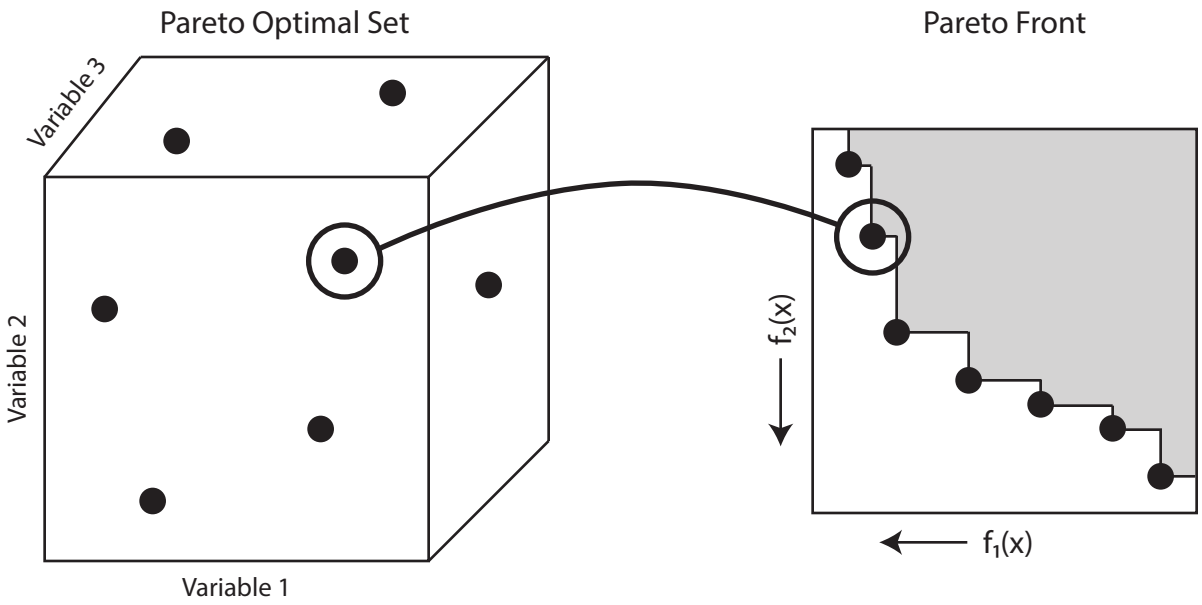


Figure 1.5: Shows a hypothetical mapping between a 3-dimensional Pareto optimal set and its associated 2-dimensional Pareto front. The shaded region in the Pareto front shows the space dominated by the Pareto front.

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

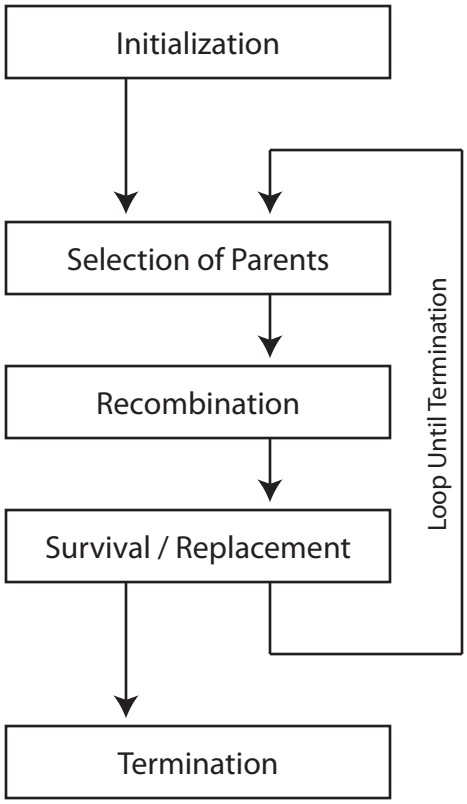


Figure 1.6: The outline of a simple EA. EAs begin with an initialization process, where the initial search population is generated. They next enter a loop of selecting parent individuals from the search population, applying a recombination operator (such as crossover and mutation in genetic algorithms) 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:

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

and the weighted Chebyshev method:

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

where $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_M)$ are the weights and $\mathbf{z}^* = (z_1^*, z_2^*, \ldots, z_M^*)$ 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 Niched-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).

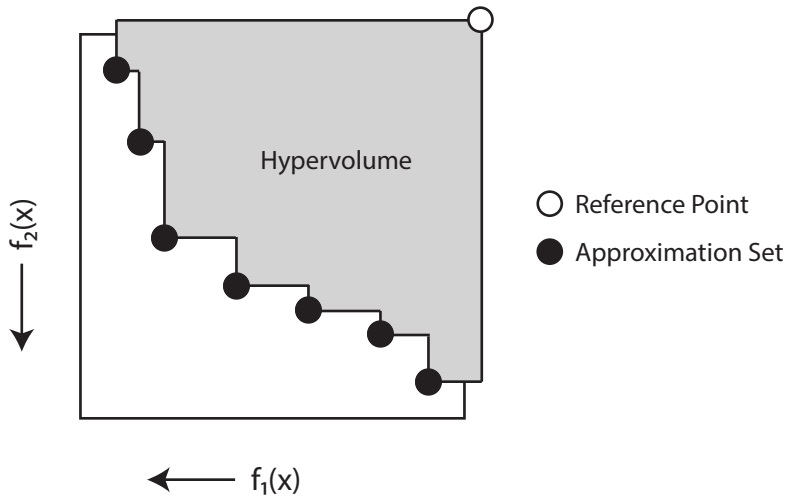


Figure 1.7: Hypervolume measures the volume of the space dominated by the approximation set, bounded by a reference point. This reference point is typically the nadir point (i.e., the worst-case value for each objective) of the reference set plus some fixed delta. This delta ensures extremal points contribute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

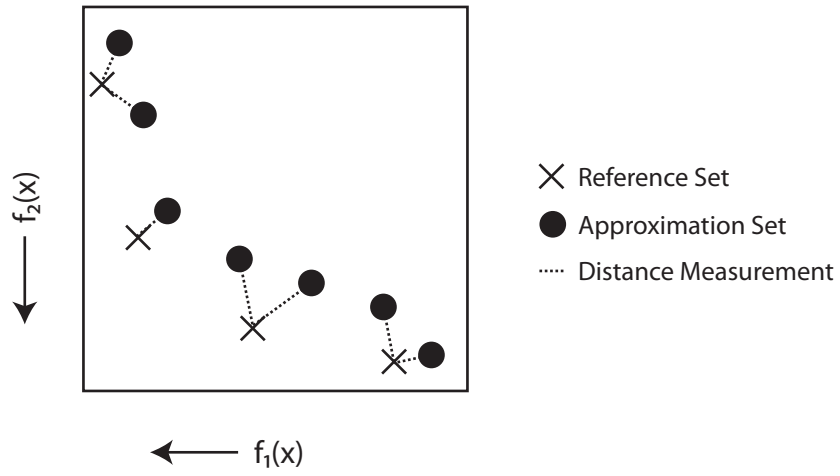


Figure 1.8: Generational distance is the average distance from every solution in the approximation set to the nearest solution in the reference set.

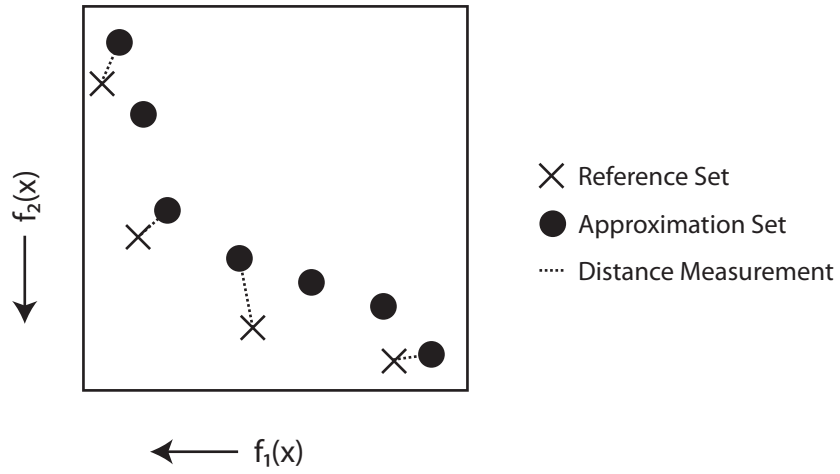


Figure 1.9: Inverted generational distance is the average distance from every solution in the reference set to the nearest solution in the approximation set.

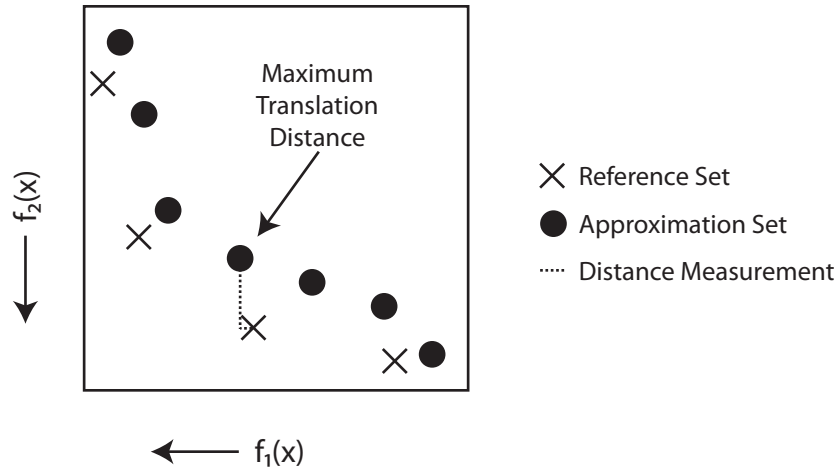


Figure 1.10: ϵ_+-indicator (also known as the additive ϵ-indicator) is the smallest distance ϵ that the approximation set must be translated by in order to completely dominate the reference set (Coello Coello et al., 2007).

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,

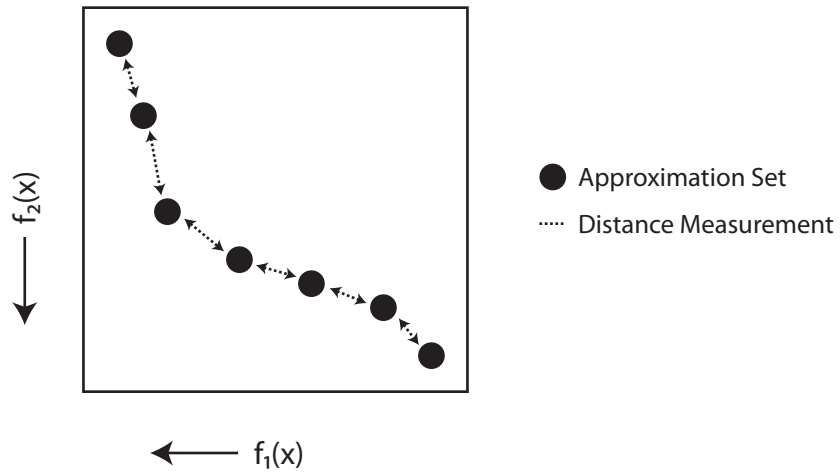


Figure 1.11: Spacing measures the uniformity of the spacing between solutions in an approximation set.

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:

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

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

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

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.

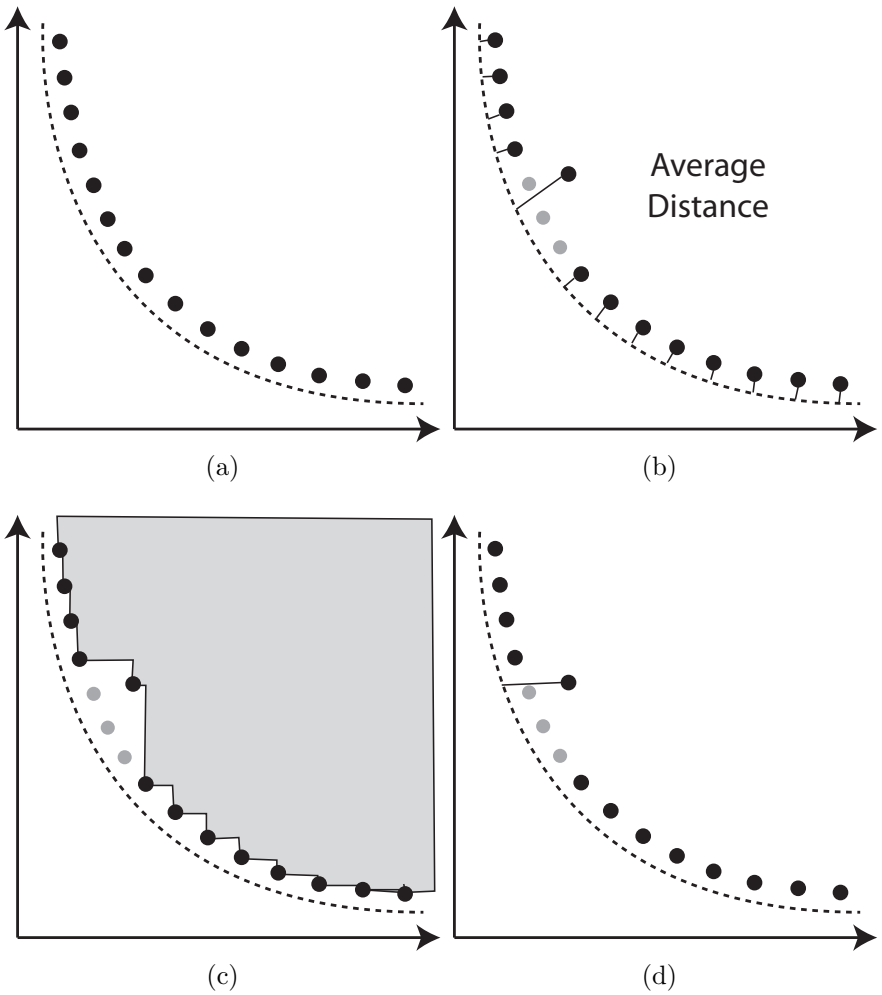


Figure 1.12: Demonstrates the importance of ϵ-indicator as a measure of consistency. (a) A good approximation set to the reference set, indicated by the dashed line. (b) Generational distance averages the distance between the approximation set and reference set, reducing the impact of large gaps. The 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`.

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.

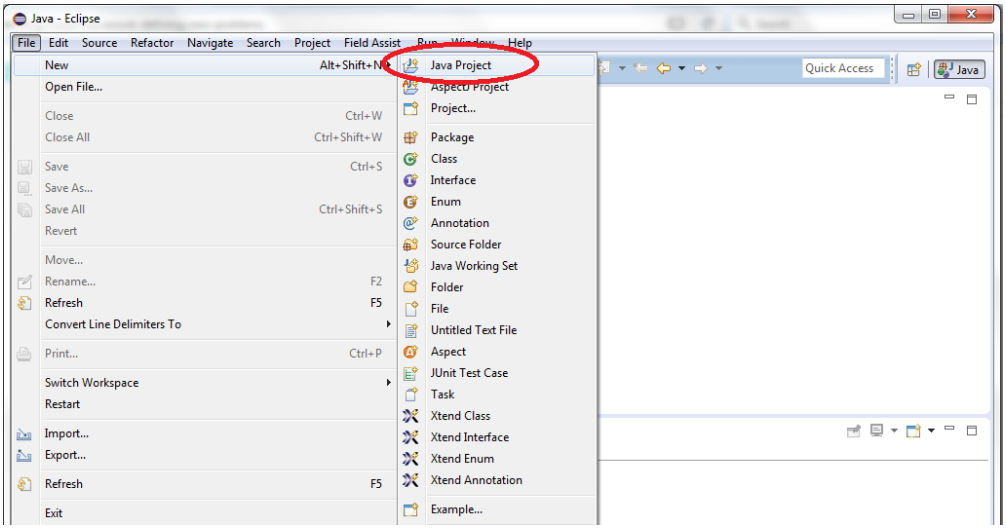
2.3 Setting up the MOEA Framework

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: `http://bit.ly/1N5sHjO`.

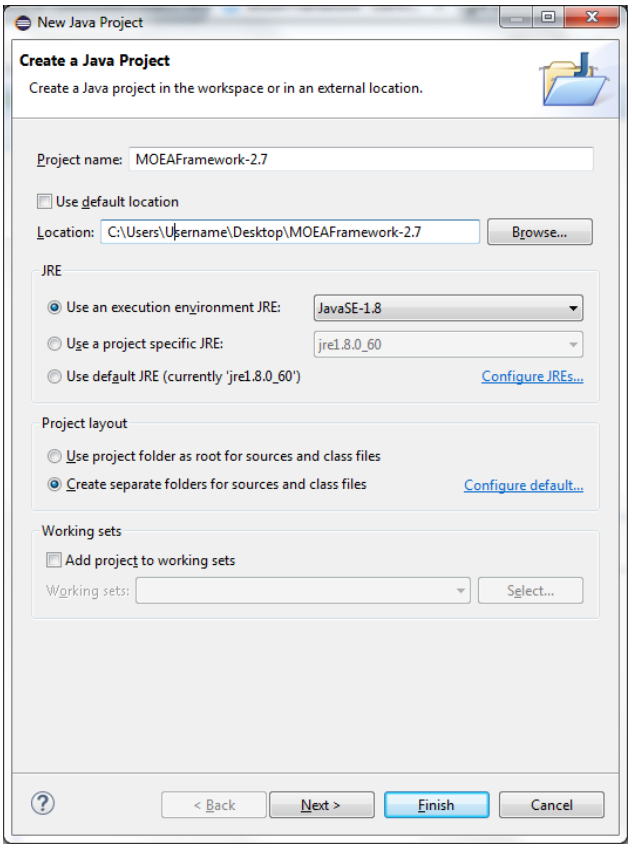
Note: that ends with the letter O and not the number 0. As you read this book, find the appropriate Java file within the `book` folder to follow along.

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.

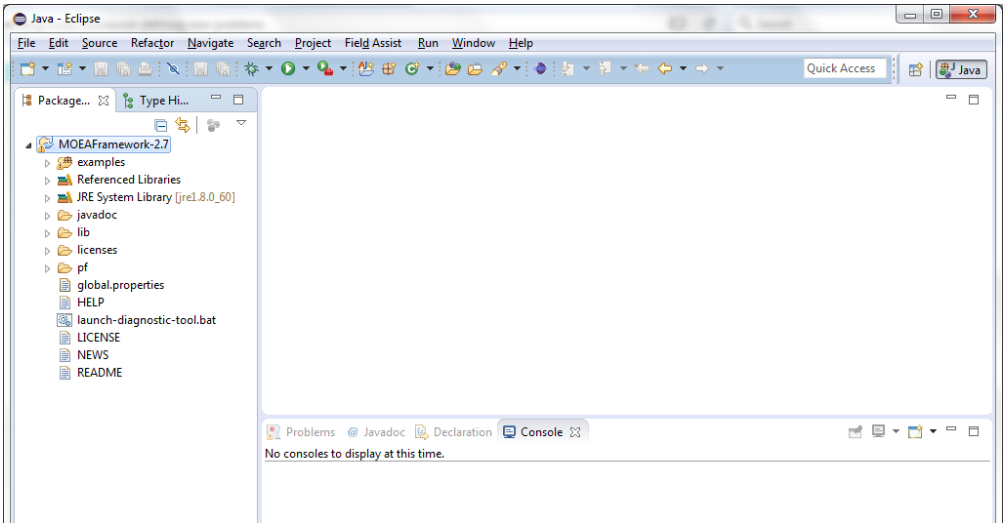
Next, we need to create a project within Eclipse. Select File → New Java Project from the menu.



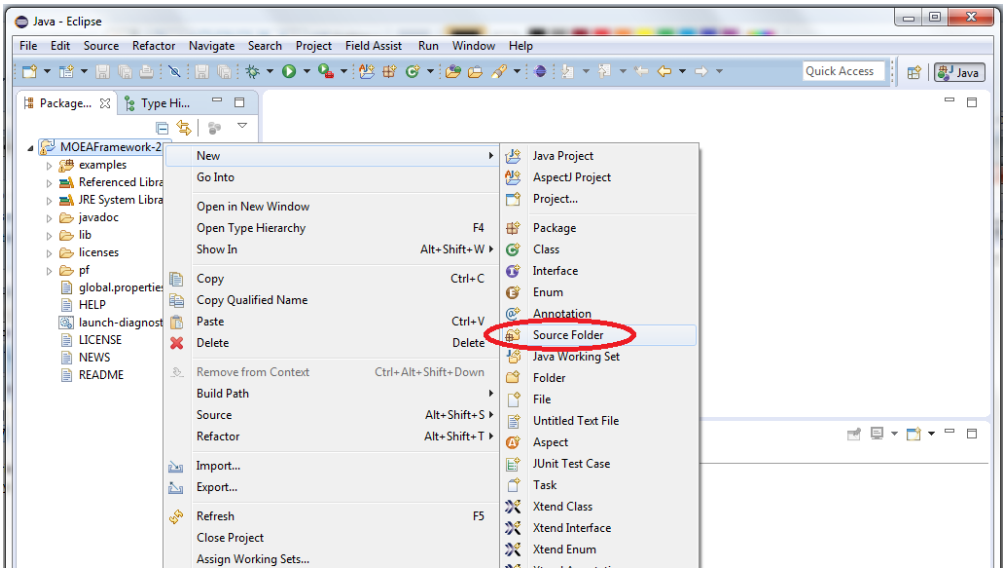
In the window that appears, uncheck "Use default location". Click the "Browse..." button and select the extracted MOEA Framework folder. Click Finish.



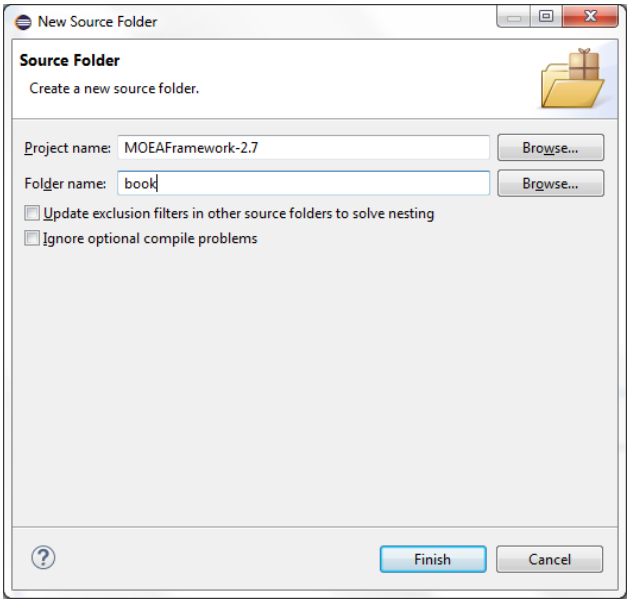
The MOEA Framework project will now appear within the "Package Explorer" in Eclipse, as shown below.



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 were our examples will reside. Right-click on the project and select New → Source Folder.

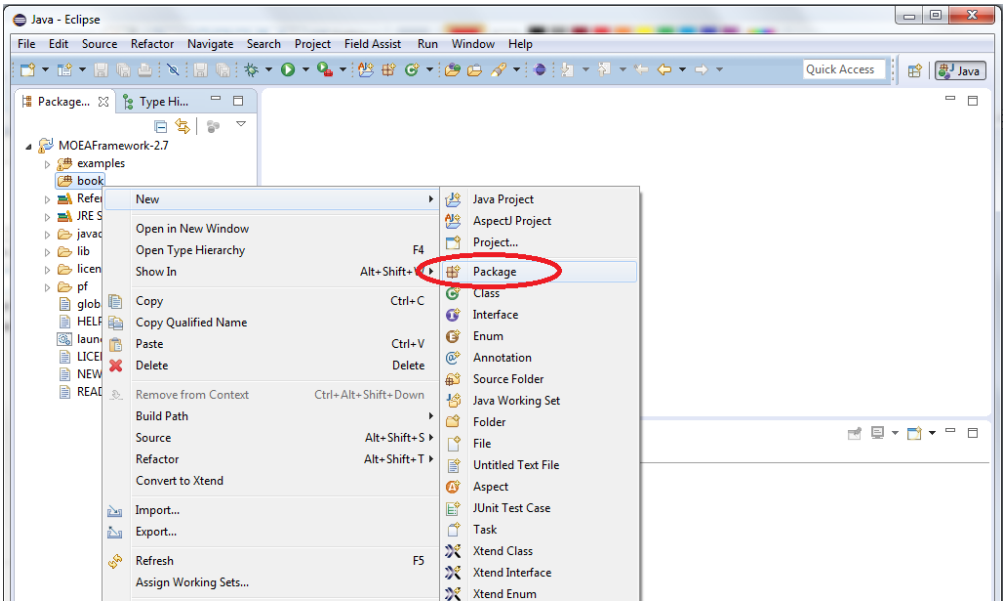


Give the new source folder the name "book" and click Finish.

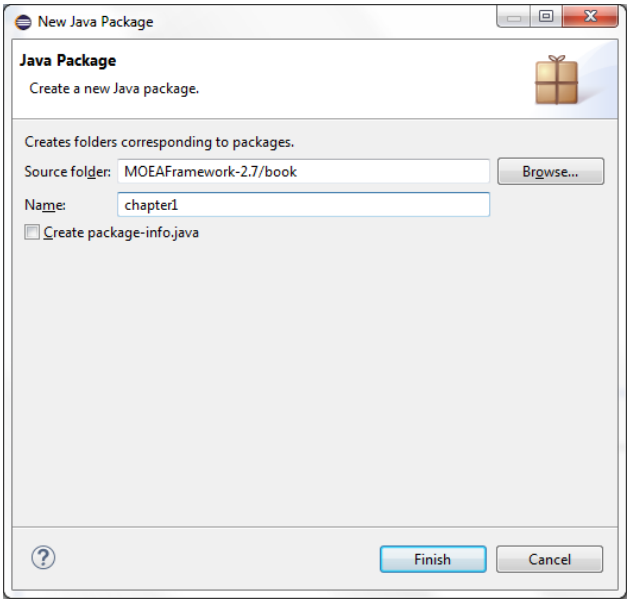


2.4 Your First Example

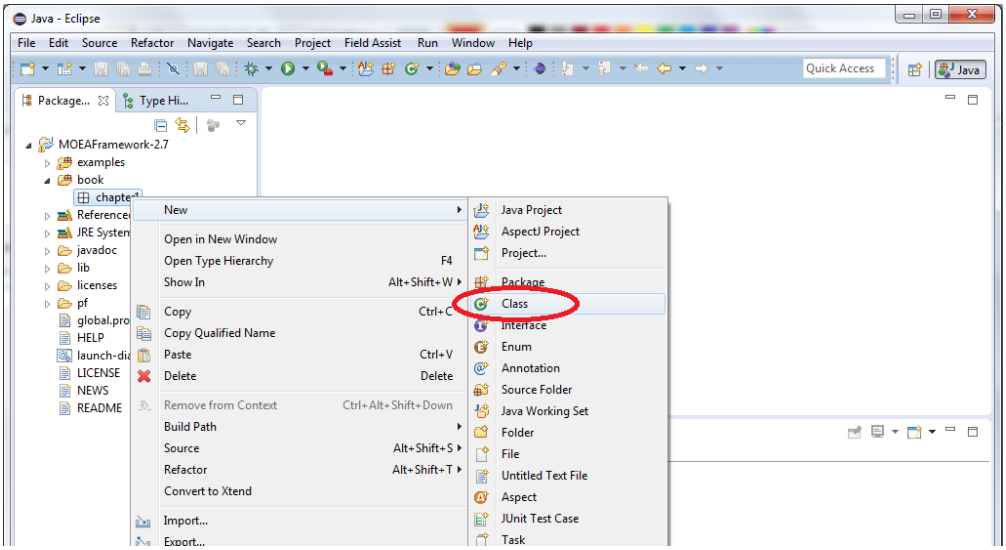
In Java, packages are used to organize source code into a hierarchical structure. We will organize the examples from each chapter into its own package. Thus, for the first chapter, we will create a package named `chapter2`. Right-click on the source folder we just created and select New → Package.



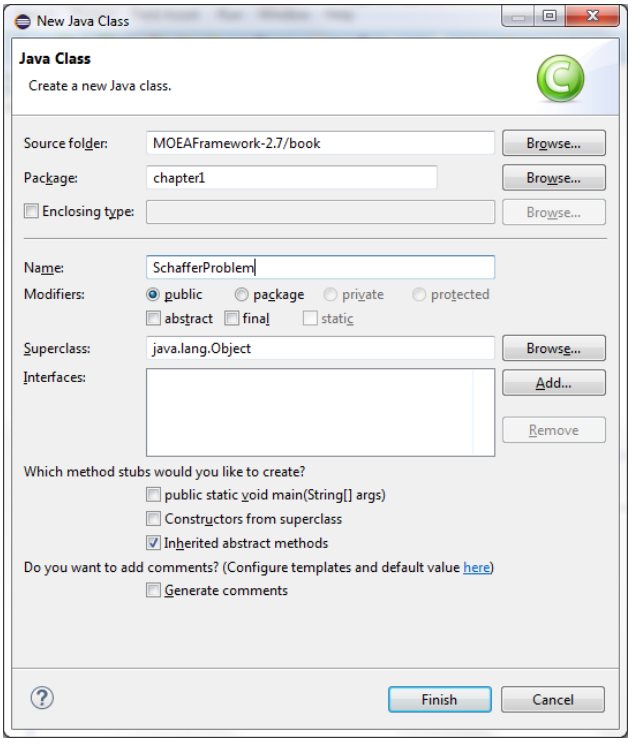
Enter the name `chapter2` and click Finish.



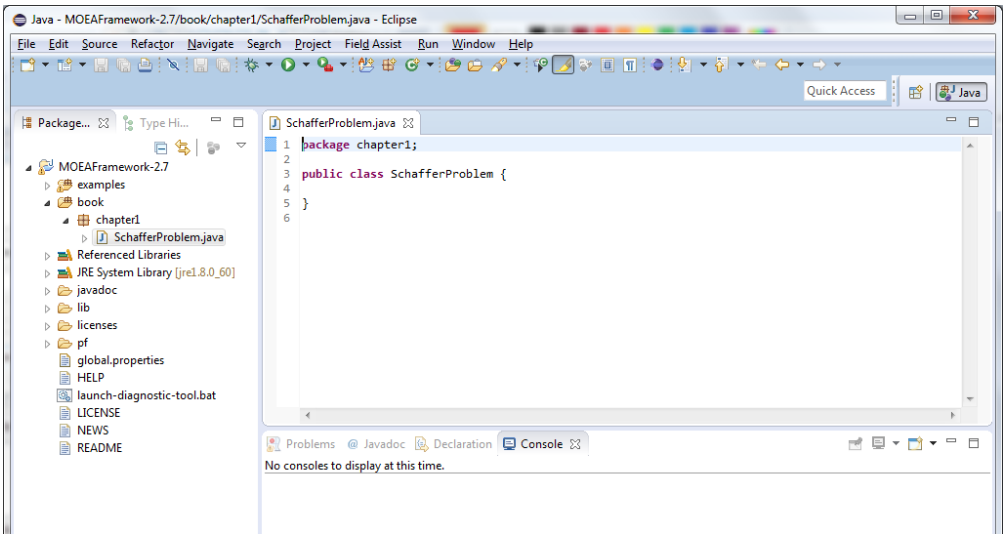
Finally, we create the Java file for the actual code. In Java, these are called classes. Right-click the `chapter2` folder and select New → Class.



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 that 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 were 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.

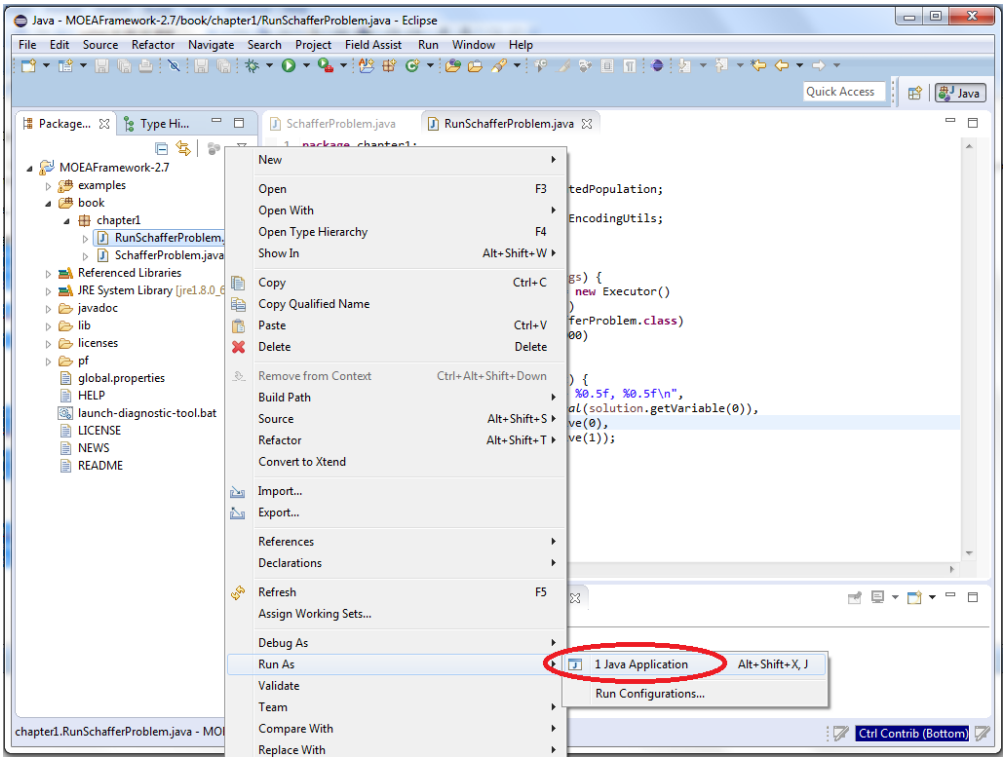
At this point, the problem is defined, but we also need to create the code to solve the problem. To begin, create a new class called `RunSchafferProblem` with the following code:

```
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 }
```

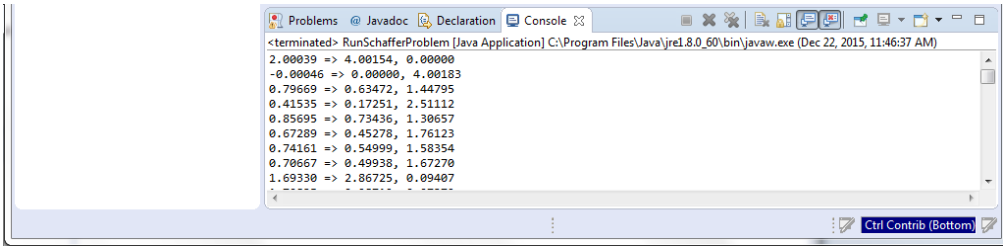
MOEAFramework/book/chapter2/RunSchafferProblem.java

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.RunShafferProblem`. 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

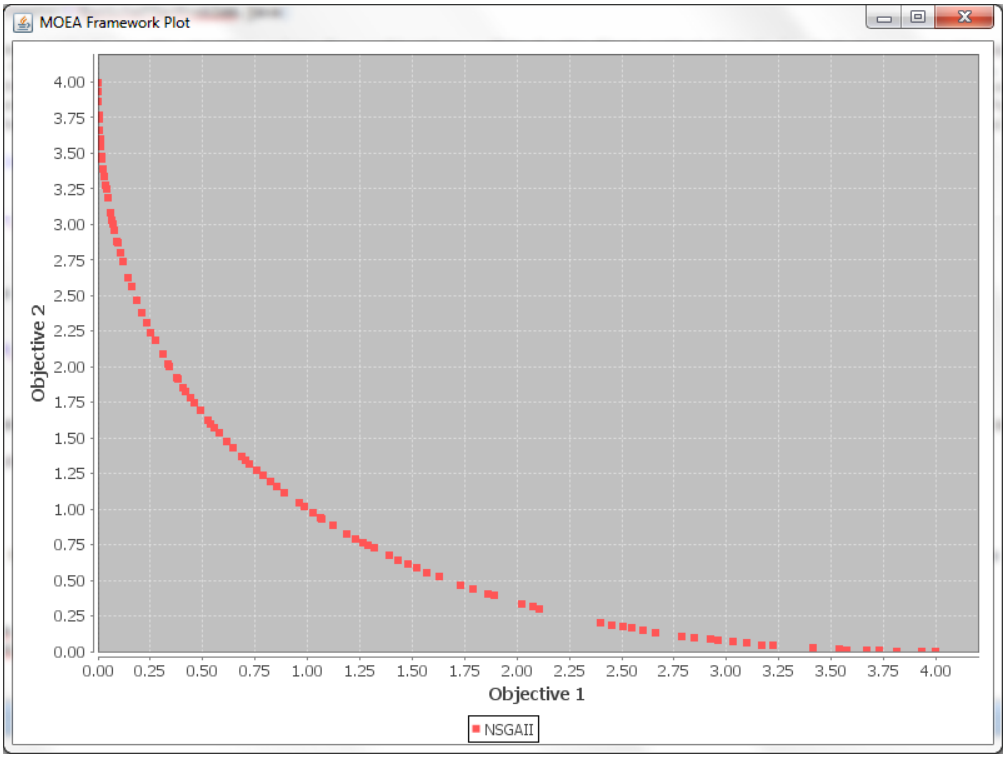
In the previous example, we output the two objectives to the console. Outputting the raw data like this is useful as you can save the data to a text file, load the data into Excel, etc. For problems like this with only two objectives, we can plot the solutions as points in a scatter plot, as shown below:

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

```java
public class PlotSchafferProblem {

  public static void main(String[] args) {
    NondominatedPopulation result = new Executor()
        .withAlgorithm("NSGAII")
        .withProblemClass(SchafferProblem.class)
        .withMaxEvaluations(10000)
        .run();

    new Plot()
        .add("NSGAII", result)
        .show();
  }

}
```

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.

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.

$$F = (f_1(x,y), f_2(x,y)), \text{ where} \qquad -20 \leq x, y \leq 20,$$

$$\begin{aligned} f_1(x,y) &= (x-2)^2 + (y-1)^2 + 2, \\ f_2(x,y) &= 9x - (y-1)^2 \end{aligned} \qquad \begin{aligned} 0 &\geq x^2 + y^2 - 225, \\ 0 &\geq x - 3y + 10 \end{aligned}$$

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 <= 0 ? 0 : x - 3y + 10`. This expression begins with the comparator `x - 3y + 10 <= 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));
```

```
    double y = EncodingUtils.getReal(solution.getVariable(1));
    double f1 = Math.pow(x - 2.0, 2.0) + Math.pow(y - 1.0, 2.0) + 2.0;
    double f2 = 9.0*x - Math.pow(y - 1.0, 2.0);
    double c1 = Math.pow(x, 2.0) + Math.pow(y, 2.0) - 225.0;
    double c2 = x - 3.0*y + 10.0;

    solution.setObjective(0, f1);
    solution.setObjective(1, f2);
    solution.setConstraint(0, c1 <= 0.0 ? 0.0 : c1);
    solution.setConstraint(1, c2 <= 0.0 ? 0.0 : c2);
  }
```

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:

```
    double c = Math.pow(x, 2.0) + y - 10;
    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.

```
  public Solution newSolution() {
    Solution solution = new Solution(2, 2, 2);

    solution.setVariable(0, EncodingUtils.newReal(-20.0, 20.0));
    solution.setVariable(1, EncodingUtils.newReal(-20.0, 20.0));

    return solution;
  }
```

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

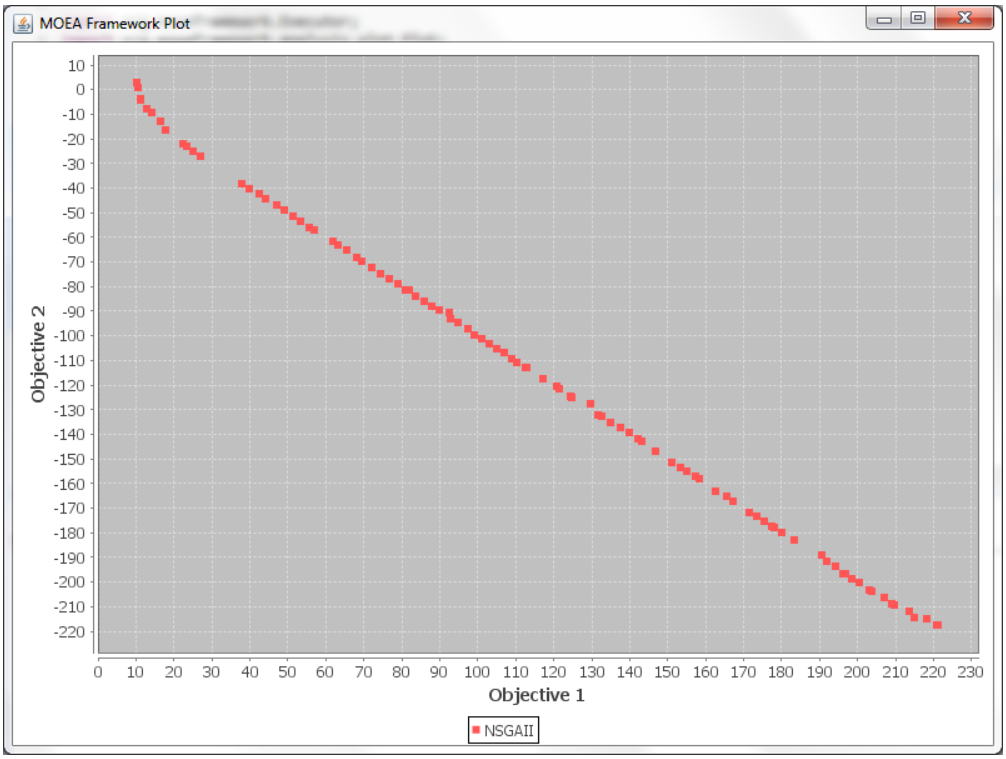
There are a few changes to this code from the previous chapter. First, on line 12 we use the new `SrinivasProblem` class. Second, on line 17, we check if the solutions are feasible prior to printing the output. Most algorithms will only store feasible solutions, but it's always good practice to check 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("NSGAII", 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 $i = 1, 2, \ldots, N$. 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:

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

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 thell 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.

Chapter 4

Choice of Optimization Algorithm

Up to this point, we have been using one of the most popular MOEAs: NSGA-II. NSGA-II is frequently used in practice as it shows strong performance on a range of problems and has been studied extensively in the academic literature. However, the MOEA Framework is not limited to these algorithms. This chapter will discuss how to run different algorithms, how to parameterize the algorithms, and how to compare the performance of algorithms.

4.1 Running Different Algorithms

The MOEA Framework contains an assortment of optimization algorithms for your use, each identified by name. For example, the following optimizes the Schaffer problem using the NSGA-II algorithm:

```
NondominatedPopulation result = new Executor()
    .withProblemClass(SchafferProblem.class)
    .withAlgorithm("NSGAII")
    .withMaxEvaluations(10000)
    .run();
```

Likewise, we can optimize the problem with GDE3 as demonstrated below. Observe that we only needed to change the name of the algorithm.

```
NondominatedPopulation result = new Executor()
    .withProblemClass(SchafferProblem.class)
    .withAlgorithm("GDE3")
    .withMaxEvaluations(10000)
    .run();
```

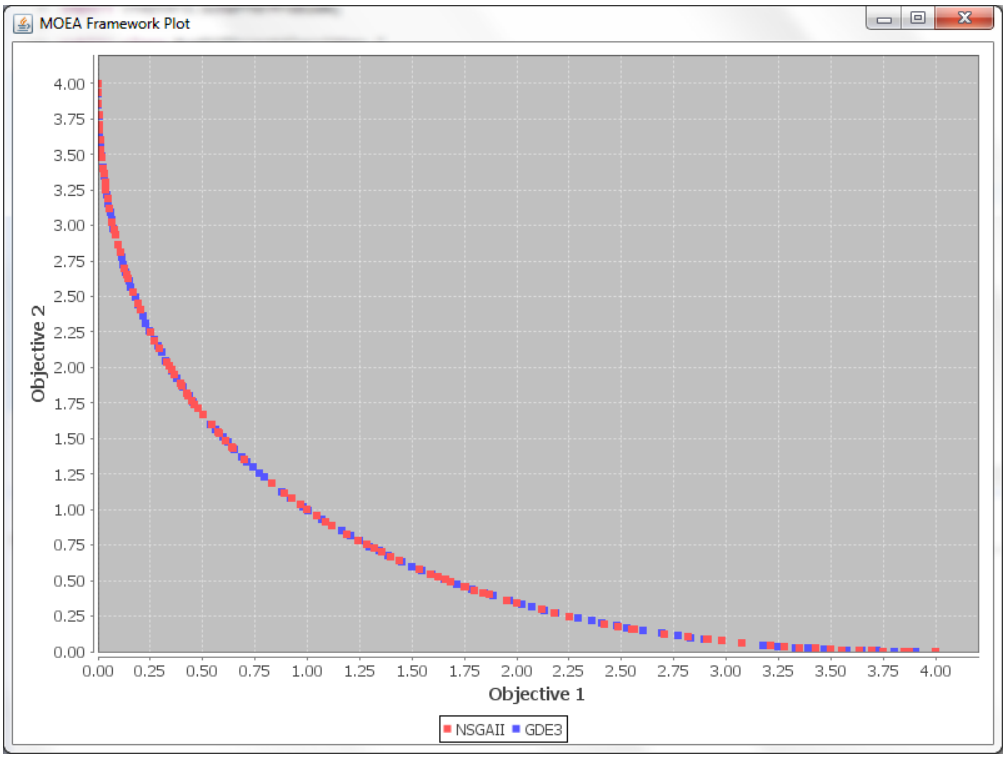
The MOEA Framework handles the rest. It ensures the algorithms are configured appropriately for the given problem or issues a warning or error if it detects any problems. The

full code to run this example and plot the results is provided below.

```
package chapter4;

import java.io.IOException;

import org.moeaframework.Executor;
import org.moeaframework.analysis.plot.Plot;
import org.moeaframework.core.NondominatedPopulation;
import chapter2.SchafferProblem;

public class RunDifferentAlgorithms {

  public static void main(String[] args) throws IOException {
    NondominatedPopulation result1 = new Executor()
        .withProblemClass(SchafferProblem.class)
        .withAlgorithm("NSGAII")
        .withMaxEvaluations(10000)
        .run();

    NondominatedPopulation result2 = new Executor()
        .withProblemClass(SchafferProblem.class)
        .withAlgorithm("GDE3")
        .withMaxEvaluations(10000)
        .run();

    new Plot()
        .add("NSGAII", result1)
        .add("GDE3", result2)
        .show();
  }

}
```

MOEAFramework/book/chapter4/RunDifferentAlgorithms.java

The output, which shows the two algorithms colored red and blue, is shown below. Visually, we can see the two algorithms produce nearly identical results. This visual approach is useful to quickly observe the relative performance of two or more algorithms. Later in this chapter we'll demonstrate how to statistically compare the performance of algorithms.



Note that not all algorithms can solve all problems. Table 4.1 provides a list of the algorithms. The first column provides the name (which you specify in the string to `withAlgorithm`), and the remaining columns indicate if the algorithm supports different encodings and constraints.

In addition to these algorithms, which are implemented natively within the MOEA Framework, algorithms from the population JMetal and PISA libraries can also be executed within the MOEA Framework with little additional work. Refer to the User Manual for additional details.

4.2 Parameterization

In addition to selecting the specific algorithm to use, each algorithm has its own parameters that can be customized. If left untouched, the algorithm uses default parameters based on best practices from the literature. All parameters can also be overridden. Appendix A lists the parameters for each algorithm, provides a short description, and indicates the default value. For example, lets compare the default parameters for NSGA-II to a custom set of parameters. Here, we are reducing the population size from 100 (the default) to 50 and increasing the distribution indices which affect how offspring are generated. We would expect these custom parameters to perform worse in practice, but lets see for ourselves:

```
1 package chapter4;
2
```

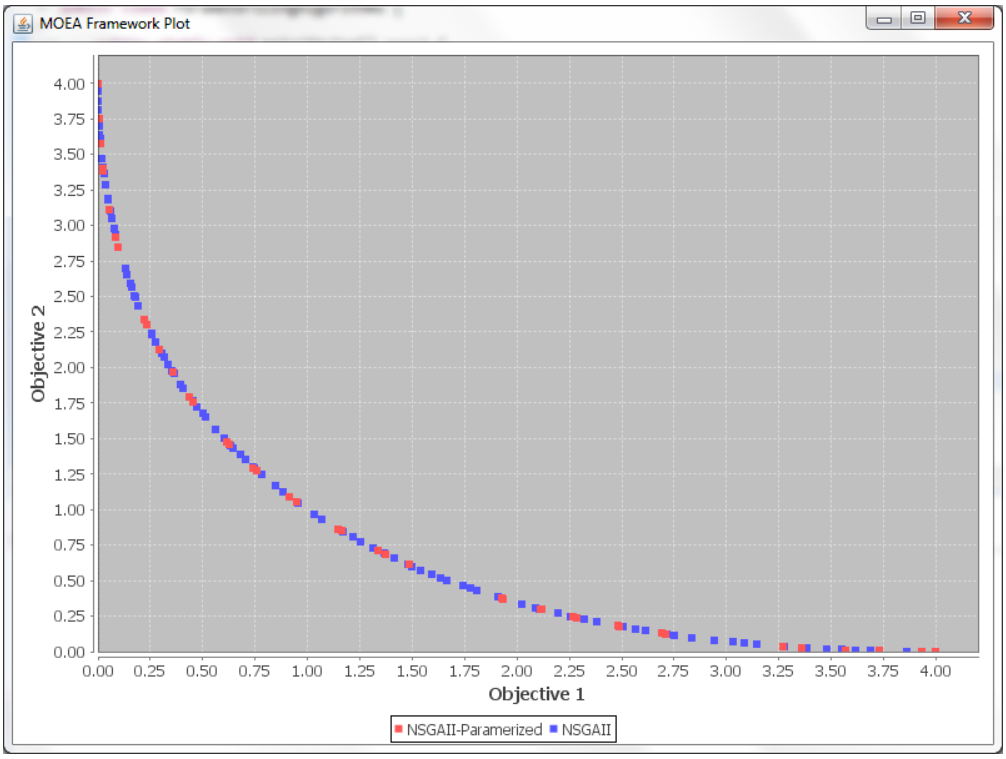
Table 4.1: List of available optimization algorithms.

Algorithm	Type	Real	Binary	Permutation	Subset	Grammar	Program	Constraints
CMA-ES	Evolutionary Strategy	Yes	No	No	No	No	No	Yes
DBEA	Decomposition	Yes	Yes	Yes	Yes	Yes	Yes	Yes
eMOEA	ϵ-Dominance	Yes	Yes	Yes	Yes	Yes	Yes	Yes
eNSGAII	ϵ-Dominance	Yes	Yes	Yes	Yes	Yes	Yes	Yes
GDE3	Differential Evolution	Yes	No	No	No	No	No	Yes
IBEA	Indicator-Based	Yes	Yes	Yes	Yes	Yes	Yes	No
MOEAD	Decomposition	Yes	No	No	No	No	No	Yes
NSGAII	Genetic Algorithm	Yes	Yes	Yes	Yes	Yes	Yes	Yes
NSGAIII	Reference Point	Yes	Yes	Yes	Yes	Yes	Yes	Yes
OMOPSO	Particle Swarm	Yes	No	No	No	No	No	Yes
PAES	Evolutionary Strategy	Yes	Yes	Yes	Yes	Yes	Yes	Yes
PESA2	Genetic Algorithm	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Random	Random Search	Yes	Yes	Yes	Yes	Yes	Yes	Yes
SMPSO	Particle Swarm	Yes	No	No	No	No	No	Yes
SMSEMOA	Indicator-Based	Yes	Yes	Yes	Yes	Yes	Yes	Yes
SPEA2	Genetic Algorithm	Yes	Yes	Yes	Yes	Yes	Yes	Yes
VEGA	Genetic Algorithm	Yes	Yes	Yes	Yes	Yes	Yes	No

```java
import org.moeaframework.Executor;
import org.moeaframework.analysis.plot.Plot;
import org.moeaframework.core.NondominatedPopulation;

import chapter2.SchafferProblem;

public class ParameterizingAlgorithms {

  public static void main(String[] args) {
    NondominatedPopulation result1 = new Executor()
        .withProblemClass(SchafferProblem.class)
        .withAlgorithm("NSGAII")
        .withMaxEvaluations(10000)
        .run();

    NondominatedPopulation result2 = new Executor()
        .withProblemClass(SchafferProblem.class)
        .withAlgorithm("NSGAII")
        .withMaxEvaluations(10000)
        .withProperty("populationSize", 50)
        .withProperty("sbx.rate", 1.0)
        .withProperty("sbx.distributionIndex", 250.0)
        .withProperty("pm.rate", 0.0)
        .withProperty("pm.distributionIndex", 300.0)
        .run();

    new Plot()
        .add("NSGAII-Paramerized", result2)
        .add("NSGAII", result1)
        .show();
  }

}
```

MOEAFramework/book/chapter4/ParameterizingAlgorithms.java

As shown below, the custom parameters (shown in red) does not approximate the Pareto surface as well as the default parameters (shown in blue). The smaller population size results in fewer points, and the larger distribution indices prevents the algorithm from effectively generating new offspring.



Many optimization algorithms support customizable crossover and mutation operators, and these operators have their own parameters founds in Appendix B. For example, NSGA-II has a single parameter, `"populationSize"` and uses the Simulated Binary Crossover and Polynomial Mutation operators for real-valued problems. These two operators have two parameters each. We can customize these parameters as demonstrated below.

```
NondominatedPopulation result = new Executor()
    .withProblemClass(SchafferProblem.class)
    .withAlgorithm("NSGAII")
    .withMaxEvaluations(10000)
    .withProperty("populationSize", 200)
    .withProperty("sbx.rate", 0.9)
    .withProperty("sbx.distributionIndex", 25.0)
    .withProperty("pm.rate", 0.1)
    .withProperty("pm.distributionIndex", 30.0)
    .run();
```

Some algorithms also allow overriding which operators are used. For example, we can choose to use Parent Centric Crossover with NSGA-II:

```
NondominatedPopulation result = new Executor()
    .withProblemClass(SchafferProblem.class)
    .withAlgorithm("NSGAII")
    .withMaxEvaluations(10000)
```

```
    .withProperty("populationSize", 200)
    .withProperty("operator", "pcx")
    .withProperty("pcx.parents", 5)
    .withProperty("pcx.offspring", 1)
    .withProperty("pcx.eta", 0.1)
    .withProperty("pcx.zeta", 0.1)
    .run();
```

Note the use of the `"operator"` parameter to specify the use of Parent Centric Crossover with the string `"pcx"`. Refer to Appendix B for a list of available operators and their string abbreviations. Operators can be combined using the + symbol. For example, by default we use Simulated Binary Crossover (SBX) and Polynomial Mutation (PM), which can be expressed as `"sbx+pm"`. This says to first apply Simulated Binary Crossover on two parents to produce two offspring, then apply Polynomial Mutation to each offspring. Alternatively, we could combine Differential Evolution (DE) and Polynomial Mutation (PM) using the string `"de+pm"`. This is the operator used by GDE3, but as shown below, it can also be used with NSGA-II:

```
NondominatedPopulation result = new Executor()
    .withProblemClass(SchafferProblem.class)
    .withAlgorithm("NSGAII")
    .withProperty("operator", "de+pm")
    .withMaxEvaluations(10000)
    .run();
```

Several precautions should be observed. First, not all combinations of operators are valid. You will see an exception if you request an invalid combination, typically saying *invalid number of parents*. See the User Manual for more details. Second, be careful when passing in the parameters. Parameter names are case sensitive. `"populationsize"` is not the same as `"populationSize"`. Additionally, you will not be notified if an invalid parameter is provided, it will simply be ignored.

4.3 Comparing Algorithms

In academic work, it is often useful to compare the performance of two or more MOEAs. For instance, you could have developed a new search operator or algorithm and wish to compare the performance against existing MOEAs to determine the level of improvement.

The MOEA Framework provides the `Analyzer` class, which is typically used in conjunction with the `Executor` class we have seen previously, to analyze the performance of algorithms. Additionally, it can statistically compare the performance of two or more algorithms to determine, for a given confidence level, if there is any statistical difference between the two algorithms. For example, the following code compares three algorithms — NSGA-II,

GDE3, and ϵ-MOEA — on the UF1 test problem.

```
1  package chapter4;
2
3  import java.io.IOException;
4
5  import org.moeaframework.Analyzer;
6  import org.moeaframework.Executor;
7  import org.moeaframework.analysis.plot.Plot;
8
9  public class ComparingAlgorithms {
10
11   public static void main(String[] args) throws IOException {
12     String problem = "UF1";
13     String[] algorithms = { "NSGAII", "GDE3", "eMOEA" };
14
15     //setup the experiment
16     Executor executor = new Executor()
17         .withProblem(problem)
18         .withMaxEvaluations(10000);
19
20     Analyzer analyzer = new Analyzer()
21         .withSameProblemAs(executor)
22         .includeHypervolume()
23         .showStatisticalSignificance();
24
25     //run each algorithm for 50 seeds
26     for (String algorithm : algorithms) {
27       analyzer.addAll(algorithm,
28           executor.withAlgorithm(algorithm).runSeeds(50));
29     }
30
31     //print the results
32     analyzer.printAnalysis();
33
34     //plot the results
35     new Plot()
36         .add(analyzer)
37         .show();
38   }
39
40 }
```

MOEAFramework/book/chapter4/ComparingAlgorithms.java

On lines 16-18, we setup the `Executor` for our test problem, UF1. Note that unlike the previous examples of using the `Executor`, we do not specify the algorithm. We will do this later. Next, on lines 20-23, the setup the `Analyzer`. We again must indicate on line 21 which problem we are solving so it can compute the performance metrics correctly. On line 22 we specify that we are interested in the hypervolume performance metric and on line 23 request that we compute the statistical significance of the results (with 95% confidence level

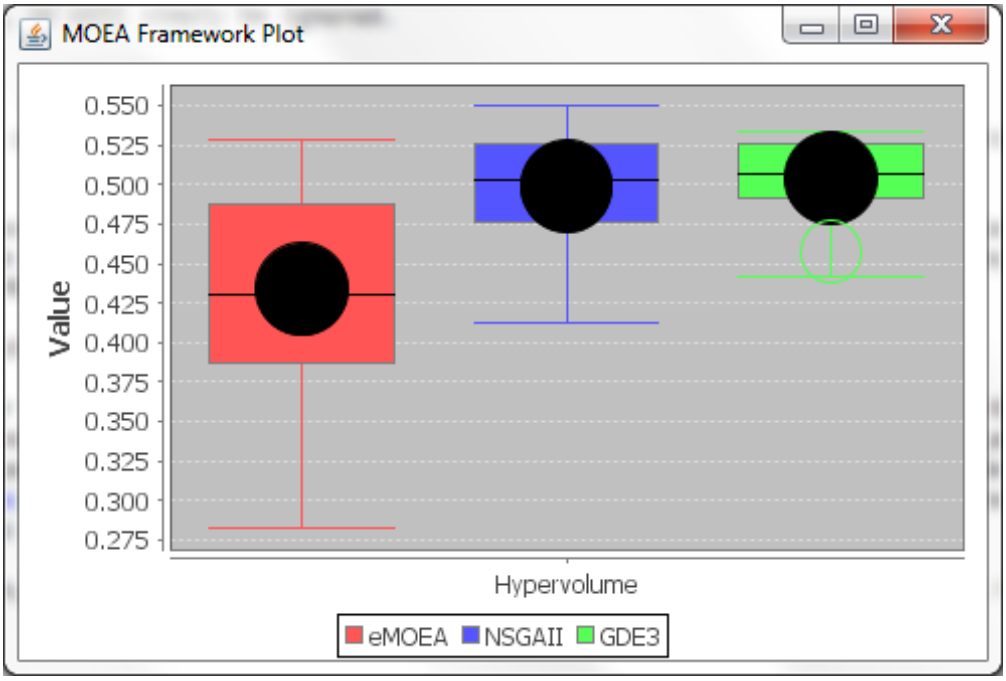
by default). The for loop in lines 26-29 actually performs the analysis. Looping over each algorithm in our study, we setup the `Executor` to run the given algorithm for 50 seeds on line 28. The results of the 50 seeds is added to the analyzer on line 27. Finally, we print the output on line 32. Alternatively, we can plot the differences visually on lines 35-37.

The text output of this code displays the minimum, median, and maximum hypervolume value achieved by each algorithm and the statistical significance.

```
GDE3:
    Hypervolume:
        Min: 0.46862049421426344
        Median: 0.5035596089917667
        Max: 0.534267455034584
        Count: 50
        Indifferent: []
eMOEA:
    Hypervolume:
        Min: 0.3784906829802386
        Median: 0.4503635874660789
        Max: 0.5322735014135916
        Count: 50
        Indifferent: [NSGAII]
NSGAII:
    Hypervolume:
        Min: 0.33173049116232906
        Median: 0.45913525261150123
        Max: 0.5370813537070005
        Count: 50
        Indifferent: [eMOEA]
```

When checking for the statistical significance as we have done, the output will contain the `Indifferent` field listing, if any, the algorithms with statistically indifferent performance. That is to say, with 95% confidence, the performance of NSGA-II and eMOEA are identical, or not statistically different.

The graphical output of this code produces the box-and-whisker plot shown below. The results of each algorithm is represented by a colored box. The filled, rectangular box shows the range encompassed by the 25% to 75% percentile (i.e., the middle 50% of the data) and the horizontal line shows the median. The colored whiskers (the thin lines at the top and bottom of the rectangular box) show the regular min and max of the data. The solid black circle shows the mean. Using this plot, we can see quickly that ϵ-MOEA performs worse than GDE3 and NSGA-II (larger values are preferred).



In addition to the hypervolume performance metric show above, the MOEA Framework also supports generational distance, inverted generational distance, additive ϵ-indicator, maximum Pareto front error, spacing, contribution, and the R2 indicator (Knowles and Corne, 2002; Coello Coello et al., 2007; Hansen et al., 1998). The R1 and R3 indicators are also available, but are not often used in practice as the R2 indicator exhibits better properties.

Table 4.2 shows the supported indicators and key properties. Pareto compliant implies that better indicator values correspond to approximation sets that are preferred by weak Pareto dominance. This is a good property for performance metrics as better values (closer to the target) indicate better Pareto approximate sets.

Each indicator is enabled by calling the corresponding include method, such as `includeHypervolume()` for the hypervolume metric. In this example, we analyzed the UF1 test problem which has a known reference set. The reference set contains all Pareto optimal solutions for a given problem (or a good approximation of the set). The MOEA Framework automatically loads the reference sets for built-in problems. For custom problems, there are two options. First, if you do not explicitly load a reference set, then the `Analyzer` automatically combines all results to produce a reference set. In this case, the reference set contains all Pareto optimal solutions generated by the runs. Alternatively, you can call `analyzer.withReferenceSet(new File("set.txt"))` to load the reference set from a file. This file simply contains the objective values for each Pareto optimal solution. For example, if the Pareto set was a straight line from (0, 1) to (1, 0), then the reference set file would be:

```
0.0 1.0
0.1 0.9
0.2 0.8
```

Table 4.2: List of available performance metrics.

Indicator	Pareto Compliant	Reference Set Required	Normalized	Target
Hypervolume	Yes	Yes	Yes	Maximize $\rightarrow$ 1
Generational Distance	No	Yes	Yes	Minimize $\rightarrow$ 0
Inverted Generational Distance	No	Yes	Yes	Minimize $\rightarrow$ 0
Additive ϵ-Indicator	Yes	No	Yes	Minimize $\rightarrow$ 0
Maximum Pareto Front Error	No	Yes	Yes	Minimize $\rightarrow$ 0
Spacing	No	No	No	Minimize $\rightarrow$ 0
Contribution	Yes	Yes	No	Maximize $\rightarrow$ 1
R2 Indicator	No	Yes	Yes	Minimize $\rightarrow$ -1

```
0.3 0.7
0.4 0.6
0.5 0.5
0.6 0.4
0.7 0.3
0.8 0.2
0.9 0.1
1.0 0.0
```

The ordering of solutions in the reference set file does not matter.

4.4 Runtime Dynamics

In the previous section, we saw how to statistically compare the performance of two or more algorithms. In doing so, we are only comparing the end-of-run performance of the algorithm using only the final Pareto approximation. It can also be useful to investigate the intermediate performance of an algorithm. That is, we want to see how the performance of the algorithm evolves over time. Additionally, we can also inspect other properties of the algorithm to see how, internally, they adapt over time.

To collect runtime dynamics, we introduce the `Instrumenter` class. The `Instrumenter` gets its name from its ability to add instrumentation, which are pieces of code that record information, to an algorithm. A variety of data can be collected using the `Instrumenter`, including:

1. Elapsed time
2. Population size / archive size
3. The approximation set
4. Performance metrics
5. Restart frequency

The `Instrumenter` works hand-in-hand with the `Executor` to collect its data. The `Executor` is responsible for configuring and running the algorithm, but it allows the `Instrumenter` to record the necessary data while the algorithm is running. To start collecting run-time dynamics, we first create and configure an `Instrumenter` instance.

```
1    Instrumenter instrumenter = new Instrumenter()
2        .withProblem("UF1")
3        .withFrequency(100)
4        .attachElapsedTimeCollector()
5        .attachGenerationalDistanceCollector();
```

First, line 1 creates the new `Instumenter` instance. Next, line 2 specifies the problem. This allows the instrumenter to access the known reference set for this problem, which is necessary for evaluating performance metrics. Third, line 3 sets the frequency at which the data is recorded. In this example, data is recorded every 100 evaluations. Lastly, lines 4 and 5 indicate that only the elapsed time and generational distance should be recorded. Calling `.attachAll()` will collect all available data.

Next, we create and configure the Executor instance with the following code snippet:

```
1    new Executor()
2        .withSameProblemAs(instrumenter)
3        .withAlgorithm("NSGAII")
4        .withMaxEvaluations(10000)
5        .withInstrumenter(instrumenter)
6        .run();
```

This code snippet is similar to the previous examples of the Executor, but includes the addition of line 5. Line 5 tells the executor that all algorithms it executes will be instrumented with our Instrumenter instance. Once the instrumenter is set and the algorithm is configured, we can run the algorithm on line 6.

When the run completes, we can access the data collected by the instrumenter. The data is stored in an `Accumulator` object. The `Accumulator` for the run we just executed can be retrieved with the following line:

```
1    Accumulator accumulator = instrumenter.getLastAccumulator();
```

An `Accumulator` is similar to a Map in that it stores key-value pairs. The key identifies the type of data recorded. However, instead of storing a single value, the `Accumulator` stores many values, one for each datapoint collected by the `Instrumenter`. Recall that in this example, we are recording a datapoint every 100 evaluations (i.e., the frequency). We could choose to either print the contents to the console:

```
1    System.out.println(accumulator.toCSV());
```

or save the contents to a CSV file:

```
1    accumulator.saveCSV(new File("output.csv"));
```

or plot the data in a line graph:

```
1    new Plot()
2        .add(accumulator)
```

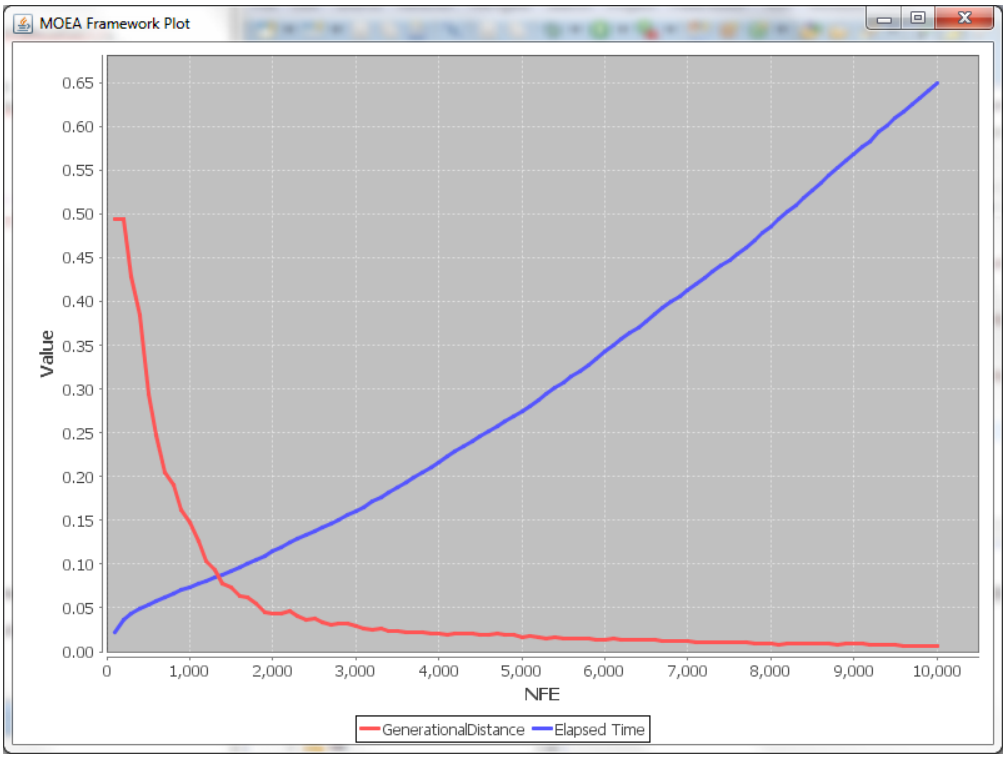
```
        .show();
```

The full code example for collecting the runtime dynamics and plotting the results is shown below:

```
package chapter4;

import java.io.IOException;

import org.moeaframework.Executor;
import org.moeaframework.Instrumenter;
import org.moeaframework.analysis.collector.Accumulator;
import org.moeaframework.analysis.plot.Plot;

public class RuntimeDynamics {

  public static void main(String[] args) throws IOException {
    Instrumenter instrumenter = new Instrumenter()
        .withProblem("UF1")
        .withFrequency(100)
        .attachElapsedTimeCollector()
        .attachGenerationalDistanceCollector();

    new Executor()
        .withSameProblemAs(instrumenter)
        .withAlgorithm("NSGAII")
        .withMaxEvaluations(10000)
        .withInstrumenter(instrumenter)
        .run();

    Accumulator accumulator = instrumenter.getLastAccumulator();

    new Plot()
        .add(accumulator)
        .show();
  }

}
```

MOEAFramework/book/chapter4/RuntimeDynamics.java

This code produces the following plot:



Rather than plotting all the data in an accumulator, you can also plot specific metrics. As shown below, we only plot the generational distance:

```
1    new Plot()
2        .add("GD", accumulator, "GenerationalDistance")
3        .show();
```

Chapter 5

Customizing Algorithms

In previous chapters, we used the `Executor` class to optimize problems. The `Executor` is great for quickly solving problems using predefined optimization algorithms. However, it is not very customizable. This chapter discusses step for customizing algorithms.

5.1 Manually Running Algorithms

Before we discuss how to customize an algorithm, lets start by looking at what happens behind the scenes in the `Executor` class. Take the following code, for example.

```
1     NondominatedPopulation result = new Executor()
2         .withAlgorithm("NSGAII")
3         .withProblemClass(SchafferProblem.class)
4         .withMaxEvaluations(10000)
5         .run();
```

This code creates a new instance of the NSGA-II algorithm and uses it to optimize the Schaffer problem. During optimization, it spends at most 10,000 evaluations. Finally, the Pareto approximate result is saved to the `result` variable. Underneath the hood, these same steps can be accomplished as shown below:

```
1  package chapter5;
2
3  import org.moeaframework.algorithm.NSGAII;
4  import org.moeaframework.core.Algorithm;
5  import org.moeaframework.core.Initialization;
6  import org.moeaframework.core.NondominatedPopulation;
7  import org.moeaframework.core.NondominatedSortingPopulation;
8  import org.moeaframework.core.Problem;
9  import org.moeaframework.core.Variation;
10 import org.moeaframework.core.comparator.ChainedComparator;
11 import org.moeaframework.core.comparator.CrowdingComparator;
```

```java
import org.moeaframework.core.comparator.ParetoDominanceComparator;
import org.moeaframework.core.operator.GAVariation;
import org.moeaframework.core.operator.RandomInitialization;
import org.moeaframework.core.operator.TournamentSelection;
import org.moeaframework.core.operator.real.PM;
import org.moeaframework.core.operator.real.SBX;
import chapter2.SchafferProblem;

public class ManualRun {

  public static void main(String[] args) {
    // define the problem
    Problem problem = new SchafferProblem();

    // create an initial random population of 100 individuals
    Initialization initialization = new RandomInitialization(
        problem,
        100);

    // define the selection operator
    TournamentSelection selection = new TournamentSelection(2,
        new ChainedComparator(
            new ParetoDominanceComparator(),
            new CrowdingComparator()));

    // define the crossover / mutation operator
    Variation variation = new GAVariation(
        new SBX(1.0, 25.0),
        new PM(1.0 / problem.getNumberOfVariables(), 30.0));

    // construct the algorithm
    Algorithm algorithm = new NSGAII(
        problem,
        new NondominatedSortingPopulation(),
        null, // no archive
        selection,
        variation,
        initialization);

    // run the algorithm for 10,000 evaluations
    while (algorithm.getNumberOfEvaluations() < 10000) {
      algorithm.step();
    }

    // get the Pareto approximate results
    NondominatedPopulation result = algorithm.getResult();
  }

}
```

MOEAFramework/book/chapter5/ManualRun.java

As can be seen, we must define each component of the algorithm manually, as listed below:

Line 24 Problem definition

Lines 27-29 Initialization operator - Produces a randomly-generated initial population with 100 individuals

Lines 32-35 Selection operator - Binary tournament selecting using Pareto dominance and crowding distance

Lines 38-40 Variation operator - Simulated binary crossover with polynomial mutation

Lines 43-49 Algorithm definition - NSGA-II with the given operators

Lines 52-54 Runs the algorithm for 10,000 evaluations

Line 57 Retrieve the Pareto approximate results

Customizing an algorithm requires changing the components above as desired. In the remainder of this chapter we'll consider several customizations.

5.2 Custom Initialization

Lets say instead of the randomly-generated initial population, we would rather use the Latin hypercube distribution. We would begin by implementing the `Initialization` interface and generating the initial population using a Latin hypercube distribution:

```
1  package chapter5;
2
3  import org.moeaframework.core.Initialization;
4  import org.moeaframework.core.Problem;
5  import org.moeaframework.core.Solution;
6  import org.moeaframework.core.variable.RealVariable;
7  import org.moeaframework.util.sequence.LatinHypercube;
8
9  public class LatinHypercubeInitialization implements Initialization {
10
11   private Problem problem;
12
13   private int size;
14
15   public LatinHypercubeInitialization(Problem problem, int size) {
16     super();
17     this.problem = problem;
18     this.size = size;
19   }
20
21   @Override
```

```
22   public Solution[] initialize() {
23     LatinHypercube lhs = new LatinHypercube();
24     double[][] samples = lhs.generate(size, problem.getNumberOfObjectives());
25     Solution[] result = new Solution[size];
26
27     for (int i = 0; i < size; i++) {
28       Solution solution = problem.newSolution();
29
30       for (int j = 0; j < problem.getNumberOfVariables(); j++) {
31         RealVariable variable = (RealVariable)solution.getVariable(j);
32         variable.setValue(samples[i][j] * (variable.getUpperBound()-variable.
               getLowerBound())
33             + variable.getLowerBound());
34       }
35
36       result[i] = solution;
37     }
38
39     return result;
40   }
41
42 }
```

MOEAFramework/book/chapter5/LatinHypercubeInitialization.java

When implementing the `Initialization` interface, we must include the `Solution [] initialize()` method to return the initial population. Fortunately, we already have code that produces the Latin hypercube samples (LHS), but they are scaled on the range $[0, 1]$. Thus, we first generate the LHS samples on the range $[0, 1]$ on lines 23-24. Next, we loop over each of these samples on line 27. For each sample, we create a new `Solution` instance (line 28). On lines 30-34, we assign the value of each decision variable, being sure to scale from the range $[0, 1]$ to the lower and upper bounds of the decision variable. Each solution is added to the `result` array and returned on line 39.

We then replace the `RandomInitialization` class with our new `LatinHypercubeInitialization` on line 27-29 below:

```
1  package chapter5;
2
3  import org.moeaframework.algorithm.NSGAII;
4  import org.moeaframework.core.Algorithm;
5  import org.moeaframework.core.Initialization;
6  import org.moeaframework.core.NondominatedPopulation;
7  import org.moeaframework.core.NondominatedSortingPopulation;
8  import org.moeaframework.core.Problem;
9  import org.moeaframework.core.Variation;
10 import org.moeaframework.core.comparator.ChainedComparator;
11 import org.moeaframework.core.comparator.CrowdingComparator;
12 import org.moeaframework.core.comparator.ParetoDominanceComparator;
13 import org.moeaframework.core.operator.GAVariation;
14 import org.moeaframework.core.operator.TournamentSelection;
```

```java
import org.moeaframework.core.operator.real.PM;
import org.moeaframework.core.operator.real.SBX;

import chapter2.SchafferProblem;

public class CustomInitialization {

  public static void main(String[] args) {
    // define the problem
    Problem problem = new SchafferProblem();

    // create an LHS initial population with 100 individuals
    Initialization initialization = new LatinHypercubeInitialization(
        problem,
        100);

    // define the selection operator
    TournamentSelection selection = new TournamentSelection(2,
        new ChainedComparator(
            new ParetoDominanceComparator(),
            new CrowdingComparator()));

    // define the crossover / mutation operator
    Variation variation = new GAVariation(
        new SBX(1.0, 25.0),
        new PM(1.0 / problem.getNumberOfVariables(), 30.0));

    // construct the algorithm
    Algorithm algorithm = new NSGAII(
        problem,
        new NondominatedSortingPopulation(),
        null, // no archive
        selection,
        variation,
        initialization);

    // run the algorithm for 10,000 evaluations
    while (algorithm.getNumberOfEvaluations() < 10000) {
      algorithm.step();
    }

    // get the Pareto approximate results
    NondominatedPopulation result = algorithm.getResult();
  }

}
```

MOEAFramework/book/chapter5/CustomInitialization.java

5.3 Custom Algorithms

It is very convenient to develop new optimization algorithms within the MOEA Framework. Not only can you build the algorithm using the existing components already built into the MOEA Framework, but your algorithm will work with all of the analytical and plotting tools available.

All optimization algorithms implemented within the MOEA Framework must implement the `Algorithm` interface. This interfaces defines several methods that must be implemented. For convenience, several abstract classes are also provided, including:

Algorithm - Used when implementing an algorithm that can not use any of the abstract classes discussed below. Provides the greatest flexibility, but requires the most development.

AbstractAlgorithm - Used when creating iterative optimization algorithms. The developer must provide the initialize() and iterate() method.

AbstractEvolutionaryAlgorithm - Used when creating evolutionary algorithms that typically optimize a population of candidate solutions.

AbstractPSOAlgorithm - Used when implementing particle swarm optimization algorithms. Provides structures for storing and updating particle positions, velocities, etc.

To demonstrate the creation of a new algorithm, we will construct a very simple optimization algorithm called the `RandomWalker`. The overall algorithm is as follows:

1. Initialize the population with random solutions
2. While NFE $<$ MaxNFE
   (a) Randomly pick a solution from the population
   (b) Mutate the individual with polynomial mutation
   (c) Add the offspring to the population
   (d) Use non-dominated sorting to remove the worst member of the population

From the above description, we see that the algorithm uses a population of solutions. Therefore, we will extend the `AbstractEvolutionaryAlgorithm`, as shown below.

```
1 package chapter5;
2
3 import org.moeaframework.algorithm.AbstractEvolutionaryAlgorithm;
4 import org.moeaframework.core.Initialization;
5 import org.moeaframework.core.NondominatedSortingPopulation;
6 import org.moeaframework.core.PRNG;
7 import org.moeaframework.core.Problem;
```

```
8  import org.moeaframework.core.Solution;
9  import org.moeaframework.core.Variation;
10
11 public class RandomWalker extends AbstractEvolutionaryAlgorithm {
12
13   private final Variation variation;
14
15   public RandomWalker(Problem problem, Initialization initialization,
16       Variation variation) {
17     super(problem, new NondominatedSortingPopulation(), null, initialization);
18     this.variation = variation;
19   }
20
21   @Override
22   protected void iterate() {
23     // get the current population
24     NondominatedSortingPopulation population = (NondominatedSortingPopulation)
         getPopulation();
25
26     // randomly select a solution from the population
27     int index = PRNG.nextInt(population.size());
28     Solution parent = population.get(index);
29
30     // mutate the selected solution
31     Solution offspring = variation.evolve(new Solution[] { parent })[0];
32
33     // evaluate the objectives/constraints
34     evaluate(offspring);
35
36     // add the offspring to the population
37     population.add(offspring);
38
39     // use non-dominated sorting to remove the worst solution
40     population.truncate(population.size()-1);
41   }
42
43 }
```

MOEAFramework/book/chapter5/RandomWalker.java

Observe that our constructor on line 15 accepts three arguments: the problem definition, the initialization procedure, and the variation operator. We pass the problem and initialization procedure to the super class (`AbstractEvolutionaryAlgorithm`). Also note that we pass a new `NondominatedSortingPopulation` to the super class, which is used to store the population. `AbstractEvolutionaryAlgorithm` will handle all of the messy details of initializing and maintaining the population.

Since we extended the `AbstractEvolutionaryAlgorithm` class, we only need to implement the `iterate()` method. The `iterate()` method, starting on line 22, describes one iteration of the algorithm. Depending on the type of algorithm, the `iterate()` method could perform a little or a lot of work. For example, in our `RandomWalker` example, each

iteration will generate one offspring. In a generational algorithm like NSGA-II, all members of the population are evolved every iteration. It is up to the designer of the algorithm to determine how much work is performed in the `iterate()` method. At the very least, however, at least one offspring should be generated each iteration.

We can test our `RandomWalker` class as follows:

```
1  package chapter5;
2
3  import org.moeaframework.Executor;
4  import org.moeaframework.core.NondominatedPopulation;
5  import org.moeaframework.core.spi.AlgorithmFactory;
6  import chapter2.SchafferProblem;
7
8  public class RunHyperheuristic {
9
10   public static void main(String[] args) {
11     AlgorithmFactory.getInstance().addProvider(new HyperheuristicProvider());
12
13     NondominatedPopulation result = new Executor()
14         .withAlgorithm("hyperheuristic")
15         .withProblemClass(SchafferProblem.class)
16         .withMaxEvaluations(10000)
17         .run();
18   }
19
20 }
```

MOEAFramework/book/chapter5/RunHyperheuristic.java

5.4 Creating Service Providers

The MOEA Framework is designed to be flexible. It uses what's called a Service Provider Interface, or SPI, to dynamically load algorithms. This allows new optimization algorithms to be introduced within the MOEA Framework without modifying the MOEA Framework's code. To do this, we will create a new Java class called `RandomWalkerProvider` and extend the `AlgorithmProvider` interface. The contents of this file are shown below.

```
1  package chapter5;
2
3  import java.util.Properties;
4
5  import org.moeaframework.core.Algorithm;
6  import org.moeaframework.core.Initialization;
7  import org.moeaframework.core.Problem;
8  import org.moeaframework.core.Variation;
9  import org.moeaframework.core.operator.RandomInitialization;
10 import org.moeaframework.core.spi.AlgorithmProvider;
```

```java
import org.moeaframework.core.spi.OperatorFactory;
import org.moeaframework.util.TypedProperties;

public class RandomWalkerProvider extends AlgorithmProvider {

  @Override
  public Algorithm getAlgorithm(String name, Properties properties, Problem
      problem) {
    if (name.equalsIgnoreCase("RandomWalker")) {
      // if the user requested the RandomWalker algorithm
      TypedProperties typedProperties = new TypedProperties(properties);

      // allow the user to customize the population size (default to 100)
      int populationSize = typedProperties.getInt("populationSize", 100);

      // initialize the algorithm with randomly-generated solutions
      Initialization initialization = new RandomInitialization(problem,
          populationSize);

      // use the operator factory to create a polynomial mutation operator
      Variation variation = OperatorFactory.getInstance().getVariation("pm",
          properties, problem);

      // construct and return the RandomWalker algorithm
      return new RandomWalker(problem, initialization, variation);
    } else {
      // return null if the user requested a different algorithm
      return null;
    }
  }

}
```

MOEAFramework/book/chapter5/RandomWalkerProvider.java

If the user requests an optimization algorithm with the name `"RandomWalker"`, the above class will create and return an instance of our `RandomWalker` class. Note that a properties object is passed to the `getAlgorithm` method. This properties object contains any custom properties set by the user, which allows the user to provide special parameters for the algorithm such as the population size.

Before we can use our new algorithm, we must register it with the MOEA Framework. The code below shows how to register and use our new optimization algorithm:

```java
package chapter5;

import org.moeaframework.Executor;
import org.moeaframework.core.NondominatedPopulation;
import org.moeaframework.core.spi.AlgorithmFactory;

import chapter2.SchafferProblem;
```

```
8
9  public class RunWithProvider {
10
11   public static void main(String[] args) {
12     AlgorithmFactory.getInstance().addProvider(new RandomWalkerProvider());
13
14     NondominatedPopulation result = new Executor()
15         .withAlgorithm("RandomWalker")
16         .withProblemClass(SchafferProblem.class)
17         .withMaxEvaluations(10000)
18         .run();
19   }
20
21 }
```

MOEAFramework/book/chapter5/RunWithProvider.java

Great! Now we can test our new algorithm with the `Analyzer` class or inspect its runtime dynamics with the `Instrumenter` class. For example, below plots the convergence of the algorithm using generational distance:

```
1  package chapter5;
2
3  import org.moeaframework.Executor;
4  import org.moeaframework.Instrumenter;
5  import org.moeaframework.analysis.collector.Accumulator;
6  import org.moeaframework.analysis.plot.Plot;
7  import org.moeaframework.core.spi.AlgorithmFactory;
8
9  public class PlotDynamicsWithProvider {
10
11   public static void main(String[] args) {
12     AlgorithmFactory.getInstance().addProvider(new RandomWalkerProvider());
13
14     Instrumenter instrumenter = new Instrumenter()
15         .withProblem("UF1")
16         .withFrequency(100)
17         .addAllowedPackage("chapter5")
18         .attachGenerationalDistanceCollector();
19
20     new Executor()
21         .withSameProblemAs(instrumenter)
22         .withAlgorithm("RandomWalker")
23         .withMaxEvaluations(10000)
24         .withInstrumenter(instrumenter)
25         .run();
26
27     Accumulator accumulator = instrumenter.getLastAccumulator();
28
29     new Plot().add(accumulator).show();
30   }
```

```
31
32 }
```

MOEAFramework/book/chapter5/PlotDynamicsWithProvider.java

Note we must call `addAllowedPackage("chapter5")` on line 17 to allow the instrumenter to work with code in our custom package.

5.5 Hyperheuristics

A hyperheuristic combines two or more other heuristics[1]. The general idea is that a given heuristic works well one some problems but performs poorly on others. By utilizing more than one heuristic, we avoid being stuck using a poorly performing heuristic. There are many examples of this in the literature, including work on AMALGAM (Vrugt and Robinson, 2007; Vrugt et al., 2009) and Borg (Hadka and Reed, 2013). These hyperheuristics employ some learning mechanism to determine which heuristics perform better on a given problem. In this book, for simplicity, we won't use any learning mechanism and instead simply run both heuristics an equal number of times. In this example, we will combine NSGA-II and GDE3. This exercise is intended to demonstrate how using modular designs is beneficial. We will combine these two algorithms with a minimal amount of code:

```
1  package chapter5;
2
3  import org.moeaframework.algorithm.AbstractEvolutionaryAlgorithm;
4  import org.moeaframework.algorithm.GDE3;
5  import org.moeaframework.algorithm.NSGAII;
6  import org.moeaframework.core.Initialization;
7  import org.moeaframework.core.NondominatedSortingPopulation;
8  import org.moeaframework.core.Problem;
9  import org.moeaframework.core.Variation;
10 import org.moeaframework.core.comparator.ChainedComparator;
11 import org.moeaframework.core.comparator.CrowdingComparator;
12 import org.moeaframework.core.comparator.ParetoDominanceComparator;
13 import org.moeaframework.core.operator.GAVariation;
14 import org.moeaframework.core.operator.TournamentSelection;
15 import org.moeaframework.core.operator.real.DifferentialEvolution;
16 import org.moeaframework.core.operator.real.DifferentialEvolutionSelection;
17 import org.moeaframework.core.operator.real.PM;
18 import org.moeaframework.core.operator.real.SBX;
19
20 public class Hyperheuristic extends AbstractEvolutionaryAlgorithm {
21
22   private NSGAII nsgaii;
23
24   private GDE3 gde3;
25
```

[1]Heuristic is another name for an MOEA

```
26    private int iteration;
27
28    public Hyperheuristic(Problem problem, NondominatedSortingPopulation
          population,
29        Initialization initialization) {
30      super(problem, population, null, initialization);
31
32      // define our NSGAII instance
33      TournamentSelection selection = new TournamentSelection(2,
34          new ChainedComparator(
35              new ParetoDominanceComparator(),
36              new CrowdingComparator()));
37
38      Variation variation = new GAVariation(
39          new SBX(1.0, 25.0),
40          new PM(1.0 / problem.getNumberOfVariables(), 30.0));
41
42      nsgaii = new NSGAII(
43          problem,
44          population,
45          null, // no archive
46          selection,
47          variation,
48          initialization);
49
50      // define our GDE3 instance
51      gde3 = new GDE3(problem,
52          population,
53          new ParetoDominanceComparator(),
54          new DifferentialEvolutionSelection(),
55          new DifferentialEvolution(0.1, 0.9),
56          initialization);
57    }
58
59    @Override
60    protected void iterate() {
61      if (iteration % 2 == 0) {
62        nsgaii.iterate();
63      } else {
64        gde3.iterate();
65      }
66
67      numberOfEvaluations = nsgaii.getNumberOfEvaluations() +
68          gde3.getNumberOfEvaluations();
69
70      iteration++;
71    }
72
73  }
```

MOEAFramework/book/chapter5/Hyperheuristic.java

Here, we manually create the NSGA-II and GDE3 instance on lines 33-48 and 51-56, respectively. The primary difference is that we share the same `population` instance between both algorithm. This way, the algorithms are evolving the same population. In the `iterate` method on lines 60-71, we iterate between running NSGA-II and GDE3. Note in particular how we must update the `numberOfEvaluations` variable during each iteration. This is necessary since each algorithm may produce a different number of offspring and we must ensure the number of evaluations is tabulated correctly.

Lets create a custom provider:

```
1  package chapter5;
2
3  import java.util.Properties;
4
5  import org.moeaframework.core.Algorithm;
6  import org.moeaframework.core.Initialization;
7  import org.moeaframework.core.NondominatedSortingPopulation;
8  import org.moeaframework.core.Problem;
9  import org.moeaframework.core.operator.RandomInitialization;
10 import org.moeaframework.core.spi.AlgorithmProvider;
11 import org.moeaframework.util.TypedProperties;
12
13 public class HyperheuristicProvider extends AlgorithmProvider {
14
15   @Override
16   public Algorithm getAlgorithm(String name, Properties properties, Problem
         problem) {
17     if (name.equalsIgnoreCase("hyperheuristic")) {
18       TypedProperties typedProperties = new TypedProperties(properties);
19
20       int populationSize = typedProperties.getInt("populationSize", 100);
21
22       Initialization initialization = new RandomInitialization(
23           problem,
24           populationSize);
25
26       Algorithm algorithm = new Hyperheuristic(
27           problem,
28           new NondominatedSortingPopulation(),
29           initialization);
30
31       return algorithm;
32     } else {
33       return null;
34     }
35   }
36
37 }
```

MOEAFramework/book/chapter5/HyperheuristicProvider.java

and test our new algorithm:

```
package chapter5;

import org.moeaframework.Executor;
import org.moeaframework.core.NondominatedPopulation;
import org.moeaframework.core.spi.AlgorithmFactory;
import chapter2.SchafferProblem;

public class RunHyperheuristic {

  public static void main(String[] args) {
    AlgorithmFactory.getInstance().addProvider(new HyperheuristicProvider());

    NondominatedPopulation result = new Executor()
        .withAlgorithm("hyperheuristic")
        .withProblemClass(SchafferProblem.class)
        .withMaxEvaluations(10000)
        .run();
  }

}
```

MOEAFramework/book/chapter5/RunHyperheuristic.java

5.6 Custom Types and Operators

The MOEA Framework provides several representations for decision variables, including real-values, integers, binary strings, permutations, and programs (expression trees). It is a good practice to reuse these built-in operators when able. However, some situations require defining new types and operators, including:

1. Data can not be represented by a composite of built-in types
2. Provide better structure (grouping) of variables
3. Define custom variation operators

Suppose we needed to represent colors. A color is commonly represented on a computer as a composite of three primary colors: red, green, and blue. The value for each primary color typically ranges between 0 and 255, where 0 indicates absence of the primary color and 255 indicates complete saturation. We typically express a color as a tuple of three values, e.g. (red, green, blue). Some common colors are shown below:

- Black - (0, 0, 0)
- Red - (255, 0, 0)

- Green - (0, 255, 0)
- Blue - (0, 0, 255)
- Yellow - (255, 255, 0)
- White - (255, 255, 255)

We could choose to represent colors as three separate real or integer values. However, we can provide better structure and enable the use of custom operators by defining a new type. First, we create the `Color` type by implementing the `Variable` interface. A variable requires two methods: `copy()` and `randomize()`. The `copy()` method is responsible for creating an exact copy of a color (which is used to create new offspring solutions) and `randomize()` creates a new random color (which is used to create the initial population). Our `Color` class is shown below:

```
1  package chapter5;
2
3  import org.moeaframework.core.PRNG;
4  import org.moeaframework.core.Variable;
5
6  public class Color implements Variable {
7
8    private static final long serialVersionUID = 846174134757847124 8L;
9
10   private int r;
11
12   private int g;
13
14   private int b;
15
16   public Color() {
17     super();
18   }
19
20   public Color(int r, int g, int b) {
21     this();
22     this.r = r;
23     this.g = g;
24     this.b = b;
25   }
26
27   protected int getR() {
28     return r;
29   }
30
31   protected void setR(int r) {
32     this.r = r;
33   }
34
```

```
35    protected int getG() {
36      return g;
37    }
38
39    protected void setG(int g) {
40      this.g = g;
41    }
42
43    protected int getB() {
44      return b;
45    }
46
47    protected void setB(int b) {
48      this.b = b;
49    }
50
51    @Override
52    public Variable copy() {
53      return new Color(r, g, b);
54    }
55
56    @Override
57    public void randomize() {
58      r = PRNG.nextInt(256);
59      g = PRNG.nextInt(256);
60      b = PRNG.nextInt(256);
61    }
62
63    @Override
64    public String toString() {
65      return "Color [r=" + r + ", g=" + g + ", b=" + b + "]";
66    }
67
68  }
```

MOEAFramework/book/chapter5/Color.java

Second, we need to define any variation operators for this type. The variation operators describe how to generate new candidate offspring during search. For simplicity, we will define a mutation operator that accepts a single parent, copies the parent, randomly perturbs the color in the copy, and returns the result. Be sure to always create copies of parents to avoid modifying the original parent solution.

```
1  package chapter5;
2
3  import org.moeaframework.core.PRNG;
4  import org.moeaframework.core.Solution;
5  import org.moeaframework.core.Variation;
6
7  public class ColorMutation implements Variation {
8
```

```
  @Override
  public int getArity() {
    return 1;
  }

  @Override
  public Solution[] evolve(Solution[] parents) {
    Solution result = parents[0].copy();

    for (int i = 0; i < result.getNumberOfVariables(); i++) {
      if (result.getVariable(i) instanceof Color) {
        Color color = (Color)result.getVariable(i);
        color.setR(Math.min(255, Math.max(0, color.getR() + PRNG.nextInt(-5,
            5))));
        color.setG(Math.min(255, Math.max(0, color.getG() + PRNG.nextInt(-5,
            5))));
        color.setB(Math.min(255, Math.max(0, color.getB() + PRNG.nextInt(-5,
            5))));
      }
    }

    return new Solution[] { result };
  }

};
```

MOEAFramework/book/chapter5/ColorMutation.java

Finally, we create an operator provider for this new type. The operator provider is responsible for specifying which variation operators are appropriate for a given problem. In this example, we check to see if the problem type contains any `Color` variables, and if so, instruct the MOEA Framework to use our variation operator.

```
package chapter5;

import java.util.Properties;

import org.moeaframework.core.Problem;
import org.moeaframework.core.Solution;
import org.moeaframework.core.Variation;
import org.moeaframework.core.spi.OperatorProvider;

public class ColorProvider extends OperatorProvider {

  @Override
  public String getMutationHint(Problem problem) {
    Solution solution = problem.newSolution();

    for (int i = 0; i < problem.getNumberOfVariables(); i++) {
      if (solution.getVariable(i) instanceof Color) {
        return "colormutation";
```

```
      }
    }

    return null;
  }

  @Override
  public String getVariationHint(Problem problem) {
    return getMutationHint(problem);
  }

  @Override
  public Variation getVariation(String name, Properties properties,
      Problem problem) {
    if (name.equalsIgnoreCase("colormutation")) {
      return new ColorMutation();
    }

    return null;
  }

}
```

MOEAFramework/book/chapter5/ColorProvider.java

At this point, we have defined our new decision variable type and the appropriate operators. We can now define a problem using this type. The problem we will solve is very simple: find N colors that, when mixed, produce a color closest to the target, assuming the background or base color is white (255, 255, 255).

```
package chapter5;
import org.moeaframework.core.Solution;
import org.moeaframework.problem.AbstractProblem;

public class ColorBlenderProblem extends AbstractProblem {

  private final Color target;

  public ColorBlenderProblem(int N, Color target) {
    super(N, 1);
    this.target = target;
  }

  public Color blend(Color c1, Color c2, double ratio) {
      double invRatio = 1.0 - ratio;
      int r = (int)((c1.getR() * invRatio) + (c2.getR() * ratio));
      int g = (int)((c1.getG() * invRatio) + (c2.getG() * ratio));
      int b = (int)((c1.getB() * invRatio) + (c2.getB() * ratio));

      return new Color(r, g, b);
  }
```

```
22
23    @Override
24    public void evaluate(Solution solution) {
25      Color result = new Color(255, 255, 255);
26
27      for (int i = 0; i < numberOfVariables; i++) {
28        result = blend(result, (Color)solution.getVariable(i), 0.5);
29      }
30
31      int diff = Math.abs(result.getR()-target.getR()) +
32          Math.abs(result.getG()-target.getG()) +
33          Math.abs(result.getB()-target.getB());
34
35      solution.setObjective(0, diff);
36    }
37
38    @Override
39    public Solution newSolution() {
40      Solution solution = new Solution(numberOfVariables, numberOfObjectives);
41
42      for (int i = 0; i < numberOfVariables; i++) {
43        solution.setVariable(i, new Color());
44      }
45
46      return solution;
47    }
48
49  }
```

MOEAFramework/book/chapter5/ColorBlenderProblem.java

Note how the constructor on line 9 accepts two arguments: `N`, the number of colors to combine, and `target`, the target color. This will allow us to run this problem with different inputs, as shown below.

```
1  package chapter5;
2  import org.moeaframework.Executor;
3  import org.moeaframework.core.NondominatedPopulation;
4  import org.moeaframework.core.Solution;
5  import org.moeaframework.core.spi.OperatorFactory;
6
7  public class RunColorBlenderProblem {
8
9    public static void main(String[] args) {
10     OperatorFactory.getInstance().addProvider(new ColorProvider());
11
12     NondominatedPopulation result = new Executor()
13         .withProblemClass(ColorBlenderProblem.class, 3, new Color(127, 127,
               127))
14         .withAlgorithm("NSGAII")
15         .withMaxEvaluations(10000)
```

```
16          .distributeOnAllCores()
17          .run();
18
19      for (Solution solution : result) {
20        System.out.print(solution.getObjective(0));
21
22        for (int i = 0; i < solution.getNumberOfVariables(); i++) {
23          System.out.print(" ");
24          System.out.print(solution.getVariable(i));
25        }
26      }
27    }
28
29  }
```

MOEAFramework/book/chapter5/RunColorBlenderProblem.java

As before, we must register our custom operator provider on line 11. Once registered, we can optimize the problem. Also observe on line 13 how we pass the arguments when calling `withProblemClass`. The result is shown below:

```
0.0 Color [r=148, g=20, b=132] Color [r=158, g=252, b=127] Color [r=76, g=61,
    b=94]
```

The three colors (148, 20, 132), (158, 252, 127) and (76, 61, 94) combine on a white background to produce the target color (127, 127, 127) exactly. You will likely see different results if running this example since there are many color combinations that produce the same result.

5.7 Learning the API

You may be wondering how we know which classes to implement when defining custom components. We make available online the entire application programming interface (API) for the MOEA Framework. The API describes in detail every class, function, and argument in the code. You can find the API online at `http://moeaframework.org/javadoc/index.html`.

Chapter 6

The Diagnostic Tool

We'll take a brief break from code and turn our attention to the MOEA Framework's diagnostic tool. The diagnostic tool provides a graphical interface to quickly run and analyze MOEAs on a test problems. We have seen many of the functions provided by the diagnostic tool in the preceeding chapters, but the diagnostic tool provides an easy-to-use point-and-click interface for accessing these features.

6.1 Using the Diagnostic Tool

To run the diagnostic tool, navigate to the MOEA Framework folder on your computer and double-click the `launch-diagnostic-tool.bat` file. You can manually run the diagnostic tool with the following command:

```
java -Djava.ext.dirs=lib
    org.moeaframework.analysis.diagnostics.LaunchDiagnosticTool
```

Figure 6.1 provides a screenshot of the diagnostic tool window. This window is composed of the following sections:

1. The configuration panel. This panel lets you select the algorithm, problem, number of repetitions (seeds), and maximum number of function evaluations (NFE).

2. The execution panel. Clicking run will execute the algorithm as configured in the configuration panel. Two progress bars display the individual run progress and the total progress for all seeds. Any in-progress runs can be canceled.

3. The displayed results table. This table displays the completed runs. The entries which are selected/highlighted are displayed in the charts. You can click an individual line to show the data for just that entry, click while holding the Alt key to select multiple entries, or click the Select All button to select all entries.

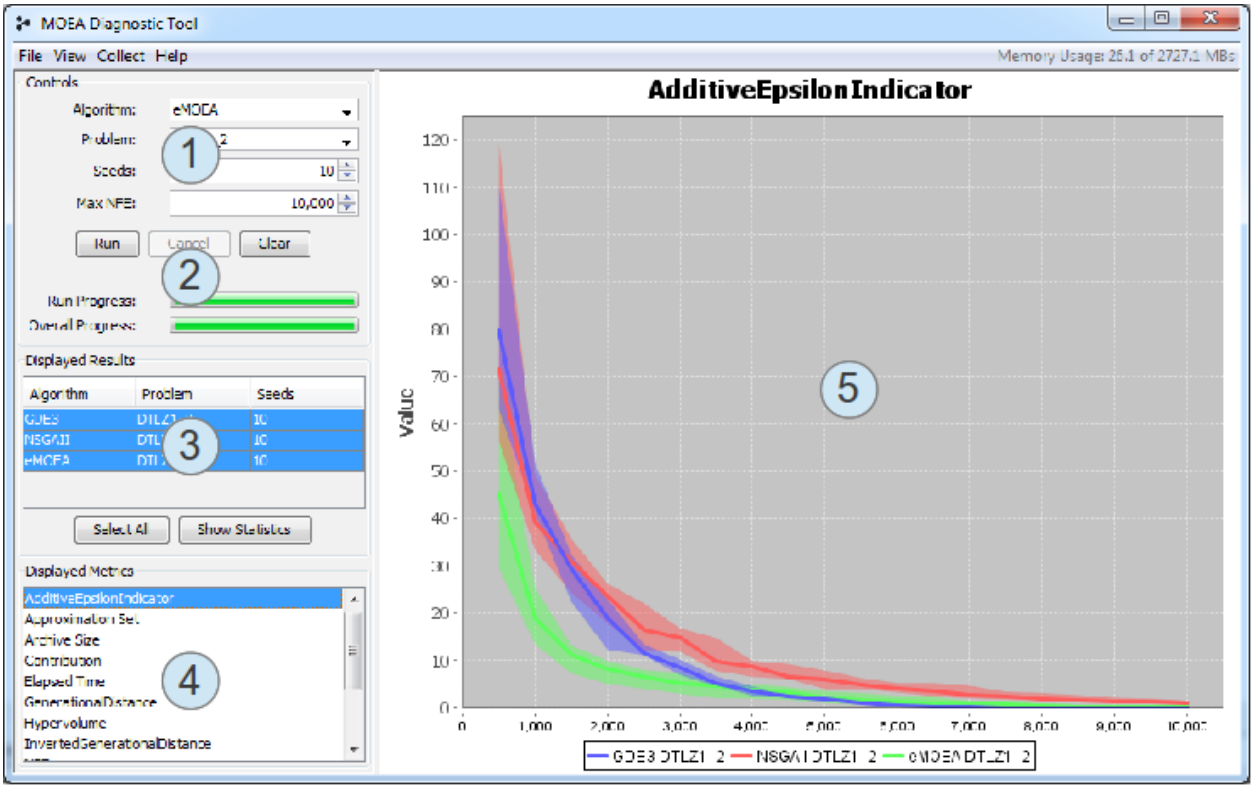


Figure 6.1: The main window of the diagnostic tool.

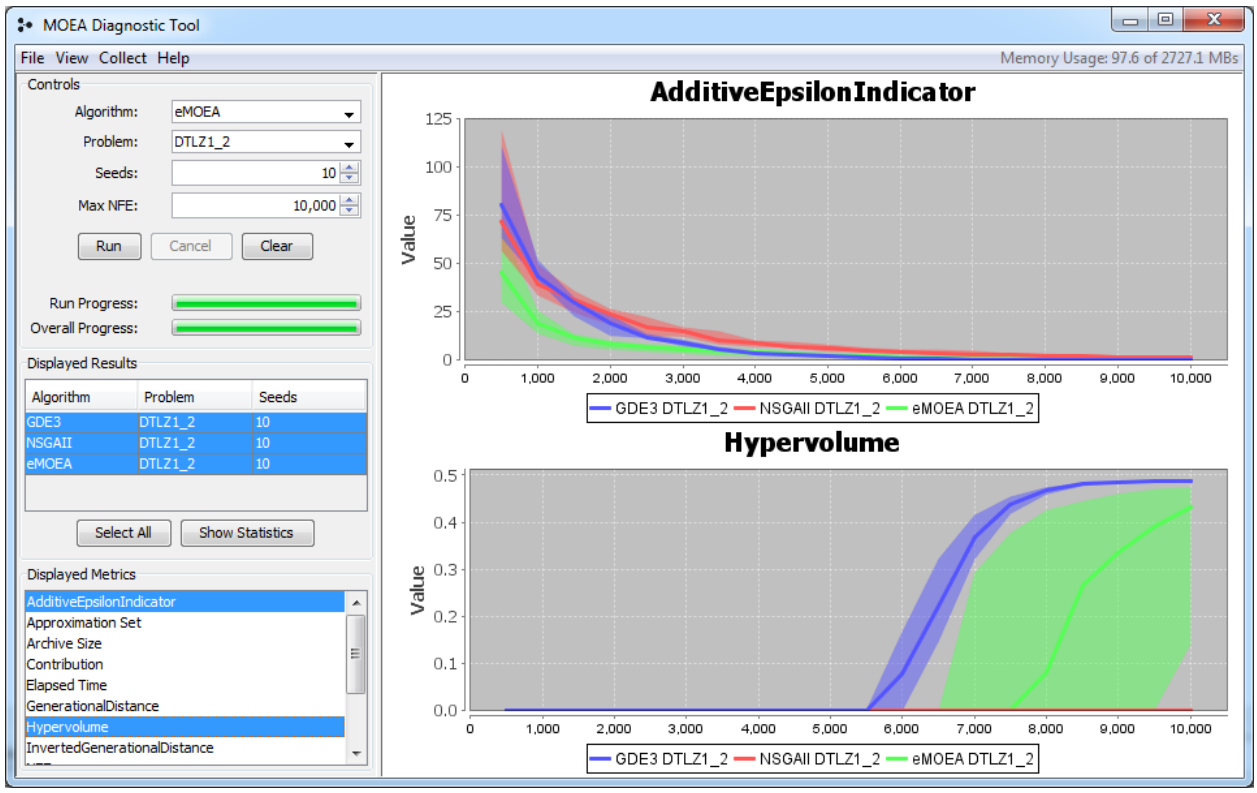


Figure 6.2: Screenshot of the diagnostic tool displaying two side-by-side metrics. You can select as many metrics to display by holding down the Alt key and clicking a row in the displayed metrics table.

4. The displayed metrics table. Similar to the displayed results table, the selected metrics are displayed in the charts. You can select one metric or multiple metrics by holding the Alt key while clicking.

5. The actual charts. A chart will be generated for each selected metric. Thus, if two metrics are selected, then two charts will be displayed side-by-side. See Figure 6.2 for an example.

Some algorithms do not provide certain metrics. When selecting a specific metric, only those algorithms that provide that metric will be displayed in the chart.

Quantile Plots vs Individual Traces

By default, the charts displayed in the diagnostic tool show the statistical 25%, 50% and 75% quantiles. The 50% quantile is the thick colored line, and the 25% and 75% quantiles are depicted by the colored area. This quantile view allows you to quickly distinguish the performance between multiple algorithms, particularly when there is significant overlap between two or more algorithms.

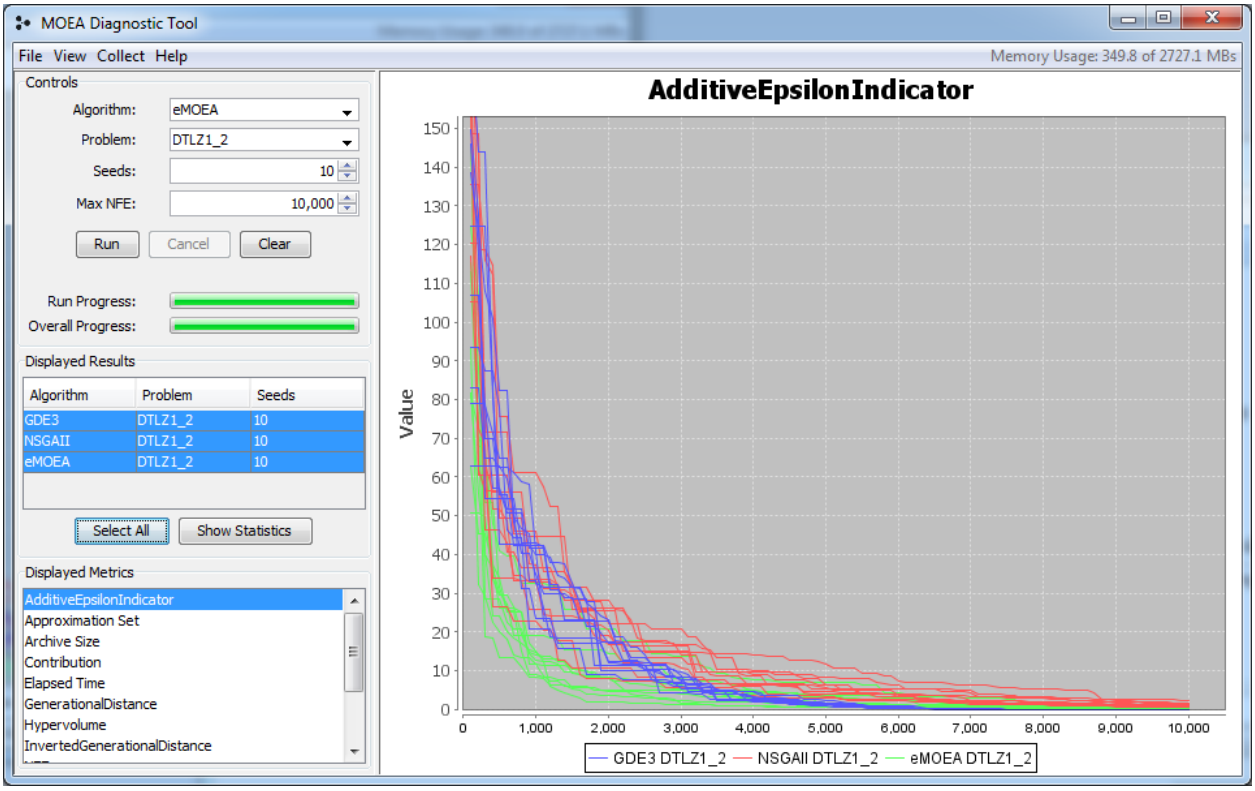


Figure 6.3: Screenshot of the diagnostic tool displaying the individual traces rather than the quantile view. The individual traces provide access to the raw data, but the quantile view is often easier to interpret.

You can alternatively view the raw, individual traces by selecting 'Show Individual Traces' in the View menu. Each colored line represents one seed. Figure 6.3 provides an example of plots showing individual traces. You can always switch back to the quantile view using the View menu.

Viewing Approximation Set Dynamics

Another powerful feature of the diagnostic tool is the visualization of approximation set dynamics. The approximation set dynamics show how the algorithm's result (its approximation set) evolved throughout the run. To view the approximation set dynamics, right-click on one of the rows in the displayed results table. A menu will appear with the option to show the approximation set. A window similar to Figure 6.4 will appear.

This window displays the following items:

1. The approximation set plot. This plot can only show two dimensions. If available, the reference set for the problem will be shown as black points. All other points are the solutions produced by the algorithm. Different seeds are displayed in different colors.

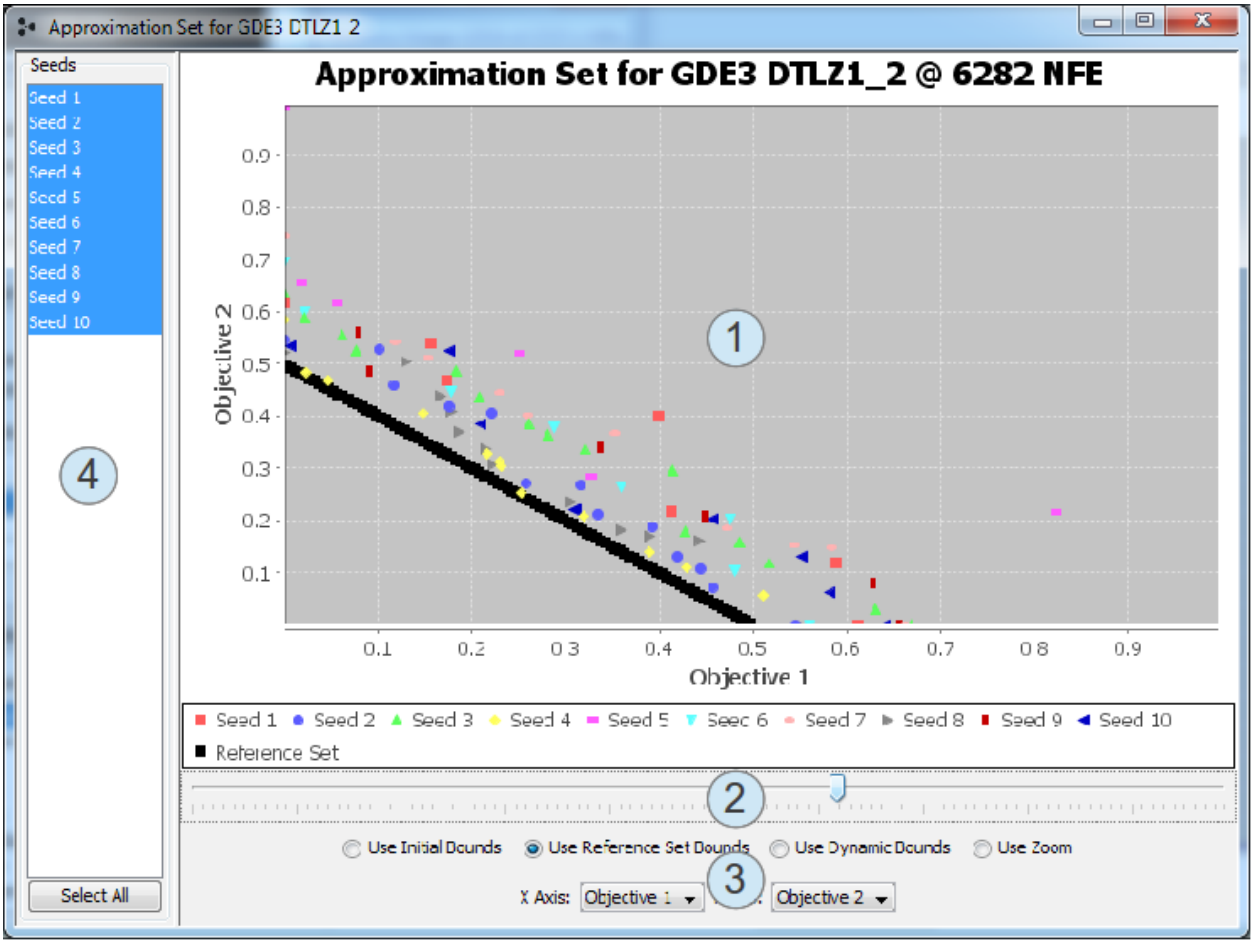


Figure 6.4: Screenshot of the approximation set viewer. This allows you to view the approximation set at any point in the algorithm's execution.

2. The evolution slider. Dragging the slider to the left or right will show the approximation set from earlier or later in the evolution.

3. The display controls. These controls let you adjust how the data is displayed. Each of the radio buttons switches between different scaling options. The most common option is 'Use Reference Set Bounds', which scales the plot so that the reference set fills most of the displayed region.

4. The displayed seeds table. By default, the approximation sets for all seeds are displayed and are distinguished by color. You can also downselect to display one or a selected group of seeds by selecting entries in this table. Multiple entries can be selected by holding the Alt key while clicking.

You can manually zoom to any portion in these plots (both in the approximation set viewer and the plots in the main diagnostic tool window) by positioning the cursor at the top-left corner of the zoom region, pressing and holding down the left-mouse button, dragging the cursor to the bottom-right corner of the zoom region, and releasing the left-mouse button. You can reset the zoom by pressing and holding the left-mouse button, dragging the cursor to the top-left portion of the plot, and releasing the left-mouse button.

Statistical Results

The diagnostic tool also allows you to exercise the statistical testing tools provided by the MOEA Framework with the click of a button. If you have two or more entries selected in the displayed results table, the 'Show Statistics' button will become enabled. The show statistics button also requires each of the selected entries to use the same problem. The button will remain disabled unless this condition is satisfied. If the button is disabled, please ensure you have two or more rows selected and all selected entries are using the same problem. Figure 6.5 shows the example output from clicking this button.

Improving Performance and Memory Efficiency

By default, the diagnostic tool collects and displays all available data. If you know ahead of time that certain pieces of data are not needed for your experiments, you can often increase the performance and memory efficiency of the program by disabling unneeded data. You can enable or disable the collection of data by checking or unchecking the appropriate item in the Collect menu.

6.2 Adding Custom Algorithms

What makes the diagnostic tool particularly powerful is that it can incorporate custom algorithms and problems. Lets take, for example, the `RandomWalker` and `Hyperheuristic` algorithms we created in the previous chapter. We can incorporate these algorithms into the diagnostic tool with the code below:

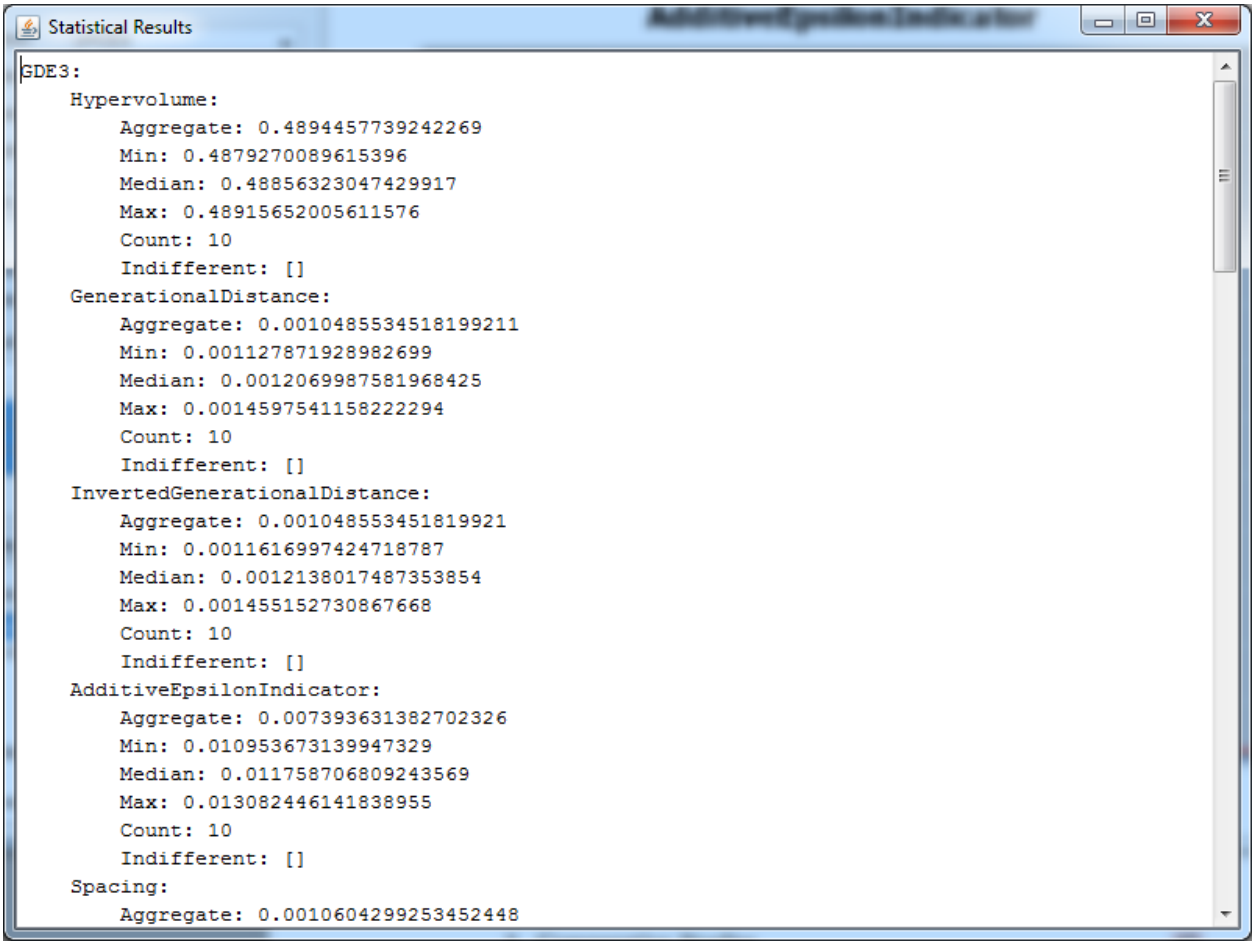


Figure 6.5: Screenshot of the statistics output by the diagnostic tool.

```
1  package chapter6;
2
3  import org.moeaframework.analysis.diagnostics.LaunchDiagnosticTool;
4  import org.moeaframework.core.Settings;
5  import org.moeaframework.core.spi.AlgorithmFactory;
6
7  import chapter5.HyperheuristicProvider;
8  import chapter5.RandomWalkerProvider;
9
10 public class RunDiagnosticTool {
11
12   public static void main(String[] args) throws Exception {
13     AlgorithmFactory.getInstance().addProvider(new RandomWalkerProvider());
14     AlgorithmFactory.getInstance().addProvider(new HyperheuristicProvider());
15
16     Settings.PROPERTIES.setString(
17         Settings.KEY_DIAGNOSTIC_TOOL_ALGORITHMS,
18         "RandomWalker, Hyperheuristic, NSGAII, GDE3");
19
20     Settings.PROPERTIES.setString(
21         Settings.KEY_ALLOWED_PACKAGES,
22         "chapter5");
23
24     LaunchDiagnosticTool.main(args);
25   }
26
27 }
```

MOEAFramework/book/chapter6/RunDiagnosticTool.java

As before, we must register our custom algorithm provides with the MOEA Framework on lines 13 and 14. Lines 16-22 set two properties to configure the diagnostic tool. The first property defines which algorithms appear in the tool. We add our two new algorithms to this list. The second property allows our custom algorithms to be instrumented. Without specifying the allowed packages property, the diagnostic tool will be unable to collect any runtime data for our two custom algorithms. Finally, on line 24, we launch the diagnostic tool window.

Now we want to run each of the four algorithms on a test problem so we can view their plots. We will use the UF1 problem, but feel free to test with other problems. Select each algorithm from the drop-down and click the Run button.

Figure 6.6 shows the resulting plot. Here, we've selected inverted generational distance. As expected, the `RandomWalker` does not perform very well — after all, it's just taking random steps. The other algorithms perform very similarly. Using the mouse button, we can zoom in, as shown in Figure 6.7. Now we can see a difference between the algorithms. NSGA-II in green and GDE3 and red show differences (the difference is slight, but it is noticeable). Our hyperheuristic, which combines NSGA-II and GDE3, matches the performance of the better individual algorithm, GDE3. Even with our simple hyperheuristic, we can see that the hyperheuristic was able to perform at least as good as its component heuristics.

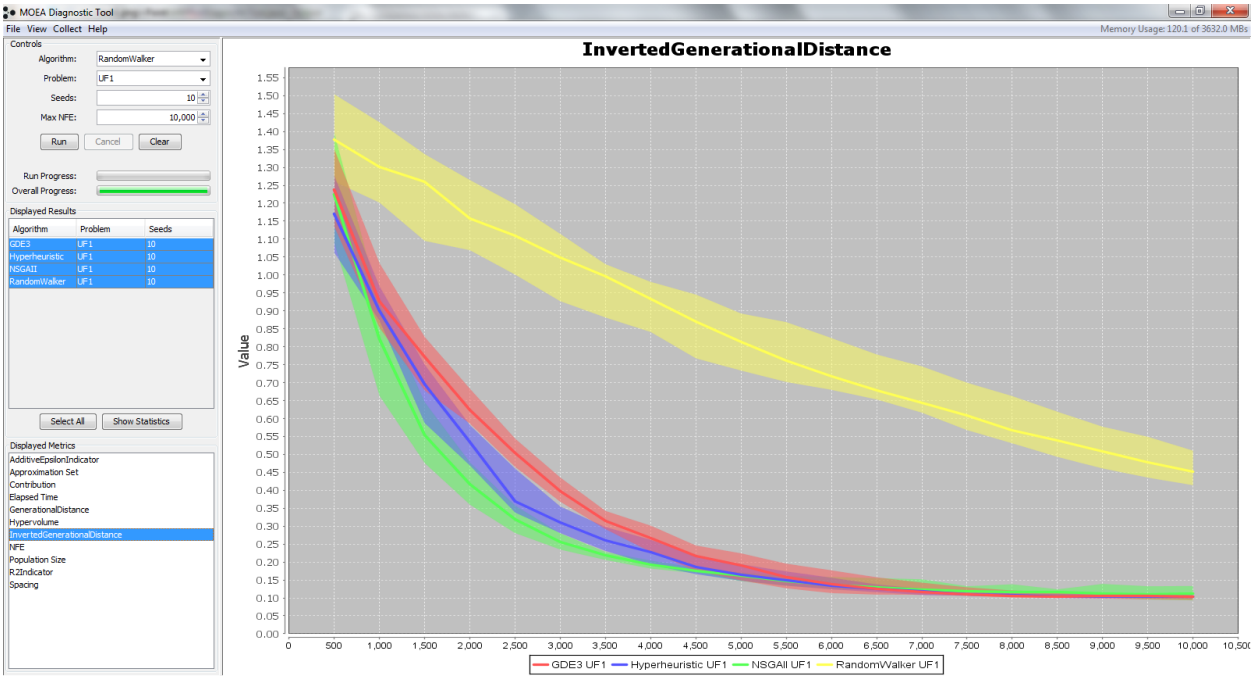


Figure 6.6: Comparison of our two custom algorithms with NSGA-II and GDE3

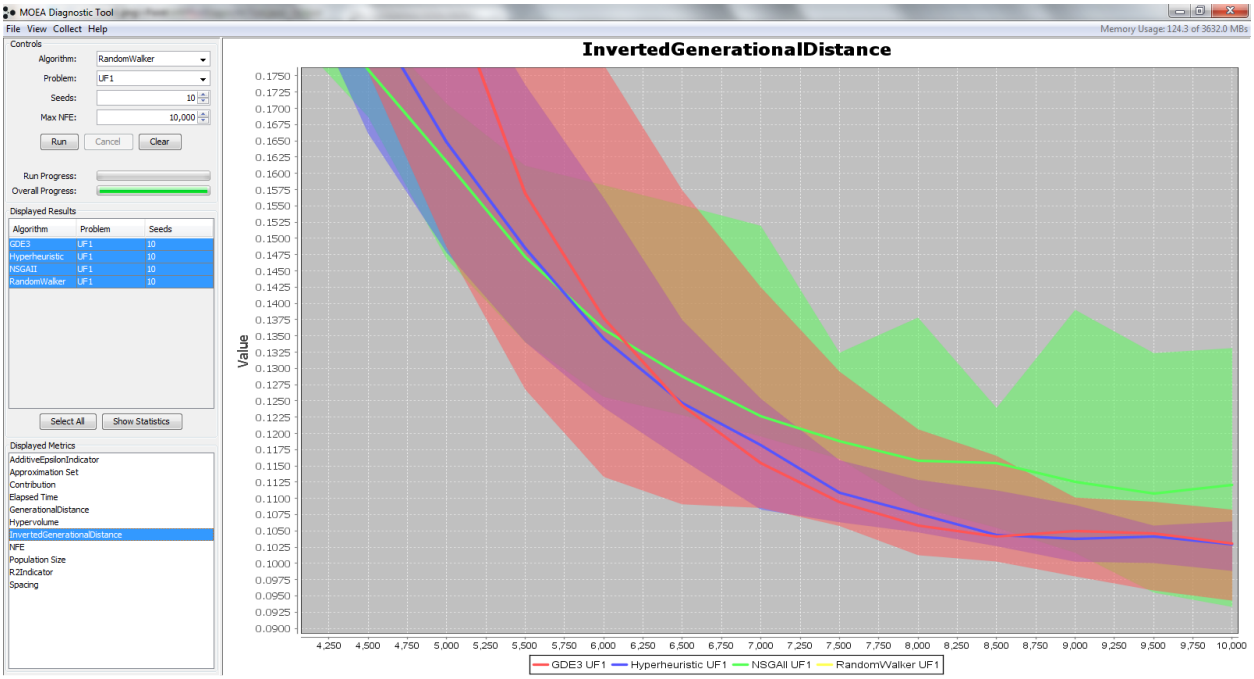


Figure 6.7: Enhanced view of the comparison.

6.3 Adding Custom Problems

Similar to how we added custom algorithms by registering a provider with the MOEA Framework, user-defined problems are defined using the `ProblemProvider`. For example, below we create a problem provider for the Schaffer problem we created in Chapter 2:

```
1  package chapter6;
2
3  import java.io.File;
4  import java.io.IOException;
5
6  import org.moeaframework.core.FrameworkException;
7  import org.moeaframework.core.NondominatedPopulation;
8  import org.moeaframework.core.PopulationIO;
9  import org.moeaframework.core.Problem;
10 import org.moeaframework.core.spi.ProblemProvider;
11
12 import chapter2.SchafferProblem;
13
14 public class SchafferProblemProvider extends ProblemProvider {
15
16   @Override
17   public Problem getProblem(String name) {
18     if (name.equalsIgnoreCase("MySchafferProblem")) {
19       return new SchafferProblem();
20     } else {
21       return null;
22     }
23   }
24
25   @Override
26   public NondominatedPopulation getReferenceSet(String name) {
27     if (name.equalsIgnoreCase("MySchafferProblem")) {
28       try {
29         return new NondominatedPopulation(
30             PopulationIO.readObjectives(new File("./pf/Schaffer.pf")));
31       } catch (IOException e) {
32         throw new FrameworkException(e);
33       }
34     } else {
35       return null;
36     }
37   }
38
39 }
```

MOEAFramework/book/chapter6/SchafferProblemProvider.java

A `ProblemProvider` must define two methods: `getProblem` and `getReferenceSet`. `getProblem` returns an instance of the problem if the name matches. `getReferenceSet` returns the reference set for the problem. Both methods

must return **null** if the name does not match any supported problem.

With the problem provider found, we can register it with the MOEA Framework and launch the diagnostic tool:

```
1  package chapter6;
2
3  import org.moeaframework.analysis.diagnostics.LaunchDiagnosticTool;
4  import org.moeaframework.core.Settings;
5  import org.moeaframework.core.spi.AlgorithmFactory;
6  import org.moeaframework.core.spi.ProblemFactory;
7
8  import chapter5.HyperheuristicProvider;
9  import chapter5.RandomWalkerProvider;
10
11 public class CustomProblem {
12
13   public static void main(String[] args) throws Exception {
14     AlgorithmFactory.getInstance().addProvider(new RandomWalkerProvider());
15     AlgorithmFactory.getInstance().addProvider(new HyperheuristicProvider());
16
17     ProblemFactory.getInstance().addProvider(new SchafferProblemProvider());
18
19     Settings.PROPERTIES.setString(
20         Settings.KEY_DIAGNOSTIC_TOOL_ALGORITHMS,
21         "RandomWalker, Hyperheuristic, NSGAII, GDE3");
22
23     Settings.PROPERTIES.setString(
24         Settings.KEY_DIAGNOSTIC_TOOL_PROBLEMS,
25         "MySchafferProblem");
26
27     Settings.PROPERTIES.setString(
28         Settings.KEY_ALLOWED_PACKAGES,
29         "chapter5");
30
31     LaunchDiagnosticTool.main(args);
32   }
33
34 }
```

MOEAFramework/book/chapter6/CustomProblem.java

Chapter 7

Subsets, Permutations, and Programs

We have seen thus far real-valued and bit string encoding problems. The MOEA Framework supports three additional, commonly-used representations: subsets, permutations, and programs. Subsets are similar to bit string encodings, except they represent situations where only a fixed number of items can be selected (whereas bit strings allow a variable number of items). Permutations are often used in problems where ordering is important. Programs are used when one must construct code, rule systems, or decision trees to solve a task. This chapter discusses these representations.

7.1 Subsets

Previously we saw with the bit string representation how we can solve the Knapsack problem. Recall that in the Knapsack problem we are tasked with identifying the items to place in a Knapsack to maximize value constrained only by the capacity of the Knapsack. By using the bit string (binary) representation, we allowed the optimization algorithm to select anywhere from 0 up to N items.

Subsets are good for a similar but slightly different class of problems. Instead of allowing one to select anywhere from 0 up to N items, what if they must select exactly k items. When the subset is a fixed size, we use the `Subset` class. Subsets are similar to selection without replacement. It will pick k items from a set containing N items without repeating any of the selected items. (Tip: if you need selection with replacement, where items can be selected more than once, just use k integer variables).

To demonstrate subsets, we will use a slight twist on the classic subset-sum problem, which we call the fixed-size subset-sum problem. In the fixed-size subset-sum problem, we are given N integers, both positive and negative. The goal is to find a subset of the integers of size k that sums to 0. For example, if we have the numbers $-5, -3, -1, 2, 4, 6$ and $k = 4$, we could pick $-5, -3, 2, 6$ since $-5 + -3 + 2 + 6 = 0$.

The full code for the fixed-size subset-sum problem is shown below:

```
1 package chapter7;
```

```
2
3  import org.moeaframework.core.Solution;
4  import org.moeaframework.core.variable.EncodingUtils;
5  import org.moeaframework.core.variable.Subset;
6  import org.moeaframework.problem.AbstractProblem;
7
8  public class FixedSubsetSumProblem extends AbstractProblem {
9
10   private int k;
11
12   private int[] values;
13
14   public FixedSubsetSumProblem(int k, int[] values) {
15     super(1, 1);
16     this.k = k;
17     this.values = values;
18   }
19
20   @Override
21   public void evaluate(Solution solution) {
22     int[] subset = EncodingUtils.getSubset(solution.getVariable(0));
23     int sum = 0;
24
25     for (int i = 0; i < subset.length; i++) {
26       sum += values[subset[i]];
27     }
28
29     solution.setObjective(0, Math.abs(sum));
30   }
31
32   @Override
33   public Solution newSolution() {
34     Solution solution = new Solution(1, 1);
35     solution.setVariable(0, new Subset(k, values.length));
36     return solution;
37   }
38
39 }
```

MOEAFramework/book/chapter7/FixedSubsetSumProblem.java

We construct the subset with `new Subset(k, values.length)` to indicate we want to form subsets of size `k` from a set of size `values.length`. Note that the subset does not actually store the values. Instead, it stores the indices of the selected items. When evaluating the solution, we must be careful to convert from the indices in the subset to the original values using `values[subset[i]]`.

When our problem implemented, we can then solve it:

```
1  package chapter7;
2
3  import java.util.Arrays;
```

```
4
5  import org.moeaframework.Executor;
6  import org.moeaframework.core.NondominatedPopulation;
7  import org.moeaframework.core.Solution;
8  import org.moeaframework.core.variable.EncodingUtils;
9
10 public class SolveFixedSubsetSumProblem {
11
12   private static int[] values = { -1, 1, 3, 5, 9, -4, -2, -8, 17, 24, 18, -16,
         -20 };
13
14   public static void main(String[] args) {
15     NondominatedPopulation result = new Executor()
16         .withAlgorithm("NSGAII")
17         .withProblemClass(FixedSubsetSumProblem.class, 6, values)
18         .withMaxEvaluations(10000)
19         .run();
20
21
22
23     for (Solution solution : result) {
24       int[] subset = EncodingUtils.getSubset(solution.getVariable(0));
25       System.out.println(Arrays.toString(toValues(subset, values)) +
26           " => " + solution.getObjective(0));
27     }
28   }
29
30   public static int[] toValues(int[] subset, int[] values) {
31     int[] result = new int[subset.length];
32
33     for (int i = 0; i < subset.length; i++) {
34       result[i] = values[subset[i]];
35     }
36
37     return result;
38   }
39
40 }
```

MOEAFramework/book/chapter7/SolveFixedSubsetSumProblem.java

You should see output similar to the following:

```
[9, 3, -20, -8, 17, -1] => 0.0
```

7.2 Permutations

Permutations are commonly seen in problems involving ordering. For example, job scheduling often uses permutations to specify the scheduling priority for jobs. Or, in the famous travelling salesman problem[1], we seek the shortest route that visits all cities. We could use a permutation to represent the path. We will demonstrate the use of permutations in this section by constructing a simple solver for traveling salesman problems.

The full code for the travelling salesman problem is provided below.

```
1  package chapter7;
2
3  import org.moeaframework.core.Solution;
4  import org.moeaframework.core.variable.EncodingUtils;
5  import org.moeaframework.problem.AbstractProblem;
6
7  public class TravellingSalesmanProblem extends AbstractProblem {
8
9    private double[][] cities;
10
11   public TravellingSalesmanProblem(double[][] cities) {
12     super(1, 1);
13     this.cities = cities;
14   }
15
16   @Override
17   public Solution newSolution() {
18     Solution solution = new Solution(1, 1);
19     solution.setVariable(0, EncodingUtils.newPermutation(cities.length));
20     return solution;
21   }
22
23   private double distance(int i, int j) {
24     return Math.sqrt(
25         Math.pow(cities[i][0]-cities[j][0], 2.0) +
26         Math.pow(cities[i][1]-cities[j][1], 2.0));
27   }
28
29   @Override
30   public void evaluate(Solution solution) {
31     int[] path = EncodingUtils.getPermutation(solution.getVariable(0));
32     double totalDistance = 0.0;
33
34     for (int i = 0; i < path.length; i++) {
35       totalDistance += distance(path[i], path[(i+1) % path.length]);
36     }
37
38     solution.setObjective(0, totalDistance);
39   }
```

[1] https://en.wikipedia.org/wiki/Travelling_salesman_problem

```
40
41 }
```

MOEAFramework/book/chapter7/TravellingSalesmanProblem.java

On lines 11-14 we construct the problem. For flexibility, we'll make the list of cities an argument. The cities are stored in a Nx2 array storing the x and y coordinates of N cities. The `newSolution` method on lines 17-21 shows how to construct a solution. A permutation is stored in a single decision variable. Lastly, we define the `evaluate` method on lines 30-39. First, we extract the permutation on line 31. Second, on lines 32-36, we sum up the total distance travelled by the salesman. Finally, on line 38, we set the objective value.

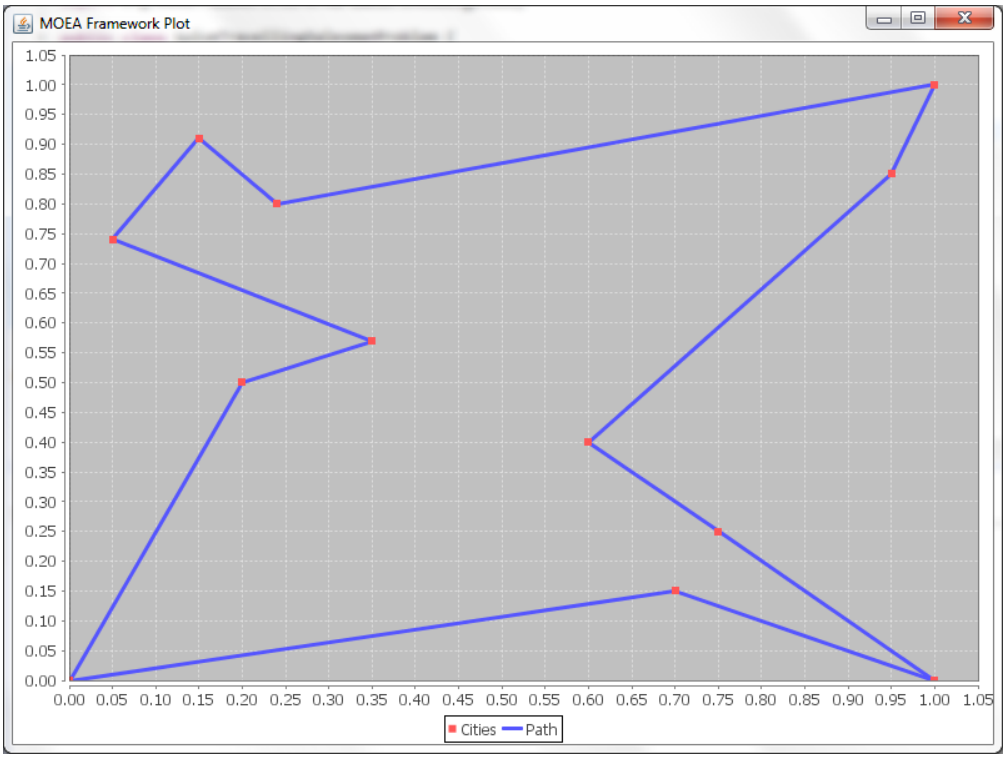
The code for optimizing this problem is shown below.

```
1  package chapter7;
2
3  import org.moeaframework.Executor;
4  import org.moeaframework.analysis.plot.Plot;
5  import org.moeaframework.core.NondominatedPopulation;
6  import org.moeaframework.core.variable.EncodingUtils;
7
8  public class SolveTravellingSalesmanProblem {
9
10   public static double[][] CITIES = {
11       { 0.0,  0.0  },
12       { 1.0,  1.0  },
13       { 0.75, 0.25 },
14       { 1.0,  0.0  },
15       { 0.95, 0.85 },
16       { 0.2,  0.5  },
17       { 0.7,  0.15 },
18       { 0.6,  0.4  },
19       { 0.15, 0.91 },
20       { 0.24, 0.8  },
21       { 0.05, 0.74 },
22       { 0.35, 0.57 }
23   };
24
25   public static void main(String[] args) {
26     NondominatedPopulation result = new Executor()
27         .withAlgorithm("NSGAII")
28         .withProblemClass(TravellingSalesmanProblem.class, CITIES)
29         .withMaxEvaluations(10000)
30         .run();
31
32     int[] bestPath = EncodingUtils.getPermutation(result.get(0).getVariable(0)
           );
33
34     new Plot()
35         .scatter("Cities", getCoordinate(0), getCoordinate(1))
36         .line("Path", getCoordinate(0, bestPath), getCoordinate(1, bestPath))
```

```
37          .show();
38    }
39
40    public static double[] getCoordinate(int index) {
41      double[] result = new double[CITIES.length];
42
43      for (int i = 0; i < CITIES.length; i++) {
44        result[i] = CITIES[i][index];
45      }
46
47      return result;
48    }
49
50    public static double[] getCoordinate(int index, int[] path) {
51      double[] result = new double[path.length+1];
52
53      for (int i = 0; i < path.length; i++) {
54        result[i] = CITIES[path[i]][index];
55      }
56
57      result[path.length] = CITIES[path[0]][index];
58
59      return result;
60    }
61
62  }
```

MOEAFramework/book/chapter7/SolveTravellingSalesmanProblem.java

We define two helper methods, both called `getCoordinate`, for extracting the coordinates of the cities for plotting. The resulting plot looks like:



7.3 Programs

My problems require developing computer code, rule systems, or decision trees. One example would be developing a robot to navigate an environment and avoid hitting obstacles. Available to the robots is input from various sensors which it uses to control steering and speed. We won't be so bold in this book to build such a robot, but we will explore here two problems. The first is a symbolic regression problem, where we attempt to find an expression to fit a set of points. The second is the ant problem, where we program a robotic ant to find food in a virtual world.

7.3.1 Type-based Rule System

To accomplish these tasks, the MOEA Framework contains a simple but extensible LISP-like programming language. The first step towards evolving programs is to define the rules (i.e., the syntax and semantics). When defining the rules, two important properties should be kept in mind: *closure* and *sufficiency*.

The closure property requires all program element to be able to accept as arguments any value and data type that could possibly be returned by any other function or terminal. All programs generated or evolved by the MOEA Framework are strongly typed. No program produced by the MOEA Framework will pass an argument to a function that is an incorrect type. Furthermore, all functions guard against invalid inputs. For example, the `log` of a

negative number is undefined. Rather then causing an error, the `log` method will guard itself and return `0.0`. This allows the rest of the calculation to continue unabated. With these two behaviors built into the MOEA Framework, the closure property is guaranteed.

The sufficiency property states that the rule set must contain all the necessary functions and terminals necessary to produce a solution to the problem. Ensuring this property holds is more challenging as it will depend on the problem domain. For instance, the operators `And`, `Or` and `Not` are sufficient to produce all boolean expressions. It may not be so obvious in other problem domains which program elements are required to ensure sufficiency. Additionally, it is often helpful to restrict the rule set to those program elements that are sufficient, thus reducing the search space for the evolutionary algorithm.

The MOEA Framework comes packaged with over 45 pre-defined program elements for defining constants, variables, arithmetic operators, control structures, functions, etc. These program elements are listed below.

Operator	Description
Abs	Calculates the absolute value of a number
Acos	Calculates the arc cosine of a number
Acosh	Calculates the hyperbolic arc cosine of a number
Add	Adds two numbers
And	Calculates the logical AND of two boolean values
Asin	Calculates the arc sine of a number
Asinh	Calculates the hyperbolic arc sine of a number
Atan	Calculates the arc tangent of a number
Atanh	Calculates the hyperbolic arc tangent of a number
Call	Calls a named function
Ceil	Calculates the smallest integer value that is greater than or equal to a number
Constant	Defines a constant value
Cos	Calculates the trigonometric cosine of an angle specified in radians
Cosh	Calculates the hyperbolic cosine of a number
Define	Defines a callable, named function
Divide	Divides two numbers
Equals	Compares two numbers for equality
Exp	Calculates the result of Euler's number e raised to the power of a number
Floor	Calculates the largest integer value that is less than or equal to a number
For	Executes an expression for a given number of iterations
Get	Reads the value stored in a named variable within the current scope
GreaterThan	Compares if one number is greater than another
GreaterThanOrEqual	Compares if a number is greater than or equal to another

IfElse	Executes one of two expressions depending on the result of a boolean expression
Lambda	Defines an immutable, anonymous function
LessThan	Compares if a number is less than another
LessThanOrEqual	Compares if a number is less than or equal to another
Log	The node for calculating the natural logarithm of a number
Log10	Calculates the base-10 logarithm of a number
Max	Calculates the maximum value of the two arguments
Min	Calculates the minimum value of the two arguments
Modulus	Calculates the modulus, or remainder, of two numbers
Multiply	Multiplies two numbers
NOP	Defines an empty expression
Not	Calculates the logical NOT of a boolean value
Or	Calculates the logical OR of two boolean values
Power	Calculates the power of a base number and exponent
Round	Rounds a number to the nearest integer
Sequence	Executes two or more expressions in sequence
Set	Assigns the value of a named variable within the current scope
Sign	Calculates the sign of a number
Sin	Calculates the trigonometric sine of an angle specified in radians
Sinh	Calculates the hyperbolic sine of a number
Square	Calculates the square of a number
SquareRoot	Calculates the square root of a number
Subtract	Subtracts two numbers
Tan	Calculates the trigonometric tangent of an angle specified in radians
Tanh	Calculates the hyperbolic tangent of a number
Truncate	Truncates, or bounds, a number within a range
While	Repeatedly executes an expression while a condition, a boolean expression, remains true

Lets consider the problem of symbolic regression. Suppose we have a bunch of data points, say $[(0,0),(1,2),(2,4),(3,6),(4,8),(5,10)]$. The goal of symbolic regression is to find a mathematical expression that fits the data points. In this example, the function $f(x) = x^2$ would work. There is typically no single, unique function that runs through all points, thus creating issues like overfitting the data. However, this topic is outside the scope of this manual.

For this problem, we could construct a rule set using several arithmetic operators. One terminal is included, the variable `x`. We will assign this variable later when evaluating the program. The last setting required is the return type of the program. In this case, the program will return a number.

```
1    Rules rules = new Rules();
```

```
    rules.add(new Add());
    rules.add(new Multiply());
    rules.add(new Subtract());
    rules.add(new Divide());
    rules.add(new Sin());
    rules.add(new Cos());
    rules.add(new Exp());
    rules.add(new Log());
    rules.add(new Get(Number.class, "x"));
    rules.setReturnType(Number.class);
```

7.3.2 Defining the Problem

With the rules defined, we can now define the problem. Programs require only a single decision variable of type `Program`. For example, the `newSolution` method would appear as follows:

```
public Solution newSolution() {
    Solution solution = new Solution(1, 1);
    solution.setVariable(0, new Program(rules));
    return solution;
}
```

Inside the `evaluate` method, we must execute the program. Each program executes inside an environment. The environment holds all of the variables and other identifiers that the program can access throughout its execution. Since we previously defined the variable `x` (with the `Get` node), we want to initialize the value of `x` in the environment. Once the environment is initialized, we can evaluate the program. Since we set the return type to be a number in the rule set, we cast the output from the program's evaluation to a number.

```
public void evaluate(Solution solution) {
    Program program = (Program)solution.getVariable(0);

    Environment environment = new Environment();
    environment.set("x", 15);

    double result = (Number)program.evaluate(environment)).doubleValue();
}
```

Returning to our symbolic regression example, we can now create the problem definition. Our rules will be a set of mathematical operators. Our `evaluate` method will compare the original function to the approximated function. We will compare the two functions by sampling the values at 100 different locations and compute the mean square error. By

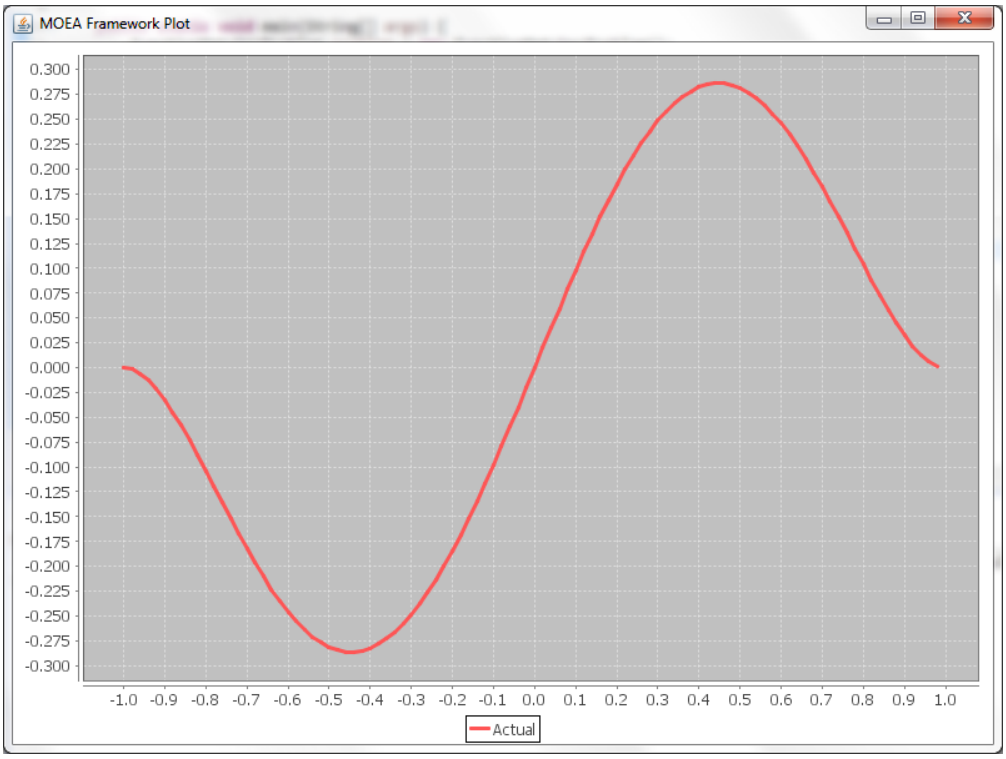
minimizing the mean square error, we minimize the difference between the two functions at the sampled points. The full problem class is shown below:

```
1  package chapter7;
2
3  import org.moeaframework.core.Solution;
4  import org.moeaframework.core.variable.Program;
5  import org.moeaframework.problem.AbstractProblem;
6  import org.moeaframework.util.tree.Add;
7  import org.moeaframework.util.tree.Cos;
8  import org.moeaframework.util.tree.Divide;
9  import org.moeaframework.util.tree.Environment;
10 import org.moeaframework.util.tree.Exp;
11 import org.moeaframework.util.tree.Get;
12 import org.moeaframework.util.tree.Log;
13 import org.moeaframework.util.tree.Multiply;
14 import org.moeaframework.util.tree.Rules;
15 import org.moeaframework.util.tree.Sin;
16 import org.moeaframework.util.tree.Subtract;
17
18 public class FunctionMatcherProblem extends AbstractProblem {
19
20   public double[] x;
21
22   public double[] y;
23
24   private Rules rules;
25
26   public FunctionMatcherProblem() {
27     super(1, 1);
28
29     rules = new Rules();
30     rules.add(new Add());
31     rules.add(new Multiply());
32     rules.add(new Subtract());
33     rules.add(new Divide());
34     rules.add(new Sin());
35     rules.add(new Cos());
36     rules.add(new Exp());
37     rules.add(new Log());
38     rules.add(new Get(Number.class, "x"));
39     rules.setReturnType(Number.class);
40     rules.setMaxVariationDepth(10);
41
42     x = new double[100];
43     y = new double[100];
44
45     for (int i = 0; i < 100; i++) {
46       x[i] = 2.0*(i / 100.0) - 1.0; // range from -1 to 1
47       y[i] = Math.pow(x[i], 5) - 2.0*Math.pow(x[i], 3) + x[i];
48     }
```

```
49    }
50
51    @Override
52    public Solution newSolution() {
53      Solution solution = new Solution(1, 1);
54      solution.setVariable(0, new Program(rules));
55      return solution;
56    }
57
58    public double[] calculate(Program program) {
59      double[] approximatedY = new double[x.length];
60
61      for (int i = 0; i < x.length; i++) {
62        Environment environment = new Environment();
63        environment.set("x", x[i]);
64        approximatedY[i] = ((Number)program.evaluate(environment)).doubleValue()
              ;
65      }
66
67      return approximatedY;
68    }
69
70    @Override
71    public void evaluate(Solution solution) {
72      Program program = (Program)solution.getVariable(0);
73
74      // calculate the difference between the approximation and actual
75      double[] approximatedY = calculate(program);
76      double difference = 0.0;
77
78      for (int i = 0; i < x.length; i++) {
79        difference += Math.pow(Math.abs(y[i] - approximatedY[i]), 2.0);
80      }
81
82      difference = Math.sqrt(difference);
83
84      // protect against NaN
85      if (Double.isNaN(difference)) {
86        difference = Double.POSITIVE_INFINITY;
87      }
88
89      solution.setObjective(0, difference);
90    }
91
92  }
```

MOEAFramework/book/chapter7/FunctionMatcherProblem.java

The target function is defined on lines 45-48. For this example, we are trying to match the function $x^5 - 2x^3 + x$, plotted below.



Use the following code to run this example:

```
package chapter7;

import org.moeaframework.Executor;
import org.moeaframework.analysis.plot.Plot;
import org.moeaframework.core.NondominatedPopulation;
import org.moeaframework.core.variable.Program;

public class RunFunctionMatcherProblem {

  public static void main(String[] args) {
    FunctionMatcherProblem problem = new FunctionMatcherProblem();

    NondominatedPopulation result = new Executor()
        .withAlgorithm("NSGAII")
        .withProblemClass(FunctionMatcherProblem.class)
        .withMaxEvaluations(10000)
        .run();

    // get the resulting function
    Program program = (Program)result.get(0).getVariable(0);

    // print the function
    System.out.println(program);
    System.out.println("Distance: " + result.get(0).getObjective(0));

```

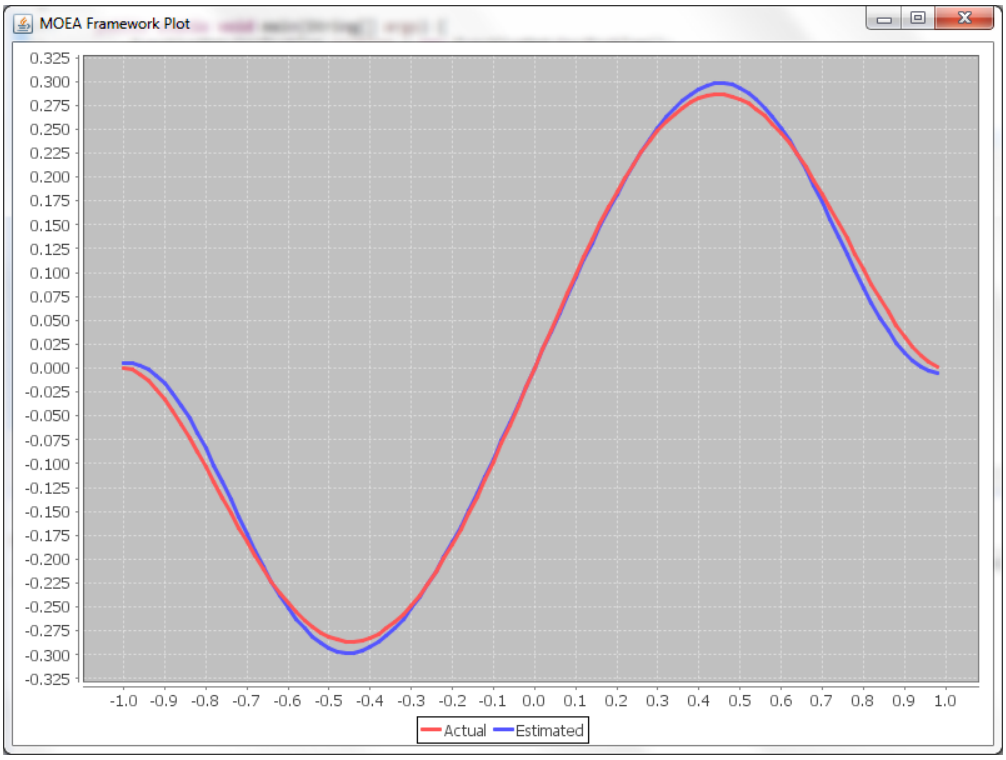
```
26      // display a plot comparing the two functions
27      new Plot()
28          .line("Actual", problem.x, problem.y)
29          .line("Estimated", problem.x, problem.calculate(program))
30          .show();
31    }
32
33  }
```

MOEAFramework/book/chapter7/RunFunctionMatcherProblem.java

The result of running this code is shown below (you may see different results each time you run this program). The program whose output best matches the target function is:

> (Program (Subtract (Sin x) (Multiply (Sin x) (Sin (Multiply (Sin x) (Add (Multiply x (Sin (Divide x (Sin x)))) (Multiply x (Sin (Divide x (Sin x))))))))))

which results in a mean square error of only 0.0275. Not too bad. The corresponding plot of the actual and estimated functions is shown below:



You may observe that the resulting function is more complicated as the original function. It is certainly possible for the optimization algorithm to find the original function, but it may require an extensive amount of time. In reality, we wouldn't know the original function, but would instead only have a bunch of points sampled at different locations.

7.3.3 Custom Operators

In addition to the built-in operators, you can also define your own custom operators. Suppose we want to define an operator to compute the hypotenuse. The operator accepts two inputs, x and y, and returns $\sqrt{x^2 + y^2}$. Our new operator, called `Hypot`, must extend the `Node` class, as shown below.

```
1  package chapter7;
2
3  import org.moeaframework.util.tree.Environment;
4  import org.moeaframework.util.tree.Node;
5
6  public class Hypot extends Node {
7
8    public Hypot() {
9      super(Number.class, Number.class, Number.class);
10   }
11
12   @Override
13   public Node copyNode() {
14     return new Hypot();
15   }
16
17   @Override
18   public Object evaluate(Environment environment) {
19     Number x = (Number)getArgument(0).evaluate(environment);
20     Number y = (Number)getArgument(1).evaluate(environment);
21
22     return Math.hypot(x.doubleValue(), y.doubleValue());
23   }
24
25 }
```

MOEAFramework/book/chapter7/Hypot.java

Three methods are required. The constructor, on lines 8-10, at a minimum must call the constructor of the `Node` class by calling **`super`**. The first argument to **`super`** is the output/return type, and the remaining arguments are the input types. Thus, by calling **`super`**`(Number.class, Number.class, Number.class)`, we are defining an operator with two numeric inputs. The `copyNode` method on lines 13-15 returns a duplicate copy of the node. Finally, on lines 18-23, the `evaluate` method reads the arguments, computes and returns the result.

The standard way to read arguments is shown on lines 19 and 20 and involves three steps. Step 1 is reading the argument by calling `getArgument`. The argument will be another `Node`, which itself could be an constant value or a more complex expression. Step 2 recursively evaluates the arguments in the given environment. Step 3 casts the result to the appropriate type. This should be the same type we defined in the constructor.

For flexibility, we recommend using the abstract `Number` class to represent numeric values rather than `Integer` or `Double`. Doing so allows the operators to work on any numeric inputs. The operator should automatically convert integers to doubles if required.

After defining our `Hypot` class, we add it to the rules:

```
1    rules.add(new Hypot());
```

7.3.4 Ant Problem

A good example of defining custom operators is the Ant problem. The ant problem defines a world containing food at certain positions, and aims to design a robotic ant to navigate through the world and eat the food. This robotic ant is controlled by a program consisting of several operations:

1. if-else condition,
2. turn left,
3. turn right,
4. move forward, and
5. check if food is ahead.

We want to use the MOEA Framework to develop a program consisting of these operations, allowing the robotic ant to eat all the food. The complete code for the ant problem can be found in the `examples/org/moeaframework/examples/gp/ant` folder. Each of the operations is defined in a separate Java class, such as `IsFoodAhead.java` and `TurnLeft.java`. These operators act on a special world environment defined in `World.java`. `World.java` defines the size of the world, the current position of the ant, and the location of food. For example, the world initially starts off empty with some locations containing food, as indicated by the # symbol. For example, the world could appear as follows:

```
###
  #
  #                      ###
  #                     #   #
  #                     #   #
  #### #####       ##
           #                  #
           #       #          #
           # #### #
```

The ant starts at location (0, 0) and, using the program, tries to follow the path of food. As it encounters food, it changes the symbol to @ to indicate the food has been eaten. For example, a successful ant would results in the state:

```
Moves: 78 / 500
Food: 36 / 36
 @@@
   @
   @                        @@@
   @                       @     @
   @                       @     @
   @@@@ @@@@@         @@
             @                    @
             @         @          @
             @ @@@@ @
```

The rules for the ant program are shown below:

```
1     rules = new Rules();
2     rules.add(new TurnLeft());
3     rules.add(new TurnRight());
4     rules.add(new MoveForward());
5     rules.add(new IsFoodAhead());
6     rules.add(new IfElse(Void.class));
7     rules.add(new Sequence(Void.class, Void.class));
8     rules.setReturnType(Void.class);
```

So how does the MOEA Framework know how to combine these operations? The type system. First lets consider `IfElse`. By passing in the type `Void.class`, we construct an if-else structure with three inputs — (condition, if-case, else-case) — with types (`Boolean.class`, `Void.class`, `Void.class`) and a return type of `Void.class`. The condition argument is always type `Boolean.java`, which implies any operation that returns type `Boolean.class` can fit. In this example, only one operation matches: `IsFoodAhead`. The other operations — `TurnLeft`, `TurnRight`, and `MoveForward` — all have a return type of `Void.class`. The type `Void.class` indicates that the expression does not return a value.

`IfElse` and `Sequence` are special structures. You can customize the type of the operator. For example, calling `new IfElse(Void.class)` says that any expression in the if-case or else-case must return type `Void.class`. Likewise, calling `new Sequence(Void.class, Void.class)` allows sequencing together two expressions with return type `Void.class`. This means that the if-else cases and sequences can only contain operators `TurnLeft`, `TurnRight`, `MoveForward`, and nested `IfElse` statements or `Sequences`. Finally, the overall program has a return type of `Void.class` Note the return type is `Void.class`, which indicates the program should not return a value.

Within `examples/org/moeaframework/examples/gp/ant` are two larger examples which can be executed by running `SantaFeExample.java` and `LosAltosExample.java`. You may observe that different programs are generated for these two examples. Since we are evolving a program to solve a single example, the program tends to be "overfit". In other words, the program works well on the example it was designed for, but will perform poorly on other examples. One way to combat overfitting is to evaluate the program on multiple example. The results from each individual evaluation could be averaged. It is also common to use a minimax approach, where you try to minimize the maximum value (or maximimize the minimum value). In this way, you are searching for a program with the best worst-case performance.

Chapter 8

Integers and Mixed Integer Programming

Another discrete type commonly seen in optimization is integers. We devote a chapter to integers because there are actually two ways to represent integers, as we will discuss below. Additionally, it is commonly to see problems with integers and real-values mixed together, forming what is called mixed integer programming.

8.1 Two Representations

The MOEA Framework supports two representations for integers. One is backed by real-values and the other by bit strings. The real-value representation is straightforward. Simply encode your integer as a real-value and cast/truncate to an integer. The `EncodingUtils.newInt(a, b)` and `EncodingUtils.getInt(...)` methods conveniently perform this function. The bit string representation on the other hand is similar to the binary representation of numbers in a computer. There are caveats to using both methods, which must be carefully understood to avoid potential pitfalls.

Real-Valued Representation for Integers

When using `EncodingUtils.newInt(a, b)`, the underlying variable is actually a `RealVariable` bounded by $[a, b + 1)$. When reading the value using `EncodingUtils.getInt(...)`, the real-value is truncated using the `floor` operation to convert it to an integer.

Advantages:

- Works with all algorithms and operators for real-values.
- Can be combined with other real-valued decision variables without changing operators.

Disadvantages:

- Small changes to the underlying real value will not cause any changes when truncated to an integer.

Bit String Representation for Integers

When using `EncodingUtils.newBinaryInt(a, b)`, the underlying variable is actually a `BinaryVariable`. The length of the bit string is chosen to encompass the bounds of the integer. The corresponding value is read using `EncodingUtils.getInt(...)`. There are two methods to encode the bit string. First is the traditional binary encoding used by computers. The disadvantage of binary encoding is that a single bit flip can cause a substantial change to the resulting integer. Alternatively, you can use gray coding, which is designed so that there exists a single bit flip to move between adjacent integers. This enables smoother transitions between values, but large jumps can still occur.

Advantages:

- Works with all algorithms and operators for bit strings (binary).
- Mutation operators are guaranteed to change the value of the integer.

Disadvantages:

- If $b - a < 2^n$, then some values will occur with higher probability (since more than one bit pattern maps to the integer).
- Large changes to the integer value can be caused by small changes to the underlying bit string.

Gray coding is used by default. Binary coding can be enabled by calling the constructor `new BinaryIntegerVariable(a, b, false)`.

Demonstration

Lets look at how both representations work on a problem. We will modify the Schaffer problem from Chapter 2 to use integer values instead of real values. For example, here is the version using the real-valued representation for integers:

```
1 package chapter8;
2
3 import org.moeaframework.core.Solution;
4 import org.moeaframework.core.variable.EncodingUtils;
5 import org.moeaframework.problem.AbstractProblem;
6
7 public class RealIntegerSchafferProblem extends AbstractProblem {
```

```
8
9   public RealIntegerSchafferProblem() {
10    super(1, 2);
11  }
12
13  @Override
14  public void evaluate(Solution solution) {
15    double x = EncodingUtils.getInt(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.newInt(-10, 10));
25    return solution;
26  }
27
28 }
```

MOEAFramework/book/chapter8/RealIntegerSchafferProblem.java

Alternatively, here is the same class using the bit string representation:

```
1  package chapter8;
2
3  import org.moeaframework.core.Solution;
4  import org.moeaframework.core.variable.EncodingUtils;
5  import org.moeaframework.problem.AbstractProblem;
6
7  public class BinaryIntegerSchafferProblem extends AbstractProblem {
8
9    public BinaryIntegerSchafferProblem() {
10     super(1, 2);
11   }
12
13   @Override
14   public void evaluate(Solution solution) {
15     double x = EncodingUtils.getInt(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.newBinaryInt(-10, 10));
25     return solution;
26   }
```

```
27
28 }
```

MOEAFramework/book/chapter8/BinaryIntegerSchafferProblem.java

Observe that only line 24 changes. Since both representations are read using `EncodingUtils.getInt(...)`, the `evaluate` method remains unchanged. We can then run both versions:

```
1  package chapter8;
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 RunIntegerSchafferProblems {
9
10   public static void main(String[] args) {
11     NondominatedPopulation result1 = new Executor()
12         .withAlgorithm("NSGAII")
13         .withProblemClass(BinaryIntegerSchafferProblem.class)
14         .withMaxEvaluations(10000)
15         .run();
16
17     NondominatedPopulation result2 = new Executor()
18         .withAlgorithm("NSGAII")
19         .withProblemClass(RealIntegerSchafferProblem.class)
20         .withMaxEvaluations(10000)
21         .run();
22
23     System.out.println("Binary Integer Encoding:");
24     for (Solution solution : result1) {
25       System.out.printf("  %d => %.5f, %.5f\n",
26           EncodingUtils.getInt(solution.getVariable(0)),
27           solution.getObjective(0),
28           solution.getObjective(1));
29     }
30
31     System.out.println();
32     System.out.println("Real Integer Encoding:");
33     for (Solution solution : result2) {
34       System.out.printf("  %d => %.5f, %.5f\n",
35           EncodingUtils.getInt(solution.getVariable(0)),
36           solution.getObjective(0),
37           solution.getObjective(1));
38     }
39   }
40
41 }
```

MOEAFramework/book/chapter8/RunIntegerSchafferProblems.java

Which produces the following output:

```
Binary Integer Encoding:
  0 => 0.00000, 4.00000
  1 => 1.00000, 1.00000
  2 => 4.00000, 0.00000

Real Integer Encoding:
  0 => 0.00000, 4.00000
  1 => 1.00000, 1.00000
  2 => 4.00000, 0.00000
```

On a simple problem such as this, both representations produce identical results. On more complex problems, you may observe some differences in performance based on which version is used. Experimentation is a good way to determine which version is appropriate for your problem.

8.2 Mixed Integer Programming

Real-world problems tend to not exclusively use only real-values or integers, but instead are formulated as a combination of decision variable types. When mixing real values and integers, one must be mindful of the underlying representation for integers. For example, lets modify the Srinivas problem from Chapter 3 to include one integer and one real-valued decision variable:

```
1  package chapter8;
2
3  import org.moeaframework.core.Solution;
4  import org.moeaframework.core.variable.EncodingUtils;
5  import org.moeaframework.problem.AbstractProblem;
6
7  public class MixedIntegerSrinivasProblem extends AbstractProblem {
8
9    public MixedIntegerSrinivasProblem() {
10     super(2, 2, 2);
11   }
12
13   @Override
14   public void evaluate(Solution solution) {
15     int x = EncodingUtils.getInt(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.newBinaryInt(-20, 20));
33      solution.setVariable(1, EncodingUtils.newReal(-20.0, 20.0));
34
35      return solution;
36    }
37
38  }
```

MOEAFramework/book/chapter8/MixedIntegerSrinivasProblem.java

If using the real-valued representation for integers, then the same operators can be shared for both real-values and integers. However, if you try to combine the bit string representation with other real-valued variables, you will see the following error:

```
Exception in thread "main" org.moeaframework.core.spi.
    ProviderNotFoundException: no provider for NSGAII
  at org.moeaframework.algorithm.StandardAlgorithms.getAlgorithm(
      StandardAlgorithms.java:248)
  at org.moeaframework.core.spi.AlgorithmFactory.instantiateAlgorithm(
      AlgorithmFactory.java:175)
  at org.moeaframework.core.spi.AlgorithmFactory.getAlgorithm(AlgorithmFactory
      .java:137)
  at org.moeaframework.Executor.runSingleSeed(Executor.java:773)
  at org.moeaframework.Executor.run(Executor.java:730)
  at chapter8.RunMixedIntegerSrinivasProblem.main(
      RunMixedIntegerSrinivasProblem.java:16)
Caused by: org.moeaframework.core.spi.ProviderLookupException: unable to find
    suitable variation operator
  at org.moeaframework.core.spi.OperatorFactory.lookupVariationHint(
      OperatorFactory.java:347)
  at org.moeaframework.core.spi.OperatorFactory.getVariation(OperatorFactory.
      java:182)
  at org.moeaframework.core.spi.OperatorFactory.getVariation(OperatorFactory.
      java:160)
  at org.moeaframework.algorithm.StandardAlgorithms.newNSGAII(
      StandardAlgorithms.java:329)
  at org.moeaframework.algorithm.StandardAlgorithms.getAlgorithm(
      StandardAlgorithms.java:207)
```

```
  ... 5 more
```

Note the line that says "unable to find suitable variation operator". This is an indication that no operator is compatible with the decision variables in the problem. The MOEA Framework automatically determines the appropriate operators given the decision variable types. However, this only works when all decision variables are the same type. If you try to mix the bit string representation of integers with other real-valued variables, you will run into this error. However, in most cases we can address this error by explicitly specifying the operators to use:

```
1  package chapter8;
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 RunMixedIntegerSrinivasProblem {
9
10   public static void main(String[] args) {
11     NondominatedPopulation result = new Executor()
12         .withAlgorithm("NSGAII")
13         .withProblemClass(MixedIntegerSrinivasProblem.class)
14         .withMaxEvaluations(10000)
15         .withProperty("operator", "sbx+hux+pm+bf")
16         .run();
17
18     for (Solution solution : result) {
19       if (!solution.violatesConstraints()) {
20         System.out.format("%3d %7.3f => %7.3f %7.3f%n",
21             EncodingUtils.getInt(solution.getVariable(0)),
22             EncodingUtils.getReal(solution.getVariable(1)),
23             solution.getObjective(0),
24             solution.getObjective(1));
25       }
26     }
27   }
28
29 }
```

MOEAFramework/book/chapter8/RunMixedIntegerSrinivasProblem.java

Here, we explicitly specify the operators to use on line 15. Since we are combining bit string and real-valued variables, we must include operators for both types using the operator string `"sbx+hux+pm+bf"`. Each operator is applied in the order they appear in the string. Furthermore, we want crossover operators to appear at the beginning and mutation operators at the end. All operators are type-safe, meaning they will only affect decision variables of the supported type. For example, the `"hux"` and `"bf"` (bit flip) operators will only affect

bit strings, while "sbx" and "pm" will only evolve real-values.

Such overloading of operators is also necessary when mixing other decision variable types. Again, just follow the rules above for setting the operator string. In the event that you provide an invalid combination of operators, then you will be given an error message describing the problem.

Chapter 9

I/O Basics

This chapter talks all about I/O: input and output. We'll walk through several use cases and provides example code.

9.1 Printing Solutions

The first thing you'll likely want to do after solving a problem with the MOEA Framework is viewing your solutions. Suppose we want to print the decision variables and objectives for all solutions in a result set:

```
1  package chapter9;
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
9  import chapter2.SchafferProblem;
10
11 public class PrintSolutions {
12
13   public static void main(String[] args) throws IOException {
14     NondominatedPopulation result = new Executor()
15         .withAlgorithm("NSGAII")
16         .withProblemClass(SchafferProblem.class)
17         .withMaxEvaluations(10000)
18         .run();
19
20     for (int i = 0; i < result.size(); i++) {
21       Solution solution = result.get(i);
22       System.out.print("Solution " + i + ":");
23
24       for (int j = 0; j < solution.getNumberOfVariables(); j++) {
```

```
25          System.out.print(" ");
26          System.out.print(solution.getVariable(j));
27        }
28
29        System.out.print(" =>");
30
31        for (int j = 0; j < solution.getNumberOfObjectives(); j++) {
32          System.out.print(" ");
33          System.out.print(solution.getObjective(j));
34        }
35
36        System.out.println();
37      }
38    }
39
40  }
```

MOEAFramework/book/chapter9/PrintSolutions.java

The following output is produced:

```
Solution 0: 2.0004392968937066 => 4.001757380556588 1.929817608202561E-7
Solution 1: -3.7964704972906493E-4 => 1.441318823679831E-7 4.001518732330799
Solution 2: 0.06057459075229121 => 0.0036692810448075643 3.7613709180356425
Solution 3: 1.657358848462215 => 2.7468383525759994 0.11740295872713932
Solution 4: 0.9436776287650016 => 0.8905274670315362 1.11581695197153
Solution 5: 1.8601691079468476 => 3.460229110159771 0.019552678372380346
Solution 6: 1.4155832357245992 => 2.003875897264526 0.34154295436612936
...
```

We could clean up this output in several ways. First, the output shows way too many decimal digits. We can print formatted numbers as follows:

```
1     for (int i = 0; i < result.size(); i++) {
2       Solution solution = result.get(i);
3       System.out.print("Solution " + i + ":");
4
5       for (int j = 0; j < solution.getNumberOfVariables(); j++) {
6         System.out.printf(" %.3f", EncodingUtils.getReal(solution.getVariable(
              j)));
7       }
8
9       System.out.print(" =>");
10
11      for (int j = 0; j < solution.getNumberOfObjectives(); j++) {
12        System.out.printf(" %.3f", solution.getObjective(j));
13      }
14
15      System.out.println();
16    }
```

```
17    }
```

MOEAFramework/book/chapter9/PrintFormattedSolutions.java

This is better:

```
Solution 0: 0.000 => 0.000 4.001
Solution 1: 2.000 => 4.001 0.000
Solution 2: 1.048 => 1.099 0.906
Solution 3: 0.374 => 0.140 2.643
Solution 4: 0.335 => 0.113 2.771
Solution 5: 0.418 => 0.175 2.503
Solution 6: 1.947 => 3.790 0.003
...
```

A second improvement would be to order the solutions. Right now, the solutions appear in a random order. However, we could sort them lexicographically by their objective values.

```
1     result.sort(new LexicographicalComparator());
```

MOEAFramework/book/chapter9/PrintLexicographicalOrdering.java

```
Solution 0: 0.000 => 0.000 4.000
Solution 1: 0.020 => 0.000 3.919
Solution 2: 0.024 => 0.001 3.903
Solution 3: 0.043 => 0.002 3.831
Solution 4: 0.047 => 0.002 3.815
Solution 5: 0.075 => 0.006 3.707
Solution 6: 0.103 => 0.011 3.599
...
```

Great. Now we can clearly see an inverse relationship between the two objectives. As the first objective increases, the second objective decreases.

Note: When printing result sets, always remember to check `solution.violatesConstraints()` if the problem has constraints. Otherwise, you may display infeasible solutions in the output.

9.2 Files

Instead of printing the information to the console, we can save the solutions directly to a file. There are advantages and disadvantages of using files. The primary advantage is that you can load the saved data. For example, you could save the results from several runs, then load them in later to generate plots. The primary disadvantage is that the generated file is not human-readable. You can't open the saved file in a text editor to see its contents. It can only be reloaded into the MOEA Framework.

We use the `PopulationIO` class to read and write files. For example, here we save the result to a file:

```
    try {
      PopulationIO.write(new File("solutions.dat"), result);
      System.out.println("Saved " + result.size() + " solutions!");
    } catch (IOException e) {
      e.printStackTrace();
    }
```

MOEAFramework/book/chapter9/SaveToFile.java

The file can then be loaded back into the MOEA Framework:

```
    try {
      Population result = PopulationIO.read(new File("solutions.dat"));

      System.out.println("Read " + result.size() + " solutions!");
    } catch (IOException e) {
      e.printStackTrace();
    }
```

MOEAFramework/book/chapter9/LoadFromFile.java

If you need to save the solutions in a human-readable format, use the printing commands from the previous section.

9.3 Checkpoints

If checkpoints are enabled, a running algorithm will periodically save checkpoint files. The checkpoint file stores the current state of the algorithm. If the run is interrupted, such as during a power outage, the run can be resumed at the last saved checkpoint. Checkpointing can be enabled when using the `Executor` to run algorithms. Calling `setCheckpointFile` sets the file location for the checkpoint file, and `checkpointEveryIteration` or `setCheckpointFrequency` control how frequently the checkpoint file is saved.

Resuming a run from a checkpoint occurs automatically. If the checkpoint file does not exist, a run starts from the beginning. However, if the checkpoint file exists, then the run is automatically resumed at that checkpoint. For this reason, care must be taken when using checkpoints as they can be a source of confusion for new users. For instance, using the same checkpoint file from an unrelated run can cause unexpected behavior or an error. For this reason, checkpoints are recommended only when solving time-consuming problems.

```
package chapter9;

import java.io.File;

import org.moeaframework.Executor;
```

```java
import org.moeaframework.core.NondominatedPopulation;

import chapter2.SchafferProblem;

public class Checkpoints {

  public static void main(String[] args) {
    File checkpointFile = new File("checkpoint.dat");
    long start = System.currentTimeMillis();

    if (checkpointFile.exists()) {
      System.out.println("Checkpoint file exists, will resume from prior run!");
    }

    NondominatedPopulation result = new Executor()
        .withAlgorithm("NSGAII")
        .withProblemClass(SchafferProblem.class)
        .withMaxEvaluations(1000000)
        .withCheckpointFrequency(10000)
        .withCheckpointFile(checkpointFile)
        .run();

    System.out.println("Elapsed time: " + (System.currentTimeMillis() - start)
        / 1000 + "s");
  }

}
```

MOEAFramework/book/chapter9/Checkpoints.java

The first time you run this code, it'll take several seconds to produce the result. If you run the code a second time, it should terminate immediately since the checkpoint file exists. Note that checkpoint files are never deleted by the MOEA Framework. Each time you run this example, it will resume from its last save point. If you want to run this example from the beginning, you must delete the checkpoint file manually.

9.4 Creating Reference Sets

All of the test problems provided by the MOEA Framework are accompanied by a predefined reference set. The reference set contains the Pareto optimal solutions to a problem, or if the set is too large, an approximation of the Pareto optimal solutions. If you are working on a new problem, you may not have a reference set available. A common method of generating reference sets, particularly for real-world problems, is to solve the problem using many different algorithms repetitively. The combined result from all runs becomes your reference set.

Lets start by generating the reference set for the Schaffer problem:

```java
package chapter9;

import java.io.File;
import java.io.IOException;

import org.moeaframework.Executor;
import org.moeaframework.core.NondominatedPopulation;
import org.moeaframework.core.PopulationIO;

import chapter2.SchafferProblem;

public class CreatingReferenceSet {

  public static void main(String[] args) throws IOException {
    int nseeds = 25;
    String[] algorithms = new String[] { "NSGAII", "GDE3", "OMOPSO" };

    NondominatedPopulation referenceSet = new NondominatedPopulation();

    Executor executor = new Executor()
        .withProblemClass(SchafferProblem.class)
        .withMaxEvaluations(10000);

    for (String algorithm : algorithms) {
      executor.withAlgorithm(algorithm);

      for (int i = 0; i < nseeds; i++) {
        referenceSet.addAll(executor.run());
      }
    }

    PopulationIO.writeObjectives(new File("Schaffer.pf"), referenceSet);
  }

}
```

MOEAFramework/book/chapter9/CreatingReferenceSet.java

On line 18 we create the `NondominatedPopulation` that will store the combined Pareto solutions from all runs. The loop on line 24 runs each of our test algorithms for 25 repetitions. Finally, we save the reference set to a file on line 32.

When using a `NondominatedPopulation`, an issue one frequently encounters is that the sets can become arbitrarily large. There is no bound on the size of the set. The only restriction is that all solutions must be non-dominated. After running the above code, we observed that the set contained 9700 solutions. This can become cumbersome to work with in practice!

A common technique for reducing the size of a reference set is to use ϵ-dominance (Laumanns et al., 2002). ϵ-dominance is similar to Pareto dominance, except the ϵ defines a minimum resolution of solutions. Larger ϵs produce smaller sets. With a small tweak to the previous code, we can define ϵs:

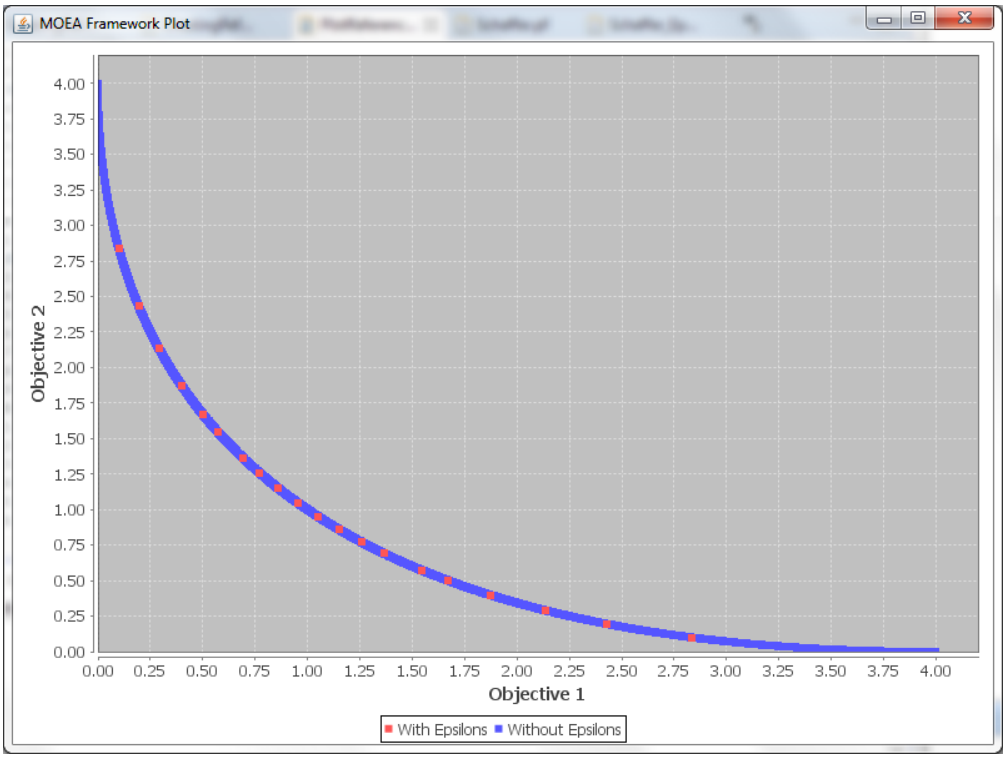
```
    NondominatedPopulation referenceSet = new EpsilonBoxDominanceArchive(
        new double[] { 0.1, 0.1 });
```

MOEAFramework/book/chapter9/CreatingReferenceSetWithEpsilons.java

With ϵs, the resulting reference set contains only 20 solutions. Plotting the two sets shows the drastic difference:

```
package chapter9;

import java.io.File;
import java.io.IOException;

import org.moeaframework.analysis.plot.Plot;
import org.moeaframework.core.Population;
import org.moeaframework.core.PopulationIO;

public class PlotReferenceSet {

  public static void main(String[] args) throws IOException {
    Population withEpsilons = PopulationIO.readObjectives(
        new File("Schaffer_Epsilon.pf"));

    Population withoutEpsilons = PopulationIO.readObjectives(
        new File("Schaffer.pf"));

    new Plot()
        .add("With Epsilons", withEpsilons)
        .add("Without Epsilons", withoutEpsilons)
        .show();
  }

}
```

MOEAFramework/book/chapter9/PlotReferenceSet.java



Observe that the red and blue points lie on the same curve, except we approximate the curve with significantly fewer red points. If using this reference set in practice, we would probably want to pick slightly smaller ϵs, since 20 points may be too small. For this problem, 100 would be ideal.

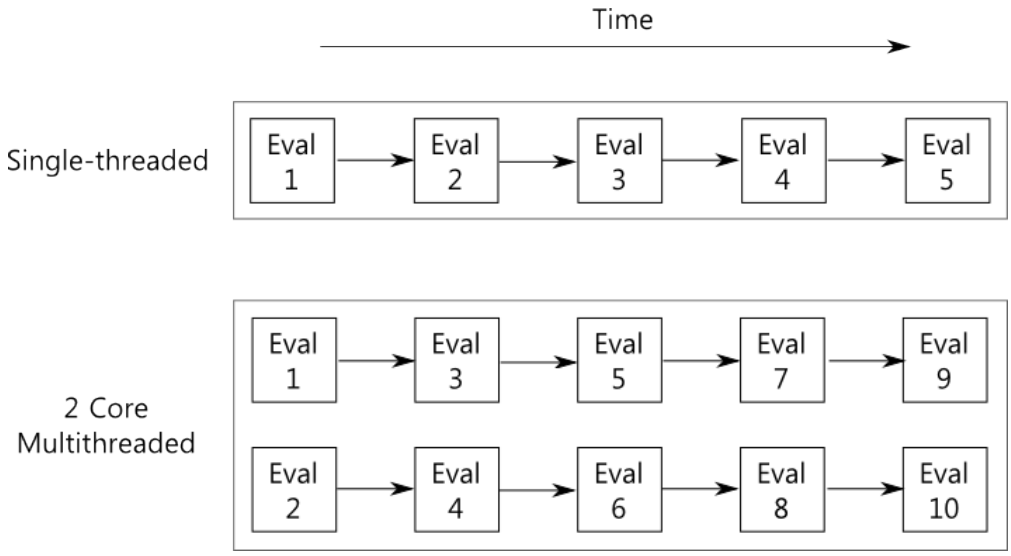
Chapter 10

Performance Enhancements

This chapter details two ways to improve the performance of your optimization. The first utilizes multithreading on your computer to utilize all available computing cores. The second rewrites the problem definition in another language. We consider compiling to a native library using C/C++ and writing functions in Python. Other languages can be supported following similar techniques.

10.1 Multithreading

When you typically optimize a program, you evaluate one solution at a time. This is a single-threaded program, where each step is performed sequentially. However, most modern day computers have processors with multiple computing cores. Suppose a computer has 4 cores. A single-threaded program would only utilize 1 of the 4 cores, or 25% of the available resources. Multithreaded programs are designed to split the computations into multiple execution threads, each of which can be executed on a separate core simultaneously. In the context of an MOEA, as shown below, the objective function evaluations can be multithreaded.



This diagram depicts the 2 core multithreaded program achieving twice as many evaluations as the single threaded program. In reality, it is a bit more nuanced. First, not all MOEAs can support multithreading. Some are inherently single threaded. The MOEA Framework uses special "future" objects to automatically detect if an MOEA support multithreading. Second, there is often additional overhead encountered when using multithreading or any type of parallelization. This overhead comes from a variety of sources, but perhaps the most common source of overhead is from communicating data between the two cores in memory. Memory access tends to be significantly slower than CPU processing. If the problem consists only of a few lines of code, then likely the evaluation time is less than the communication overhead. Enabling multithreading would slow down the program in this case. Third, if you attempt to utilize all cores on a computer, particularly one with iterative user sessions such as Windows, you will have to compete with other programs for time on the CPU.

To demonstrate this, we will modify our Schaffer problem. The original Schaffer problem is very simple and very fast to compute. To slow down the evaluations, lets add a loop to add some additional computing time to each function evaluation:

```
    double sum = 0.0;

    for (int i = 0; i < 100000; i++) {
      sum += i;
    }
```

On a modern computer, this adds approximately 0.1 milliseconds per evaluation. Thus, our new Schaffer problem, which we call `ExpensiveSchafferProblem`, would appear as follows:

```
package chapter10;
```

```java

import org.moeaframework.core.Solution;
import org.moeaframework.core.variable.EncodingUtils;
import org.moeaframework.problem.AbstractProblem;

public class ExpensiveSchafferProblem extends AbstractProblem {

  public ExpensiveSchafferProblem() {
    super(1, 2);
  }

  @Override
  public void evaluate(Solution solution) {
    double x = EncodingUtils.getReal(solution.getVariable(0));

    // perform some expensive calculation
    double sum = 0.0;

    for (int i = 0; i < 100000; i++) {
      sum += i;
    }

    solution.setObjective(0, Math.pow(x, 2.0));
    solution.setObjective(1, Math.pow(x - 2.0, 2.0));
  }

  @Override
  public Solution newSolution() {
    Solution solution = new Solution(1, 2);
    solution.setVariable(0, EncodingUtils.newReal(-10.0, 10.0));
    return solution;
  }

}
```

MOEAFramework/book/chapter10/ExpensiveSchafferProblem.java

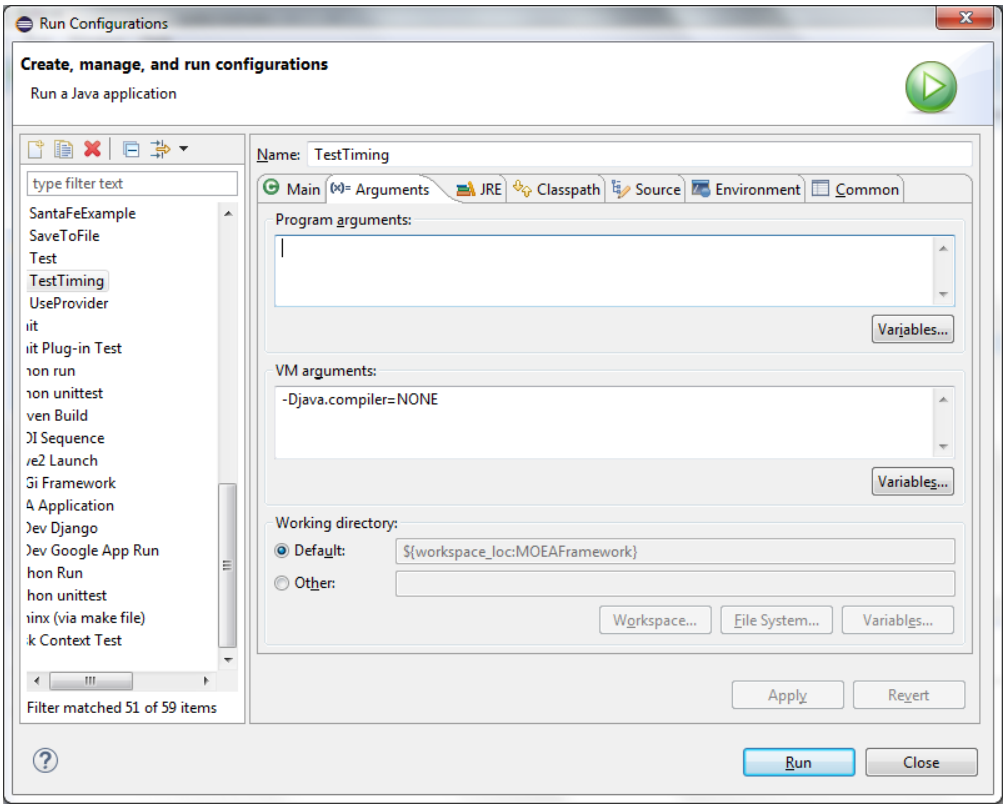
Enabling multithreading in the MOEA Framework is straightforward: when using the `Executor` simply call `distributeOnAllCores()` to run on all available cores, or call `distributeOn(N)` to run on N cores. Any optimization algorithm that supports multithreading will automatically distribute evaluations on the requested number of cores on your computer. In the following code, we compare the runtime between the single threaded version and a multithreaded version:

```java
package chapter10;

import org.moeaframework.Executor;

public class TestTiming {

  public static void main(String[] args) {
```

```
8       long startTime;
9
10      // run without multithreading
11      startTime = System.currentTimeMillis();
12
13      new Executor()
14          .withAlgorithm("NSGAII")
15          .withProblemClass(ExpensiveSchafferProblem.class)
16          .withMaxEvaluations(10000)
17          .run();
18
19      System.out.println("Single threaded: " + (System.currentTimeMillis()-
            startTime));
20
21      // run with mulithreading
22      startTime = System.currentTimeMillis();
23
24      new Executor()
25          .withAlgorithm("NSGAII")
26          .withProblemClass(ExpensiveSchafferProblem.class)
27          .withMaxEvaluations(10000)
28          .distributeOnAllCores()
29          .run();
30
31      System.out.println("Multithreading: " + (System.currentTimeMillis()-
            startTime));
32    }
33
34  }
```

MOEAFramework/book/chapter10/TestTiming.java

Prior to running the example, it is important to set the `-Djava.compiler=NONE` Java VM option. Without this option, Java will attempt to optimize our program and will remove the for loop. In practice, this optimization is very useful, but for this demonstration we must temporarily disable it. In Eclipse, you set it in the run configuration window, as shown below.



Running this program produces the output below:

```
Single threaded: 14339
Multithreading: 4679
```

It took 14.3 seconds for the single threaded program and 4.6 seconds for the multithreaded program on an 4 core machine. We would expect the multithreaded program to run in $14.3/4 = 3.575$ seconds, but due to overhead it takes much longer. When comparing single threaded and multithreaded programs, there are two useful calculations. The first is speedup:

$$S = T_s/T_p, \tag{10.1}$$

where S is the parallel speedup, T_s is the total runtime for the single threaded (or serial) version, and T_p is the total runtime for the multithreaded (or parallel) version. In our example, the speedup is $14.3/4.6 = 3.1$, meaning that our multithreaded version runs approximately three times faster. The second calculation is efficiency:

$$E = S/p, \tag{10.2}$$

where E is the parallel efficiency, S is the parallel speedup, and p is the number of processors or cores. Thus, in our example, the efficiency is $3.1/4 = 0.78$ or 78%. In other words, we

utilized 78% of the available computing resources. The remaining 22% is lost due to overhead. Larger overheads require longer evaluation times in order to achieve higher efficiency. Conversely, smaller overheads make multithreading and parallelization more efficient. If the overhead is quite large or the evaluation time is too short, it is possible for the multithreaded version to take longer to run! If considering the use of multithreading, it is good practice to test the speedup and efficiency on a small run.

10.2 Termination Conditions

In all of the examples seen thus far, we have run the algorithm for a fixed number of evaluations using `withMaxEvaluations(...)`. Running for a fixed number of evaluations is commonly used when comparing different algorithms, since each algorithm will be given the same number of evaluations. However, in practical situations, one is often limited by computing resources. For example, management may require an engineering team to devise a feasible solution within a day or just a few hours. This can be achieved by replacing `withMaxEvaluations(...)` with `withMaxTime(...)`.

```
1     new Executor()
2         .withAlgorithm("NSGAII")
3         .withProblemClass(ExpensiveSchafferProblem.class)
4         .withMaxTime(30000) // 30 seconds
5         .run();
```

MOEAFramework/book/chapter10/TerminationCondition.java

In this example, we are running the algorithm for 30 seconds, as indicated by `withMaxTime(30000)`. Note that the time is specified in milliseconds (1 second = 1000 milliseconds).

You may notice that the algorithm takes slightly longer than 30 seconds to run. This is because the algorithm is allowed to finish the current iteration before terminating. For problems with longer evaluation times, this can become more noticeable. Be aware of this when scheduling jobs on computing resources that have hard time limits.

10.3 Native Compilation

While Java itself is a fast language compared to interpreted languages like Python, it still remains around 2-3 times slower than C, C++, and Fortran. If your evaluation contains a non-trivial amount of code, you could experience speedup by writing the evaluation code in one of these languages and compiling the code natively. Java Native Access (JNA)[1] is an open source library that lets a Java program invoke natively compiled code. If your problem is already written in one of these native languages, you can also use JNA to call the functions directly.

[1] `https://github.com/java-native-access/jna`

Lets continue with the Schaffer problem. Suppose we wanted to re-write the Schaffer problem in C. We would need a C function with two arguments: an array of decision variables and an array of the objectives. If the problem has constraints, we would include a third array. Our function, called `schaffer`, would look like:

```
#include <math.h>

void schaffer(double* vars, double* objs) {
  objs[0] = pow(vars[0], 2.0);
  objs[1] = pow(vars[0] - 2.0, 2.0);
}
```

MOEAFramework/book/chapter10/schaffer.c

We can then compile this code to a native shared library as shown below.

```
gcc -O3 -shared -o schaffer.dll schaffer.c -lm
```

When compiling code natively, there are many issues that can arise. First, you must determine if your computer is 32-bit or 64-bit. If you have a 64-bit computer, you will want to ensure that your installed version of Java is 64-bit. You will likely also need to include the `-m64` option when compiling the example code, as shown below. Failing to do so will produce an `UnsatisfiedLinkException` error message when running this example.

```
gcc -m64 -O3 -shared -o schaffer.dll schaffer.c -lm
```

Second, you must install an appropriate compiler for your system. If using a 64-bit machine, be sure to get a 64-bit compatible compiler. On Windows, we recommend MinGW[2]. Third, you will need to use an appropriate shared library extension for your operating system. As shown in these examples, we use the `.dll` extension on Windows. Following a Unix/Linux naming convention, you would call the library `libschaffer.so`.

Next, we need to write a Java class that interfaces with the native library we just compiled, as shown below:

```
package chapter10;

import org.moeaframework.core.Solution;
import org.moeaframework.core.variable.EncodingUtils;
import org.moeaframework.problem.AbstractProblem;

import com.sun.jna.Native;

public class JNASchafferProblem extends AbstractProblem {
```

[2] http://www.mingw.org/

```
10
11    public static native double schaffer(double[] vars, double[] objs);
12
13    static {
14      System.setProperty("jna.library.path", "./book/chapter9");
15      Native.register("schaffer");
16    }
17
18    public JNASchafferProblem() {
19      super(1, 2);
20    }
21
22    @Override
23    public void evaluate(Solution solution) {
24      double[] vars = EncodingUtils.getReal(solution);
25      double[] objs = new double[solution.getNumberOfObjectives()];
26
27      schaffer(vars, objs);
28
29      solution.setObjectives(objs);
30    }
31
32    @Override
33    public Solution newSolution() {
34      Solution solution = new Solution(1, 2);
35      solution.setVariable(0, EncodingUtils.newReal(-10.0, 10.0));
36      return solution;
37    }
38
39  }
```

MOEAFramework/book/chapter10/JNASchafferProblem.java

As with previous problems, we extend the `AbstractProblem`. However, unlike previous examples, we need to connect this code to the native library. First, on line 11, we create a "native" version of the `schaffer` function. The name and arguments to this function must match the name and arguments of the C function. In this case, a double array (`double[]`) maps to a double pointer in C (`double*`).

Second, on lines 13-16, we connect our Java class with the native library using JNA. We first set the `jna.library.path` system property on line 14 to include the path to the compiled library (i.e., `schaffer.dll`). This is necessary for JNA to locate the library. Then, on line 15, we call `Native.register("schaffer")` register the code with JNA. JNA searches its library path for the shared library, called `schaffer.dll` (on Windows) or `libschaffer.so` (on Linux). Upon finding the library, it loads it into memory and configures Java so that calls to the `schaffer` method in Java are forwarded to the native library.

Finally, in the `evaluate` method on lines 23-30, we call the `schaffer` method to evaluate the problem. On line 24, we create a double array with the decision variables, and on line 25 we create an empty array to store the resulting objectives. The `schaffer`

method is invoked on line 27 to call the underlying native code. Lastly, we set the objectives for the solution. One would handle constraints similarly.

We can then optimize the problem as we have done in the past:

```
1     NondominatedPopulation result = new Executor()
2         .withAlgorithm("NSGAII")
3         .withProblemClass(JNASchafferProblem.class)
4         .withMaxEvaluations(10000)
5         .run();
```

MOEAFramework/book/chapter10/ComparingCInterface.java

10.4 Standard I/O

If native compilation is not an option, the MOEA Framework also supports running models via standard I/O or network sockets with the `ExternalProblem` class. Here, we will discuss standard I/O using Python.

The `ExternalProblem` class defines a very simple interface for communicating with models over standard input and output (I/O). When the optimization starts, the MOEA Framework launches the process for evaluating solutions. For each solution, it sends a single line to the process containing the decision variables using whitespace to separate values. For example, it would pass five real-valued decision variables as:

```
0.109 0.912 0.3 0.291 0.154
```

The process parses the input line, evaluates the problem, and writes the objectives and constraints to its standard output, again separated by whitespace. For example, a problem with two objectives would write:

```
1.05 0.9
```

Any constraints would appear on the same line following the objective values. If the process receives a blank line, it terminates as the optimization has finished. For example, our Schaffer problem could be written in Python as follows:

```
1 from sys import *
2 from math import *
3
4 while True:
5     # Read the next line from standard input
6     line = raw_input()
7
8     # If line is empty, stop
```

```
9     if line == "":
10        break
11
12    # Parse the decision variables from the input
13    vars = map(float, line.split())
14
15    # Evaluate the Schaffer problem
16    objs = (vars[0]**2, (vars[0] - 2)**2)
17
18    # Print objectives to standard output, flush to write immediately
19    print "%f %f" % objs
20    stdout.flush()
```

MOEAFramework/book/chapter10/schaffer.py

Line 4 starts an infinite loop that reads each line of input (line 6) and terminates only if a blank line is encountered (line 9). Line 13 parses each value from the input into the array `vars`. Next, the objectives are calculated on line 16 and written to the process' output on line 19. Always flush the output to immediately print the objectives as shown on line 20, otherwise the program may stall.

In addition to the Python code, we must also create a Java class to represent the problem. Unlike the previous examples, when connecting to an external process in this manner, we want to extend the `ExternalProblem` class, as shown below:

```
1  package chapter10;
2
3  import java.io.IOException;
4
5  import org.moeaframework.core.Solution;
6  import org.moeaframework.core.variable.EncodingUtils;
7  import org.moeaframework.problem.ExternalProblem;
8
9  public class StdioSchafferProblem extends ExternalProblem {
10
11   public StdioSchafferProblem() throws IOException {
12     super("python", "book/chapter9/schaffer.py");
13   }
14
15   @Override
16   public String getName() {
17     return "ExternalSchafferProblem";
18   }
19
20   @Override
21   public int getNumberOfConstraints() {
22     return 0;
23   }
24
25   @Override
26   public int getNumberOfObjectives() {
```

```
27      return 2;
28    }
29
30    @Override
31    public int getNumberOfVariables() {
32      return 1;
33    }
34
35    @Override
36    public Solution newSolution() {
37      Solution solution = new Solution(1, 2);
38      solution.setVariable(0, EncodingUtils.newReal(-10.0, 10.0));
39      return solution;
40    }
41
42  }
```

MOEAFramework/book/chapter10/StdioSchafferProblem.java

When extending the `ExternalProblem` class, we need to specify in the constructor the program to execute. In this case, we are running a Python script, so we need to invoke the Python interpreter on our `schaffer.py` file (line 12). However, once the Java class is defined, we optimize it like any other problem:

```
1     new Executor()
2         .withAlgorithm("NSGAII")
3         .withProblemClass(StdioSchafferProblem.class)
4         .withMaxEvaluations(10000)
5         .run();
```

MOEAFramework/book/chapter10/ComparingCInterface.java

Lets take a quick look at how these interface options compare. Below we show the runtime for the Schaffer problem when it is implemented completely in Java, when we connect to a native library using JNA, and when we communicate via standard I/O.

```
Pure Java: 262
Via JNA: 377
Via Standard I/O: 532
```

Due to the simplicity of our problem, the pure Java option is fastest. For more complex problems, we would expect the C version to run faster. The JNA version adds an overhead of approximately 0.0115 milliseconds per evaluation, whereas the standard I/O version adds an overhead of approximately 0.027 milliseconds per evaluation.

10.5 A Note on Concurrency

We demonstrated two ways to interface with problems written in different languages: JNA and Standard I/O. One of the advantages of JNA is that it can be combined with multi-threaded evaluations. This requires that the native code you write in C or other language be reentrant. The term reentrant means that the function can be called multiple times, before prior calls have exited, and produce correct results. In order for a function to be reentrant, several rules should be followed[3]:

- The function must to access any static or global data that is not a constant.
- The function must not modify its own code
- The function must not call other non-reentrant functions

In our example C code, the `schaffer` function is entirely self-contained. It only reads the decision variables provided as an input to the function and returns the objectives. Thus, this function is reentrant.

If you have a problem that is non-reentrant, it is good practice to make Java's `evaluate` method synchronized. In you problem definition, you would define the `evaluate` method as follows:

```
1 public synchronized void evaluate(Solution solution) {
2   // body of method
3 }
```

The **`synchronized`** keyword ensures that the `evaluate` method can be invoked one at a time. Thus, the `synchronize` keyword will prevent someone from accidentally parallelizing your problem.

[3] `https://en.wikipedia.org/wiki/Reentrancy_(computing)`

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

[1] Parameter 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 then 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 it 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:

$$H = \binom{M + divisions - 1}{divisions} \tag{A.1}$$

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

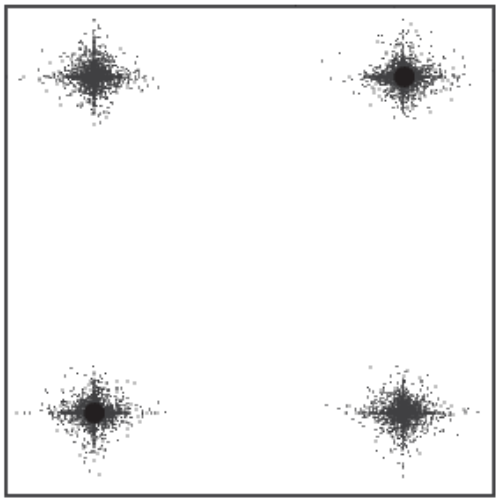
The following crossover and mutation operators are supported by the MOEA Framework.

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)

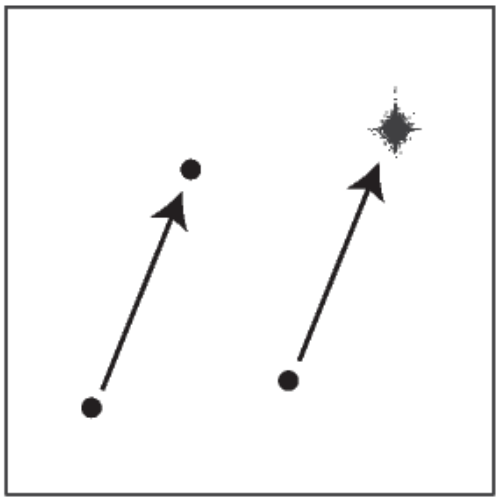
PM attempts to simulate the offspring distribution of binary-encoded bit-flip mutation on real-valued decision variables (Deb and Goyal, 1996). Similar to SBX, PM favors offspring nearer to the parent. It is recommended each decision variable is mutated with a probability of $1/N$, where N is the number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

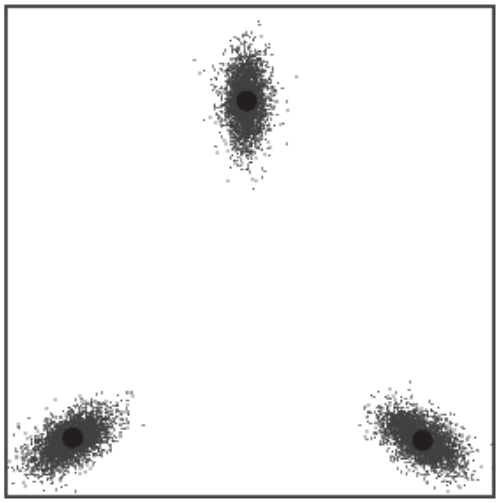


The scaling factor parameter adjusts the magnitude of the difference vector, allowing the user to decrease or increase the magnitude in relation to the actual difference between the individuals (Storn and Price, 1997). The crossover rate parameter controls the fraction of decision variables which are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

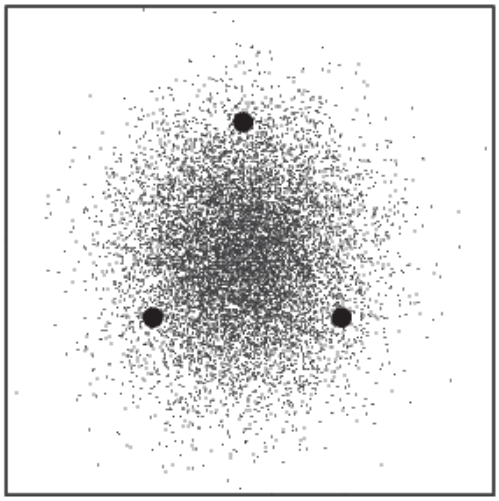
PCX is a multiparent operator, allowing a user-defined number of parents and offspring (Deb et al., 2002). Offspring are clustered around the parents, as depicted in the figure below.



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)

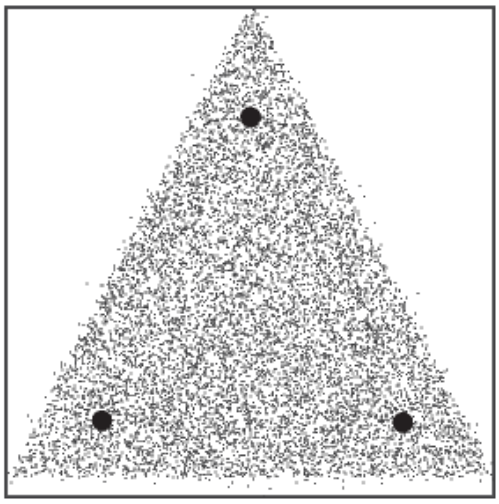
UNDX is a multiparent operator, allowing a user-defined number of parents and offspring (Kita et al., 1999; Deb et al., 2002). Offspring are centered around the centroid, forming a normal distribution whose shape is controlled by the positions of the parents, as depicted in the figure below.



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 satisifed, 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)

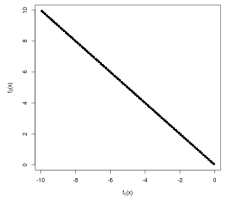
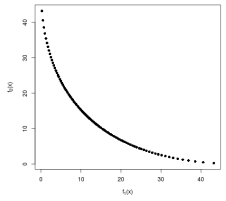
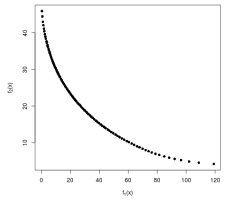
Crossover operator where each index is swapped with a specified probability.

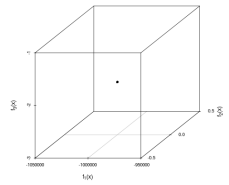
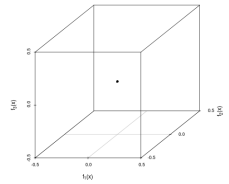
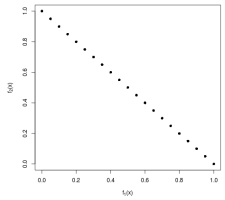
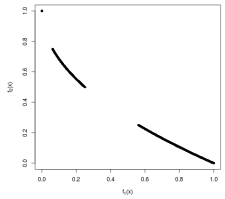
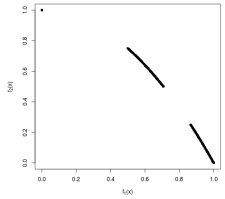
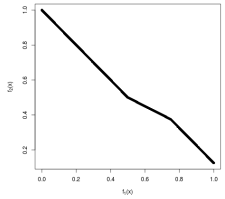
Parameters	Description	Default Value
ux.rate	The probability that uniform crossover is applied to produce offspring	1.0

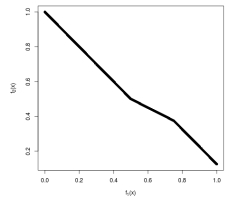
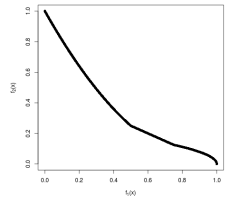
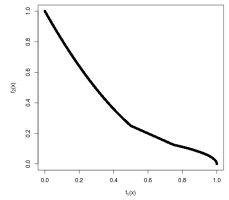
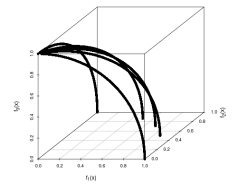
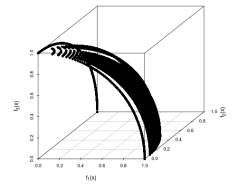
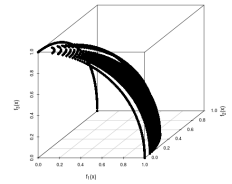
Appendix C

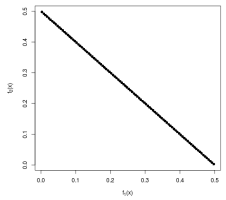
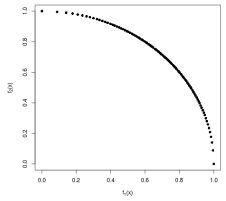
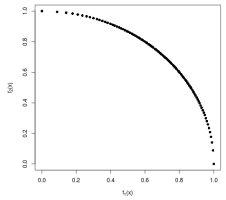
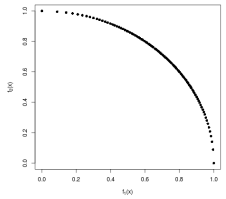
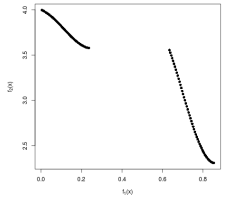
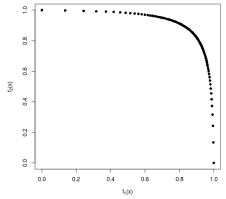
List of Problems

The following test problems are packaged with the MOEA Framework. Problems marked with † have maximized objectives. The MOEA Framework negates the values of maximized objectives. Note that while many problems have similar Pareto fronts, their underlying problem definitions and properties can differ greatly.

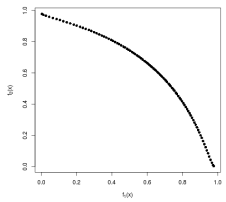
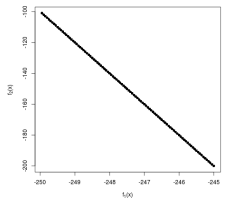
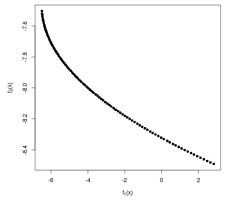
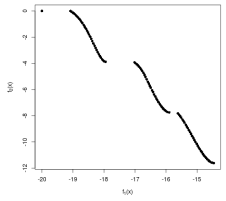
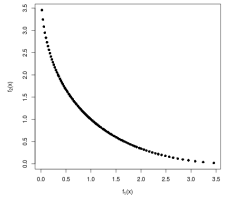
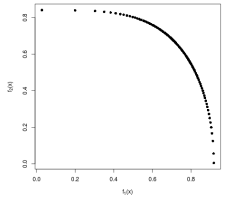
Problem	# of Vars	# of Objs	# of Constrs	Type	Pareto Front
Belegundu	2	2	2	Real	
Binh	2	2	0	Real	
Binh2	2	2	2	Real	

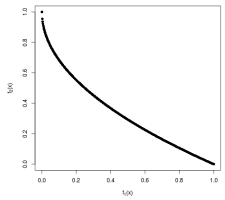
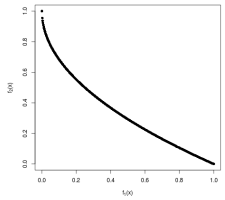
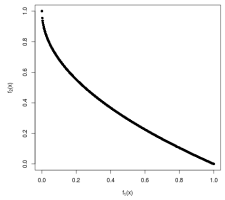
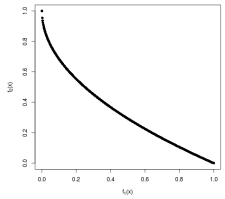
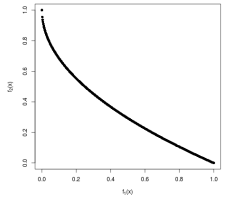
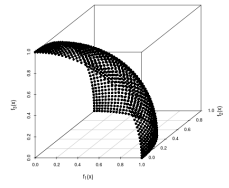
Problem	# of Vars	# of Objs	# of Constrs	Type	Pareto Front
Binh3	2	3	0	Real	
Binh4	2	3	2	Real	
CF1	10	2	1	Real	
CF2	10	2	1	Real	
CF3	10	2	1	Real	
CF4	10	2	1	Real	

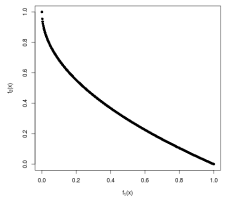
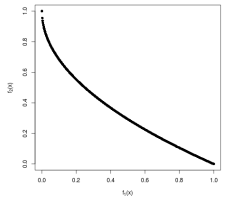
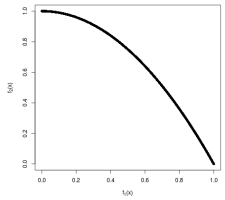
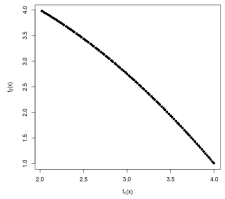
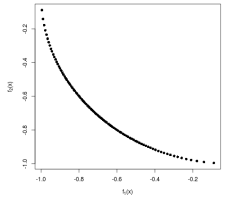
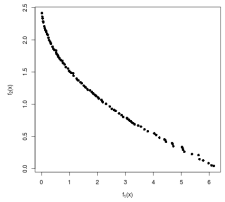
Problem	# of Vars	# of Objs	# of Constrs	Type	Pareto Front
CF5	10	2	1	Real	
CF6	10	2	2	Real	
CF7	10	2	2	Real	
CF8	10	3	1	Real	
CF9	10	3	1	Real	
CF10	10	3	1	Real	

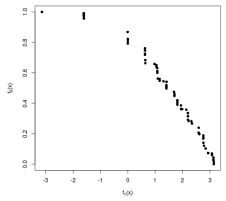
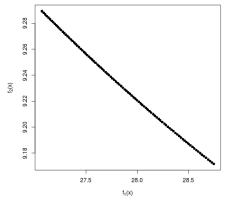
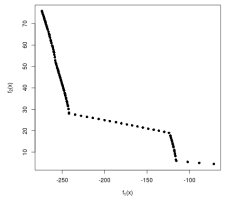
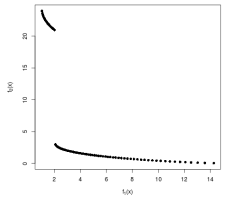
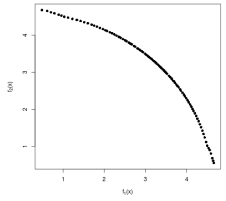
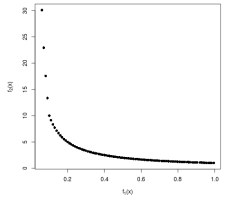
Problem	# of Vars	# of Objs	# of Constrs	Type	Pareto Front
DTLZ1_N[1]	$4 + N$	N	0	Real	
DTLZ2_N	$9 + N$	N	0	Real	
DTLZ3_N	$9 + N$	N	0	Real	
DTLZ4_N	$9 + N$	N	0	Real	
DTLZ7_N	$19 + N$	N	0	Real	
Fonseca	2	2	0	Real	

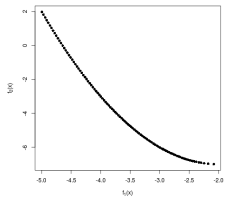
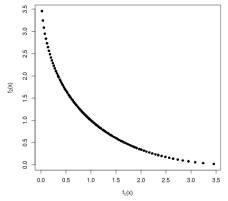
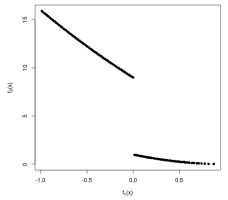
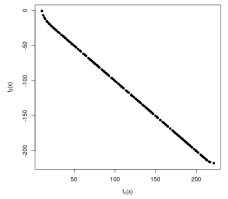
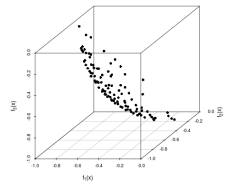
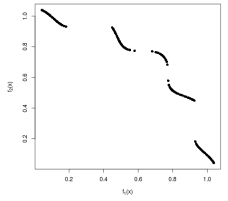
[1]DTLZ problems are scalable to any number of objectives. Replace N with the number of objectives.

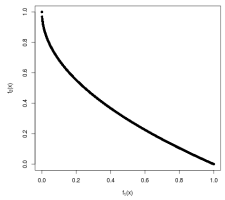
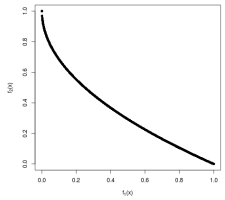
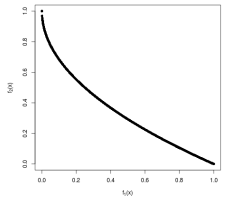
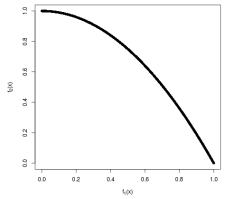
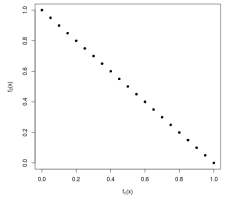
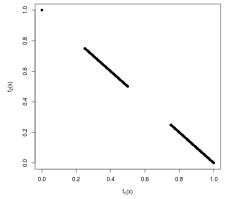
Problem	# of Vars	# of Objs	# of Constrs	Type	Pareto Front
Fonseca2	3	2	0	Real	
Jimenez †	2	2	4	Real	
Kita †	2	2	3	Real	
Kursawe	3	2	0	Real	
Laumanns	2	2	0	Real	
Lis	2	2	0	Real	

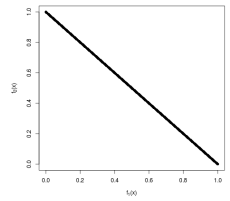
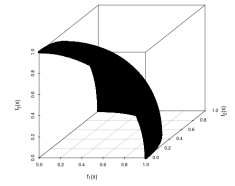
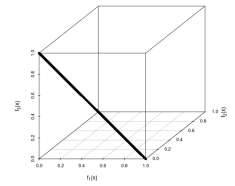
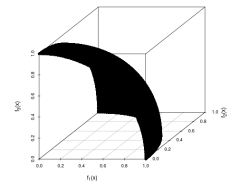
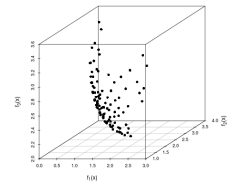
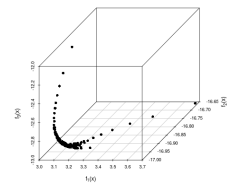
Problem	# of Vars	# of Objs	# of Constrs	Type	Pareto Front
LZ1	30	2	0	Real	
LZ2	30	2	0	Real	
LZ3	30	2	0	Real	
LZ4	30	2	0	Real	
LZ5	30	2	0	Real	
LZ6	10	3	0	Real	

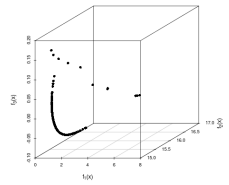
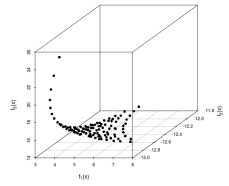
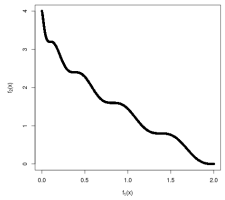
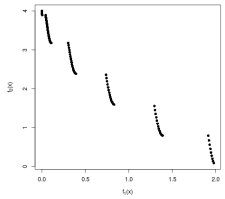
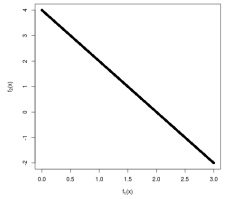
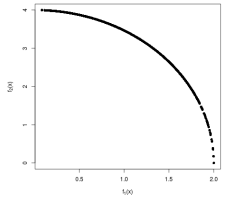
Problem	# of Vars	# of Objs	# of Constrs	Type	Pareto Front
LZ7	10	2	0	Real	
LZ8	10	2	0	Real	
LZ9	30	2	0	Real	
Murata	2	2	0	Real	
Obayashi †	2	2	1	Real	
OKA1	2	2	0	Real	

Problem	# of Vars	# of Objs	# of Constrs	Type	Pareto Front
OKA2	3	2	0	Real	
Osyczka	2	2	2	Real	
Osyczka2	6	2	6	Real	
Poloni †	2	2	0	Real	
Quagliarella	16	2	0	Real	
Rendon	2	2	0	Real	

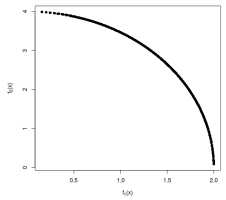
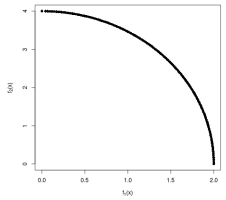
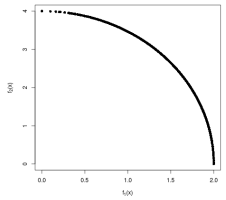
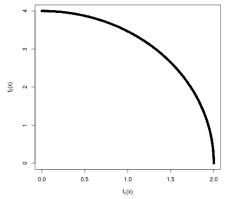
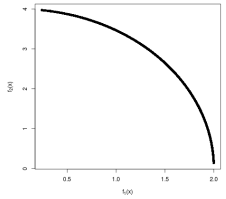
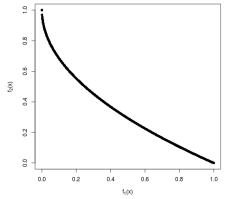
Problem	# of Vars	# of Objs	# of Constrs	Type	Pareto Front
Rendon2	2	2	0	Real	
Schaffer	1	2	0	Real	
Schaffer2	1	2	0	Real	
Srinivas	2	2	2	Real	
Tamaki †	3	3	1	Real	
Tanaka	2	2	2	Real	

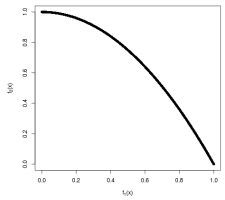
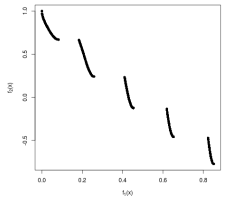
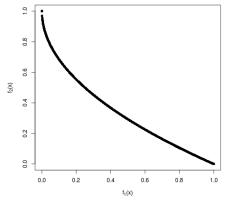
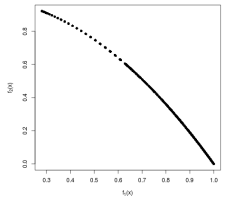
Problem	# of Vars	# of Objs	# of Constrs	Type	Pareto Front
UF1	30	2	0	Real	
UF2	30	2	0	Real	
UF3	30	2	0	Real	
UF4	30	2	0	Real	
UF5	30	2	0	Real	
UF6	30	2	0	Real	

Problem	# of Vars	# of Objs	# of Constrs	Type	Pareto Front
UF7	30	2	0	Real	
UF8	30	3	0	Real	
UF9	30	3	0	Real	
UF10	30	3	0	Real	
UF11	30	5	0	Real	
UF12	30	5	0	Real	
UF13	30	5	0	Real	
Viennet	2	3	0	Real	
Viennet2	2	3	0	Real	

Problem	# of Vars	# of Objs	# of Constrs	Type	Pareto Front
Viennet3	2	3	0	Real	
Viennet4	2	3	0	Real	
WFG1_N[2]	$9+N$	N	0	Real	
WFG2_N	$9+N$	N	0	Real	
WFG3_N	$9+N$	N	0	Real	
WFG4_N	$9+N$	N	0	Real	

[2]WFG problems are scalable to any number of objectives. Replace N with the number of objectives.

Problem	# of Vars	# of Objs	# of Constrs	Type	Pareto Front
WFG5_N	$9 + N$	N	0	Real	
WFG6_N	$9 + N$	N	0	Real	
WFG7_N	$9 + N$	N	0	Real	
WFG8_N	$9 + N$	N	0	Real	
WFG9_N	$9 + N$	N	0	Real	
ZDT1	30	2	0	Real	

Problem	# of Vars	# of Objs	# of Constrs	Type	Pareto Front
ZDT2	30	2	0	Real	
ZDT3	30	2	0	Real	
ZDT4	10	2	0	Real	
ZDT5	80	2	0	Binary	
ZDT6	10	2	0	Real	

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., Jaszkiewicz, A., and Jaszkiewicz, 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 Niched 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.