Use R!

Paulo Cortez

Modern Optimization with R

Springer

Use R!

Series Editors:
Robert Gentleman Kurt Hornik Giovanni Parmigiani

More information about this series at http://www.springer.com/series/6991

Use R!

Albert: Bayesian Computation with R (2nd ed. 2009)
Bivand/Pebesma/Gómez-Rubio: Applied Spatial Data Analysis with R (2nd ed. 2013)
Cook/Swayne: Interactive and Dynamic Graphics for Data Analysis: With R and GGobi
Hahne/Huber/Gentleman/Falcon: Bioconductor Case Studies
Paradis: Analysis of Phylogenetics and Evolution with R (2nd ed. 2012)
Pfaff: Analysis of Integrated and Cointegrated Time Series with R (2nd ed. 2008)
Sarkar: Lattice: Multivariate Data Visualization with R
Spector: Data Manipulation with R

Paulo Cortez

Modern Optimization with R

Springer

Paulo Cortez
Department of Information Systems
University of Minho
Guimarães, Portugal

ISSN 2197-5736 ISSN 2197-5744 (electronic)
ISBN 978-3-319-08262-2 ISBN 978-3-319-08263-9 (eBook)
DOI 10.1007/978-3-319-08263-9
Springer Cham Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014945630

Mathematics Subject Classification (2010): 68T20, 60-04, 62M10, 62M45, 65C05, 68T05, 97R40

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Currently, we are in the Information Age, where more organizational and individual activities and processes are based on Information Technology. We are also in a fast changing world. Due to several factors, such as globalization, technological improvements, and more recently the 2008 financial crisis, both organizations and individuals are pressured for improving their efficiency, reducing costs, and making better-informed decisions. This is where optimization methods, supported by computational tools, can play a key role.

Optimization is about minimizing or maximizing a goal (or goals) and it is useful in several domains, including Agriculture, Banking, Control, Engineering, Finance, Marketing, Production, and Science. Examples of real-world applications include the optimization of construction works, financial portfolios, marketing campaigns, and water management in agriculture, just to name a few.

Modern optimization, also known as metaheuristics, is related with general purpose solvers based on computational methods that use few domain knowledge, iteratively improving an initial solution (or population of solutions) to optimize a problem. Modern optimization is particularly useful for solving complex problems for which no specialized optimization algorithm has been developed, such as problems with discontinuities, dynamic changes, multiple objectives, or hard and soft restrictions, which are more difficult to be handled by classical methods.

Although modern optimization often incorporates random processes within their search engines, the overall optimization procedure tends to be much better than pure random (Monte Carlo) search. Several of these methods are naturally inspired. Examples of popular modern methods that are discussed in this book are simulated annealing, tabu search, genetic algorithms, genetic programming, NSGA II (multi-objective optimization), differential evolution, and particle swarm optimization.

R is a free, open source, and multiple platform tool (e.g., *Windows*, *Linux*, *MacOS*) that was specifically developed for statistical analysis. Currently, there is an increasing interest in using R to perform an intelligent data analysis. While it is difficult to know the real number of R users (e.g., it may range from 250,000 to 2 million), several estimates show a clear growth in the R popularity. In effect,

the R community is very active and new packages are being continuously created, with more than 5800 packages available, thus enhancing the tool capabilities. In particular, several of these packages implement modern optimization methods.

There are several books that discuss either modern optimization methods or the R tool. However, within the author's knowledge, there is no book that integrates both subjects and under a practical point of view, with several application R code examples that can be easily tested by the readers. Hence, the goal of this book is to gather in a single document (self-contained) the most relevant concepts related with modern optimization methods, showing how such concepts and methods can be addressed using the R tool.

This book is addressed for several target audience groups. Given that the R tool is free, this book can be easily adopted in several bachelor or master level courses in areas such as "Operations Research", "Decision Support", "Business Intelligence", "Soft Computing", or "Evolutionary Computation". Thus, this book should be appealing for bachelor's or master's students in Computer Science, Information Technology, or related areas (e.g., Engineering or Science). The book should also be of interest for two types of practitioners: R users interested in applying modern optimization methods and non R expert data analysts or optimization practitioners who want to test the R capabilities for optimizing real-world tasks.

How to Read This Book

This book is organized as follows:

Chapter 1 introduces the motivation for modern optimization methods and why the R tool should be used to explore such methods. Also, this chapter discusses key modern optimization topics, namely the representation of a solution, the evaluation function, constraints, and an overall view of modern optimization methods. This chapter ends with the description of the optimization tasks that are used for tutorial purposes in the next chapters.
Chapter 2 presents basic concepts about the R tool. This chapter is particularly addressed for non R experts, including the necessary knowledge that is required to understand and test the book examples. R experts should skip this chapter.
Chapter 3 is about how blind search can be implemented in R. This chapter details in particular three approaches: pure blind, grid, and Monte Carlo search.
Chapter 4 introduces local search methods, namely hill climbing, simulated annealing, and tabu search. Then, an example comparison between several local search methods is shown.
Chapter 5 presents population-based search methods, namely genetic and evolutionary algorithms, differential evolution, particle swarm optimization and

estimation of distribution algorithm. Then, two additional examples are discussed, showing a comparison between population-based methods and how to handle constraints.

Chapter 6 is dedicated to multi-objective optimization. This chapter first presents three demonstrative multi-objective tasks and then discusses three multi-objective optimization approaches: weighted-formula, lexicographic, and Pareto (e.g., NSGA-II algorithm).

Chapter 7 presents three real-world applications of previously discussed methods, namely traveling salesman problem, time series forecasting, and wine quality classification.

Each chapter starts with an introduction, followed by several chapter topic related sections and ends with an R command summary and exercise sections. Throughout the book, several examples of R code are shown. The code was run using a 64 bit R (version 3.0.2) under on a *MacOS* laptop. Nevertheless, these examples should be easily reproduced by the readers on other systems, possibly resulting in slight numerical (32 bit version) or graphical differences for deterministic examples. Also, given that a large portion of the discussed methods are stochastic, it is natural that different executions of the same code and under the same system will lead to small differences in the results.

It is particularly recommended that students should execute the R code and try to solve the proposed exercises. Examples of solutions are presented at the end of this book. All these code files and data examples are available at: http://www3.dsi.uminho.pt/pcortez/mor.

Production

Several contents of this book were taught by the author in the last 5 years in distinct course units of master and doctoral programs. At the master's level, it included the courses "Adaptive Business Intelligence" (Masters in Information Technology, University of Minho, Portugal) and "Business Intelligence" (Masters in Information Systems Management, Lisbon University Institute, Portugal). The doctoral course was "Adaptive Business Intelligence" (Doctoral Program in Computer Science, Universities of Minho, Aveiro and Porto, Portugal). Also, some material was lectured at a tutorial given in the European Simulation and Modelling Conference (ESM 2011), held at Guimarães.

This book was written in LaTeX, using the **texstudio** editor (http://texstudio.sourceforge.net) and its US English spell checker. Most figures were made in R, while some of the figures were designed using xfig (http://www.xfig.org), an open source vector graphical tool.

Guimarães, Portugal Paulo Cortez

Contents

List of Figures

List of Algorithms

Chapter 1
Introduction

1.1 Motivation

A vast number of real-world (often complex) tasks can be viewed as an optimization problem, where the goal is to minimize or maximize a given goal. In effect, optimization is quite useful in distinct application domains, such as Agriculture, Banking, Control, Engineering, Finance, Marketing, Production and Science. Moreover, due to advances in Information Technology, nowadays it is easy to store and process data. Since the 1970s, and following the Moore's law, the number of transistors in computer processors has doubled every 2 years, resulting in more computational power at a reasonable price. And it is estimated that the amount of data storage doubles at a higher rate. Furthermore, organizations and individual users are currently pressured to increase efficiency and reduce costs. Rather than taking decisions based on human experience and intuition, there is an increasing trend for adopting computational tools, based on optimization methods, to analyze real-world data in order to make better informed decisions.

Optimization is a core topic of the Operations Research field, which developed several classical techniques, such as linear programming (proposed in the 1940s) and branch and bound (developed in the 1960s) (Schrijver 1998). More recently, in the last decades, there has been an emergence of new optimization algorithms, often termed "modern optimization" (Michalewicz et al. 2006), "modern heuristics" (Michalewicz and Fogel 2004), or "metaheuristics" (Luke 2012). In this book, we adopt the first term, modern optimization, to describe these algorithms.

In contrast with classical methods, modern optimization methods are general purpose solvers, i.e., applicable to a wide range of distinct problems, since few domain knowledge is required. For instance, the optimization problem does not need to be differentiable, which is required by classical methods such as gradient descent. There are only two main issues that need to be specified by the user when adopting modern heuristic methods (Michalewicz et al. 2006): the representation of the solution, which defines the search space and its size; and the evaluation function, which defines how good a particular solution is, allowing to compare

P. Cortez, *Modern Optimization with R*, Use R!, DOI 10.1007/978-3-319-08263-9_1

different solutions. In particular, modern methods are useful for solving complex problems for which no specialized optimization algorithm has been developed (Luke 2012). For instance, problems with discontinuities, dynamic changes, multiple objectives or hard and soft restrictions, which are more difficult to be handled by classical methods (Michalewicz et al. 2006). Also in contrast with classical methods, modern optimization does not warranty that the optimal solution is always found. However, often modern methods achieve high quality solutions with a much more reasonable use of computational resources (e.g., memory and processing effort) (Michalewicz and Fogel 2004).

There is a vast number of successful real-world applications based on modern optimization methods. Examples studied by the author of this book include (among others): sitting guests at a wedding party (Rocha et al. 2001); time series forecasting (Cortez et al. 2004); optimization of data mining classification and regression models (Rocha et al. 2007); and improvement of quality of service levels in computer networks (Rocha et al. 2011).

1.2 Why R?

The R tool (R Core Team 2013) is an open source, high-level matrix programming language for statistical and data analysis. The tool runs on multiple platforms, including *Windows*, *MacOS*, *Linux*, *FreeBSD*, and other UNIX systems. R is an interpreted language, meaning that the user gets an immediate response of the tool, without the need of program compilation. The most common usage of R is under a console command interface, which often requires a higher learning curve from the user when compared with other more graphical user interface tools. However, after mastering the R environment, the user achieves a better understanding of what is being executed and higher control when compared with graphical interface based products.

The R base distribution includes a large variety of statistical techniques (e.g., distribution functions, statistical tests), which can be useful for inclusion in modern optimization methods and to analyze their results. Moreover, the tool is highly extensible by creating packages. The R community is very active and new packages are being continuously created, with more than 5,800 packages available at the Comprehensive R Archive Network (CRAN): http://www.r-project.org/. By installing these packages, users get access to new features, such as: data mining/machine learning algorithms; simulation and visualization techniques; and also modern optimization methods. New algorithms tend to be quickly implemented in R, thus this tool can be viewed as worldwide gateway for sharing computational algorithms (Cortez 2010). While it is difficult to know the real number of R users (e.g., it may range from 250,000 to 2 million), several estimates show a clear growth in the R popularity (Vance 2009; Muenchen 2013). A useful advantage of using R is that it is possible to execute quite distinct computational tasks under the same tool, such as

combining optimization with statistical analysis, visualization, simulation, and data mining (see Sect. 7.4 for an example that optimizes data mining models).

To facilitate the usage of packages, given that a large number is available, several R packages are organized into CRAN task views. The Optimization and Mathematical Programming view is located at http://cran.r-project.org/web/views/Optimization.html and includes more than 60 packages. In this book, we explore several of these CRAN view packages (and others) related with modern optimization methods.

1.3 Representation of a Solution

A major decision when using modern optimization methods is related with how to represent a possible solution (Michalewicz et al. 2006). Such decision sets the search space and its size, thus producing an impact on how new solutions are searched. To represent a solution, there are several possibilities. Binary, integer, character, real value and ordered vectors, matrices, trees and virtually any computer based representation form (e.g., computer program) can be used to encode solutions. A given representation might include a mix of different encodings (e.g., binary and real values). Also, a representation might be of fixed (e.g., fixed binary vectors) or of variable length (e.g., trees).

Historically, some of these representation types are attached with specific optimization methods. For instance, binary encodings are the basis of Genetic Algorithms (Holland 1975). Tabu search was designed to work on discrete spaces (Glover and Laguna 1998). Real-value encodings are adopted by several evolutionary algorithms (e.g., evolution strategy) (Bäck and Schwefel 1993), differential evolution, and particle swarms. Tree structures are optimized using genetic programming (Banzhaf et al. 1998). It should be noted that often these optimization methods can be adapted to other representations. For instance, a novel particle swarm was proposed for discrete optimization in Chen et al. (2010).

In what concerns this book, the representations adopted are restricted by the implementations available in the explored R tool packages, which mainly adopt binary or real values. Thus, there will be more focus on these type of representations. Nevertheless, Sect. 7.2 shows a ordered vector representation optimization example. More details about other representations, including their algorithmic adjustments, can be found in Luke (2012).

1.4 Evaluation Function

Another important decision for handling optimization tasks is the definition of the evaluation function, which should translate the desired goal (or goals) to be maximized or minimized. Such function allows to compare different solutions, by providing either a rank (ordinal evaluation function) or a quality measure score

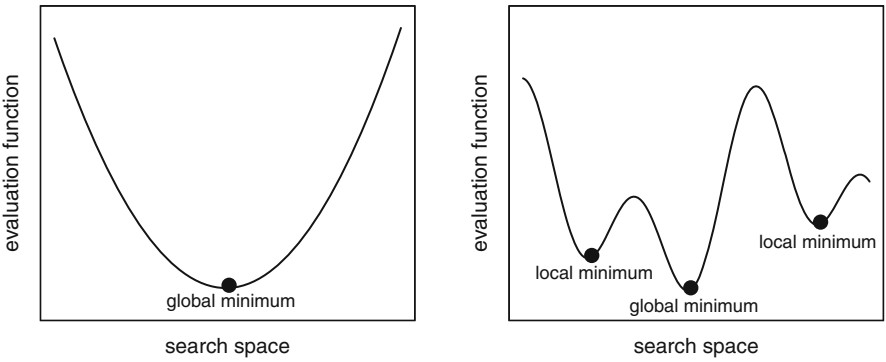

Fig. 1.1 Example of a convex (*left*) and non-convex (*right*) function landscapes

(numeric function) (Michalewicz et al. 2006). When considering numeric functions, the shape can be convex or non-convex, with several local minima/maxima (Fig. 1.1). Convex tasks are much easier to solve and there are specialized algorithms (e.g., least squares, linear programming) that are quite effective for handling such problems (Boyd and Vandenberghe 2004). However, many practical problems are non-convex, often including noisy or complex function landscapes, with discontinuities. Optimization problems can even be dynamic, changing through time. For all these complex problems, an interesting alternative is to use modern optimization algorithms that only search a subset of the search space but tend to achieve near optimum solutions in a reasonable time.

By default, some implementations of optimization methods only perform a minimization of a numerical evaluation function. In such cases, a simple approach is to transform the maximization function $\max(f(s))$ into the equivalent minimization task $-\min(f'(s))$, by adopting $f'(s) = -f(s)$, where s denotes the solution.

In several application fields (e.g., Control, Engineering, Finance) there are two or more goals that need to be optimized. Often, these goals conflict and trade-offs need to be set, since optimizing solutions under a single objective can lead to unacceptable outcomes in terms of the remaining goals. In such cases, a much better approach is to adopt a multi-objective optimization. In this book, we devote more attention to single response evaluation functions, since multi-objective optimization is discussed in a separated chapter (Chap. 6).

1.5 Constraints

There are two main types of constraints (Michalewicz 2008): hard and soft. Hard constraints cannot be violated and are due to factors such as laws or physical restrictions. Soft constraints are related with other (often non-priority) user goals, such as increasing production efficiency while reducing environmental costs.

Soft restrictions can be handled by adopting a multi-objective approach (Chap. 6), while hard constraints may originate infeasible solutions that need to

be treated by the optimization procedure. To deal with infeasible solutions, several methods can be adopted (Michalewicz et al. 2006): death-penalty, penalty-weights, repair and only generate feasible solutions.

Death-penalty is a simple method, which involves assigning a very large penalty value, such that infeasible solutions are quickly discarded by the search (see Sect. 4.4 for an example). However, this method is not very efficient and often puts the search engine effort in discarding solutions and not finding the optimum value. Penalty-weights is a better solution, also easy to implement. For example, quite often, the shape of an evaluation function can be set within the form $f(s) = Objective(s) - Penalty(s)$ (Rocha et al. 2001). For instance, for a given business, a possible evaluation function could be $f = w_1 \times Profit(s) - w_2 \times Cost(s)$. The main problem with penalty-weights is that often it is difficult to find the ideal weights, in particular when several constraints are involved. The repair approach transforms an infeasible solution into a feasible one. Often, this is achieved by using domain dependent information (such as shown in Sect. 5.2) or by applying a local search (e.g., looking for a feasible solution in the solution space neighborhood, see Sect. 5.7). Finally, the approaches that only generate feasible solutions are based in decoders and special operators. Decoders work only in a feasible search space, by adopting an indirectly representation, while special operators use domain knowledge to create new solutions from previous ones.

1.6 Optimization Methods

There are different dimensions that can be used to classify optimization methods. Three factors of analysis are adopted in this book: the type of guided search; the search is deterministic or stochastic based; and if the method is inspired by physical or biological processes.

The type of search can be blind or guided. The former assumes the exhaustion of all alternatives for finding the optimum solution, while the latter uses previous searches to guide current search. Modern methods use a guided search, which often is subdivided into two main categories: local, which searches within the neighborhood of an initial solution, and global search, which uses a population of solutions. In most practical problems, with high-dimensional search spaces, pure blind search is not feasible, requiring too much computational effort. Local (or single-state) search presents in general a much faster convergence, when compared with global search methods. However, if the evaluation landscape is too noisy or complex, with several local minima (e.g., right of Fig. 1.1), local methods can easily get stuck. In such cases, multiple runs, with different initial solutions, can be executed to improve convergence. Although population based algorithms tend to require more computation than local methods, they perform a simultaneous search in distinct regions of the search space, thus working better as global optimization methods.

The distinct search types can even be combined. For instance, a two-phase search can be set, where a global method is employed at a first step, to quickly identify

interesting search space regions, and then, as the second step, the best solutions are improved by employing a local search method. Another alternative is to perform a tight integration of both approaches, such as under a Lamarckian evolution or Baldwin effect (Rocha et al. 2000). In both cases, within each cycle of the population based method, each new solution is used as the initial point of a local search and the evaluation function is computed over the improved solution. Lamarckian evolution replaces the population original solution by its improved value (Sect. 7.2 presents a Lamarckian evolution example), while the Baldwinian strategy keeps the original point (as set by the population based method).

Several modern methods employ some degree of randomness, thus belonging to the family of stochastic optimization methods, such as simulated annealing and genetic algorithms (Luke 2012). Also, several of these methods are naturally inspired (e.g., genetic algorithms, particle swarm optimization) (Holland 1975; Eberhart et al. 2001). Figure 1.2 shows the full taxonomy of the optimization methods presented in this book (with respective R packages).

The distinct modern optimization methods share some common features. Algorithm 1 shows (in pseudo-code) a generic modern optimization method that is applicable to all methods discussed in this book. This global algorithm receives two inputs, the evaluation function (f) and a set of control parameters (C), which includes not only the method's internal parameters (e.g., initial temperature, population size) but it is also related with the representation of the solution (e.g., lower and upper bounds, representation type, and length). In all modern optimization methods, there is an initial setup followed by a main loop cycle that ends once a given termination criterian is met. Distinct criteria can be adopted (or even combined):

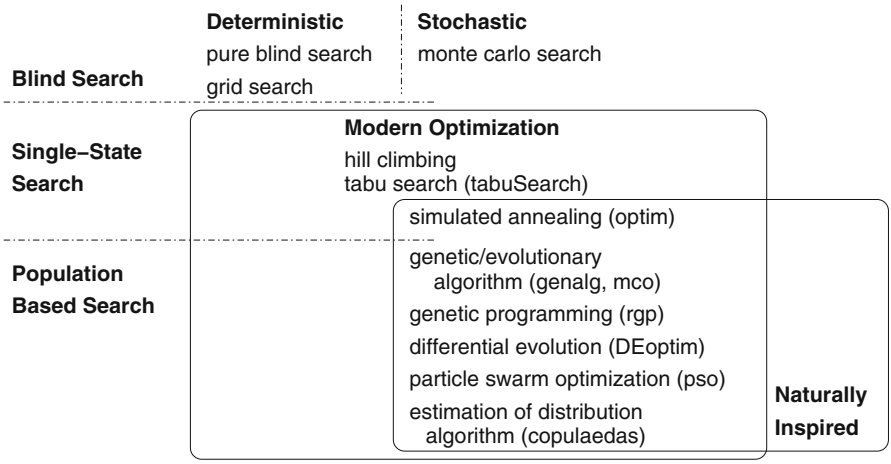

Fig. 1.2 Classification of the optimization methods presented in this book (related R packages are in *brackets*)

Algorithm 1 Generic modern optimization method

Inputs: f, C ▷ f is the evaluation function, C includes control parameters
$S \leftarrow initialization(C)$ ▷ S is a solution or population
$i \leftarrow 0$ ▷ i is the number of iterations of the method
while not $termination\_criteria(S, f, C, i)$ **do**
  $S' \leftarrow change(S, f, C, i)$ ▷ new solution or population
  $B \leftarrow best(S, S', f, C, i)$ ▷ store the best solution
  $S \leftarrow select(S, S', f, C, i)$ ▷ solution or population for next iteration
  $i \leftarrow i + 1$
end while
Output: B ▷ the best solution

- maximum computational measures—such as number of iterations, evaluation function calculations, time elapsed;
- target measures—such as to stop if best value is higher or equal to a given threshold;
- convergence measures—such as number of iterations without any improvement in the solution or average enhancement achieved in the last iterations; and
- distribution measures—such as measuring how spread are the last tested solutions in the search space and stop if the dispersion is smaller than a threshold.

What distinguishes the methods is related with two main aspects. First, if in each iteration there is a single-state (local based) or a population of solutions. Second, the way new solutions are created (function *change*) and used to guide in the search (function *select*). In the generic pseudo-code, the number of iterations (i) is also included as an input of the *change*, *best*, and *select* functions because it is assumed that the behavior of these functions can be dynamic, changing as the search method evolves.

1.7 Demonstrative Problems

This section includes examples of simple optimization tasks that were selected mainly from a tutorial perspective, where the aim is to easily show the capabilities of the optimization methods. The selected demonstrative problems include 2 binary, 1 integer and 2 real value tasks. More optimization tasks are presented and explored in Chaps. 6 (multi-optimization tasks) and 7 (real-world tasks).

The binary problems are termed here **sum of bits** and **max sin**. The former, also known as max ones, is a simple binary maximization "toy" problem, defined as (Luke 2012):

$$f_{\text{sum of bits}}(\mathbf{x}) = \sum_{i=1}^{D} x_i \tag{1.1}$$

where $\mathbf{x} = (x_1, \ldots, x_D)$ is a boolean vector ($x_i \in \{0, 1\}$) with a dimension (or length) of D. The latter problem is another simple binary task, where the goal is to maximize (Eberhart and Shi 2011):

$$\begin{aligned} x' &= \sum_{i=1}^{D} x_i 2^{i-1} \\ f_{\text{max sin}}(\mathbf{x}) &= \sin\left(\pi \frac{x'}{2^D}\right) \end{aligned} \tag{1.2}$$

where x' is the integer representation of x.

A visualization for both binary problems is given in top of Fig. 1.3, assuming a dimension of $D = 8$. In the top left graph, x-axis denotes x', the integer representation of $\mathbf{x}$ for **sum of bits**. In the example, the optimum solution for the **sum of bits** is $\mathbf{x} = (1, 1, 1, 1, 1, 1, 1, 1)$ ($f(\mathbf{x}) = 8$), while the best solution for **max sin** is $\mathbf{x} = (1, 0, 0, 0, 0, 0, 0, 0)$, $x' = 128$ ($f(\mathbf{x}) = 1$).

The **bag prices** is an integer optimization task (proposed in this book), that mimics the decision of setting of prices for items produced in a bag factory. The factory produces up to five ($D = 5$) different bags, with a unit cost of $\mathbf{u} = (\$30, \$25, \$20, \$15, \$10)$, where u_i is the cost for manufacturing product i. The production cost is $cost(x_i) = 100 + u_i \times sales(x_i)$ for the i-th bag type. The number of expected sales, which is what the factory will produce, is dependent on the product selling price ($\mathbf{x}$) and marketing effort ($\mathbf{m}$), according to the formula $sales(x_i) = round((1000/\ln(x_i + 200) - 141) \times m_i)$, where $round$ is the ordinary rounding function and $\mathbf{m} = (2.0, 1.75, 1.5, 1.25, 1.0)$. The manager at the factory needs to decide the selling prices for each bag (x_i, in \$), within the range \$1 to \$1,000, in order to maximize the expected profit related with the next production cycle:

$$f_{\text{bag prices}} = \sum_{i=1}^{D} x_i \times sales(x_i) - cost(x_i) \tag{1.3}$$

The middle left graph of Fig. 1.3 plots the full search space for the first item of **bag prices** ($D = 1$), while the middle right plot shows a zoom near the optimum solution. As shown by the graphs, the profit function follows in general a global convex shape. However, close to the optimum ($x_1 = 414$, $f(x_1) = 11420$) point there are several local minima, under a "saw" shape that is more difficult to optimize. As shown in Chap. 3, the optimum solution for five different bags ($D = 5$) is $\mathbf{x} = c(414, 404, 408, 413, 395)$, which corresponds to an estimated profit of \$43,899.

Turning to the real value tasks, two popular benchmarks are adopted, namely **sphere** and **rastrigin** (Tang et al. 2009), which are defined by:

$$\begin{aligned} f_{\text{sphere}}(\mathbf{x}) &= \sum_{i=1}^{D} x_i^2 \\ f_{\text{rastrigin}}(\mathbf{x}) &= \sum_{i=1}^{D} (x_i^2 - 10\cos 2\pi x_i + 10) \end{aligned} \tag{1.4}$$

where $\mathbf{x} = (x_1, \ldots, x_D)$ is a real value vector ($x_i \in \Re$). For both tasks, the goal is to find the minimum value, which is the origin point (e.g., if $D = 4$ and $\mathbf{x} = (0, 0, 0, 0)$, then $f(\mathbf{x}) = 0$). The **sphere** task is more simpler, mainly used for demonstration purposes, while the **rastrigin** is much more difficult multi-modal

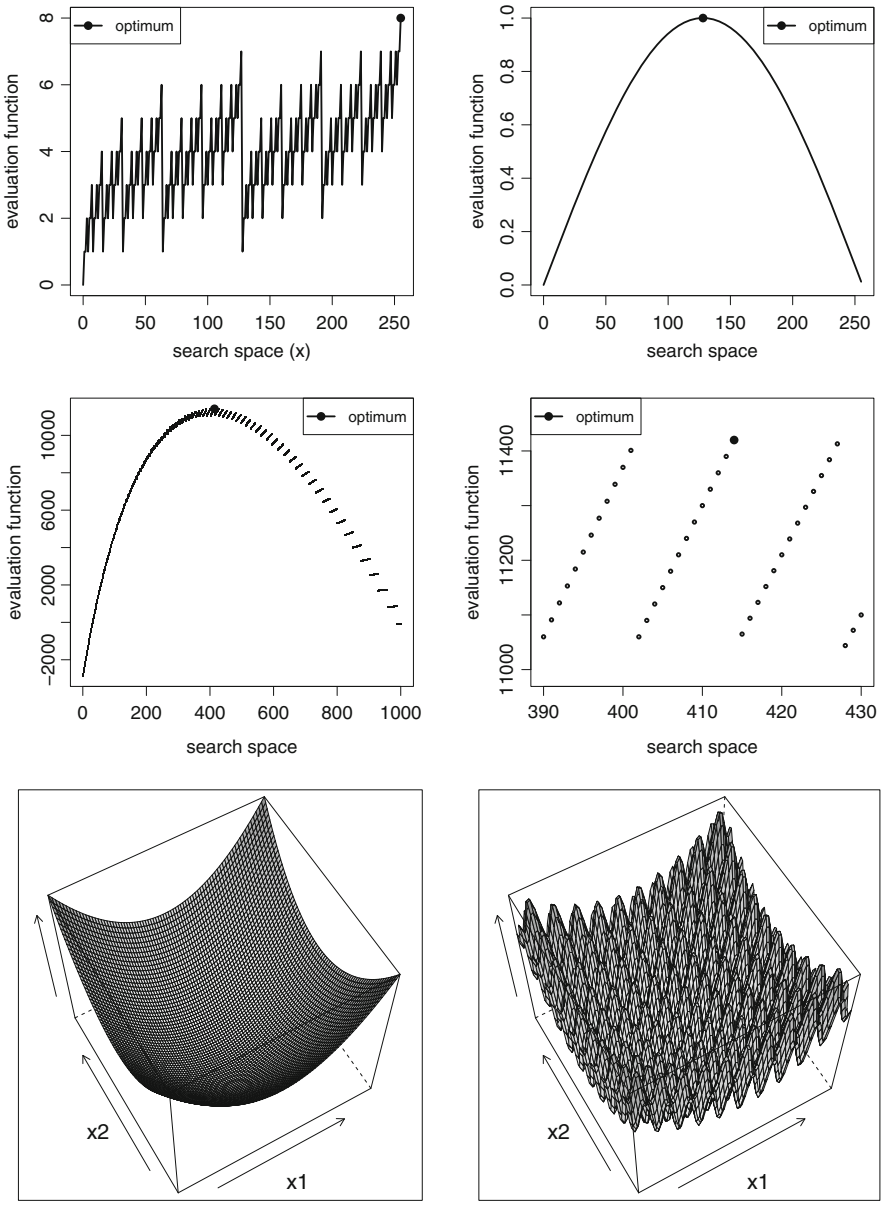

Fig. 1.3 Example of the binary (**sum of bits**—*top left*; **max sin**—*top right*), integer (**bag prices**—*middle*) and real value (**sphere**—*bottom left*; **rastrigin**—*bottom right*) task landscapes

problem, given that the number of local minima grows exponentially with the increase of dimensionality (D). The differences between **sphere** and **rastrigin** are clearly shown in the two graphs at the bottom of Fig. 1.3.

Chapter 2
R Basics

2.1 Introduction

As explained in the preface of this book, the goal of this chapter is to briefly present the most relevant R tool aspects that need to be learned by non-experts in order to understand the examples discussed in this book. For a more detailed introduction to the tool, please consult (Paradis 2002; Zuur et al. 2009; Venables et al. 2013).

R is language and a computational environment for statistical analysis that was created by Ihaka and Gentleman in 1991 and that was influenced by the S and Scheme languages (Ihaka and Gentleman 1996). R uses a high-level language, based in objects, that is flexible and extensible (e.g., by the development of packages) and allows a natural integration of statistics, graphics, and programming. The R system offers an integrated suite with a large and coherent collection of tools for data manipulation, analysis, and graphical display.

The tool is freely distributed under a GNU general public license and can be easily installed from the official web page (http://www.r-project.org), with several binary versions available for most commonly adopted operating systems (e.g., *Windows*, *MacOS*). R can be run under the console (e.g., common in *Linux* systems) or using Graphical User Interface (GUI) applications (e.g., R for Mac OS X GUI). There are also several independent Integrated Development Environments (IDE) for R, such as RStudio (http://www.rstudio.com/) and Tinn-R (http://nbcgib.uesc.br/lec/software/editores/tinn-r/en). Figure 2.1 shows an example of the R console and GUI applications for the *Mac OS* system.

R works mostly under a console interface (Fig. 2.1), where commands are typed after the prompt (`>`). An extensive help system is included. There are two console alternatives to get help on a particular function. For instance, `help(barplot)` or `?barplot` returns the help for the `barplot` function. It is also possible to search for a text expression, such as `help.search("linear models")` or `??"linear models"`. For each function, the help system includes often a short description, the function main arguments, details, return value, references, and examples. The last item is quite useful for an immediate user perception of

P. Cortez, *Modern Optimization with R*, Use R!, DOI 10.1007/978-3-319-08263-9_2

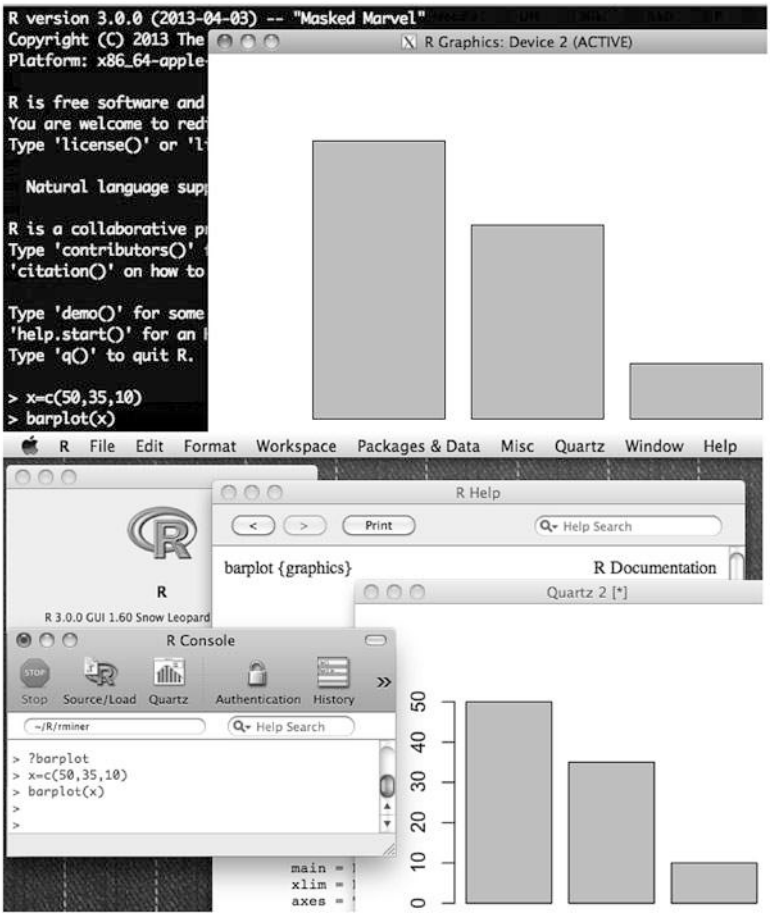

Fig. 2.1 Example of the R console (*top*) and GUI 3.0 versions (*bottom*) for *Mac OS X*

the function capabilities and can be accessed directly in the console, such as by using `> example(barplot)`. Some demonstrations of interesting R scripts are available with the command `demo`, such as `> demo(graphics)` or `> demo()`.

The tool capabilities can be extended by installing packages. The full list of packages is available at CRAN (http://cran.r-project.org). Packages can be installed on the console, using the command `install.packages`, or GUI system, using the application menus. After installing a package, the respective functions and help is only available if the package is loaded using the `library` command. For example, the following sequence shows the commands needed to install the particle swarm package and get the help on its main function:

```
> install.packages("pso")
> library(pso)
> ?pso
```

A good way to get help on a particular *package* is to use `> help(package=` *package*`)`.

R instructions can be separated using the `;` or newline character. Everything that appears after the # character in a line is considered a comment. R commands can be introduced directly in the console or edited in a script source file (e.g., using RStudio). The common adopted extension for R files is `.R` and these can be loaded with the `source` command. For example, `source("code.R")` executes the file `code.R`.

By default, the R system searches for files (e.g., code, data) in the current working directory, unless an explicit path is defined. The definition of such path is operating system dependent. Examples of paths are: `"~/directory/"` (*Unix* or *Mac OS*, where ~ means the user's home directory); `"C:/Documents and Settings/User/directory/"` (*Windows*); and `"../directory"` (relative path should work in all systems). The working directory can be accessed and changed using the R GUI (e.g., `"~/R/rminer/"` at the bottom of Fig. 2.1) or the console, under the `getwd()` and `setwd()` functions, such as `> setwd("../directory")`.

There is a vast amount of R features (e.g., functions, operators), either in the base version or its contributing packages. In effect, the number of features offered in R is such large that often users face the dilemma between spending time coding a procedure or searching if such procedure has already been implemented. In what concerns this book, the next sections describe some relevant R features that are required to understand the remaining chapters of this book. Explanation is given mostly based on examples, given that the full description of each R operator or function can be obtained using the help system, such as `help(":")` or `help("sort")`.

2.2 Basic Objects and Functions

R uses objects to store items, such as data and functions. The = (or `<-`[1]) operator can be used to assign an object to a variable. The class of an object is automatically assumed, with atomic objects including the `logical` (i.e., `FALSE`, `TRUE`), `character` (e.g., `"day"`), `integer` (e.g., `1L`), and `numeric` (e.g., `0.2`, `1.2e-3`) types. The type of any R object can be accessed by using the function `class`. There are also several constants defined, such as: `pi`—π; `Inf`—infinity; `NaN`—not a number; `NA`—missing value; and `NULL`—empty or null object.

[1] Although the `<-` operator is commonly used in R, this book adopts the smaller = character.

The R system includes an extensive list of functions and operators that can be applied over a wide range of object types, such as:

- `class()`—get type of object;
- `summary()`—show a summary of the object;
- `print()`—shows the object;
- `plot()`—plots the object; and
- `is.na()`, `is.nan()`, `is.null()`—check if object is `NA`, `NaN`, or `NULL`.

Another useful function is `ls()`, which lists all objects defined by the user. An example of a simple R session is shown here:

```
> s="hello world"
> print(class(s))    # character
[1] "character"
> print(s)           # "hello world"
[1] "hello world"
> x=1.1
> print(class(x))    # numeric
[1] "numeric"
> print(summary(x)) # summary of x
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    1.1     1.1     1.1     1.1     1.1     1.1
> plot(x)
> print(x)           # 1.1
[1] 1.1
> print(pi)          # 3.141593
[1] 3.141593
> print(sqrt(-1))    # NaN
[1] NaN
Warning message:
In sqrt(-1) : NaNs produced
> print(1/0)         # Inf
[1] Inf
```

There are also several containers, such as: `vector`, `factor`, `ordered`, `matrix`, `array`, `data.frame`, and `list`. Vectors, matrices, and arrays use an indexed notation (`[]`) to store and access several atoms. A factor is a special vector that contains only discrete values from a domain set, while ordered is a special factor whose domain levels have an order. A data frame is a special matrix where the columns (made of vectors or factors) have names. Finally, a list is a collection of distinct objects (called components). A list can include indexed elements (of any type, including containers) under the `[[]]` notation.

There is a large number of functions and operators that can be used to manipulate R objects (including containers). Some useful functions are:

- `c()`—concatenate several elements;
- `seq()`—create a regular sequence;
- `sample()`, `runif()`, `rnorm()`—create random samples;
- `set.seed()`—set the random generation seed number;
- `str()`—show the internal structure of the object;

- `length()`, `sum()`, `mean()`, `median()`, `min()`, `max()`—computes the length, sum, average, median, minimum or maximum of all elements of the object;
- `names()`—get and set the names of an object;
- `sort()`—sorts a vector or factor;
- `which()`—returns the indexes of an object that follow a logical condition;
- `which.min()`, `which.max()`—returns the index of the minimum or maximum value;
- `sqrt()`—square root of a number; and
- `sin()`, `cos()`, `tan()`—trigonometric functions.

Examples of operators are:

- `$`—get and set a list component;
- `:`—generate regular sequences;
- `+`, `-`, `*`, `/`—simple arithmetic operators;
- `^` (or `**`)—power operator; and
- `%%`—rest of an integer division.

R also offers vast graphical based features. Examples of useful related functions are:

- `plot`—generic plotting function;
- `barplot`—bar plots;
- `pie`—pie charts;
- `hist`—histograms;
- `boxplot`—box-and-whisker plot; and
- `wireframe`—3D scatter plot (package `lattice`).

A graph can be sent to screen (default) or redirected to a device (e.g., PDF file). The description of all these graphical features is out of scope of this book, but a very interesting sample of R based graphs and code can be found at the R Graph Gallery https://www.facebook.com/pages/R-Graph-Gallery/169231589826661.

An example of an R session that uses factors and vectors is presented next (Fig. 2.2 shows the graphs created by such code):

```
> f=factor(c("a","a","b","b","c")); print(f) # create factor
[1] a a b b c
Levels: a b c
> f[1]="c"; print(f)                          # change factor
[1] c a b b c
Levels: a b c
> print(levels(f))          # show domain levels: "a" "b" "c"
[1] "a" "b" "c"
> print(summary(f))                    # show a summary of y
a b c
1 2 2
> plot(f)                                      # show y barplot
> x=c(1.1,2.3,-1,4,2e-2) # creates vector x
> summary(x)             # show summary of x
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -1.000   0.020   1.100   1.284   2.300   4.000
```

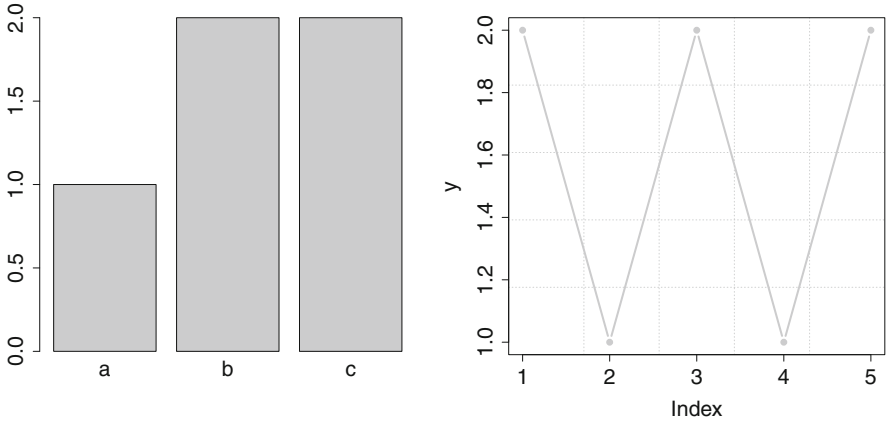

Fig. 2.2 Examples of a plot of a factor (*left*) and a vector (*right*) in R

```
> print(x)                # show x
[1]  1.10  2.30 -1.00  4.00  0.02
> str(x)                  # show x structure
 num [1:5] 1.1 2.3 -1 4 0.02
> length(x)               # show the length of x
[1] 5
> x[2]                    # show second element of x
[1] 2.3
> x[2:3]=(2:3)*1.1        # change 2nd and 3rd elements
> x[length(x)]=5          # change last element to 5
> print(x)                # show x
[1] 1.1 2.2 3.3 4.0 5.0
> print(x>3)              # show which x elements > 3
[1] FALSE FALSE  TRUE  TRUE  TRUE
> print(which(x>3))       # show indexes of x>3 condition
[1] 3 4 5
> names(x)=c("1st","2nd","3rd","4th","5th") # change names of x
> print(x)                # show x
1st 2nd 3rd 4th 5th
1.1 2.2 3.3 4.0 5.0
> print(mean(x))          # show the average of x
[1] 3.12
> print(summary(x))       # show a summary of x
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.10    2.20    3.30    3.12    4.00    5.00
> y=vector(length=5); print(y)      # FALSE, FALSE, ..., FALSE
[1] FALSE FALSE FALSE FALSE FALSE
> y[]=1; print(y)                   # all elements set to 1
[1] 1 1 1 1 1
> y[c(1,3,5)]=2; print(y)           # 2,1,2,1,2
[1] 2 1 2 1 2
> # fancier plot of y:
> plot(y,type="b",lwd=3,col="gray",pch=19,panel.first=grid(5,5))
```

Typically, R functions can receive several arguments, allowing to detail the effect of the function (e.g. `help(plot)`). To facilitate the use of functions, most of the parameters have default values (which are available in the help system). For instance, replacing the last line of above code with `plot(y)` will also work, although with a simpler effect.

Another R example for manipulating vectors is shown here:

```
> x=sample(1:10,5,replace=TRUE)  # 5 random samples from 1 to 10
                                 #   with replacement
> print(x)                       # show x
[1] 10  5  5  1  2
> print(min(x))                  # show min of x
[1] 1
> print(which.min(x))     # show index of x that contains min
[1] 4
> print(sort(x,decreasing=TRUE)) # show x in decreasing order
[1] 10  5  5  2  1
> y=seq(0,20,by=2); print(y)     # y = 0, 2, ..., 20
 [1]  0  2  4  6  8 10 12 14 16 18 20
> print(y[x])                    # show y[x]
[1] 18  8  8  0  2
> print(y[-x])              # - means indexes excluded from y
[1]  4  6 10 12 14 16 20
> x=runif(5,0.0,10.0);print(x) # 5 uniform samples from 0 to 10
[1] 1.011359 1.454996 6.430331 9.395036 6.192061
> y=rnorm(5,10.0,1.0);print(y) # normal samples (mean 10, std 1)
[1] 10.601637  9.231792  9.548483  9.883687  9.591727
> t.test(x,y)                  # t-student paired test

   Welch Two Sample t-test

data:  x and y
t = -3.015, df = 4.168, p-value = 0.03733
alternative hypothesis: true difference in means is not equal to
     0
95 percent confidence interval:
 -9.2932638 -0.4561531
sample estimates:
mean of x mean of y
 4.896757  9.771465
```

The last R function (`t.test()`) checks if the differences between the x and y averages are statistically significant. Other statistical tests are easily available in R, such as `wilcox.test` (Wilcoxon) or `chisq.test` (Pearson's chi-squared). In the above example, x is created using a uniform distribution $\mathscr{U}(0, 10)$, while y is created using a normal one, i.e., $\mathscr{N}(10, 1)$. Given that R is a strong statistical tool, there is an extensive list of distribution functions (e.g., binomial, Poisson), which can be accessed using `help("Distributions")`.

The next R session is about `matrix` and `data.frame` objects:

```
> m=matrix(ncol=3,nrow=2); m[,]=0; print(m) # 3x2 matrix
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
> m[1,]=1:3; print(m)                       # change 1st row
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    0    0    0
> m[,3]=1:2; print(m)                       # change 3rd column
     [,1] [,2] [,3]
[1,]    1    2    1
[2,]    0    0    2
> m[2,1]=3; print(m)                        # change m[2,1]
     [,1] [,2] [,3]
[1,]    1    2    1
[2,]    3    0    2
> print(nrow(m))                            # number of rows
[1] 2
> print(ncol(m))                            # number of columns
[1] 3
> m[nrow(m),ncol(m)]=5; print(m)            # change last
    element
     [,1] [,2] [,3]
[1,]    1    2    1
[2,]    3    0    5
> m[nrow(m)-1,ncol(m)-1]=4; print(m)        # change m[1,2]
     [,1] [,2] [,3]
[1,]    1    4    1
[2,]    3    0    5
> print(max(m))                             # show maximum of m
[1] 5
> m=sqrt(m); print(m)                       # change m
         [,1] [,2]     [,3]
[1,] 1.000000    2 1.000000
[2,] 1.732051    0 2.236068
> m[1,]=c(1,1,2013); m[2,]=c(2,2,2013)      # change m
> d=data.frame(m)                           # create data.frame
> names(d)=c("day","month","year")          # change names
> d[1,]=c(2,1,2013); print(d)               # change 1st row
  day month year
1   2     1 2013
2   2     2 2013
> d$day[2]=3; print(d)                      # change d[1,2]
  day month year
1   2     1 2013
2   3     2 2013
> d=rbind(d,c(4,3,2014)); print(d)          # add row to d
  day month year
1   2     1 2013
2   3     2 2013
3   4     3 2014
> # change 2nd column of d to factor, same as d[,2]=factor(...
```

```
> d$month=factor(c("Jan","Feb","Mar"))
> print(summary(d))                          # summary of d
      day      month        year
 Min.   :2.0   Feb:1   Min.   :2013
 1st Qu.:2.5   Jan:1   1st Qu.:2013
 Median :3.0   Mar:1   Median :2013
 Mean   :3.0           Mean   :2013
 3rd Qu.:3.5           3rd Qu.:2014
 Max.   :4.0           Max.   :2014
```

The last section example is related with lists:

```
> l=list(a="hello",b=1:3) # list with 2 components
> print(summary(l))       # summary of l
  Length Class  Mode
a 1      -none- character
b 3      -none- numeric
> print(l)                # show l
$a
[1] "hello"

$b
[1] 1 2 3

> l$b=l$b^2+1;print(l)      # change b to (b*b)+1
$a
[1] "hello"

$b
[1]  2  5 10

> v=vector("list",3)  # vector list
> v[[1]]=1:3          # change 1st element of v
> v[[2]]=2            # change 2nd element of v
> v[[3]]=l            # change 3rd element of v
> print(v)            # show v
[[1]]
[1] 1 2 3

[[2]]
[1] 2

[[3]]
[[3]]$a
[1] "hello"

[[3]]$b
[1] 2 3 4

> print(length(v))    # length of v
[1] 3
```

2.3 Controlling Execution and Writing Functions

The R language contains a set of control-flow constructs that are quite similar to other imperative languages (e.g., C, Java). Such constructs can be accessed using the console command line `help("Control")` and include:

- `if`(*condition*) *expression*—if *condition* is true then execute *expression*;
- `if`(*condition*) *expression1* `else` *expression2*—another conditional execution variant, where *expression1* is executed if *condition* is `TRUE`, else *expression2* is executed;
- `switch(...)`—conditional control function that evaluates the first argument and based on such argument chooses one of the remaining arguments.
- `for`(*variable* `in` *sequence*) *expression*—cycle where *variable* assumes in each iteration a different value from *sequence*.
- `while`(*condition*) *expression*—loop that is executed while *condition* is true;
- `repeat` *expression*—execute expression (stops only if there is a break);
- `break`—breaks out a loop; and
- `next`—-skips to next iteration.

A *condition* in R is of the `logical` type, assumed as the first `TRUE` or `FALSE` value. Similarly to other imperative languages, several logical operators can be used within a condition (use `help("Logic")` for more details):

- $x==y$, $x!=y$, $x>y$, $x>=y$ $x<y$ and $x>=y$—equal, different, higher, higher or equal, lower, lower or equal to;
- $!x$—negation of x;
- $x\&y$ and $x|y$—x and y elementwise logical AND and OR (may generate several `TRUE` or `FALSE` values);
- $x\&\&y$ and $x||y$—left to right examination of logical AND and OR (generates only one `TRUE` or `FALSE` value);
- `xor`(x, y) —elementwise exclusive OR.

Regarding the *expression*, it can include a single command, *expression1*, or a compound expression, under the form { *expression1* ; *expression2* }. An R session is presented to exemplify how control execution is performed:

```
# two if else examples:
> x=0; if(x>0) cat("positive\n") else if(x==0) cat("neutral\n")
    else cat("negative\n")
neutral
> if(xor(x,1)) cat("XOR TRUE\n") else cat("XOR FALSE\n")
XOR TRUE
> print(switch(3,"a","b","c"))        # numeric switch example
[1] "c"
> x=1; while(x<3) { print(x); x=x+1;} # while example
[1] 1
[1] 2
> for(i in 1:3) print(2*i)            # for example #1
[1] 2
```

```
[1] 4
[1] 6
> for(i in c("a","b","c")) print(i)   # for example #2
[1] "a"
[1] "b"
[1] "c"
> for(i in 1:10) if(i%%3==0) print(i) # for example #3
[1] 3
[1] 6
[1] 9
                                      # character switch example:
> var="sin";x=1:3;y=switch(var,cos=cos(x),sin=sin(x))
> cat("the",var,"of",x,"is",round(y,digits=3),"\n")
the sin of 1 2 3 is 0.841 0.909 0.141
```

This example introduces two new functions: `cat` and `round`. Similarly to `print`, the `cat` function concatenates and outputs several objects, where `"\n"` means the newline character[2], while `round` rounds the object values with the number of digits defined by the second argument.

The elementwise logical operators are useful for filtering containers, such as shown in this example:

```
> x=1:10;print(x)
 [1]  1  2  3  4  5  6  7  8  9 10
> print(x>=3&x<8)                          # select some elements
 [1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
> I=which(x>=3&x<8);print(I)               # indexes of selection
[1] 3 4 5 6 7
> d=data.frame(x=1:4,f=factor(c(rep("a",2),rep("b",2))))
> print(d)
  x f
1 1 a
2 2 a
3 3 b
4 4 b
> print(d[d$x<2|d$f=="b",])                # select rows
  x f
1 1 a
3 3 b
4 4 b
```

The `rep` function replicates elements of vectors and lists. For instance, `rep(1:3,2)` results in the vector: `1 2 3 1 2 3`.

The power of R is highly enhanced by the definition of functions, which similarly to other imperative languages (e.g., C, Java) define a portion of code that can be called several times during program execution. A function is defined as: *name*=`function(`*arg1*, *arg2*, `...)` *expression*, where: *name* is the function

[2]For C language users, there is also the sprintf function (e.g., `sprintf("float: %.2f string: %s",pi,"pi")`).

name; *arg1*, *arg2*, ... are the arguments; and *expression* is a single command or compound expression. Arguments can have default values by using *arg=arg expression* in the function definition. Here, *arg expression* can be a constant or an arbitrary R expression, even involving other arguments of the same function. The three dots special argument (...) means several arguments that are passed on to other functions (an example of ... usage is shown in Sect. 3.2).

The scope of a function code is local, meaning that any assignment (=) within the function is lost when the function ends. If needed, global assignment in R is possible using the «- operator (see Sect. 4.5 for an example). Also, a function can only return one object, which is the value set under the `return` command or, if not used, last line of the function. Functions can be recursive, i.e., a function that calls the same function but typically with different arguments. Moreover, a function may define other functions within itself.

As an example, the following code was edited in a file named `functions.R` and computes the profit for the **bag prices** task (Sect. 1.7) and defines a recursive function:

```
### functions.R file ###
# compute the bag factory profit for x:
#    x - a vector of prices
profit=function(x)     # x - a vector of prices
{ x=round(x,digits=0) # convert x into integer
  s=sales(x)          # get the expected sales
  c=cost(s)           # get the expected cost
  profit=sum(s*x-c)   # compute the profit
  return(profit)
# local variables x, s, c and profit are lost from here
}

# compute the cost for producing units:
#    units - number of units produced
#    A - fixed cost, cpu - cost per unit
cost=function(units,A=100,cpu=35-5*(1:length(units)))
{ return(A+cpu*units) }

# compute the estimated sales for x:
#    x - a vector of prices, m - marketing effort
#    A, B, C - constants of the estimated function
sales=function(x,A=1000,B=200,C=141,
               m=seq(2,length.out=length(x),by=-0.25))
{ return(round(m*(A/log(x+B)-C),digits=0))}

# example of a simple recursive function:
fact=function(x=0) # x - integer number
{ if(x==0) return(1) else return(x*fact(x-1))}
```

In this example, although object `x` is changed inside function `profit`, such change is not visible outside the function scope. The code also presents several examples of default arguments, such as constants (e.g., `C=141`) and more complex expressions (e.g., `cpu=35-5*(1:length(units))`). The last function

(`fact`) was included only for demonstrative purposes of a recursive function, since it only works for single numbers. It should be noted that R includes the enhanced `factorial` function that works with both single and container objects.

When invoking a function call, arguments can be given in any order, provided the argument name is explicitly used, under the form *argname=object*, where *argname* is the name of the argument. Else, arguments are assumed from left to right. The following session loads the previous code and executes some of its functions:

```
> source("functions.R") # load the code
> cat("class of profit is:",class(profit),"\n") # function
class of profit is: function
> x=c(414.1,404.2,408.3,413.2,395.0)
> y=profit(x); cat("maximum profit:",y,"\n")
maximum profit: 43899
> cat("x is not changed:",x,"\n")
x is not changed: 414.1 404.2 408.3 413.2 395
> cat("cost(x=",x,")=",cost(x),"\n")
cost(x= 414.1 404.2 408.3 413.2 395 )= 12523 10205 8266 6298
    4050
> cat("sales(x=",x,")=",sales(round(x)),"\n")
sales(x= 414.1 404.2 408.3 413.2 395 )= 30 27 23 19 16
> x=c(414,404); # sales for 2 bags:
> cat("sales(x=",x,")=",sales(x),"\n")
sales(x= 414 404 )= 30 27
> cat("sales(x,A=1000,m=c(2,1.75))=",sales(x,1000,m=c(2,1.75)),"
    \n")
sales(x,A=1000,m=c(2,1.75))= 30 27
> # show 3! :
> x=3; cat("fact(",x,")=",fact(x),"\n")
fact( 3 )= 6
```

R users tend to avoid the definition of loops (e.g., `for`) in order to reduce the number of lines of code and mistakes. Often, this can be achieved by using special functions that execute an argument function over all elements of a container (e.g., vector or matrix), such as: `sapply`, which runs over a vector or list; and `apply`, which runs over a matrix or array. An example that demonstrates these functions is shown next:

```
> source("functions.R")          # load the code
> x=1:5                          # show the factorial of 1:5
> cat(sapply(x,fact),"\n")       # fact is a function
1 2 6 24 120
> m=matrix(ncol=5,nrow=2)
> m[1,]=c(1,1,1,1,1)             # very cheap bags
> m[2,]=c(414,404,408,413,395)   # optimum
# show profit for both price setups:
> y=apply(m,1,profit); print(y) # profit is a function
[1] -7854 43899
```

The second argument of `apply()` is called `MARGIN` and indicates if the function (third argument) is applied over the rows (`1`), columns (`2`), or both (`c(1,2)`).

2.4 Importing and Exporting Data

The R tool includes several functions for importing and exporting data. Any R object can be saved into a binary or ASCII (using an R external representation) with the `save` function and then loaded with the `load` command. All R session objects, i.e., the current workspace, can be saved with `save.image()`, which also occurs with `q("yes")` (quiting function). Such workspace is automatically saved into a `.RData` file. Similarly to when reading R source files, file names are assumed to be found in the current working directory (corresponding to `getwd()`), unless the absolute path is specified in the file names.

Text files can be read by using the `readLines` (all file or one line at the time) functions. A text file can be generated by using a combination of: `file` (create or open a connection), `writeLines` (write lines into a connection) and `close` (close a connection); or `sink` (divert output to a connection) and console writing functions (e.g., `print` or `cat`).

The next example shows how to save/load objects and text files:

```
> x=list(a=1:3,b="hello!")           # x is a list
> save(x,file="x.Rdata",ascii=TRUE) # save into working
                                      directory
> rm(x)                              # remove an object
> print(x)                           # gives an error
Error in print(x) : object 'x' not found
> load("x.Rdata")                    # x now exists!
> print(x)                           # show x
$a
[1] 1 2 3

$b
[1] "hello!"

> t=readLines("x.Rdata")             # read all text file
> cat("first line:",t[1],"\n")       # show 1st line
first line: RDA2
> cat("first line:",readLines("x.Rdata",n=1),"\n")
first line: RDA2
> # write a text file using writeLines:
> conn=file("demo.txt")             # create a connection
> writeLines("hello!", conn)         # write something
> close(conn)                        # close connection
> # write a text file using sink:
> sink("demo2.txt")                  # divert output
> cat("hello!\n")                    # write something
> sink()                             # stop sink
```

A common way of loading data is to read tabular or spreadsheet data (e.g., CSV format) by using the `read.table` function (and its variants, such as `read.csv`). It should be noted that `read.table` can also read files directly from the Web, as shown in Sect. 7.4. The reverse operation is performed using the `write.table` command. The "R data import/export" section of the R manual

(accessed using `help.start()`) includes a wide range of data formats that can be accessed by installing the `foreign` package, such as `read.spss`, `read.mtp` (Minitab Portable Worksheet format), and `read.xport` (SAS XPORT format). Other file formats can be read using other packages, such as Excel files (`gdata` package and function `read.xls`), Web content (`RCurlb` package and `getURL` function), relational databases (e.g., MySQL using package `RMySQL`), and text corpus (`tm` package). A demonstrative example for reading and writing tabular data is shown here:

```
> # create and write a simple data.frame:
> d=data.frame(day=1:2,mon=factor(c("Jan","Feb")),year=c(12,13))
> print(d)
  day mon year
1   1 Jan   12
2   2 Feb   13
> write.table(d,file="demo.csv",row.names=FALSE,sep=";")
> # read the created data.frame:
> d2=read.table("demo.csv",header=TRUE,sep=";")
> print(d2)
  day mon year
1   1 Jan   12
2   2 Feb   13
> # read white wine quality dataset from UCI repository:
> library(RCurl)
> URL="http://archive.ics.uci.edu/ml/machine-learning-databases/
    wine-quality/winequality-white.csv"
> wine=getURL(URL)
# write "winequality-white.csv" to working directory:
> write(wine,file="winequality-white.csv")
# read file:
> w=read.table("winequality-white.csv",header=TRUE,sep=";")
> cat("wine data (",nrow(w),"x",ncol(w),")\n") # show nrow x
    ncol
wine data ( 4898 x 12 )
```

Any R graphic can be saved into a file by changing the output device driver, creating the graphic and then closing the device (`dev.off()`). Several graphic devices are available, such as `pdf`, `png`, `jpeg`, and `tiff`. The next example shows the full code used to create the top left graph of Fig. 1.3:

```
# create PDF file:
DIR=""              # change if different directory is used
pdf(paste(DIR,"sumbits.pdf",sep=""),width=5,height=5)

sumbinint=function(x)          # sum of bits of an integer
{ return(sum(as.numeric(intToBits(x))))}

sumbits=function(x)              # sum of bits of a vector
{ return(sapply(x,sumbinint))}

D=8; x=0:(2^D-1)# x is the search space (integer representation)
y=sumbits(x)     # y is the number of binary bits of x
plot(x,y,type="l",ylab="evaluation function",
     xlab="search space (x)",lwd=2)
```

```
pmax=c(x[which.max(y)],max(y)) # maximum point coordinates
points(pmax[1],pmax[2],pch=19,lwd=2)  # plot maximum point
legend("topleft","optimum",pch=19,lwd=2)    # add a legend
dev.off()                                  # close the device
```

This examples introduces the functions: `intToBits`, which converts an integer into a binary representation; `as.numeric`, which converts an object into numeric; and `legend`, which adds legends to plots.

2.5 Additional Features

This section discusses four additional R features: command line execution of R, parallel computing, getting R source code of a function, and interfacing with other computer languages.

The R environment can be executed directly from the operating system console, under two possibilities:

- `R [options] [< infile] [> outfile]`; or
- `R CMD command [arguments]`.

The full details can be accessed by running `$ R -help` in the operating system console (`$` is the *Mac OS* prompt). This direct mode can be used for several operations, such as compiling files for use in R or executing an R file in batch processing, without manual intervention. For example, if the previous shown code for creating a pdf file is saved in a file called `sumbits.R`, then such code can be directly executed in R by using: `$ R --vanilla --slave < sumbits.R`.

There is a CRAN task view for high-performance and parallel computing with R (http://cran.r-project.org/web/views/HighPerformanceComputing.html). The view includes several packages that are useful for high-performance computing, such as `multicore` and `parallel`. An example of `multicore` use is shown next:

```
> library(multicore)        # load the package
> x1=1:5;x2=5:10     # create 2 objects
> p1=parallel(factorial(x1)) # run in parallel
> p2=parallel(factorial(x2)) # run in parallel
> collect(list(p1,p2))       # collect results
$`8995`
[1]   1   2   6  24 120

$`8996`
[1]     120     720    5040   40320  362880 3628800
```

Given that all functions are stored as objects, it is easy in R to access the full code of any given function, including built-in functions, by using the `methods` and `getAnywhere` commands, such as:

```
methods(mean) # list all available methods for mean function
getAnywhere(mean.default) # show R code for default mean
    function
```

The R environment can interface with other programming languages, such as Fortran, C, and Java. Examples of interfaces with the C and Java languages can be found in:

- C—http://adv-r.had.co.nz/C-interface.html; see also the "Writing R Extensions" user manual, by typing `help.start()`;
- Java—http://www.rforge.net/rJava/.

2.6 Command Summary

`Inf`	Infinity value
`NA`	Missing value
`NULL`	Empty or null object
`NaN`	Not a number constant
`RCurl`	Package for network (HTTP/FTP/. . .) interface
`apply()`	Apply a function over a matrix or array
`as.numeric()`	Converts an object into numeric
`barplot()`	Draw a bar plot
`boxplot()`	Plot a box-and-whisker graph
`c()`	Concatenate values into a vector
`cat()`	Concatenate and output command
`chisq.test()`	Pearson's chi-squared test
`class()`	Get class of object
`close()`	Close a file connection
`cos()`	Cosine trigonometric function
`dev.off())`	Close a graphical device
`example()`	Show examples of a command
`factorial()`	Compute the factorial of an object
`file()`	Create or open a file connection
`for()`	Loop execution command
`function()`	Defines a function
`getAnywhere()`	Retrieve an R object
`getURL()`	Get Web content (package `RCurl`)
`getwd()`	Get working directory

`help()`	Get help on a particular subject
`help.search()`	Get help on a text expression
`help.start()`	Get the full R manual
`hist()`	Plot a histogram
`if()`	Conditional execution command
`install.packages()`	Install a package
`intToBits()`	Convert integer to binary representation
`is.na()`	Check missing data
`is.nan()`	Check if NaN
`is.null()`	Check if NULL
`jpeg()`	Set graphical device to jpeg file
`lattice`	Package with high-level data visualization functions
`legend()`	Add a legend to a plot
`length()`	Number of elements of an object
`library()`	Load a package
`load()`	Load an object from file
`ls()`	List created objects
`max()`	Maximum of all values
`mean()`	Mean of all values
`median()`	Median of all values
`methods()`	List methods for functions
`min()`	Minimum of all values
`multicore`	Package for parallel processing
`names()`	Get and set the names of an object
`parallel()`	Execute expression in a separate process (package `multicore`)
`pdf()`	Set graphical device to pdf file
`pi`	Mathematical π value
`pie()`	Plot a pie chart
`plot()`	Generic plot of a object
`png()`	Set graphical device to png file
`print()`	Show an object
`read.table()`	Read a tabular file (e.g., CSV)
`readLines()`	Read lines from text file
`rep()`	Function that replicates elements of vectors and lists
`return()`	Returns an item from a function
`rnorm()`	Create normal distribution random samples
`round()`	Rounds the first argument values

`runif()`	Create real value uniform random samples
`sample()`	Create integer uniform random samples
`sapply()`	Apply a function over a vector
`save()`	Save an object into a file
`save.image()`	Save workspace
`seq()`	Create a regular sequence
`setwd()`	Set the working directory
`set.seed()`	Set the random generation number (used by `sample`, `runif`, ...)
`sin()`	Sine trigonometric function
`sink()`	Divert output to a file connection
`sort()`	Sorts a vector or factor
`source()`	Execute R code from a file
`sqrt()`	Square root of a number
`str()`	Show internal structure of object
`sum()`	Sum of all values
`summary()`	Show a summary of the object
`switch()`	Conditional control function
`t.test()`	Performs a t-student test
`tan()`	Tangent trigonometric function
`tiff()`	Set graphical device to tiff file
`wilcox.test()`	Wilcoxon test
`which()`	Returns the indexes of an object that follow a logical condition
`which.max()`	Returns the indexes of the maximum value
`which.min()`	Returns the indexes of the minimum value
`while()`	Loop execution command
`wireframe()`	Draw a 3D scatter plot (package `lattice`)
`writeLines()`	Write lines into a text file
`write.table()`	Write object into tabular file

2.7 Exercises

2.1. In the R environment console, create a vector `v` with 10 elements, all set to 0. Using a single command, replace the indexes 3, 7, and 9 values of `v` with 1.

2.2. Create a vector `v` with all even numbers between 1 and 50.

2.3. Create matrix `m` of size 3×4, such that:

1. the first row contains the sequence 1, 2, 3, 4;
2. the second row contains the square root of the first row;

3. the third row is computed after step 2 and contains the square root of the second row; and
4. the fourth column is computed after step 3 and contains the squared values of the third column.

Then, show matrix values and its row and column sums (use `apply` function) with a precision of two decimal digits.

2.4. Create function `counteven(x)` that counts how many even numbers are included in a vector `x`, using three approaches:

1. use a `for()` cycle with an `if()` condition;
2. use `sapply()` function; and
3. use a condition that is applied directly to `x` (without `if`).

Test the function over the object `x=1:10`.

2.5. Write in a file `maxsin.R` the full R code that is needed to create the `maxsin.pdf` PDF file that appears in top right plot of Fig. 1.3 (Sect. 1.7 and Eq. (1.2) explain how **max sin** is defined). Execute the R source file and check if the PDF file is identical to the top right plot of Fig. 1.3.

2.6. Forest fires data exercise:

1. If needed, install and load the `RCurl` package.
2. Use the `getURL` and `write` functions to write the forest fires data http://archive.ics.uci.edu/ml/machine-learning-databases/forest-fires/forestfires.csv into a local file.
3. Load the local CSV file (`forestfires.csv`, the separator character is `","`) into a data frame.
4. Show the average temperature in August.
5. Select ten random samples of temperatures from the months February, July, and August; then check if the average temperature differences are significant under 95 % confidence level t-student paired tests.
6. Show all records from August and with a burned area higher than 100.
7. Save the records obtained previously (6) into a CSV file named `aug100.csv`.

Chapter 3
Blind Search

3.1 Introduction

Full blind search assumes the exhaustion of all alternatives, where any previous search does not affect how next solutions are tested (left of Fig. 3.1). Given that the full search space is tested, the optimum solution is always found. Blind search is only applicable to discrete search spaces and it is easy to encode in two ways. First, by setting the full search space in a matrix and then sequentially testing each row (solution) of this matrix. Second, in a recursive way, by setting the search space as a tree, where each branch denotes a possible value for a given variable and all solutions appear at the leaves (at the same level). Examples of two quite known blind methods based on tree structures are depth-first and breadth-first algorithms. The former starts at the root of the tree and traverses through each branch as far as possible, before backtracking. The latter also starts at the root but searches on a level basis, searching first all succeeding nodes of the root and then the next succeeding nodes of the root succeeding nodes, and so on.

The major disadvantage of pure blind search is that it is not feasible when the search space is continuous or too large, a situation that often occurs with real-world tasks. Consider, for instance, the **bag prices** toy problem defined in Sect. 1.7, even with a small search dimension ($D = 5$) the full search space is quite large for the R tool (i.e., $1000^5 = 10^{15} = 1000 \times 10^{12} = 1000$ billion of searches!). Hence, pure blind search methods are often adapted, by setting thresholds (e.g., depth-first with a maximum depth of K), reducing the space searched or using heuristics. Grid search (Hsu et al. 2003) is an example of a search space reduction method. Monte Carlo search (Caflisch 1998), also known as random search, is another popular blind method. The method is based on a repeated random sampling, with up to N sampled points. This method is popular since it is computationally feasible and quite easy to encode.

P. Cortez, *Modern Optimization with R*, Use R!, DOI 10.1007/978-3-319-08263-9_3

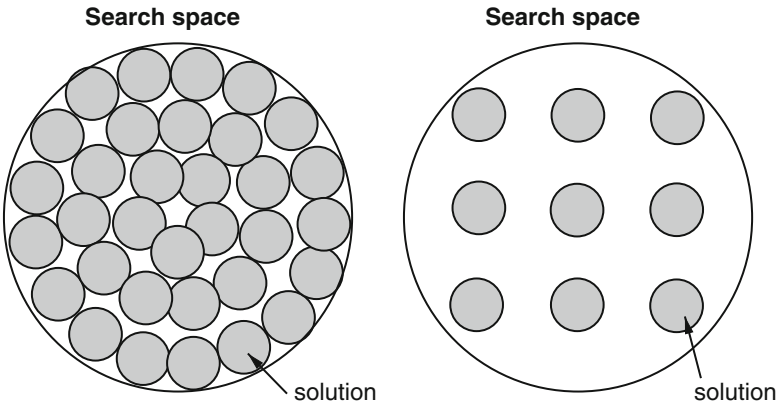

Fig. 3.1 Example of pure blind search (*left*) and grid search (*right*) strategies

The next sections present R implementations of three blind search methods: full blind search, grid search, and Monte Carlo search. Also, these implementations are tested on the demonstrative problems presented in Chap. 1.

3.2 Full Blind Search

This section presents two blind search functions: `fsearch` and `dfsearch`. The former is a simpler function that requires the search space to be explicitly defined in a matrix in the format *solutions* $\times D$ (argument `Search`), while the latter performs a recursive implementation of the depth-first search and requires the definition of the domain values for each variable to be optimized (argument `domain`). Both functions receive as arguments the evaluation function (`FUN`), the optimization type (`type`, a character with `"min"` or `"max"`) and extra arguments, (denoted by `...` and that might be used by the evaluation function `FUN`). These functions were encoded in a file named `blind.R`:

```
### blind.R file ###

# full bind search method
#    search - matrix with solutions x D
#    FUN - evaluation function
#    type - "min" or "max"
#    ... - extra parameters for FUN
fsearch=function(search,FUN,type="min",...)
{
 x=apply(search,1,FUN,...) # run FUN over all search rows
 ib=switch(type,min=which.min(x),max=which.max(x))
 return(list(index=ib,sol=search[ib,],eval=x[ib]))
}
```

```
# depth-first full search method
#    l - level of the tree
#    b - branch of the tree
#    domain - vector list of size D with domain values
#    FUN - eval function
#    type - "min" or "max"
#    D - dimension (number of variables)
#    x - current solution vector
#    bcur - current best sol
#    ... - extra parameters for FUN
dfsearch=function(l=1,b=1,domain,FUN,type="min",
  D=length(domain),
                x=rep(NA,D),
                bcur=switch(type,min=list(sol=NULL,eval=Inf),
                                 max=list(sol=NULL,eval=-Inf)),
                 ...)
{ if((l-1)==D) # "leave" with solution x to be tested:
    { f=FUN(x,...);fb=bcur$eval
      ib=switch(type,min=which.min(c(fb,f)),
                     max=which.max(c(fb,f)))
      if(ib==1) return (bcur) else return(list(sol=x,eval=f))
    }
  else # go through sub branches
    { for(j in 1:length(domain[[l]]))
        { x[l]=domain[[l]][j]
          bcur=dfsearch(l+1,j,domain,FUN,type,D=D,
                        x=x,bcur=bcur,...)
        }
      return(bcur)
    }
}
```

where `dfsearch` is a recursive function that tests if the tree node is a leave, computing the evaluation function for the respective solution, else traverses through the node sub branches. This function requires some memory state variables (`l`, `b`, `x` and `bcur`) that are changed each time a new recursive call is executed. The domain of values is stored in a vector list of length D, since the elements of this vector can have different lengths, according to their domain values.

The next R code tests both blind search functions for the **sum of bits** and **max sin** tasks (Sect. 1.7, $D = 8$):

```
### binary-blind.R file ###

source("blind.R") # load the blind search methods

# read D bits from integer x:
binint=function(x,D)
{ x=rev(intToBits(x)[1:D]) # get D bits
  # remove extra 0s from raw type:
  as.numeric(unlist(strsplit(as.character(x),""))[(1:D)*2])
}
```

```
# convert binary vector into integer: code inspired in
# http://stackoverflow.com/questions/12892348/
# in-r-how-to-convert-binary-string-to-binary-or-decimal-value
intbin=function(x) sum(2^(which(rev(x==1))-1))

# sum a raw binary object x (evaluation function):
sumbin=function(x) sum(as.numeric(x))

# max sin of binary raw object x (evaluation function):
maxsin=function(x,Dim) sin(pi*(intbin(x))/(2^Dim))

D=8 # number of dimensions
x=0:(2^D-1) # integer search space
# set full search space in solutions x D:
search=t(sapply(x,binint,D=D))
# set the domain values (D binary variables):
domain=vector("list",D)
for(i in 1:D) domain[[i]]=c(0,1) # bits

# sum of bits, fsearch:
S1=fsearch(search,sumbin,"max") # full search
cat("fsearch best s:",S1$sol,"f:",S1$eval,"\n")

# sum of bits, dfsearch:
S2=dfsearch(domain=domain,FUN=sumbin,type="max")
cat("dfsearch best s:",S2$sol,"f:",S2$eval,"\n")

# max sin, fsearch:
S3=fsearch(search,maxsin,"max",Dim=8) # full search
cat("fsearch best s:",S3$sol,"f:",S3$eval,"\n")

# max sin, dfsearch:
S4=dfsearch(domain=domain,FUN=maxsin,type="max",Dim=8)
cat("dfsearch best s:",S4$sol,"f:",S4$eval,"\n")
```

where `binint` is an auxiliary function that selects only D bits from the `raw` object returned by `intToBits`. The `intToBits` returns 32 bits in a reversed format, thus the `rev` R function is also applied to set correctly the bits order. Given that the `raw` type includes two hex digits, the purpose of the last line of function `binint` is to remove extra 0 characters from the raw object. Such line uses some R functions that were not described in the previous chapter: `as.character`—convert to character type; `strsplit`—split a character vector into substrings; and `unlist`—transforms a list into a vector. The following R session exemplifies the effect of the `binint` code (and newly introduced R functions):

```
> x=intToBits(7)[1:4]; print(x)
[1] 01 01 01 00
> x=rev(x); print(x)
[1] 00 01 01 01
> x=strsplit(as.character(x),""); print(x)
[[1]]
[1] "0" "0"

[[2]]
```

```
 [1] "0" "1"

 [[3]]
 [1] "0" "1"

 [[4]]
 [1] "0" "1"

 > x=unlist(x); print(x)
 [1] "0" "0" "0" "1" "0" "1" "0" "1"
 > x=as.numeric(x[(1:4)*2]); print(x)
 [1] 0 1 1 1
```

The generic `sapply` function uses the defined `binint` function in order to create the full binary search space from an integer space. Given that `sapply` returns a $D\times$ *solutions* matrix, the `t` R function is used to transpose the matrix into the required *solutions* $\times D$ format. The result of executing file `binary-blind.R` is:

```
> source("binary-blind.R")
fsearch best s: 1 1 1 1 1 1 1 1 f: 8
dfsearch best s: 1 1 1 1 1 1 1 1 f: 8
fsearch best s: 1 0 0 0 0 0 0 0 f: 1
dfsearch best s: 1 0 0 0 0 0 0 0 f: 1
```

where both methods (`fsearch` and `dfsearch`) return the optimum **sum of bits** and **max sin** solutions.

Turning to the **bag prices** task (Sect. 1.7), as explained previously, the search of all space of solutions (1000^5) is not feasible in practical terms. However, using domain knowledge, i.e., the original problem formulation assumes that the price for each bag can be optimized independently of other bag prices, it is easy to get the optimum solution, as shown in file `bag-blind.R`:

```
### bag-blind.R file ###

source("blind.R") # load the blind search methods
source("functions.R") # load profit(), cost() and sales()

# auxiliary function that sets the optimum price for
# one bag type (D), assuming an independent influence of
# a particular price on the remaining bag prices:
ibag=function(D) # D - type of bag
{ x=1:1000 # price for each bag type
  # set search space for one bag:
  search=matrix(ncol=5,nrow=1000)
  search[]=1; search[,D]=x
  S1=fsearch(search,profit,"max")
  S1$sol[D] # best price
}

# compute the best price for all bag types:
S=sapply(1:5,ibag)
# show the optimum solution:
cat("optimum s:",S,"f:",profit(S),"\n")
```

The result of executing file `bag-blind.R` is:

```
> source("bag-blind.R")
optimum s: 414 404 408 413 395 f: 43899
```

It should be noted that while the original formulation of **bag prices** assumes an independence when optimizing each bag price variable (and optimum profit is 43,899), there are other variations presented in this book where this assumption is not true (see Sects. 5.7 and 6.2).

Given that pure blind search cannot be applied to real value search spaces ($\Re$), no code is shown here for the **sphere** and **rastrigin** tasks. Nevertheless, these two real value optimization tasks are handled in the next two sections.

3.3 Grid Search

Grid search reduces the space of solutions by implementing a regular hyper dimensional search with a given step size. The left of Fig. 3.1 shows an example of a two dimensional (3×3) grid search. Grid search is particularly used for hyperparameter optimization of machine learning algorithms, such as neural networks or support vector machines.

There are several grid search variants. Uniform design search (Huang et al. 2007) is similar to the standard grid search method, except that it uses a different type of grid, with lesser search points. Nested grid search is another variant that uses several grid search levels. The first level is used with a large step size. Then, a second grid level is applied over the best point, searching over a smaller area and with a lower grid size. And so on. Nested search is not a pure blind method, since it incorporates a greedy heuristic, where the next level search is guided by the result of the current level search.

Depending on the grid step size, grid search is often much faster than pure bind search. Also, depending on the number of levels and initial grid step size, nested search might be much faster than standard grid search, but it also can get stuck more easily on local minima. The main disadvantage of the grid search approach is that it suffers from the curse of dimensionality, i.e., the computational effort complexity is very high when the number of dimensions (variables to optimize) is large. For instance, the standard grid search computational complexity is $\mathcal{O}(L^D)$, where L is the number of grid search levels and D the dimension (variables to optimize). If only $L = 3$ levels are considered and with a dimension of $D = 30$, this leads to $3^{30} \approx$ 206 billion searches, which is infeasible under the R tool. Other disadvantages of grid search methods include the additional parameters that need to be set (e.g., grid search step, number of nested levels) and also the adopted type of blind search that does not warranty achieving the optimum solution and, more importantly, might not be particularly efficient in several practical applications.

The next code implements two functions for the standard grid search method (`gsearch` and `gsearch2`) and one for the nested grid search (`ngsearch`):

```
### grid.R file ###

# standard grid search method (uses fsearch)
#    step - vector with step size for each dimension D
#    lower - vector with lowest values for each dimension
#    upper - vector with highest values for each dimension
#    FUN - evaluation function
#    type - "min" or "max"
#    ... - extra parameters for FUN
gsearch=function(step,lower,upper,FUN,type="min",...)
{ D=length(step) # dimension
  domain=vector("list",D) # domain values
  L=vector(length=D) # auxiliary vector
  for(i in 1:D)
     { domain[[i]]=seq(lower[i],upper[i],by=step[i])
       L[i]=length(domain[[i]])
     }
  LS=prod(L)
  s=matrix(ncol=D,nrow=LS) # set the search space
  for(i in 1:D)
    {
     if(i==1) E=1 else E=E*L[i-1]
     s[,i]=rep(domain[[i]],length.out=LS,each=E)
    }
  fsearch(s,FUN,type,...) # best solution
}

# standard grid search method (uses dfsearch)
gsearch2=function(step,lower,upper,FUN,type="min",...)
{ D=length(step) # dimension
  domain=vector("list",D) # domain values
  for(i in 1:D) domain[[i]]=seq(lower[i],upper[i],by=step[i])
  dfsearch(domain=domain,FUN=FUN,type=type,...) # solution
}

# nested grid search method (uses fsearch)
#    levels - number of nested levels
ngsearch=function(levels,step,lower,upper,FUN,type,...)
{ stop=FALSE;i=1 # auxiliary objects
  bcur=switch(type,min=list(sol=NULL,eval=Inf),
                   max=list(sol=NULL,eval=-Inf))
  while(!stop) # cycle while stopping criteria is not met
  {
     s=gsearch(step,lower,upper,FUN,type,...)
     # if needed, update best current solution:
     if( (type=="min" && s$eval<bcur$eval)||
         (type=="max" && s$eval>bcur$eval)) bcur=s
     if(i<levels) # update step, lower and upper:
       { step=step/2
         interval=(upper-lower)/4
         lower=sapply(lower,max,s$sol-interval)
         upper=sapply(upper,min,s$sol+interval)
       }
```

```
    if(i>=levels || sum((upper-lower)<=step)>0) stop=TRUE
    else i=i+1
  }
  return(bcur) # best solution
}
```

All functions require the setting of the grid step (`step`, numeric vector) and lower and upper bounds (vectors `lower` and `upper`). The first function uses the `fsearch` function, while the second one uses the recursive blind variant (`dfsearch`), both described in Sect. 3.2. The `gsearch` function contains more code in comparison with `gsearch2`, since it requires setting first the search space. This is achieved by using the useful `rep` function with the `each` argument. For example, `rep(1:2,each=2)` returns the vector: `1 1 2 2`. The second grid search function (`gsearch2`) is simpler to implement, given that it performs a direct call of the depth-first search.

The nested grid function uses a simple cycle that calls `gsearch` and whose maximum number of iterations depends on the `levels` argument. The cycle also stops when the range set by the upper and lower bounds is lower than the step size. The next level search is set around the best solution of the current grid search and with half of the current step size. The lower and upper bounds are changed accordingly and the `min` and `max` functions are used to avoid setting a search space larger than the original bounds. For some configurations of the `step`, `lower` and `upper` arguments, this nested function might repeat on the next level the evaluation of solutions that were previously evaluated. For the sake of simplicity, the nested grid code is kept with this handicap, although it could be enhanced by implementing a cache that stores previous tested solutions in memory and only computes the evaluation function for new solutions.

The next code explores the three implemented grid search methods for the **bag prices** task of Sect. 1.7:

```
### bag-grid.R file ###

source("blind.R") # load the blind search methods
source("grid.R") # load the grid search methods
source("functions.R") # load the profit function

# grid search for all bag prices, step of 100$
PTM=proc.time() # start clock
S1=gsearch(rep(100,5),rep(1,5),rep(1000,5),profit,"max")
sec=(proc.time()-PTM)[3] # get seconds elapsed
cat("gsearch best s:",S1$sol,"f:",S1$eval,"time:",sec,"s\n")

# grid search 2 for all bag prices, step of 100$
PTM=proc.time() # start clock
S2=gsearch2(rep(100,5),rep(1,5),rep(1000,5),profit,"max")
sec=(proc.time()-PTM)[3] # get seconds elapsed
cat("gsearch2 best s:",S2$sol,"f:",S2$eval,"time:",sec,"s\n")
```

```
# nested grid with 3 levels and initial step of 500$
PTM=proc.time() # start clock
S3=ngsearch(3,rep(500,5),rep(1,5),rep(1000,5),profit,"max")
sec=(proc.time()-PTM)[3] # get seconds elapsed
cat("ngsearch best s:",S3$sol,"f:",S3$eval,"time:",sec,"s\n")
```

This code includes the `proc.time` R function, which returns the time elapsed (in seconds) by the running process and that is useful for computational effort measurements. The result of executing file `bag-grid.R` is:

```
> source("bag-grid.R")
gsearch best s: 401 401 401 401 501 f: 43142 time: 4.149 s
gsearch2 best s: 401 401 401 401 501 f: 43142 time: 5.654 s
ngsearch best s: 376.375 376.375 376.375 501.375 501.375 f:
    42823 time: 0.005 s
```

Under the tested settings, the pure grid search methods execute 10 searches per dimension, leading to a total of $10^5 = 100{,}000$ evaluations, achieving the same solution (43,142) under a similar computational effort.[1] The nested grid achieves a worst solution (42,823) but under much less evaluations (2 searches for per dimension and level, total of $2^5 \times 3 = 96$ tested solutions).

Regarding the real value optimization tasks (**sphere** and **rastrigin**, Sect. 1.7), these can be handled by grid search methods, provided that the dimension adopted is small. The next code shows an example for $D = 2$ and range of $[-5.2, 5.2]$ (commonly used within these benchmark functions):

```
### real-grid.R file ###

source("blind.R") # load the blind search methods
source("grid.R") # load the grid search methods

# real-value functions: sphere and rastrigin:
sphere=function(x) sum(x^2)
rastrigin=function(x) 10*length(x)+sum(x^2-10*cos(2*pi*x))

cat("sphere:\n") # D=2, easy task
S=gsearch(rep(1.1,2),rep(-5.2,2),rep(5.2,2),sphere,"min")
cat("gsearch s:",S$sol,"f:",S$eval,"\n")
S=ngsearch(3,rep(3,2),rep(-5.2,2),rep(5.2,2),sphere,"min")
cat("ngsearch s:",S$sol,"f:",S$eval,"\n")

cat("rastrigin:\n") # D=2, easy task
S=gsearch(rep(1.1,2),rep(-5.2,2),rep(5.2,2),rastrigin,"min")
cat("gsearch s:",S$sol,"f:",S$eval,"\n")
S=ngsearch(3,rep(3,2),rep(-5.2,2),rep(5.2,2),rastrigin,"min")
cat("ngsearch s:",S$sol,"f:",S$eval,"\n")
```

[1] Slightly different execution times can be achieved by executing distinct runs under the same code and machine.

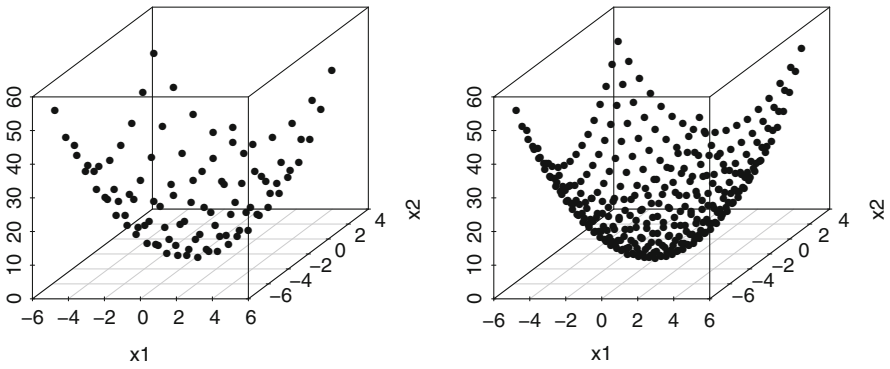

Fig. 3.2 Example of grid search using $L = 10$ (*left*) and $L = 20$ (*right*) levels for **sphere** and $D = 2$

The execution result of file `real-grid.R` is:

```
sphere:
gsearch s: 0.3 0.3 f: 0.18
ngsearch s: -0.1 -0.1 f: 0.02
rastrigin:
gsearch s: -1.9 -1.9 f: 11.03966
ngsearch s: -0.1 -0.1 f: 3.83966
```

Good solutions were achieved, close to the optimum solution of $s = (0, 0)$ and $f = 0$. Figure 3.2 shows how the space is searched when using the standard grid search for **sphere** when using $L = 10$ (left) and $L = 20$ search levels (right) for each dimension. The three dimensional plots were achieved using the `scatterplot3d` function of the `scatterplot3d` package.

3.4 Monte Carlo Search

Monte Carlo is a versatile numerical method that is easy to implement and is applicable to high-dimensional problems (in contrast with grid search), ranging from Physics to Finance (Caflisch 1998). The method consists in a random generation of N points, using a given probability distribution over the problem domain. The computational effort complexity is $\mathcal{O}(N)$.

More details about implementing Monte Carlo methods in R can be found in Robert and Casella (2009). In this book, we present a very simple implementation of the Monte Carlo search, which adopts the uniform distribution $\mathcal{U}(lower, upper)$ and includes only four lines of code:

```
### montecarlo.R file ###

# montecarlo uniform search method
#    N - number of samples
```

```
#    lower - vector with lowest values for each dimension
#    upper - vector with highest values for each dimension
#    domain - vector list of size D with domain values
#    FUN - evaluation function
#    type - "min" or "max"
#    ... - extra parameters for FUN
mcsearch=function(N,lower,upper,FUN,type="min",...)
{ D=length(lower)
  s=matrix(nrow=N,ncol=D) # set the search space
  for(i in 1:N) s[i,]=runif(D,lower,upper)
  fsearch(s,FUN,type,...) # best solution
}
```

The proposed implementation is tested here for the **bag prices** ($D = 5$) and real value tasks (**sphere** and **rastrigin**, $D \in \{2, 30\}$) by using $N = 10{,}000$ uniform samples:

```
### test-mc.R file ###

source("blind.R") # load the blind search methods
source("montecarlo.R") # load the monte carlo method
source("functions.R") # load the profit function

N=10000 # set the number of samples
cat("monte carlo search (N:",N,")\n")

# bag prices
cat("bag prices:")
S=mcsearch(N,rep(1,5),rep(1000,5),profit,"max")
cat("s:",S$sol,"f:",S$eval,"\n")

# real-value functions: sphere and rastrigin:
sphere=function(x) sum(x^2)
rastrigin=function(x) 10*length(x)+sum(x^2-10*cos(2*pi*x))

D=c(2,30)
label="sphere"
for(i in 1:length(D))
   { S=mcsearch(N,rep(-5.2,D[i]),rep(5.2,D[i]),sphere,"min")
     cat(label,"D:",D[i],"s:",S$sol[1:2],"f:",S$eval,"\n")
   }
label="rastrigin"
for(i in 1:length(D))
   { S=mcsearch(N,rep(-5.2,D[i]),rep(5.2,D[i]),rastrigin,"min")
     cat(label,"D:",D[i],"s:",S$sol[1:2],"f:",S$eval,"\n")
   }
```

To simplify the analysis of the obtained results, the code only shows the optimized values for the first two variables (x_1 and x_2). Given that Monte Carlo is a stochastic method, each run will present a different result. An example execution (one run) is:

```
> source("test-mc.R")
monte carlo search (N: 10000 )
bag prices:s: 349.7477 369.1669 396.1959 320.5007 302.3327 f:
    42508
```

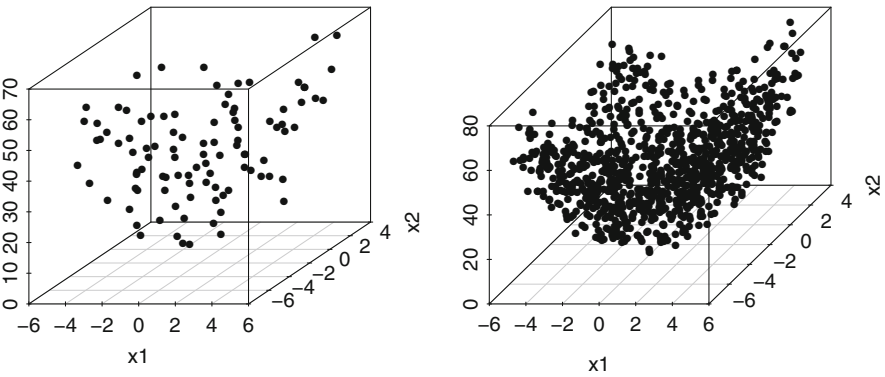

Fig. 3.3 Example of Monte Carlo search using $N = 100$ (*left*) and $N = 1{,}000$ (*right*) samples for **sphere** and $D = 2$

```
sphere D: 2 s: -0.01755296 0.0350427 f: 0.001536097
sphere D: 30 s: -0.09818928 -1.883463 f: 113.7578
rastrigin D: 2 s: -0.0124561 0.02947438 f: 0.2026272
rastrigin D: 30 s: 0.6508581 -3.043595 f: 347.1969
```

Under the tested setup ($N = 10{,}000$), interesting results were achieved for the **sphere** and **rastrigin** tasks when $D = 2$. However, when the dimension increases ($D = 30$) the optimized solutions are further away from the optimum value ($f = 0$). For demonstration purposes, Fig. 3.3 shows two examples of Monte Carlo searches for **sphere** and $D = 2$ (left plot with $N = 100$ and right graph with $N = 1{,}000$).

3.5 Command Summary

`dfsearch()`	Depth-first blind search (chapter file `"blind.R"`)
`fsearch()`	Full blind search (chapter file `"blind.R"`)
`gsearch()`	Grid search (chapter file `"grid.R"`)
`mcsearch()`	Monte Carlo search (chapter file `"montecarlo.R"`)
`ngsearch()`	Nested-grid search (chapter file `"grid.R"`)
`proc.time()`	Time elapsed by the running process
`rev()`	Reversed version of an object
`scatterplot3d`	Package that implements `scatterplot3d()`
`scatterplot3d()`	Plot a 3D point cloud (package `scatterplot3d`)
`strsplit()`	Split elements of character vector
`t()`	Matrix transpose
`unlist()`	Convert list to vector

3.6 Exercises

3.1. Explore the optimization of the binary **max sin** task with a higher dimension ($D = 16$), under pure blind, grid, and Monte Carlo methods. Show the optimized solutions, evaluation values, and time elapsed (in seconds). For the grid and Monte Carlo methods, use directly the `fsearch` function, by changing only the integer space (object x of the file `"binary-blind.R"`) with a maximum of $N = 1{,}000$ searches. Tip: use `seq()` for setting the grid search integer space and `sample()` for Monte Carlo.

3.2. Consider the **bag prices** ($D = 5$). Adapt the file `"bag-grid.R"` such that two grid searches are performed over the range $[350, 450]$ with a step size of 11$ in order to show the solution and evaluation values. The first search should be executed using `gsearch` function, while the second search should be implemented using the depth-first method (`dfsearch` function). Tip: use the `seq` function to set the domain of values used by `dfsearch`.

3.3. Consider the **rastrigin** task with a dimension of $D = 30$. Using the Monte Carlo method, explore different N values within the range {100,1,000,10,000}. Execute 30 runs for each N value and compare if the average differences are statistically significant at the 95 % confidence level under a pairwise t-student test. Also, plot the boxplots for the results due to each N value.

Chapter 4
Local Search

4.1 Introduction

In contrast with the blind search methods presented in Chap. 3, modern optimization techniques are based on a guided search, where new solutions are generated from existing solutions. Local search, often termed single-state search, includes several methods that focus their attention within a local neighborhood of a given initial solution, as shown in Fig. 4.1. *A priori* knowledge, such as problem domain heuristics, can be used to set the initial solution. A more common approach is to set the initial point randomly and perform several restarts (also known as runs).

The main differences within local methods is set on how new solutions are defined and what is kept as the current solution (corresponding to functions *change* and *select* of Algorithm 1). The next sections describe how three local search methods, namely hill climbing, simulated annealing, and tabu search, can be adopted in the R tool. This chapter also includes a section that describes how to compare search methods in R, by providing a demonstrative example that compares two local search methods with random search.

4.2 Hill Climbing

Hill climbing is a simple local optimization method that "climbs" up the hill until a local optimum is found (assuming a maximization goal). The method works by iteratively searching for new solutions within the neighborhood of current solution, adopting new solutions if they are better, as shown in the pseudo-code of Algorithm 2. The purpose of function *change* is to produce a slightly different solution, by performing a full search in the whole neighborhood or by applying a small random change in the current solution values. It should be noted that while the

P. Cortez, *Modern Optimization with R*, Use R!, DOI 10.1007/978-3-319-08263-9_4

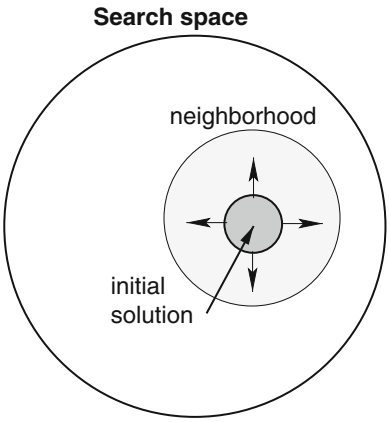

Fig. 4.1 Example of a local search strategy

Algorithm 2 Pure hill climbing optimization method

1: **Inputs:** S, f, C ▷ S is the initial solution, f is the evaluation function, C includes control parameters
2: $i \leftarrow 0$ ▷ i is the number of iterations of the method
3: **while** not $termination\_criteria(S, f, C, i)$ **do**
4: $S' \leftarrow change(S, C)$ ▷ new solution
5: $B \leftarrow best(S, S', f)$ ▷ best solution for next iteration
6: $S \leftarrow B$ ▷ deterministic select function
7: $i \leftarrow i + 1$
8: **end while**
9: **Output:** B ▷ the best solution

standard hill climbing algorithm is deterministic, when random changes are used for perturbing a solution, a stochastic behavior is achieved. This is why hill climbing is set at the middle of the deterministic/stochastic dimension in Fig. 1.2.

There are several hill climbing variants, such as steepest ascent hill climbing (Luke 2012), which searches for up to N solutions in the neighborhood of S and then adopts the best one; and stochastic hill climbing (Michalewicz et al. 2006), which replaces the deterministic select function, selecting new solutions with a probability of P (a similar strategy is performed by the simulated annealing method, discussed in the next section).

The R implementation of the standard hill climbing method is coded in file `hill.R`:

```
### hill.R file ###

# pure hill climbing:
#    par - initial solution
#    fn - evaluation function
#    change - function to generate the next candidate
#    lower - vector with lowest values for each dimension
#    upper - vector with highest values for each dimension
#    control - list with stopping and monitoring method:
```

```
#       $maxit - maximum number of iterations
#       $REPORT - frequency of monitoring information
#    type - "min" or "max"
#    ... - extra parameters for FUN
hclimbing=function(par,fn,change,lower,upper,control,
                   type="min",...)
{ fpar=fn(par,...)
  for(i in 1:control$maxit)
     {
      par1=change(par,lower,upper)
      fpar1=fn(par1,...)
      if(control$REPORT>0 &&(i==1||i%%control$REPORT==0))
        cat("i:",i,"s:",par,"f:",fpar,"s'",par1,"f:",fpar1,"\n")
      if(   (type=="min" && fpar1<fpar)
         || (type=="max" && fpar1>fpar)) { par=par1;fpar=fpar1 }
     }
  if(control$REPORT>=1) cat("best:",par,"f:",fpar,"\n")
  return(list(sol=par,eval=fpar))
}

# slight random change of vector par:
#    par - initial solution
#    lower - vector with lowest values for each dimension
#    upper - vector with highest values for each dimension
#    dist - random distribution function
#    round - use integer (TRUE) or continuous (FALSE) search
#    ... - extra parameters for dist
#    examples: dist=rnorm, mean=0, sd=1; dist=runif, min=0,max=1
hchange=function(par,lower,upper,dist,round=TRUE,...)
{ D=length(par) # dimension
  step=dist(D,...) # slight step
  if(round) step=round(step)
  par1=par+step
  # return par1 within [lower,upper]:
  return(ifelse(par1<lower,lower,ifelse(par1>upper,upper,par1)))
}
```

The main function (`hclimbing`) receives an initial search point (`par`), an evaluation function (named now `fun` for coherence purposes with `optim`, see next section), `lower` and `upper` bounds, a `control` object and optimization `type`. The `control` list is used to set the maximum number of iterations (`control$maxit`) and monitor the search, showing the solutions searched every `control$REPORT` iterations.

The change function (`hchange`) produces a small perturbation over a given solution (`par`). New values are achieved by adopting a given random distribution function (`dist`). Given the goal of getting a small perturbation, the normal (Gaussian) distribution $\mathcal{N}(0,1)$ is adopted in this book, corresponding to the arguments `dist=rnorm, mean=0, sd=1`. This means that in most cases very small changes are performed (with an average of zero), although large deviations might occur in a few cases. The new solution is kept within the range [`lower`,`upper`] by using the useful `ifelse`(*condition*, *yes*, *no*) R function

that performs a conditional element selection (returns the values of *yes* if the *condition* is true, else returns the elements of *no*). For example, the result of `x=c(-1,4,9);sqrt(ifelse(x>=0,x,NA))` is `NA 2 3`.

For demonstration purposes, the next R code executes ten iterations of a hill climbing search for the **sum of bits** task (Sect. 1.7), starting from the origin (all zero) solution:

```
### sumbits-hill.R file ###

source("hill.R") # load the hill climbing methods

# sum a raw binary object x (evaluation function):
sumbin=function(x) sum(x)

# hill climbing for sum of bits, one run:
D=8 # dimension
s=rep(0,D) # c(0,0,0,0,...)
C=list(maxit=10,REPORT=1) # maximum of 10 iterations
ichange=function(par,lower,upper) # integer change
{ hchange(par,lower,upper,rnorm,mean=0,sd=1) }

hclimbing(s,sumbin,change=ichange,lower=rep(0,D),upper=rep(1,D),
          control=C,type="max")
```

One example of such execution is:

```
> source("sumbits-hill.R")
i: 1 s: 0 0 0 0 0 0 0 0 f: 0 s' 0 0 0 1 0 0 1 0 f: 2
i: 2 s: 0 0 0 1 0 0 1 0 f: 2 s' 0 0 0 1 0 0 1 0 f: 2
i: 3 s: 0 0 0 1 0 0 1 0 f: 2 s' 0 0 0 0 1 1 0 0 f: 2
i: 4 s: 0 0 0 1 0 0 1 0 f: 2 s' 1 0 0 1 0 0 1 0 f: 3
i: 5 s: 1 0 0 1 0 0 1 0 f: 3 s' 0 0 0 0 1 0 0 1 f: 2
i: 6 s: 1 0 0 1 0 0 1 0 f: 3 s' 1 1 0 1 1 0 0 1 f: 5
i: 7 s: 1 1 0 1 1 0 0 1 f: 5 s' 0 1 0 1 0 1 0 0 f: 3
i: 8 s: 1 1 0 1 1 0 0 1 f: 5 s' 0 0 0 1 0 1 1 0 f: 3
i: 9 s: 1 1 0 1 1 0 0 1 f: 5 s' 1 0 1 1 1 0 1 0 f: 5
i: 10 s: 1 1 0 1 1 0 0 1 f: 5 s' 1 1 0 1 1 1 1 1 f: 7
best: 1 1 0 1 1 1 1 1 f: 7
```

The **sum of bits** is an easy task and after ten iterations the hill climbing method achieves a solution that is very close to the optimum ($f = 8$).

The next code performs a hill climbing for the **bag prices** ($D = 5$) and **sphere** tasks ($D = 2$):

```
### bs-hill.R file ###

source("hill.R") # load the hill climbing methods
source("functions.R") # load the profit function

# hill climbing for all bag prices, one run:
D=5; C=list(maxit=10000,REPORT=10000) # 10000 iterations
s=sample(1:1000,D,replace=TRUE) # initial search
ichange=function(par,lower,upper) # integer value change
```

```
{ hchange(par,lower,upper,rnorm,mean=0,sd=1) }
hclimbing(s,profit,change=ichange,lower=rep(1,D),
          upper=rep(1000,D),control=C,type="max")

# hill climbing for sphere, one run:
sphere=function(x) sum(x^2)
D=2; C=list(maxit=10000,REPORT=10000)
rchange=function(par,lower,upper) # real value change
{ hchange(par,lower,upper,rnorm,mean=0,sd=0.5,round=FALSE) }

s=runif(D,-5.2,5.2) # initial search
hclimbing(s,sphere,change=rchange,lower=rep(-5.2,D),
          upper=rep(5.2,D),control=C,type="min")
```

An execution example is:

```
> source("bs-hill.R")
i: 1 s: 136 332 716 748 781 f: 28946 s' 135 332 716 749 781 f:
    28890
i: 10000 s: 188 338 743 770 812 f: 31570 s' 189 336 742 769 812
    f: 31387
best: 188 338 743 770 812 f: 31570
i: 1 s: 1.884003 4.549536 f: 24.24775 s' 2.142131 4.349898 f: 23
    .51034
i: 10000 s: 0.001860534 0.009182373 f: 8.777755e-05 s' 0.5428469
     -0.304862 f: 0.3876236
best: 0.001860534 0.009182373 f: 8.777755e-05
```

Using 10,000 iterations, the hill climbing search improved the solution from 28,890 to 31,570 (**bag prices**) and from 23.51 to 0.00 (**sphere**). For demonstrative purposes, Fig. 4.2 shows the searched "down the hill" (best) points for the **sphere** task.

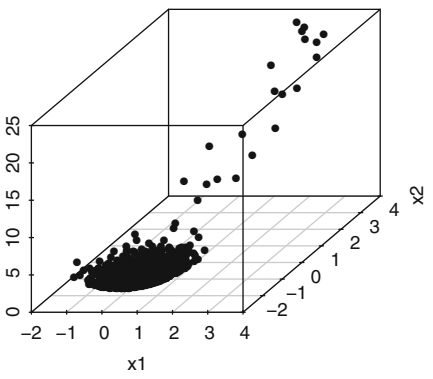

Fig. 4.2 Example of hill climbing search (only best "down the hill" points are shown) for **sphere** and $D = 2$

4.3 Simulated Annealing

Simulated annealing is a variation of the hill climbing technique that was proposed in the 1980s and that is inspired in the annealing phenomenon of metallurgy, which involves first heating a particular metal and then perform a controlled cooling (Luke 2012). This single-state method differs from the hill climbing search by adopting a control temperature parameter (T) that is used to compute the probability of accepting inferior solutions. In contrast with the stochastic hill climbing, which adopts a fixed value for T, the simulated annealing uses a variable temperature value during the search. The method starts with a high temperature and then gradually decreases (cooling process) the control parameter until a small value is achieved (similar to the hill climbing). Given that simulated annealing is a single-state method, it is described in this chapter. However, it should be noted that for high temperatures the method is almost equivalent to Monte Carlo search, thus behaving more like a global search method (in particular if the *change* function is set to perform high changes), while for low temperatures the method is similar to the hill climbing local search (Michalewicz et al. 2006).

This book adopts the simulated annealing implementation of the `optim` R function, which only performs minimization tasks and that executes several optimization methods by setting argument `method`, such as:

- `"Nelder-Mead"`—Nelder and Mead or downhill simplex method;
- `"BFGS"`—a quasi-Newton method;
- `"CG"`—conjugate gradients method;
- `"L-BFGS-B"`—modification of the BFGS method with lower and upper bounds; and
- `"SANN"`—simulated annealing.

Algorithm 3 presents the pseudo-code of the simulated annealing implementation, which is based on the variant proposed by Bélisle (1992). This implementation includes three search parameters: $maxit$—the maximum number of iterations; `temp` (T)—the initial temperature; and $tmax$—the number of evaluations at each temperature. By default, the values for the control parameters are $maxit = 10{,}000$, $T = 10$, and $tmax = 10$. Also, new search points are generated using a Gaussian Markov kernel with a scale proportional to the temperature. Nevertheless, these defaults can be changed by setting two optim arguments: the `control` list and `gr` (*change*) function. The last argument is useful for solving combinatorial problems, i.e., when the representation of the solution includes discrete values. The `optim` function returns a list with several components, such as `$par`—the optimized values and `$value`—the evaluation of the best solution.

Similarly to the hill climbing demonstration, the `sumbits-sann.R` file executes ten iterations of the simulated annealing for the **sum of bits** task:

```
### sumbits-sann.R file ###
source("hill.R") # get hchange function
# sum a raw binary object x (evaluation function):
minsumbin=function(x) (length(x)-sum(x)) # optim only minimizes!
```

Algorithm 3 Simulated annealing search as implemented by the `optim` function

Inputs: S, f, C ▷ S is the initial solution, f is the evaluation function, C contains control parameters ($maxit$, T and $tmax$)
$maxit \leftarrow get\_maxit(C)$ ▷ maximum number of iterations
$T \leftarrow get\_temperature(C)$ ▷ temperature, should be a high number
$tmax \leftarrow get\_tmax(C)$ ▷ number of evaluations at each temperature
$fs \leftarrow f(S)$ ▷ evaluation of S
$B \leftarrow S$ ▷ best solution
$i \leftarrow 0$ ▷ i is the number of iterations of the method
while $i < maxit$ **do** ▷ $maxit$ is the termination criterion
  for $j = 1 \rightarrow tmax$ **do** ▷ cycle j from 1 to $tmax$
    $S' \leftarrow change(S, C)$ ▷ new solution (might depend on T)
    $fs' \leftarrow f(S')$ ▷ evaluation of S'
    $r \leftarrow \mathscr{U}(0, 1)$ ▷ random number, uniform within [0, 1]
    $p \leftarrow \exp\left(\frac{fs'-fs}{T}\right)$ ▷ probability $P(S, S', T)$ (Metropolis function)
    if $fs' < fs \vee r < p$ **then** $S \leftarrow S'$ ▷ accept best solution or worst if $r < p$
    end if
    if $fs' < fs$ **then** $B \leftarrow S'$
    end if
    $i \leftarrow i + 1$
  end for
  $T \leftarrow \frac{T}{\log(i/tmax) \times tmax + \exp(1)}$ ▷ cooling step (decrease temperature)
end while
Output: B ▷ the best solution

```
# SANN for sum of bits, one run:
D=8 # dimension
s=rep(0,D) # c(0,0,0,0,...)
C=list(maxit=10,temp=10,tmax=1,trace=TRUE,REPORT=1)
bchange=function(par) # binary change
{ D=length(par)
  hchange(par,lower=rep(0,D),upper=rep(1,D),rnorm,mean=0,sd=1)
}
s=optim(s,minsumbin,gr=bchange,method="SANN",control=C)
cat("best:",s$par,"f:",s$value,"(max: fs:",sum(s$par),")\n")
```

Given that `optim` only performs minimization, the evaluation function needs to be adapted (as discussed in Sect. 1.3). In this example, it was set to have a minimum of zero. Also, given that `method="SANN"` does not include lower and upper bonds, it is the responsibility of the change function (`gr`) to not generate unfeasible solutions. In this case, the auxiliary binary change function (`bchange`) uses the `hchange` function (from file `hill.R`) to set the 0 and 1 bounds for all D values. An execution example of file `sumbits-sann.R` is:

```
> source("sumbits-sann.R")
sann objective function values
initial       value 8.000000
iter        1 value 7.000000
iter        2 value 2.000000
```

```
iter        3 value 2.000000
iter        4 value 1.000000
iter        5 value 1.000000
iter        6 value 1.000000
iter        7 value 1.000000
iter        8 value 1.000000
iter        9 value 1.000000
final         value 1.000000
sann stopped after 9 iterations
best: 1 1 1 1 1 1 1 0 f: 1 (max: fs: 7 )
```

The simulated annealing search is also adapted for **bag prices** ($D = 5$) and **sphere** tasks ($D = 2$), by setting $maxit = 10{,}000$, $T = 1{,}000$ and $tmax = 10$ (file `bs-sann.R`):

```
### bs-sann.R file ###

source("hill.R") # load the hchange method
source("functions.R") # load the profit function
eval=function(x) -profit(x) # optim minimizes!

# hill climbing for all bag prices, one run:
D=5; C=list(maxit=10000,temp=1000,trace=TRUE,REPORT=10000)
s=sample(1:1000,D,replace=TRUE) # initial search
ichange=function(par) # integer value change
{ D=length(par)
  hchange(par,lower=rep(1,D),upper=rep(1000,D),rnorm,mean=0,
   sd=1)
}
s=optim(s,eval,gr=ichange,method="SANN",control=C)
cat("best:",s$par,"profit:",abs(s$value),"\n")

# hill climbing for sphere, one run:
sphere=function(x) sum(x^2)
D=2; C=list(maxit=10000,temp=1000,trace=TRUE,REPORT=10000)

s=runif(D,-5.2,5.2) # initial search
# SANN with default change (gr) function:
s=optim(s,sphere,method="SANN",control=C)
cat("best:",s$par,"f:",s$value,"\n")
```

An example execution of file `bs-sann.R` is:

```
> source("bs-sann.R")
sann objective function values
initial       value -35982.000000
final         value -39449.000000
sann stopped after 9999 iterations
best: 293 570 634 606 474 profit: 39449
sann objective function values
initial       value 21.733662
final         value 1.243649
sann stopped after 9999 iterations
best: -0.6856747 -0.8794882 f: 1.243649
```

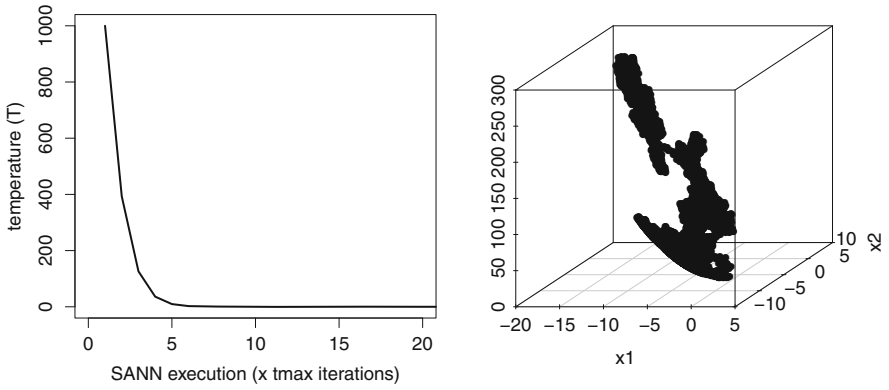

Fig. 4.3 Example of the temperature cooling (*left*) and simulated annealing search (*right*) for **sphere** and $D = 2$

In this execution, the initial solution was improved \$3,467 (**bag prices**) and 20.49 (**sphere**). For the last task execution, Fig. 4.3 shows the evolution of the temperature values (left) and points searched (right). While the initial point is within the $[-5.2, 5.2]$ range, the default and unbounded Gaussian Markov *change* function searches for several solutions outside the initial range. However, as the search proceeds, more solutions tend to converge to the origin point, which is the optimum.

4.4 Tabu Search

Tabu search was created by Glover (1986) and uses the concept of "memory" to force the search into new areas. The algorithm is a variation of the hill climbing method that includes in a tabu list of length L, which stores the most recent solutions that become "tabu" and thus cannot be used when selecting a new solution. The intention is to keep a short-term memory of recent changes, preventing future moves from deleting these changes (Brownlee 2011). Similarly to the hill climbing method, the search of solutions within the neighborhood of the current solution (function *change*) can be deterministic, including the entire neighborhood, or stochastic (e.g., small random perturbation). Also, the tabu search algorithm is deterministic, except if a stochastic change is adopted (Michalewicz et al. 2007). Hence, this method is also centered within the deterministic/stochastic factor of analysis in Fig. 1.2.

There are extensions of the original tabu search method. Tabu search was devised for discrete spaces and combinatorial problems (e.g., traveling salesman problem). However, the method can be extended to work with real valued points if a similarity function is used to check if a solution is very close to a member of the tabu list (Luke 2012). Other extensions to the original method include adding other types

Algorithm 4 Tabu search

```
1: Inputs: S, f, C   ▷ S is the initial solution, f is the evaluation function, C contains control parameters (maxit, L and N)
2: maxit ← get_maxit(C)   ▷ maximum number of iterations
3: L ← get_L(C)   ▷ length of the tabu list
4: N ← get_N(C)   ▷ number of neighbor configurations to check at each iteration
5: List ← {}   ▷ tabu list (first in, first-out queue)
6: i ← 0   ▷ i is the number of iterations of the method
7: while i < maxit do   ▷ maxit is the termination criterion
8:     for j = 1 → N do   ▷ cycle j from 1 to N
9:         S′ ← change(S, C)   ▷ new solution
10:        CList ← {}   ▷ candidate list
11:        if S′ ∉ List then CList ← CList ∪ S′   ▷ add S′ into CList
12:        end if
13:    end for
14:    S′ ← best(CList, f)   ▷ get best candidate solution
15:    if isbest(S′, S, f) then   ▷ if S′ is better than S
16:        List ← List ∪ S′   ▷ enqueue S′ into List
17:        if length(List) > L then dequeue(L)   ▷ remove oldest element
18:        end if
19:        S ← S′   ▷ set S as the best solution S′
20:    end if
21:    i ← i + 1
22: end while
23: Output: S   ▷ the best solution
```

of memory structures, such as: intermediate-term, to focus the search in promising areas (intensification phase); and long-term, to promote a wider exploration of the search space (diversification phase). More details can be found in Glover (1990).

Algorithm 4 presents a simple implementation of tabu search, in an adaptation of the pseudo-code presented in Brownlee (2011). The algorithm combines a steepest ascent hill climbing search with a short tabu memory and includes three control parameters: $maxit$—the maximum number of iterations; L—the length of the tabu list; and N—the number of neighborhood solutions searched at each iteration.

In this section, the `tabuSearch` function is adopted (as implemented in the package under the same name). This function only works with binary strings, using a stochastic generation of new solutions and assuming a maximization goal. Also, it implements a three memory scheme, under the sequence of stages: preliminary search (short-term), intensification (intermediate-term), and diversification (long-term). Some relevant arguments are:

- `size`—length of the binary solution (L_S);
- `iters`—maximum number of iterations ($maxit$) during the preliminary stage;
- `objFunc`—evaluation function (f) to be maximized;
- `config`—initial solution (S);
- `neigh`—number of neighbor configurations (N) searched at each iteration;

- `listSize`—length of the tabu list (L);
- `nRestarts`—maximum number of restarts in the intensification stage; and
- `repeatAll`—number of times to repeat the search.

The `tabuSearch` function returns a list with elements such as: `$configKeep`—matrix with stored solutions; and `$eUtilityKeep`—vector with the respective evaluations.

To demonstrate this method, file `binary-tabu.R` optimizes the binary tasks of Sect. 1.7:

```
### binary-tabu.R file ###
library(tabuSearch) # load tabuSearch package

# tabu search for sum of bits:
sumbin=function(x) (sum(x)) # sum of bits
D=8 # dimension
s=rep(0,D) # c(0,0,0,0,...)
s=tabuSearch(D,iters=2,objFunc=sumbin,config=s,neigh=2,
             listSize=4,nRestarts=1)
b=which.max(s$eUtilityKeep) # best index
cat("best:",s$configKeep[b,],"f:",s$eUtilityKeep[b],"\n")

# tabu search for max sin:
intbin=function(x) sum(2^(which(rev(x==1))-1))
maxsin=function(x) # max sin (explained in Chapter 3)
{ D=length(x);x=intbin(x); return(sin(pi*(as.numeric(x))/(2^D)))
    }
D=8
s=rep(0,D) # c(0,0,0,0,...)
s=tabuSearch(D,iters=2,objFunc=maxsin,config=s,neigh=2,
             listSize=4,nRestarts=1)
b=which.max(s$eUtilityKeep) # best index
cat("best:",s$configKeep[b,],"f:",s$eUtilityKeep[b],"\n")
```

An example of file `binary-tabu.R` execution is:

```
> source("binary-tabu.R")
best: 0 1 1 1 0 1 1 1 f: 6
best: 1 0 0 1 0 1 1 0 f: 0.9637761
```

While few iterations were used, the method optimized solutions close to the optimum values ($f = 8$ for **sum of bits** and $f = 1$ for **max sin**).

The tabu search is also demonstrated for the **bag prices** integer task ($D = 5$). Given that `tabuSearch()` imposes some restrictions, adaptations are needed. The most relevant is the use of a binary representation, with ten digits per integer value (to cover the {\$1,\$2,...,\$1,000} range). Also, since the associated search space includes infeasible solutions, a simply death penalty scheme is used (Sect. 1.5), where $f = -\infty$ if any price is above \$1,000. Finally, given that `tabuSearch()` does not include extra arguments to be passed to the evaluation

function, the arguments `D` and `Dim` need to be explicitly defined before tabu search method is executed. The adapted R code (file `bag-tabu.R`) is:

```
### bag-tabu.R file ###
library(tabuSearch) # load tabuSearch package
source("functions.R") # load the profit function

# tabu search for bag prices:
D=5 # dimension (number of prices)
MaxPrice=1000
Dim=ceiling(log(MaxPrice,2)) # size of each price (=10)
size=D*Dim # total number of bits (=50)
s=sample(0:1,size,replace=TRUE) # initial search

intbin=function(x) # convert binary to integer
{ sum(2^(which(rev(x==1))-1)) } # explained in Chapter 3
bintbin=function(x) # convert binary to D prices
{ # note: D and Dim need to be set outside this function
  s=vector(length=D)
  for(i in 1:D) # convert x into s:
  { ini=(i-1)*Dim+1;end=ini+Dim-1
    s[i]=intbin(x[ini:end])
  }
  return(s)
}
bprofit=function(x) # profit for binary x
{ s=bintbin(x)
  if(sum(s>MaxPrice)>0) f=-Inf # death penalty
  else f=profit(s)
  return(f)
}

cat("initial:",bintbin(s),"f:",bprofit(s),"\n")
s=tabuSearch(size,iters=100,objFunc=bprofit,config=s,neigh=4,
    listSize=16,nRestarts=1)
b=which.max(s$eUtilityKeep) # best index
cat("best:",bintbin(s$configKeep[b,]),"f:",s$eUtilityKeep[b],
  "\n")
```

This code introduces the `ceiling()` R function that returns the closest upper integer. An execution example of file `bag-tabu.R` is:

```
> source("bag-tabu.R")
initial: 621 1005 880 884 435 f: -Inf
best: 419 428 442 425 382 f: 43050
```

In this case, the tabu search managed to improve an infeasible initial search point into a solution that is only 2 % far from the optimum value ($f = 43{,}899$).

4.5 Comparison of Local Search Methods

The comparison of optimization methods is not a trivial task. The *no free lunch* theorem (Wolpert and Macready 1997) states that all search methods have a similar global performance when compared over all possible functions. However, the set of all functions includes random and deceptive ones, which often are not interesting to be optimized. A constructive response to the theorem is to define a subset of "searchable" functions where the theorem does not hold, comparing the average performance of a several algorithms on this subset (Mendes 2004). Yet, even if an interesting subset of functions and methods is selected, there are other relevant issues for a robust comparison: how to tune the control parameters of a method (e.g., T of simulated annealing) and which performance metrics and statistical tests should be adopted.

Hence, rather than presenting a complete comparison, this section presents an R code example of how optimization methods can be compared, assuming some reasonable assumptions (if needed, these can be changed by the readers). The example uses one task, **rastrigin** benchmark with $D = 20$ (which is the most difficult real value task from Sect. 1.7) and compares three methods: Monte Carlo (Sect. 3.4), hill climbing (Sect. 4.2), and simulated annealing (Sect. 4.3). To avoid any bias towards a method, the same *change* function is used for hill climbing and simulated annealing strategies and the default `optim` values ($T = 10, tmax = 10$) are adopted for the last search strategy. The same maximum number of iterations ($maxit = 10{,}000$) is used for all methods. Rather than comparing just the final best value, the comparison is made throughout the search execution. Some measures of search execution can be deceptive, such as time elapsed, which might be dependent on the processor workload, or number of iterations, whose computational effort depends on the type of search. Thus, the best value is stored for each evaluation function (from 1 to 10,000), as sequentially called by the method. Finally, a total of 50 runs are executed for each method, with the initial solutions randomly generated within the range $[-5.2, 5.2]$. To aggregate the results, the average and respective t-student 95 % confidence intervals curves are computed for the best values. The comparison code outputs a PDF result file (file `compare.R`):

```
### compare.R file ###

source("hill.R") # get hchange
source("blind.R") # get fsearch
source("montecarlo.R") # get mcsearch
library(rminer) # get meanint

# comparison setup:
crastrigin=function(x)
{ f=10*length(x)+sum(x^2-10*cos(2*pi*x))
  # global assignment code: <<-
  EV<<-EV+1 # increase evaluations
  if(f<BEST) BEST<<-f
```

```
  if(EV<=MAXIT) F[EV]<<-BEST
  return(f)
}
Runs=50; D=20; MAXIT=10000
lower=rep(-5.2,D);upper=rep(5.2,D)
rchange1=function(par,lower,upper) # change for hclimbing
{ hchange(par,lower=lower,upper=upper,rnorm,
          mean=0,sd=0.5,round=FALSE) }
rchange2=function(par)             # change for optim
{ hchange(par,lower=lower,upper=upper,rnorm,
          mean=0,sd=0.5,round=FALSE) }
CHILL=list(maxit=MAXIT,REPORT=0)
CSANN=list(maxit=MAXIT,temp=10,trace=FALSE)
Methods=c("monte carlo","hill climbing","simulated annealing")

# run all optimizations and store results:
RES=vector("list",length(Methods)) # all results
for(m in 1:length(Methods))
   RES[[m]]=matrix(nrow=MAXIT,ncol=Runs)
for(R in 1:Runs) # cycle all runs
{ s=runif(D,-5.2,5.2) # initial search point
  EV=0; BEST=Inf; F=rep(NA,MAXIT) # reset these vars.
  # monte carlo:
  mcsearch(MAXIT,lower=lower,upper=upper,FUN=crastrigin)
  RES[[1]][,R]=F
  # hill climbing:
  EV=0; BEST=Inf; F=rep(NA,MAXIT)
  hclimbing(s,crastrigin,change=rchange1,lower=lower,
            upper=upper,control=CHILL,type="min")
  RES[[2]][,R]=F
  # SANN:
  EV=0; BEST=Inf; F=rep(NA,MAXIT)
  optim(s,crastrigin,method="SANN",gr=rchange2,control=CSANN)
  RES[[3]][,R]=F
}

# aggregate (average and confidence interval) results:
AV=matrix(nrow=MAXIT,ncol=length(Methods))
CI=AV
for(m in 1:length(Methods))
{
 for(i in 1:MAXIT)
 {
  mi=meanint(RES[[m]][i,])
  AV[i,m]=mi$mean;CI[i,m]=mi$int
 }
}

# show comparative PDF graph:

# plot a nice confidence interval bar:
confbar=function(x,ylower,yupper,K=100)
{ segments(x-K,yupper,x+K)
```

```
  segments(x-K,ylower,x+K)
  segments(g2,ylower,g2,yupper)
}

pdf("comp-rastrigin.pdf",width=5,height=5)
par(mar=c(4.0,4.0,0.1,0.6)) # reduce default plot margin
MIN=min(AV-CI);MAX=max(AV+CI)
# 10.000 are too much points, thus two grids are used
# to improve clarity:
g1=seq(1,MAXIT,length.out=1000) # grid for lines
g2=seq(1,MAXIT,length.out=11) # grid for confbar
plot(g1,AV[g1,3],ylim=c(MIN,MAX),type="l",lwd=2,
     ylab="average best",xlab="number of evaluations")
confbar(g2,AV[g2,3]-CI[g2,3],AV[g2,3]+CI[g2,3])
lines(g1,AV[g1,2],lwd=2,lty=2)
confbar(g2,AV[g2,2]-CI[g2,2],AV[g2,2]+CI[g2,2])
lines(g1,AV[g1,1],lwd=2,lty=3)
confbar(g2,AV[g2,1]-CI[g2,1],AV[g2,1]+CI[g2,1])
legend("topright",legend=rev(Methods),lwd=2,lty=1:3)
dev.off() # close the PDF device
```

Given that some optimization functions (e.g., `optim`) are restrictive in terms of the parameters that can be used as inputs, the evaluation function is adapted to perform global assignments (operator `<<-`, Sect. 2.3) to the number of evaluations (`EV`), best value (`BEST`), and vector of best function values (`F`). The results are stored in a vector list of size 3, each element with a matrix $maxit \times runs$. Two similar *change* functions are defined, since `optim` does not allow the definition of additional arguments to be passed to `gr`. The code introduces some new R functions:

- `meanint` (from package `rminer`)—computes the mean and t-student confidence intervals;
- `segments`—draws a segment;
- `par`—sets graphical parameters used by `plot`; and
- `lines`—joints points into line segments.

The result execution of file `compare.R` is presented in Fig. 4.4. Initially, all methods present a fast and similar convergence. However, after around 1,000 evaluations, the hill climbing and simulated annealing methods start to outperform the random search (Monte Carlo). The confidence interval bars show that after around 4,000 evaluations, the local search methods are statistically better when compared with Monte Carlo. In this experiment, simulated annealing produces only a slight best average result and the differences are not statistically significant when compared with hill climbing (since confidence intervals overlap).

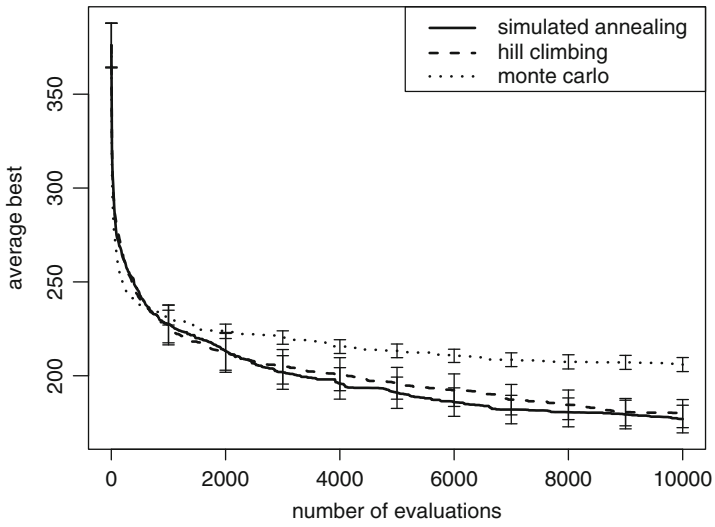

Fig. 4.4 Local search comparison example for the **rastrigin** task ($D = 20$)

4.6 Command Summary

ceiling()	returns the closest upper integer
hchange()	slight random change of a vector (chapter file "hill.R")
hclimbing()	standard hill climbing search (chapter file "hill.R")
ifelse()	conditional element selection
lines()	joints points into line segments
meanint	computes the mean and t-student confidence intervals (package rminer)
optim()	general-purpose optimization (includes simulated annealing)
par()	set or query graphical parameters used by plot()
rminer	package for simpler use of data mining (classification and regression) methods
segments	draws a segment line
tabuSearch	package for tabu search
tabuSearch()	tabu search for binary string maximization (package tabuSearch)

4.7 Exercises

4.1. Adapt the function `hclimbing` function to accept an additional control parameter (N). When $N = 0$, the function should execute the standard hill climbing, while when $N > 0$, the function should implement the steepest ascent hill climbing method. This last variant works by searching first N neighbor solutions within each iteration, in order to select the best new solution to be compared with current search point.

4.2. Explore the optimization of the binary **max sin** task with a higher dimension ($D = 16$), under hill climbing, simulated annealing, and tabu search methods. Use the zero vector as the starting point and a maximum of 20 iterations. Show the optimized solutions and evaluation values.

4.3. Execute the optimization of the **rastrigin** function ($D = 8$) with the `tabuSearch` function. Adopt a binary representation such that each dimension value is encoded into 8 bits, denoting any of the 256 regular levels within the range $[-5.2, 5.2]$. Use the control parameters: $maxit = 500$, $N = 8$, $L = 8$ and `nRestarts=1` and a randomly generated initial point.

Chapter 5
Population Based Search

5.1 Introduction

In previous chapter, several local based search methods were presented, such as hill climbing, simulated annealing, and tabu search. All these methods are single-state, thus operating their effort around the neighborhood of a current solution. This type of search is simple and quite often efficient (Michalewicz et al. 2006). However, there is another interesting class of search methods, known as population based search, that use a pool of candidate solutions rather than a single search point. Thus, population based methods tend to require more computation when compared with simpler local methods, although they tend to work better as global optimization methods, quickly finding interesting regions of the search space (Michalewicz and Fogel 2004).

As shown in Fig. 5.1, population based methods tend to explore more distinct regions of the search space, when compared with single-state methods. As consequence, more diversity can be reached in terms of setting new solutions, which can be created not only by slightly changing each individual search point but also by combining attributes related with two (or more) search points.

The main difference between population based methods is set in terms of: how solutions are represented and what attributes are stored for each search point; how new solutions are created; and how the best population individuals are selected. Most population based methods are naturally inspired (Luke 2012). Natural phenomena such as genetics, natural selection, and collective behavior of animals have led to optimization techniques such as genetic and evolutionary algorithms, genetic programming, estimation of distribution, differential evolution, and particle swarm optimization. This chapter describes all these methods and examples of their applications using the R tool.

P. Cortez, *Modern Optimization with R*, Use R!, DOI 10.1007/978-3-319-08263-9_5

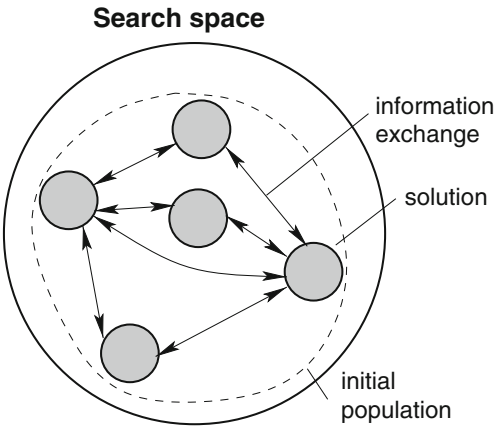

Fig. 5.1 Example of a population based search strategy

5.2 Genetic and Evolutionary Algorithms

Evolutionary computation denotes several optimization algorithms inspired in the natural selection phenomenon and that include a population of competing solutions. Although it is not always clearly defined, the distinction among these methods is mostly based on how to represent a solution and how new solutions are created. Genetic algorithms were proposed by Holland (1975). The original method worked only on binary representations and adopted massively the crossover operator for generating new solutions. More recently, the term evolutionary algorithm was adopted to address genetic algorithm variants that include real value representations and that adopt flexible genetic operators, ranging from heavy use of crossover to only mutation changes (Michalewicz 1996).

There is a biological terminology associated with evolutionary computation methods (Luke 2012). For instance, a candidate solution is often termed *individual*, while *population* denotes a pool of individuals. The *genotype*, *genome*, or *chromosome* denotes the individual data structure representation. A *gene* is a value position in such representation and an *allele* is a particular value for a gene. The evaluation function is also known as *fitness* and *phenotype* represents how the individual operates during fitness assessment. The creation of new solutions is called *breeding* and occurs due to the application of *genetic* operators, such as *crossover* and *mutation*. Crossover involves selecting two or more *parent* solutions in order to generate *children*, while mutation often performs a slight change to a single individual.

This book adopts the genetic/evolutionary algorithm as implemented by the `genalg` package (Lucasius and Kateman 1993). The package handles minimization tasks and contains two relevant functions: `rbga.bin`, for binary chromosomes; and `rbga`, for real value representations. The `genalg` main pseudo-code for `rbga.bin()` and `rbga()` is presented in Algorithm 5 and is detailed in the next paragraphs.

Algorithm 5 Genetic/evolutionary algorithm as implemented by the `genalg` package

Inputs: f, C ▷ f is the evaluation (fitness) function, C includes control parameters
$P \leftarrow initialization(C)$ ▷ random initial population
$N_P \leftarrow get\_population\_size(C)$ ▷ population size
$E \leftarrow get\_elitism(C)$ ▷ number of best individuals kept (elitism)
$i \leftarrow 0$ ▷ i is the number of iterations of the method
while $i < maxit$ **do**
  $F_P \leftarrow f(P)$ ▷ evaluate current population
  $P_E \leftarrow best(P, F_P, E)$ ▷ set the elitism population (lowest E fitness values)
  $Parents \leftarrow selectparents(P, F_P, N_P - E)$ ▷ select $N_P - E$ parents from current population
  $Children \leftarrow crossover(Parents, C)$ ▷ create $N_P - E$ children solutions
  $Children \leftarrow mutation(Children, maxit, i)$ ▷ apply the mutation operator to the children
  $P \leftarrow E \cup Children$ ▷ set the next population
  $i \leftarrow i + 1$
end while
Output: P ▷ last population

The *initialization* function creates a random population of N_P individuals (argument `popSize`), under a particular distribution (uniform or other). Each individual contains a fixed length chromosome (with L_S genes), defined by `size` for `rbga.bin` or by the length of the lower bound values (`stringMin`) for `rbga`. For `rbga`, the initial values are randomly generated within the lower (`stringMin`) and upper (`stringMax`) bounds. The optional argument `suggestions` can be used to include *a priori* knowledge, by defining an explicit initial matrix with up to N_P solutions. The function `rbga.bin()` includes also the argument `zeroToOneRatio` that denotes the probability for choosing a zero for mutations and population initialization. The ideal number of individuals (N_P, argument `popSize`) is problem dependent. Often, this value is set taking into account the chromosome length (L_S), computational effort and, preliminary experiments. Common population size values are $N_P \in \{20, 50, 100, 200, 500, 1{,}000, 2{,}000\}$.

The algorithm runs for $maxit$ generations (argument `iters`). If an *elitism* scheme is adopted ($E > 0$, value set by argument `elitism`), then the best E individuals from the current population always pass to the next generation (the default is `elitism`=20 % of the population size). The remaining next population individuals are created by applying a crossover and then a mutation. The pseudo *selectparents* function aims at the selection of $N_P - E$ parents from the current population, in order to apply the crossover operator. The `rbga` algorithm performs an uniform random choice of parents, while `rbga.bin` executes a probabilistic selection of the fittest individuals. This selection works by ranking first the current population (according to the fitness values) and then performing a random selection of parents according to a density normal distribution. The respective R code for generating such probabilities is `dnorm(1:popSize, mean = 0, sd = (popSize/3))`. For instance, if $N_P = 4$, then the probabilities for choosing

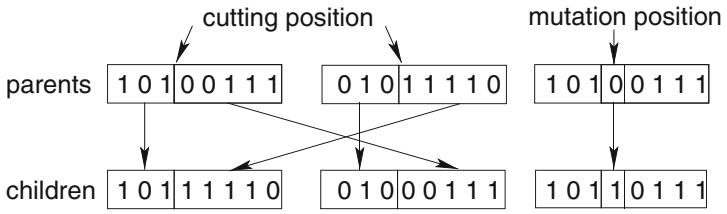

Fig. 5.2 Example of binary one-point crossover (*left*) and mutation (*right*) operators

parents from the ranked population are $(0.23, 0.1, 0.02, 0.0)$, where 0.23 denotes the probability for the best individual. It should be noted that this probabilistic selection is also known as roulette wheel selection (Goldberg and Deb 1991). There are other selection schemes, such as tournament selection, which is explained in Sect. 6.4.

To create new individuals, both `rbga` and `rbga.bin` adopt the same one-point crossover, which was originally proposed in Holland (1975). The original one-point operator works by first selecting two parents and a random cutting position and then creating two children, each with distinct portions of the parents, as exemplified in the left of Fig. 5.2. As a specific `genalg` package implementation, only the first one-point crossover child is inserted into the new population and thus $N_P - E$ crossover operations are executed (Sect. 7.2 shows an implementation that inserts the two children generated from a crossover). Next, a mutation operator is executed over the children, where each gene can be changed with a small probability (set by `mutationChange`). By default, `mutationChange` is set to `1/(size+1)`. Once a gene is mutated, the new value is set differently according to the chromosome representation type. In the binary version, the new bit value is randomly set taking into account the `zeroToOneRatio` value. The right of Fig. 5.2 shows an example of a binary bit mutation. Mutated real values are obtained by first computing $g' = 0.67 \times r_d \times d_f \times R_g$, where $r_d \in \{-1, 1\}$ is a random direction, $d_f = (maxit - i)/maxit$ is a dampening factor, and $R_g = \max(g) - \min(g)$ is the range of the gene (e.g., $\max(g)$ denotes the upper bound for g, as set by `StringMax`). If g' lies outside the lower (`StringMin`) or upper bounds (`StringMax`), then $g' = \min(g) + \mathscr{U}(0, 1) \times R_g$. More details can be checked by accessing the `rbga` package source code (`> getAnywhere(rbga.bin)` and `> getAnywhere(rbga)`).

The `rbga.bin` and `rba` functions include four additional parameters:

- `monitorFunc`—monitoring function (e.g., could be used to compute useful statistics, such as population diversity measures), applied after each generation;
- `evalFunc`—evaluation (or fitness) function;
- `showSettings`—if `TRUE`, then the genetic algorithm parameters are shown (e.g., N_P); and
- `verbose`—if `TRUE`, then more text about the search evolution is displayed.

The result of executing `rbga.bin` and `rba` is a list with components such as: `$population`—last population; `$evaluations`—last fitness values; `$best`—best value per generation; and `$mean`—mean fitness value per generation. The `genalg` package also includes functions for plotting (`plot.rbga`) and summarizing (`summary.rbga`) results. These functions adopt the useful R feature of S3 scheme of method dispatching, meaning that if `obj` is the object returned by `rbga.bin` or `rbga`, then the simpler call of `plot(obj)` (or `summary(obj)`) will execute `plot.rbga` (or `summary.rbga`).

Given that the `help(rbga.bin)` already provides an example with the **sum of bits** task, the demonstration code (file `bag-genalg.R`) for the genetic algorithm explores the **bag prices** ($D = 5$) problem of Sect. 1.7 (the code was adapted from the example given for the tabu search in Sect. 4.5):

```
### bag-genalg.R file ###
library(genalg) # load genalg package
source("functions.R") # load the profit function

# genetic algorithm search for bag prices:
D=5 # dimension (number of prices)
MaxPrice=1000
Dim=ceiling(log(MaxPrice,2)) # size of each price (=10)
size=D*Dim # total number of bits (=50)
intbin=function(x) # convert binary to integer
{ sum(2^(which(rev(x==1))-1)) } # explained in Chapter 3
bintbin=function(x) # convert binary to D prices
{ # note: D and Dim need to be set outside this function
  s=vector(length=D)
  for(i in 1:D) # convert x into s:
  { ini=(i-1)*Dim+1;end=ini+Dim-1
    s[i]=intbin(x[ini:end])
  }
  return(s)
}
bprofit=function(x) # profit for binary x
{ s=bintbin(x)
  s=ifelse(s>MaxPrice,MaxPrice,s) # repair!
  f=-profit(s) # minimization task!
  return(f)
}
# genetic algorithm execution:
G=rbga.bin(size=size,popSize=50,iters=100,zeroToOneRatio=1,
    evalFunc=bprofit,elitism=1)
# show results:
b=which.min(G$evaluations) # best individual
cat("best:",bintbin(G$population[b,]),"f:",-G$evaluations[b],
    "\n")
pdf("genalg1.pdf") # personalized plot of G results
plot(-G$best,type="l",lwd=2,ylab="profit",xlab="generations")
lines(-G$mean,lty=2,lwd=2)
legend("bottomright",c("best","mean"),lty=1:2,lwd=2)
dev.off()
summary(G,echo=TRUE) # same as summary.rbga
```

Similarly to the tabu search example, 10 binary digits are used to encode each price. The evaluation function (`bprofit`) was adapted with two changes. First, a repair strategy was adopted for handling infeasible prices, where high prices are limited into the `MaxPrice` upper bound. Second, the `profit` function is multiplied by -1, since `genalg` only handles minimization tasks. The last code lines show results in terms of the best solution and summary of the genetic algorithm execution. Also, the code creates a plot showing the evolution of the best and mean profit values. An example of executing file `bag-genalg.R` is:

```
> source("bag-genalg.R")
best: 427 431 425 355 447 f: 43671
GA Settings
  Type                  = binary chromosome
  Population size       = 50
  Number of Generations = 100
  Elitism               = 1
  Mutation Chance       = 0.0196078431372549

Search Domain
  Var 1 = [,]
  Var 0 = [,]

GA Results
  Best Solution : 0 1 1 0 1 0 1 0 1 1 0 1 1 0 1 0 1 1 1 1 0 1 1
      0 1 0 1 0 0 1 0 1 0 1 1 0 0 0 1 1 0 1 1 0 1 1 1 1 1 1
```

Using 100 generations, the genetic algorithm improved the initial population (randomly set) best profit from \$36,745 to \$43,671, with the best fitness value being very close to the optimum (profit of \$43,899). Figure 5.3 shows the best and mean profit values during the 100 generations.

The `rbga` demonstration code is related with the **sphere** ($D = 2$) task (file `sphere-genalg.R`):

```
### sphere-genalg.R file ###
library(genalg) # load genalg

# evolutionary algorithm for sphere:
sphere=function(x) sum(x^2)
D=2
monitor=function(obj)
{ if(i==1)
    { plot(obj$population,xlim=c(-5.2,5.2),ylim=c(-5.2,5.2),
           xlab="x1",ylab="x2",type="p",pch=16,
           col=gray(1-i/maxit))
    }
  else if(i%%K==0) points(obj$population,pch=16,
                          col=gray(1-i/maxit))
  i<<-i+1 # global update
}

maxit=100
K=5 # store population values every K generations
i=1 # initial generation
```

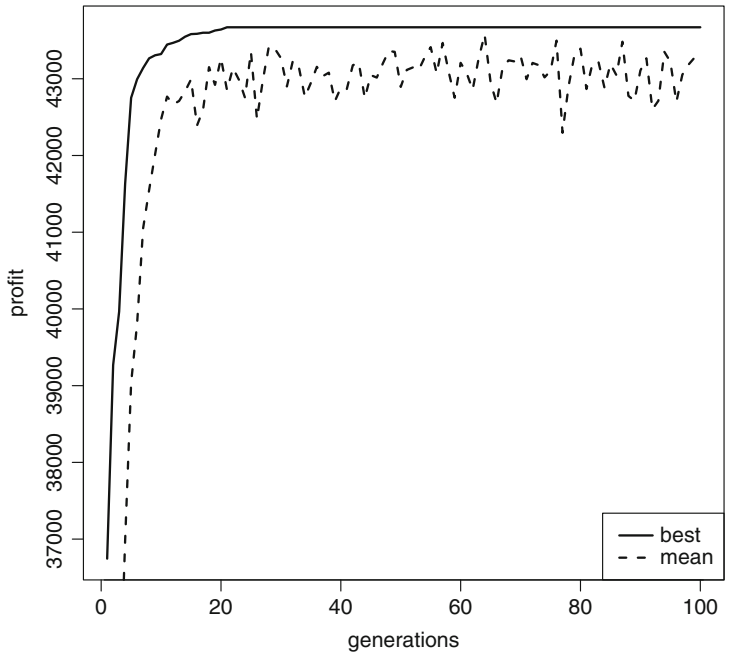

Fig. 5.3 Example of evolution of a genetic algorithm for task **bag prices**

```
# evolutionary algorithm execution:
pdf("genalg2.pdf",width=5,height=5)
set.seed(12345) # set for replicability purposes
E=rbga(rep(-5.2,D),rep(5.2,D),popSize=5,iters=maxit,
       monitorFunc=monitor,evalFunc=sphere)
b=which.min(E$evaluations) # best individual
cat("best:",E$population[b,],"f:",E$evaluations[b],"\n")
dev.off()
```

In this example, the `monitor` argument is used to plot the population of solutions every `K` generations, using a coloring scheme that ranges from light gray (initial population) to dark (last generation). This gradient coloring is achieved using the `gray()` R function, which creates gray colors between 1 (white) and 0 (black). The use of the `set.seed` command (setting the R random seed) is adopted here only for reproducibility purposes, i.e. readers who execute this code should get the same results. The execution of file `sphere-genalg.R` is:

```
> source("sphere-genalg.R")
best: 0.05639766 0.009093091 f: 0.00326338
```

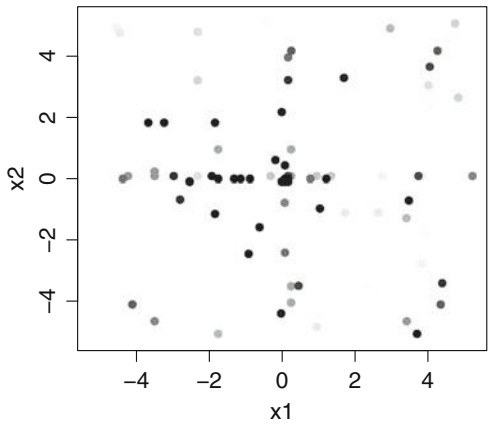

Fig. 5.4 Example of an evolutionary algorithm search for **sphere** ($D = 2$)

Although a very small population is used ($N_P = 5$, minimum value accepted by `rbga`), the evolved solution of $s = (0.009, 0.003)$ and $f = 0.056$ is very close to the optimum ($f = 0$). Figure 5.4 presents the result of the plot, showing that darker points converge towards the optimum point (origin).

5.3 Differential Evolution

Differential evolution is a global search strategy that tends to work well for continuous numerical optimization and that was proposed in Storn and Price (1997). Similarly to genetic and evolutionary algorithms, the method evolves a population of solutions, where each solution is made of a string of real values. The main difference when compared with evolutionary algorithms is that differential evolution uses arithmetic operators to generate new solutions, instead of the classical crossover and mutation operators. The differential mutation operators are based on vector addition and subtraction, thus only working in metric spaces (e.g., boolean, integer or real values) (Luke 2012).

This chapter adopts the differential evolution algorithm as implemented by the `DEoptim` package (Mullen et al. 2011), which performs a minimization goal. The respective pseudo-code is presented in Algorithm 6.

The classical differential mutation starts by first choosing three individuals (s_1, s_2, and s_3) from the population. In contrast with genetic algorithms, these three individuals are randomly selected and selection only occurs when replacing mutated individuals in the population (as shown in Algorithm 6). A trial mutant is created using (Mullen et al. 2011): $s_{m,j} = s_{1,j} + F \times (x_{2,j} - x_{3,j})$, where $F \in [0, 2]$ is a positive scaling factor, often less than 1.0, and j denotes the j-th parameter of the representation of the solution. If the trial mutant values violate the upper or lower bounds, then $s_{m,j}$ is reseted using $s_{m,j} = \max(s_j) - \mathscr{U}(0, 1)(\max(s_j) - min(s_j))$,

Algorithm 6 Differential evolution algorithm as implemented by the `DEoptim` package

Inputs: f, C ▷ f is the evaluation (fitness) function, C includes control parameters
$P \leftarrow initialization(C)$ ▷ set initial population
$B \leftarrow best(P, f)$ ▷ best solution of the initial population
$i \leftarrow 0$ ▷ i is the number of iterations of the method
while not $termination\_criteria(P, f, C, i)$ **do** ▷ DEoptim uses up to three termination criteria
  for each individual $s \in P$ **do** ▷ cycle all population individuals
    $s' \leftarrow mutation(P, C)$ ▷ differential mutation, uses parameters F and CR
    if $f(s') < f(s)$ **then** $P \leftarrow replace(P, s, s')$ ▷ replace s by s' in the population
    end if
    if $f(s') < f(B)$ **then** $B \leftarrow s'$ ▷ minimization goal
    end if
  end for
  $i \leftarrow i + 1$
end while
Output: B, P ▷ best solution and last population

if $s_{m,j} > \max(s_j)$, or $s_{m,j} = \min(s_j) + \mathscr{U}(0,1)(\max(s_j) - \min(s_j))$, if $s_{m,j} < \min(s_j)$, where $\max(s_j)$ and $\min(s_j)$ denote the upper and lower limits for the j-th parameter of the solution. The first trial mutant value (chosen at random) is always computed. Then, new mutations are generated until all string values have been mutated (total of L_S mutations) or if $r > CR$, where $r = \mathscr{U}(0,1)$ denotes a random number and CR is the crossover probability. Finally, the new child (s') is set as the generated mutant values plus the remaining ones from the current solution (s). Hence, the CR constant controls the fraction of values that are mutated.

The `DEoptim` function includes six arguments:

- `fn`—function to be minimized;
- `lower`, `upper`—lower and upper bounds;
- `control`—a list of control parameters (details are given in function `DEoptim.control`);
- `...`—additional arguments to be passed to `fn`; and
- `fnMap`—optional function that is run after each population creation but before the population is evaluated (it allows to impose integer/cardinality constraints).

The control parameters (C) are specified using the `DEoptim.control` function, which contains arguments such as:

- `VTR`—value to be reached, stop if best value is below VTR (default `VTR=-Inf`);
- `strategys`—type of differential strategy adopted, includes six different mutation strategies (classical mutation is set using `strategy=1`, default is `strategy=2`, full details can be accessed by executing `>?DEoptim.control`);
- `NP`—population size (default is `10*length(lower)`);
- `itermax`—maximum number of iterations (default is `200`);

- `CR`—crossover probability ($CR \in [0, 1]$, default is `0.5`);
- `F`—differential weighting factor ($F \in [0, 2]$, default is `0.8`);
- `trace`—a logical or integer value indicating if progress should be reported (if integer it occurs every `trace` iterations, default is `true`);
- `initialpop`—an initial population (defaults to `NULL`);
- `storepopfrom`, `storepopfreq`—from which iteration and with which frequency should the population values be stored; and
- `reltol`, `steptol`—relative convergence tolerance stop criterion, the method stops if unable to reduce the value by a factor of `reltol*(abs(value))` after `steptol` iterations.

The result of the `DEoptim` function is a list that contains two components:

- `$optim`—a list with elements, such as `$bestmem`—best solution and `$bestval`—best evaluation value;
- `$member`—a list with components, such as `bestvalit`—best value at each iteration and `pop`—last population.

Similarly to the `genalg` package, `DEoptim` also includes functions for plotting (`plot.DEoptim`) and summarizing (`summary.DEoptim`) results (under S3 scheme of method dispatching).

Price et al. (2005) advise the following general configuration for the differential evolution parameters: use the default $F = 0.8$ and $CR = 0.9$ values and set the population size to ten times the number of solution values ($N_P = 10 \times L_S$). Further details about the `DEoptim` package can be found in Mullen et al. (2011) (execute `> vignette("DEoptim")` to get an immediate access to this reference).

The demonstration `sphere-DEoptim.R` code adopts the **sphere** ($D = 2$) task:

```
### sphere-DEoptim.R file ###
library(DEoptim) # load DEoptim

sphere=function(x) sum(x^2)
D=2
maxit=100
set.seed(12345) # set for replicability
C=DEoptim.control(strategy=1,NP=5,itermax=maxit,CR=0.9,F=0.8,
                  trace=25,storepopfrom=1,storepopfreq=1)
# perform the optimization:
D=suppressWarnings(DEoptim(sphere,rep(-5.2,D),rep(5.2,D),
                           control=C))
# show result:
summary(D)
pdf("DEoptim.pdf",onefile=FALSE,width=5,height=9,
    colormodel="gray")
plot(D,plot.type="storepop")
dev.off()
cat("best:",D$optim$bestmem,"f:",D$optim$bestval,"\n")
```

The C object contains the control parameters, adjusted for the classical differential mutation and population size of 5, among other settings (the arguments storepopfrom and storepopfreq are required for the plot). Giving that DEoptim produces a warning when the population size is not set using the advised rule ($N_P = 10 \times L_S$), the suppressWarnings R function was added to ignore such warning. Regarding the plot, the informative "storepop" was selected (other options are "bestmemit"—evolution of the best parameter values; and "bestvalit"—best function value in each iteration). Also, additional arguments were used in the pdf function (onefile and colormodel) in order to adjust the file to contain just one page with a gray coloring scheme. The execution result of file sphere-DEoptim.R is:

```
> source("sphere-DEoptim.R")
Iteration: 25 bestvalit: 0.644692 bestmemit:    0.799515
     0.073944
Iteration: 50 bestvalit: 0.308293 bestmemit:    0.550749
    -0.070493
Iteration: 75 bestvalit: 0.290737 bestmemit:    0.535771
    -0.060715
Iteration: 100 bestvalit: 0.256731 bestmemit:    0.504867
    -0.042906

***** summary of DEoptim object *****
best member   :  0.50487 -0.04291
best value    :  0.25673
after         :  100 generations
fn evaluated  :  202 times
************************************
best: 0.5048666 -0.0429055 f: 0.2567311
```

The differential evolution algorithm improved the best value of the initial population from 7.37 to 0.25, leading to the optimized solution of $s = (0.50, -0.04)$. Figure 5.5 presents the result of the plot, showing a fast convergence in the population towards the optimized values (0.5 and -0.04). The first optimized value (0.5) is not very close to the optimum. However this is a tutorial example that includes a very small population size. When the advised rule is used ($N_P = 20$), the maximum distance of the optimized point to the origin is 7.53×10^{-10}!

5.4 Particle Swarm Optimization

Swarm intelligence denotes a family of algorithms (e.g., ant colony and particle swarm optimization) that are inspired in swarm behavior, which is exhibited by several animals (e.g., birds, ants, bees). These algorithms assume a population of simple agents with direct or indirect interactions that influence future behaviors. While each agent is independent, the whole swarm tends to produce a self-organized behavior, which is the essence of swarm intelligence (Michalewicz et al. 2006).

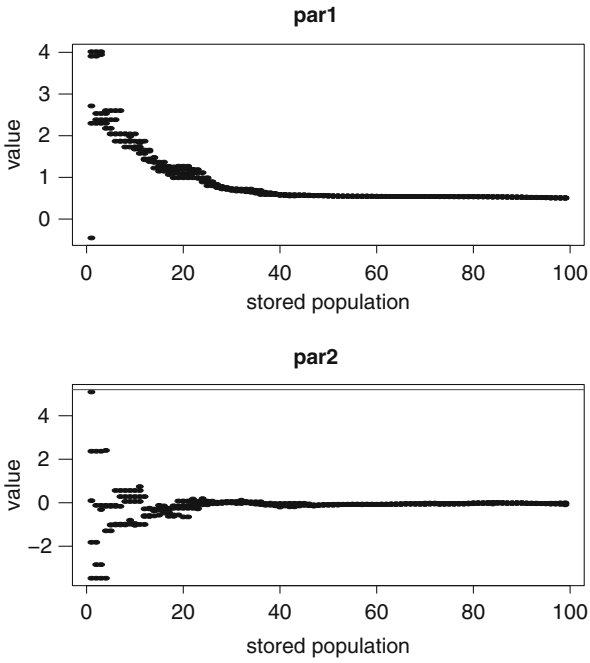

Fig. 5.5 Population evolution in terms of x_1 (*top*) and x_2 (*bottom*) values under the differential evolution algorithm for **sphere** ($D = 2$)

Particle swarm optimization is a swarm intelligence technique that was proposed by Kennedy and Eberhart (1995) for numerical optimization. Similarly to differential evolution, particle swarms operate mostly on metric spaces (Luke 2012). The algorithm is defined by the evolution of a population of particles, represented as vectors with a D-th (or L_S) dimensional space. The particle trajectories oscillate around a region that is influenced by the individual previous performance and by the success of his neighborhood (Mendes et al. 2002).

Since the original algorithm was presented in 1995, numerous variants have been proposed. This chapter adopts the `pso` package, which implements two standard versions that have been made publicly available at the Particle Swarm Central site (http://www.particleswarm.info/): SPSO 2007 and 2011. It is important to note that these SPSO variants do not claim to be the best versions on the market. Rather, SPSO implement the original particle swarm version (Kennedy and Eberhart 1995) with few improvements based on recent works. The goal is to define stable standards that can be compared against newly proposed particle swarm algorithms.

A particle moves on a step-by-step basis, where each step is also known as iteration, and contains (Clerc 2012): a position (s, inside search space), a fitness value (f), a velocity (v, used to compute next position), and a memory (p, previous best position found by the particle, and l, previous best position in the neighborhood). Each particle starts with random position and velocity values. Then, the search is performed by a cycle of iterations. During the search, the swarm assumes a topology, which denotes a set of links between particles. A link allows

Algorithm 7 Particle swarm optimization pseudo-code for SPSO 2007 and 2011

Inputs: f, C ▷ f is the fitness function, C includes control parameters
$P \leftarrow initialization(C)$ ▷ set initial swarm (topology, random position and velocity, previous best and previous best position found in the neighborhood)
$B \leftarrow best(P, f)$ ▷ best particle
$i \leftarrow 0$ ▷ i is the number of iterations of the method
while not $termination\_criteria(P, f, C, i)$ **do**
  for each particle $x = (s, v, p, l) \in P$ **do** ▷ cycle all particles
    $v \leftarrow velocity(s, v, p, l)$ ▷ compute new velocity for x
    $s \leftarrow s + v$ ▷ move the particle to new position s (*mutation*)
    $s \leftarrow confinement(s, C)$ ▷ adjust position s if it is outside bounds
    if $f(s) < f(p)$ **then** $p \leftarrow s$ ▷ update previous best
    end if
    $x \leftarrow (s, v, p, l)$ ▷ update particle
    if $f(s) < f(B)$ **then** $B \leftarrow s$ ▷ update best value
    end if
  end for
  $i \leftarrow i + 1$
end while
Output: B ▷ best solution

one particle to inform another one about its memory. The neighborhood is defined by the set of informants of a particle. The new particle position depends on its current position and velocity, while velocity values are changed using all elements of a particle (s, v, p and l). The overall pseudo-code for SPSO 2007 and 2011 is presented in Algorithm 7, which assumes a minimization goal.

This chapter highlights only the main SPSO differences. The full details are available at Clerc (2012). Several termination criteria are defined in SPSO 2007 and 2011: maximum admissible error (when optimum point is known), maximum number of iterations/evaluations, and maximum number of iterations without improvement. Regarding the swarm size (N_P), in SPSO 2007 it is automatically defined as the integer part of $10 + 2\sqrt{L_S}$ (L_S denotes the length of a solution), while in SPSO 2011 it is user defined (with suggested value of 40).

Historically, two popular particle swarm topologies (Mendes et al. 2002) were: *star*, where all particles know each other; and *ring*, where each particle has only two neighbors. However, the SPSO variants use a more recent adaptive star topology (Clerc 2012), where each particle informs itself and K randomly particles. Usually, K is set to 3. SPSO 2007 and 2011 use similar random uniform initializations for the position: $s_j = \mathscr{U}(\min(s_j), \max(s_j)), j \in \{1, \ldots, L_S\}$. However, the velocity is set differently: SPSO 2007—$v_j = \frac{\mathscr{U}(\min(s_j), \max(s_j)) - s_j}{2}$; SPSO 2011—$v_j = \mathscr{U}(\min(s_j) - s_j, \max(s_j) - s_j)$.

In SPSO 2007, the velocity update is applied dimension by dimension:

$$v_j \leftarrow wvj + \mathscr{U}(0, c)(p_j - s_j) + \mathscr{U}(0, c)(l_j - s_j) \tag{5.1}$$

where w and c are exploitation and exploration constants. The former constant sets the ability to explore regions of the search space, while the latter one defines the ability to concentrate around a promising area. The suggested values for these constants are $w = 1/(2\ln 2) \simeq 0.721$ and $c = 1/2 + \ln(2) \simeq 1.193$. A different scheme is adopted for SPSO 2011. First, the center of gravity (G_j) of three points (current position and two points, slightly beyond p and l):

$$G_j = s_j + c\frac{p_j + l_j - 2s_j}{3} \tag{5.2}$$

Then, a random point (s') is selected within the hypersphere of center G_j and radius $\| G_j - s_j \|$. Next, the velocity is updated as $v_j = wvj + s'_j - s_j$ and the new position is simply adjusted using $s_j = wv_j + s'_j$. There is a special case, when $l = p$. In such case, in SPSO 2007, the velocity is set using $v_j \leftarrow wvj + \mathscr{U}(0, c)(p_j - s_j)$, while in SPSO 2011 the center of gravity is computed using $s_j + c\frac{p_j - s_j}{2}$.

The *confinement* function is used to assure that the new position is within the admissible bounds. In both SPSO variants, when a position is outside a bound, it is set to that bound. In SPSO 2007, the velocity is also set to zero, while in SPSO 2011 it is set to half the opposite velocity ($v_j = -0.5v_j$).

The `pso` package includes the core `psoptim` function that can be used as a replacement of function `optim` (Sect. 4.3) and includes arguments, such as:

- `par`—vector defining the dimensionality of the problem (L_S), included for compatibility with optim and can include `NA` values;
- `fn`—function to be minimized;
- `lower`, `upper`—lower and upper bounds;
- `...`—additional arguments to be passed to `fn`; and
- `control`—a list of control parameters.

The control list includes components, such as:

- `$trace`—if positive, progress information is shown (default is 0);
- `$fnscale`—scaling applied to the evaluation function (if negative, transforms the problem into maximization; default is 1);
- `$maxit`—maximum number of iterations (defaults to 1,000);
- `$maxf`—maximum number of function evaluations;
- `$abstol`—stops if best fitness is less than or equal to this value (defaults to `-Inf`);
- `$reltol`—if the maximum distance between best particle and all others is less than `reltol*d`, then the algorithm restarts;
- `$REPORT`—frequency of reports if `trace` is positive;
- `$trace.stats`—if `TRUE`, then statistics at every `REPORT` step are recorded;
- `$s`—swarm size (N_P);
- `$k`—K value (defaults to 3);

- `$p`—average percentage of informants for each particle (a value of 1 implies a fully informed scheme, where all particles and not just K neighbors affect the individual velocity, defaults to `1-(1-1/s)^k`);
- `$w`—exploitation constant (if a vector of 2 elements, constant is gradually changed from `w[1]` to `w[2]`, default `1/(2*log(2))`;
- `$c.p`—local exploration constant (associated with p, defaults to `0.5+log(2)`);
- `$c.g`—global exploration constant (associated with l, defaults to `0.5+log(2)`);
- `$d`—diameter of the search space (defaults to Euclidean distance between `upper` and `lower`);
- `$v.max`—maximum admitted velocity (if not `NA` the velocity is clampled to the length of `v.max*d`, defaults to `NA`);
- `$maxit.stagnate`—maximum number of iterations without improvement (defaults to `Inf`); and
- `$type` —SPSO implementation type ("SPSO2007" or "SPSO2011," defaults to "SPSO2007").

The result is a list (compatible with `optim`) that contains:

- `$par`—best solution found;
- `$value`—best evaluation value;
- `$counts`—vector with three numbers (function evaluations, iterations, and restarts);
- `$convergence` and `$message`—stopping criterion type and message; and
- `$stats`—if `trace` is positive and `trace.stats` is true, then it contains the statistics: `it`—iteration numbers, `error`—best fitnesses, `f`—current swarm fitness vector, and `x`—current swarm position matrix.

The `sphere-psoptim.R` file adapts the `psoptim` function for the **sphere** ($D = 2$) task:

```
### sphere-psoptim.R file ###
library(pso) # load pso

sphere=function(x) sum(x^2)

D=2; maxit=10; s=5
set.seed(12345) # set for replicability
C=list(trace=1,maxit=maxit,REPORT=1,trace.stats=1,s=s)
# perform the optimization:
PSO=psoptim(rep(NA,D),fn=sphere,lower=rep(-5.2,D),
            upper=rep(5.2,D),control=C)
# result:
pdf("psoptim1.pdf",width=5,height=5)
j=1 # j-th parameter
plot(xlim=c(1,maxit),rep(1,s),PSO$stats$x[[1]][j,],pch=19,
     xlab="iterations",ylab=paste("s_",j," value",sep=""))
```

```
for(i in 2:maxit) points(rep(i,s),PSO$stats$x[[i]][j,],pch=19)
dev.off()
pdf("psoptim2.pdf",width=5,height=5)
plot(PSO$stats$error,type="l",lwd=2,xlab="iterations",
     ylab="best fitness")
dev.off()
cat("best:",PSO$par,"f:",PSO$value,"\n")
```

In this demonstration, a very small swarm size ($N_P = 5$) was adopted. Also, the control list was set to report statistics every iteration, under a maximum of ten iterations. The visual results are presented in terms of two plots. The first plot is similar to Fig. 5.5 and shows the evolution of the position particles for the first parameter. The second plot shows the evolution of the best fitness during the optimization. The execution of file `sphere-psoptim.R` is:

```
> source("sphere-psoptim.R")
S=5, K=3, p=0.488, w0=0.7213, w1=0.7213, c.p=1.193, c.g=1.193
v.max=NA, d=14.71, vectorize=FALSE, hybrid=off
It 1: fitness=3.318
It 2: fitness=0.9281
It 3: fitness=0.7925
It 4: fitness=0.4302
It 5: fitness=0.2844
It 6: fitness=0.2394
It 7: fitness=0.2383
It 8: fitness=0.2383
It 9: fitness=0.1174
It 10: fitness=0.1174
Maximal number of iterations reached
best: 0.2037517 -0.2755488 f: 0.1174419
```

The particle swarm improved the best fitness from 3.318 to 0.1174, leading to the optimized solution of $s = (0.20, -0.28)$ ($f = 0.12$). Figure 5.6 presents the result of the plots. It should be stressed that this tutorial example uses a very small swarm. When the advised rule is adopted ($N_P = 12$), the optimized values are $f = 0.03$ (10 iterations) and $f = 9.36 \times 10^{-12}$ (100 iterations).

5.5 Estimation of Distribution Algorithm

Estimation of distribution algorithms (EDA) (Larrañaga and Lozano 2002) are optimization methods that combine ideas from evolutionary computation, machine learning, and statistics. These methods were proposed in the mid-1990s, under several variants, such as population based incremental learning (PBIL) (Baluja 1994) and univariate marginal distribution algorithm (UMDA) (Mühlenbein 1997).

EDA works by iteratively estimating and sampling a probability distribution that is built from promising solutions (Gonzalez-Fernandez and Soto 2012). Other population based methods (e.g., evolutionary algorithms) create new individuals

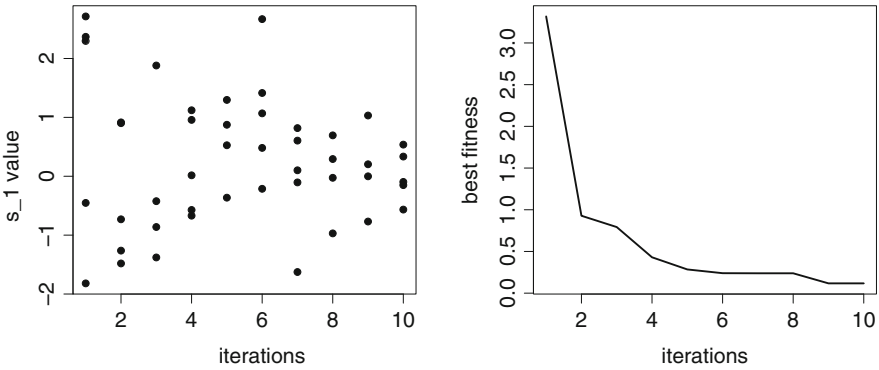

Fig. 5.6 Particle swarm optimization for **sphere** and $D = 2$ (*left* denotes the evolution of the position particles for the first parameter; *right* shows the evolution of the best fitness)

using an implicit distribution function (e.g., due to mutation and crossover operators). In contrast, EDA uses an explicit probability distribution defined by a model class (e.g., normal distribution). One main advantage of EDAs is that the search distribution may encode dependencies between the domain problem parameters, thus performing a more effective search.

The EDAs adopted in this chapter are implemented in the `copulaedas` package. The full implementation details are available at Gonzalez-Fernandez and Soto (2012). The generic EDA structure is presented in Algorithm 8. The initial population is often created by using a random seeding method. The results of global optimization methods, such as EDAs, can often be enhanced when combined with a local optimization method. Also, as described in Chap. 1, such local optimization can be useful to repair infeasible solutions (see Sect. 5.7 for an example). Then, the population of solutions are improved in a main cycle, until a termination criterion is met.

Within the main cycle, the *selection* function goal is to choose the most interesting solutions. For instance, *truncation selection* chooses a percentage of the best solutions from current population (P). The essential steps of EDA are the estimation and simulation of the search distribution, which is implemented by the *learn* and *sample* functions. The learning estimates the structure and parameters of the probabilistic model (M) and the sampling is used to generate new solutions (P') from the probabilistic model. Finally, the *replacement* function defines the next population. For instance, by replacing the full current population by the newly sampled one (P'), by maintaining only the best solutions (found in both populations) or by keeping a diverse set of solutions.

The `copulaedas` package implements EDAs based on *copula* functions (Joe 1997), under a modular object oriented implementation composed of separated generic functions that facilitates the definition of new EDAs (Gonzalez-Fernandez and Soto 2012). EDA components, such as learning and sampling methods, are independently programmed under a common structure shared by most EDAs.

Algorithm 8 Generic EDA pseudo-code implemented in `copulaedas` package, adapted from Gonzalez-Fernandez and Soto (2012)

Inputs: f, C ▷ f is the fitness function, C includes control parameters (e.g., N_P)
$P \leftarrow initialization(C)$ ▷ set initial population (*seeding* method)
if $required$ **then** $P \leftarrow local\_optimization(P, f, C)$ ▷ apply local optimization to P
end if
$B \leftarrow best(P, f)$ ▷ best solution of the population
$i \leftarrow 0$ ▷ i is the number of iterations of the method
while not $termination\_criteria(P, f, C)$ **do**
  $P' \leftarrow selection(P, f, C)$ ▷ selected population P'
  $M \leftarrow learn(P')$ ▷ set probabilistic model M using a learning method
  $P' \leftarrow sample(M)$ ▷ set sampled population from M using a sampling method
  if $required$ **then** $P' \leftarrow local\_optimization(P', f, C)$ ▷ apply local optimization to P'
  end if
  $B \leftarrow best(B, P', f)$ ▷ update best solution (if needed)
  $P \leftarrow replacement(P, P', f, C)$ ▷ create new population using a replacement method
  $i \leftarrow i + 1$
end while
Output: B ▷ best solution

The package uses S4 classes, which denotes R objects that have a formal definition of a class (type `> help("Classes")` for more details) and generic methods that can be defined by using the `setMethod` R function. An S4 instance is composed of slots, which is a class component that can be accessed and changed using the `@` symbol. An S4 class instance can be displayed at the console by using the `show()` R function.

The main function is `edaRun`, which implements Algorithm 8, assumes a minimization goal, and includes four arguments:

- `eda`—an EDA instance;
- `f`—evaluation function to be minimized; and
- `lower`, `upper`—lower and upper bounds.

The result is an `EDAResult` class with several slots, namely: `@eda`—EDA class; `@f`—evaluation function; `@lower` and `@upper`—lower and upper bounds; `@numGens`—number of generations (iterations); `@fEvals`—number of evaluations; `@bestEval`—best evaluation; `@bestSol`—best solution; and `@cpuTime`—time elapsed by the algorithm;

An EDA instance can be created using one of two functions, according to the type of model of search distributions: `CEDA`—using multivariate copula; and `VEDA`—using *vines* (graphical models that represent high-dimensional distributions and that can model a more rich variety of dependencies). The main arguments of `CEDA` are: `copula`—`"indep"` (independence or product copula) or `"normal"` (normal copula, the default); `margin`—marginal distribution (e.g., `"norm"`); and `popSize`—population size (N_P, default is 100). The `VEDA` function includes the same `margin` and `popSize` arguments and also: `vine`—`"CVine"` (canonical

vine) or "DVine" (the default); copulas—candidate copulas: "normal", "t", "clayton", "frank" or "gumbel" (default is c("normal")); and indepTestSigLevel—significance independence test level (default 0.01). The result is a CEDA (or VEDA) class with two slots: @name—the EDA name; and @parameters—the EDA parameters. Using these two functions, several EDAs can be defined, including UMDA, Gaussian copula EDA (GCEDA), C-vine EDA (CVEDA) and D-vine (DVEDA):

```
# four EDA types:
# adapted from (Gonzalez-Fernandez and Soto, 2012)
UMDA=CEDA(copula="indep",margin="norm"); UMDA@name="UMDA"
GCEDA=CEDA(copula="normal",margin="norm"); GCEDA@name="GCEDA"
CVEDA=VEDA(vine="CVine",indepTestSigLevel=0.01,
           copulas = c("normal"),margin = "norm")
CVEDA@name="CVEDA"
DVEDA=VEDA(vine="DVine",indepTestSigLevel=0.01,
           copulas = c("normal"),margin = "norm")
DVEDA@name="DVEDA"
```

The population size (N_P) is a critical factor of EDA performance, if too small then the estimate of the search distributions might be inaccurate, while a too large number increases the computational effort and might not introduce any gain in the optimization. Thus, several population size values should be tested. In particular, the copulaedas vignette (Gonzalez-Fernandez and Soto 2012) presents a bisection method that starts with an initial interval and that is implemented using the edaCriticalPopSize function (check `> ?edaCriticalPopSize`).

The copulaedas package includes several other generic methods that can be defined using the setMethod function (type `> help("EDA-class")` for more details), such as: edaSeed—initialization function (default is edaSeedUniform); edaOptimize—local optimization (disabled by default, Sect. 5.7 exemplifies how to define a different function); edaSelect—selection function (default is edaSelectTruncation); edaReplace—replacement function (default is edaReplaceComplete—P is replaced by P'); edaReport—reporting function (disabled by default); and edaTerminate—termination criteria (default edaTerminateMaxGen—maximum of iterations).

The same **sphere** ($D = 2$) task is used to demonstrate the EDA:

```
### sphere-EDA.R file ###
library(copulaedas)

sphere=function(x) sum(x^2)

D=2; maxit=10; LP=5
set.seed(12345) # set for replicability

# set termination criterion and report method:
setMethod("edaTerminate","EDA",edaTerminateMaxGen)
setMethod("edaReport","EDA",edaReportSimple)
```

```
# set EDA type:
UMDA=CEDA(copula="indep",margin="norm",popSize=LP,maxGen=maxit)
UMDA@name="UMDA (LP=5)"
# run the algorithm:
E=edaRun(UMDA,sphere,rep(-5.2,D),rep(5.2,D))
# show result:
show(E)
cat("best:",E@bestSol,"f:",E@bestEval,"\n")

# second EDA execution, using LP=100:
maxit=10; LP=100;
UMDA=CEDA(copula="indep",margin="norm",popSize=LP,maxGen=maxit)
UMDA@name="UMDA (LP=100)"
setMethod("edaReport","EDA",edaReportDumpPop) # pop_*.txt files
E=edaRun(UMDA,sphere,rep(-5.2,D),rep(5.2,D))
show(E)
cat("best:",E@bestSol,"f:",E@bestEval,"\n")

# read dumped files and create a plot:
pdf("eda1.pdf",width=7,height=7)
j=1; # j-th parameter
i=1;d=read.table(paste("pop_",i,".txt",sep=""))
plot(xlim=c(1,maxit),rep(1,LP),d[,j],pch=19,
     xlab="iterations",ylab=paste("s_",j," value",sep=""))
for(i in 2:maxit)
{ d=read.table(paste("pop_",i,".txt",sep=""))
  points(rep(i,LP),d[,j],pch=19)
}
dev.off()
```

In this example, the UMDA EDA type was selected using two different population sizes ($N_P = 5$ and $N_P = 100$, which is the default `copulaedas` value). For the last EDA, the `edaReportDumpPop` report type is adopted, which dumps each population into a different text file (e.g., the first population is stored at `pop_1.txt`). After showing the second EDA result, the dumped files are read using the `read.table` command, in order to create the plot of Fig. 5.7. The execution result of such demonstration code is:

```
> source("sphere-EDA.R")

  Generation      Minimum         Mean    Std. Dev.
           1 7.376173e+00 1.823098e+01 6.958909e+00
           2 7.583753e+00 1.230911e+01 4.032899e+00
           3 8.001074e+00 9.506158e+00 9.969029e-01
           4 7.118887e+00 8.358575e+00 9.419817e-01
           5 7.075184e+00 7.622604e+00 3.998974e-01
           6 7.140877e+00 7.321902e+00 1.257652e-01
           7 7.070203e+00 7.222189e+00 1.176669e-01
           8 7.018386e+00 7.089300e+00 4.450968e-02
           9 6.935975e+00 7.010147e+00 7.216829e-02
          10 6.927741e+00 6.946876e+00 1.160758e-02

Results for UMDA (LP=5)
```

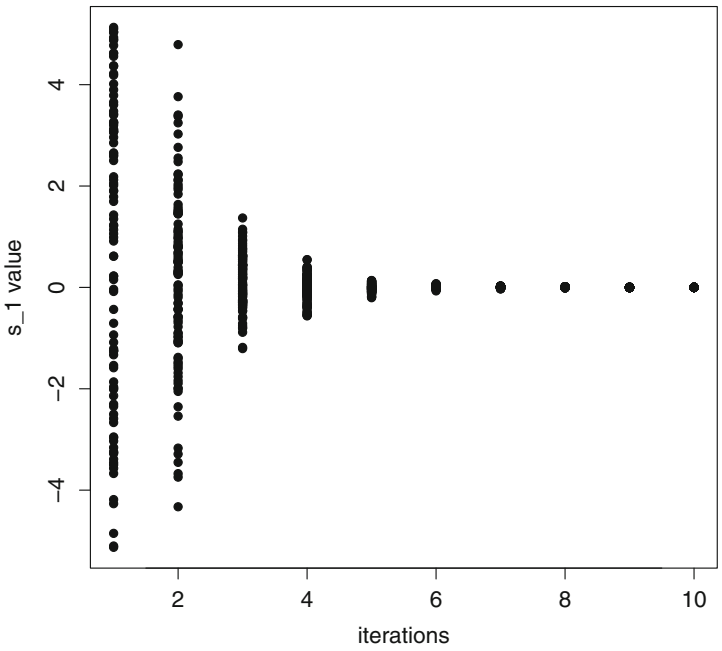

Fig. 5.7 Evolution of the first parameter population values (x_1) for EDA ($N_P = 100$)

```
Best function evaluation    6.927741e+00
No. of generations          10
No. of function evaluations 50
CPU time                    0.103 seconds
best: 1.804887 -1.915757 f: 6.927741

Results for UMDA (LP=100)
Best function evaluation    5.359326e-08
No. of generations          10
No. of function evaluations 1000
CPU time                    0.036 seconds
best: -0.00013545 0.0001877407 f: 5.359326e-08
```

When only five individuals are used, the algorithm only performs a slight optimization (from $f = 7.38$ to $f = 6.93$). However, when a higher population size is adopted ($N_P = 100$), the EDA performs a very good optimization, achieving a value of 5.36×10^{-8} in only ten generations. Figure 5.7 shows the respective evolution of the first parameter population values, showing a fast convergence towards the optimum zero value.

5.6 Comparison of Population Based Methods

The goal of this section is to compare all previously presented population based algorithms on two tasks (**rastrigin**, $D = 20$; and **bag prices**, $D = 5$). Four continuous optimization methods are compared: evolutionary algorithm, differential evolution, particle swarm optimization (SPSO 2007), and EDA (GCEDA variant). For the second task, each solution is rounded to the nearest integer value (within [$1,$1000]) before computing the profit. Each method is run fifty times for each task. To simplify the analysis, the comparison is made only in terms of aggregated results over the runs (average or percentage of successes) and no confidence intervals or statistical tests are used (check Sects. 2.2, 4.5, and 5.7 for R code examples of more robust statistical comparative analysis).

Similarly to what is discussed in Sect. 4.5, rather than executing a complete and robust comparison, the intention is more to show how population based algorithms can be compared. To provide a fair comparison and to simplify the experimentation setup, the default parameters of the methods are adopted, except for the population size, which is kept the same for all methods ($N_P = 100$ for **rastrigin** and $N_P = 50$ for **bag prices**). Also, as performed in Sect. 4.5, all methods are evaluated by storing the best value as a function of the number of evaluations and up to the same maximum number (`MAXFN=10000` for **rastrigin** and `MAXFN=5000` for **bag prices**).

The comparison code (file `compare2.R`) uses the same global variables of Sect. 4.5 (`EV`, `F` and `BEST`), to store the best value:

```
### compare2.R file ###

source("functions.R") # bag prices functions
library(genalg)
library(DEoptim)
library(pso)
library(copulaedas)

# evaluation functions: ------------------------------------
crastrigin=function(x) # adapted rastrigin
{ f=10*length(x)+sum(x^2-10*cos(2*pi*x))
  # global assignment code: <<-
  EV<<-EV+1 # increase evaluations
  if(f<BEST) BEST<<-f # minimum value
  if(EV<=MAXFN) F[EV]<<-BEST
  return(f)
}
cprofit=function(x) # adapted bag prices
{ x=round(x,digits=0) # convert x into integer
  # given that EDA occasionally produces unbounded values:
  x=ifelse(x<1,1,x)         # assure that x is within
  x=ifelse(x>1000,1000,x)   # the [1,1000] bounds
  s=sales(x)           # get the expected sales
  c=cost(s)            # get the expected cost
  profit=sum(s*x-c)    # compute the profit
  EV<<-EV+1 # increase evaluations
  if(profit>BEST) BEST<<-profit # maximum value
```

```
  if(EV<=MAXFN) F[EV]<<-BEST
  return(-profit) # minimization task!
}
# auxiliary functions: ------------------------------------
crun=function(method,f,lower,upper,LP,maxit) # run a method
{ if(method=="EA")
     rbga(evalFunc=f,stringMin=lower,stringMax=upper,popSize=LP,
          iters=maxit*1.5)
  else if(method=="DE")
     { C=DEoptim.control(itermax=maxit,trace=FALSE,NP=LP)
       DEoptim(f,lower=lower,upper=upper,control=C)
     }
  else if(method=="PSO")
     { C=list(maxit=maxit,s=LP)
       psoptim(rep(NA,length(lower)),fn=f,
               lower=lower,upper=upper,control=C)
     }
  else if(method=="EDA")
     { setMethod("edaTerminate","EDA",edaTerminateMaxGen)
       GCEDA=CEDA(copula="normal",margin="norm",popSize=LP,
                  maxGen=maxit)
       GCEDA@name="GCEDA"
       edaRun(GCEDA,f,lower,upper)
     }
}

successes=function(x,LIM,type="min") # number of successes
{ if(type=="min") return(sum(x<LIM)) else return(sum(x>LIM)) }

ctest=function(Methods,f,lower,upper,type="min",Runs, # test
               D,MAXFN,maxit,LP,pdf,main,LIM) # all methods:
{ RES=vector("list",length(Methods)) # all results
  VAL=matrix(nrow=Runs,ncol=length(Methods)) # best values
  for(m in 1:length(Methods)) # initialize RES object
   RES[[m]]=matrix(nrow=MAXFN,ncol=Runs)


  for(R in 1:Runs) # cycle all runs
    for(m in 1:length(Methods))
      { EV<<-0; F<<-rep(NA,MAXFN) # reset EV and F
       if(type=="min") BEST<<-Inf else BEST<<- -Inf # reset BEST
       suppressWarnings(crun(Methods[m],f,lower,upper,LP,maxit))
       RES[[m]][,R]=F # store all best values
        VAL[R,m]=F[MAXFN] # store best value at MAXFN
      }
  # compute average F result per method:
  AV=matrix(nrow=MAXFN,ncol=length(Methods))
  for(m in 1:length(Methods))
    for(i in 1:MAXFN)
      AV[i,m]=mean(RES[[m]][i,])
  # show results:
  cat(main,"\n",Methods,"\n")
  cat(round(apply(VAL,2,mean),digits=0)," (average best)\n")
```

```
  cat(round(100*apply(VAL,2,successes,LIM,type)/Runs,
            digits=0)," (%successes)\n")

  # create pdf file:
  pdf(paste(pdf,".pdf",sep=""),width=5,height=5,paper="special")
  par(mar=c(4.0,4.0,1.8,0.6)) # reduce default plot margin
  MIN=min(AV);MAX=max(AV)
  # use a grid to improve clarity:
  g1=seq(1,MAXFN,length.out=500) # grid for lines
  plot(g1,AV[g1,1],ylim=c(MIN,MAX),type="l",lwd=2,main=main,
       ylab="average best",xlab="number of evaluations")
  for(i in 2:length(Methods)) lines(g1,AV[g1,i],lwd=2,lty=i)
  if(type=="min") position="topright" else position=
    "bottomright"
  legend(position,legend=Methods,lwd=2,lty=1:length(Methods))
  dev.off() # close the PDF device
}

# define EV, BEST and F:
MAXFN=10000
EV=0;BEST=Inf;F=rep(NA,MAXFN)
# define method labels:
Methods=c("EA","DE","PSO","EDA")
# rastrigin comparison: ----------------------------------
Runs=50; D=20; LP=100; maxit=100
lower=rep(-5.2,D);upper=rep(5.2,D)
ctest(Methods,crastrigin,lower,upper,"min",Runs,D,MAXFN,maxit,
  LP,
      "comp-rastrigin2","rastrigin (D=20)",75)
# bag prices comparison: ---------------------------------
MAXFN=5000
F=rep(NA,MAXFN)
Runs=50; D=5; LP=50; maxit=100
lower=rep(1,D);upper=rep(1000,D)
ctest(Methods,cprofit,lower,upper,"max",Runs,D,MAXFN,maxit,LP,
      "comp-bagprices","bag prices (D=5)",43500)
```

Two important auxiliary functions were defined: `crun`—for executing a run of one of the four methods; and `ctest`—for executing several runs and showing the overall results. For the evolutionary algorithm, the maximum number of iterations is increased by 50 % to assure that at least `MAXFN` evaluations are executed (due to elitism, the number of tested solutions is lower than $N_P \times maxit$). The obtained results for each task are presented in terms of a plot and two console metrics. Each plot shows in the y-axis the evolution of the average best value, while the x-axis contains the number of evaluation functions. The two metrics used are: the average (over all runs) the best result (measured at `MAXFN`) and the percentage of successes. The last metric is measured as the proportion of best results below 75 (for **rastrigin**) or above 43,500 (for **bag prices**).

The results of the two plots are presented in Fig. 5.8, while the console results are:

```
> source("compare2.R")
rastrigin (D=20)
 EA DE PSO EDA
```

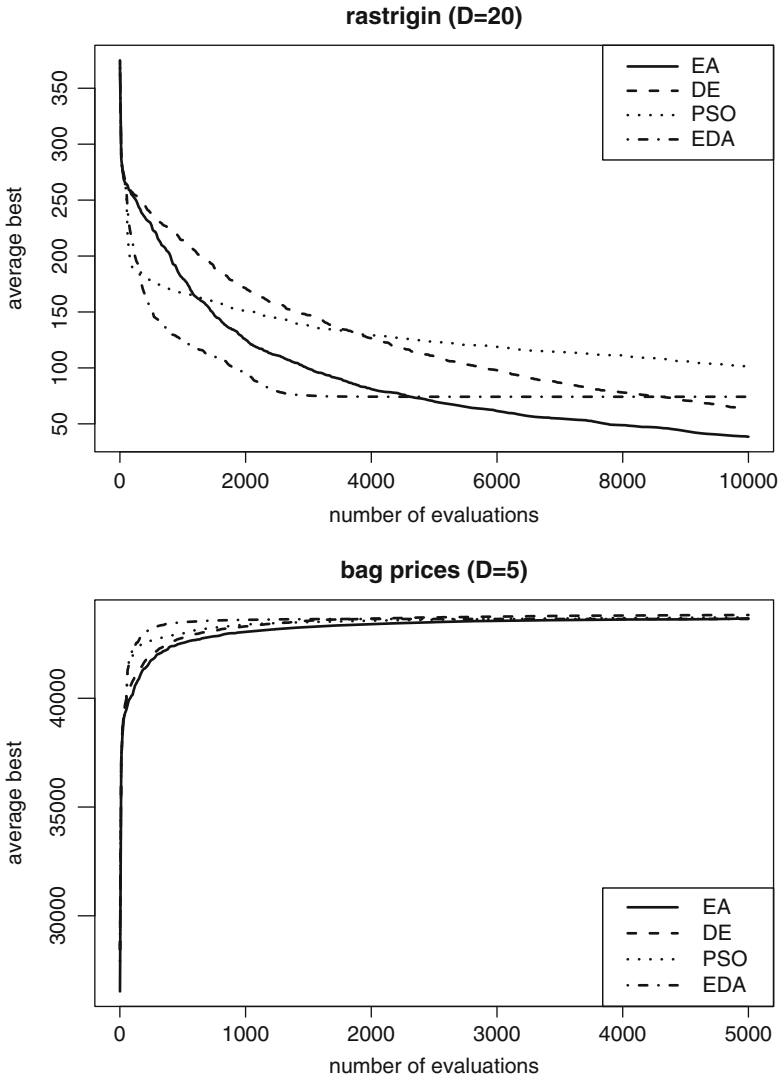

Fig. 5.8 Population based search comparison example for the **rastrigin** (*top*) and **bag prices** (*bottom*) tasks

```
38 64 101 74  (average best)
100 94 2 58  (%successes)
 EA DE PSO EDA
43674 43830 43722 43646  (average best)
96 100 100 92  (%successes)
```

The comparison of methods using the methodology related with Fig. 5.8 is interesting, as it presents the average behavior of the method throughout the number

of evaluations, which is correlated with computational effort. EDA shows a faster initial convergence when compared with other methods (see Fig. 5.8) and thus it is the best choice if few computational resources are available. However, after a while the EDA convergence gets more flat and the method is outperformed by the evolutionary algorithm for **rastrigin** (after around 5,000 evaluations) and by the differential evolution for **bag prices** (after around 2,000 evaluations). Hence, if more computation power is available, then evolutionary algorithm is the best method (at `MAXFN`) for **rastrigin**, with an average of 38 and 100 % of successes, while differential evolution is the best choice for **bag prices**, with an average of 43,830 and 100 % of successes. It should be noted that this is just a demonstrative comparison example and different results could be achieved with a distinct experimental setup (e.g., different N_P and `MAXFN` values).

5.7 Bag Prices with Constraint

This section compares two strategies for handling constraints: death penalty and repair. As explained in Sect. 1.5, death penalty is a simply strategy that can be easily applied to any optimization method. It requires only changing the evaluation function to return very high penalty value if the solution is infeasible. However, such strategy is not very efficient, since the infeasible solutions do not guide the search, thus behaving similarly to Monte Carlo random search, see Sect. 3.4. The repair alternative tends to be more efficient, as it transforms an infeasible solution into a feasible one, but often requires domain knowledge.

For this comparison, the **bag prices** task (of Sect. 1.7) is adopted with a hard constraint: the maximum number of bags that can be manufactured within a production cycle is set to 50. Also, the EDA method is used as the optimization engine, since the `copulaedas` package presents a useful feature for the repair strategy, since it is possible to add a local optimization method within the EDA main cycle (by using the `edaOptimize` generic method). The death penalty is simply implemented by returning `Inf` when a solution is infeasible, while the repair method uses a local search and domain knowledge. The code of Sect. 5.6 was adapted (e.g., same N_P, $maxit$ and `MAXFN` values) for this experiment:

```
### bag-prices-constr.R file ###

source("functions.R") # bag prices functions
library(copulaedas) # EDA

# evaluation function:  ------------------------------------
cprofit2=function(x) # bag prices with death penalty
{ x=round(x,digits=0) # convert x into integer
  x=ifelse(x<1,1,x)        # assure that x is within
  x=ifelse(x>1000,1000,x)  # the [1,1000] bounds
  s=sales(x)
  if(sum(s)>50) res=Inf # if needed, death penalty!!!
```

```
  else{ c=cost(s);profit=sum(s*x-c)
        # if needed, store best value
        if(profit>BEST) { BEST<<-profit; B<<-x}
        res=-profit # minimization task!
      }
  EV<<-EV+1 # increase evaluations
  if(EV<=MAXFN) F[EV]<<-BEST
  return(res)
}
# example of a local search method that repairs a solution:
localRepair=function(eda, gen, pop, popEval, f, lower, upper)
{
 for(i in 1:nrow(pop))
 { x=pop[i,]
   x=round(x,digits=0) # convert x into integer
   x=ifelse(x<lower[1],lower[1],x) # assure x within
   x=ifelse(x>upper[1],upper[1],x) # bounds
   s=sales(x)
   if(sum(s)>50)
   {
    x1=x
    while(sum(s)>50) # new constraint: repair
    { # increase price to reduce sales:
      x1=x1+abs(round(rnorm(D,mean=0,sd=5)))
      x1=ifelse(x1>upper[1],upper[1],x1) # bound if needed
      s=sales(x1)
    }
    x=x1 # update the new x
   }
   pop[i,]=x;popEval[i]=f(x)
 }
 return(list(pop=pop,popEval=popEval))
}

# experiment: ------------------------------------------------
MAXFN=5000
Runs=50; D=5; LP=50; maxit=100
lower=rep(1,D);upper=rep(1000,D)
Methods=c("Death","Repair")
setMethod("edaTerminate","EDA",edaTerminateMaxGen)
GCEDA=CEDA(copula="normal",margin="norm",popSize=LP,
           maxGen=maxit,fEvalStdDev=10)
GCEDA@name="GCEDA"

RES=vector("list",length(Methods)) # all results
VAL=matrix(nrow=Runs,ncol=length(Methods)) # best values
for(m in 1:length(Methods)) # initialize RES object
   RES[[m]]=matrix(nrow=MAXFN,ncol=Runs)
for(R in 1:Runs) # cycle all runs
  {
    B=NA;EV=0; F=rep(NA,MAXFN); BEST= -Inf # reset vars.
    setMethod("edaOptimize","EDA",edaOptimizeDisabled)
    setMethod("edaTerminate","EDA",edaTerminateMaxGen)
```

```
    suppressWarnings(edaRun(GCEDA,cprofit2,lower,upper))
    RES[[1]][,R]=F # store all best values
    VAL[R,1]=F[MAXFN] # store best value at MAXFN

    B=NA;EV=0; F=rep(NA,MAXFN); BEST= -Inf # reset vars.
    # set local repair search method:
    setMethod("edaOptimize","EDA",localRepair)
    # set additional termination criterion:
    setMethod("edaTerminate","EDA",
              edaTerminateCombined(edaTerminateMaxGen,
                  edaTerminateEvalStdDev))
    # this edaRun might produces warnings or errors:
    suppressWarnings(try(edaRun(GCEDA,cprofit2,lower,upper),
        silent=TRUE))
    if(EV<MAXFN) # if stopped due to EvalStdDev
       F[(EV+1):MAXFN]=rep(F[EV],MAXFN-EV) # replace NAs
    RES[[2]][,R]=F # store all best values
    VAL[R,2]=F[MAXFN] # store best value at MAXFN
  }

# compute average F result per method:
AV=matrix(nrow=MAXFN,ncol=length(Methods))
for(m in 1:length(Methods))
  for(i in 1:MAXFN)
    AV[i,m]=mean(RES[[m]][i,])
# show results:
cat(Methods,"\n")
cat(round(apply(VAL,2,mean),digits=0)," (average best)\n")
# Mann-Whitney non-parametric test:
p=wilcox.test(VAL[,1],VAL[,2],paired=TRUE)$p.value
cat("p-value:",round(p,digits=2),"(<0.05)\n")

# create pdf file:
pdf("comp-bagprices-constr.pdf",width=5,height=5,paper=
  "special")
par(mar=c(4.0,4.0,1.8,0.6)) # reduce default plot margin
# use a grid to improve clarity:
g1=seq(1,MAXFN,length.out=500) # grid for lines
plot(g1,AV[g1,2],type="l",lwd=2,
     main="bag prices with constraint",
     ylab="average best",xlab="number of evaluations")
lines(g1,AV[g1,1],lwd=2,lty=2)
legend("bottomright",legend=rev(Methods),lwd=2,lty=1:4)
dev.off() # close the PDF device
```

The two constraint handling methods are compared similarly as in Sect. 5.6, thus global `EV`, `F`, and `BEST` variables are used to store the best values at a given function evaluation. The death penalty is implemented in function `cprofit2`. The repair solution is handled by the `localRepair` function, which contains the signature (function arguments) required by `edaOptimize` (e.g., eda argument is not needed). If a solution is infeasible, then a local search is applied, where the solution prices are randomly increased until the expected sales is lower than 51. This local search uses the following domain knowledge: the effect of increasing

a price is a reduction in the number of sales. The new feasible solutions are then evaluated. To reduce the number of code lines, the same `cprofit2` function is used, although the death penalty is never applied, given the evaluated solution is feasible. The `localRepair` function returns a list with the new population and evaluation values. Such list is used by `edaRun` to replace the sampled population (P'), thus behaving as a Lamarckian global–local search hybrid (see Sect. 1.6).

The EDA applied is similar to the one presented in Sect. 5.6 (e.g., GCEDA), except for the repair strategy, which includes the explained local search and an additional termination criterion (`edaTerminateEvalStdDev`), which stops when the standard deviation of the evaluation of the solutions is too low. When more than one criterion is used in EDA, the `edaTerminateCombined` method needs to be adopted. The `edaTerminateEvalStdDev` extra criterion was added given that the repair strategy leads to a very fast convergence and thus the population quickly converges to the same solution. However, setting the right standard deviation threshold value (`fEvalStdDev`) is not an easy task (setting a too large value will stop the method too soon). With `fEvalStdDev=10`, `edaRun` still occasionally produces warnings or errors. Thus, the `suppressWarnings` and `try` R functions are used to avoid this problem. The latter function prevents the failure of the `edaRun` execution. Such failure is not problematic, giving that the results are stored in global variables. In the evaluation function (`cprofit2`), the global variable `B` is used to store the best solution, which is useful when `edaRun` fails. When a failure occurs, the code also replaces the remaining `NA` values of the `F` object with the last known evaluation value.

The obtained results are processed similarly to what is described in Sect. 5.6, except that the number of successes is not measured and the final average results are compared with a Mann–Whitney non-parametric statistical test (using the `wilcox.test` function). An example of file `bag-prices-constr.R` execution is:

```
> source("bag-prices-constr.R")
Death Repair
31175 32364  (average best)
p-value: 0 (<0.05)
```

As shown by the average results, statistical test (p-value<0.05) and plot of Fig. 5.9, the repair strategy clearly outperforms the death penalty one, with a final average difference of \$1189 (statistically significant). Nevertheless, it should be noted that this repair strategy uses domain knowledge and thus it cannot be applied directly to other tasks.

5.8 Genetic Programming

Genetic programming denotes a collection of evolutionary computation methods that automatically generate computer programs (Banzhaf et al. 1998). In general, the computer programs have a variable length and are based on lists or trees

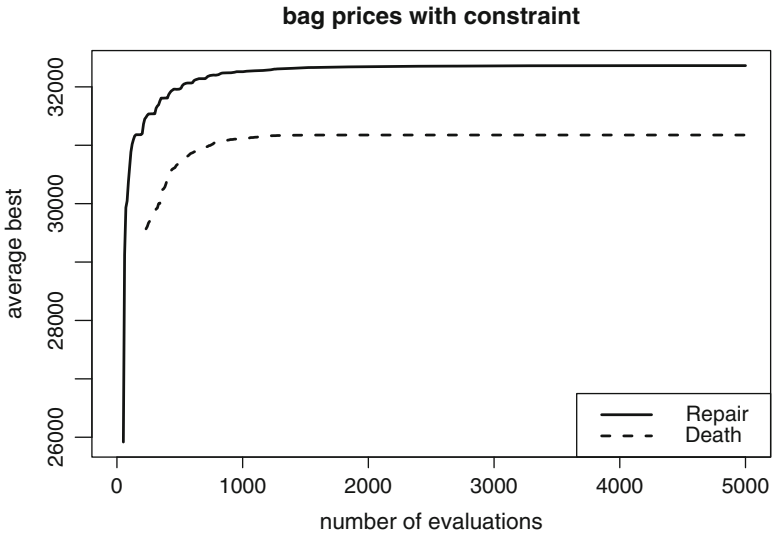

Fig. 5.9 Comparison of repair and death penalty strategies for **bag prices** task with constraint

(Luke 2012). Hence, the goal of genetic programming is quite distinct from the previous presented population based algorithms. Instead of numerical optimization, genetic programming is used in tasks such as automatic programming or discovering mathematical functions. As pointed out in Flasch (2013), the one key advantage of genetic programming is that the representation of the solutions is often easy to interpret by humans; however, the main drawback is a high computational cost, due to the high search space of potential solutions.

Genetic programming adopts the same concepts of evolutionary algorithms, with a population of solutions that compete for survival and use of genetic operators for generating new offspring. Thus, the search engine is similar to what is described in Algorithm 5. Giving that a different representation is adopted (e.g., trees), a distinct initialization function is adopted (e.g., *random growth*) and specialized genetic operators are used (Michalewicz et al. 2006). There are two main mutation types in classical genetic programming systems: replace a randomly selected value or function by another random value or function; and replace a randomly selected subtree by another generated subtree. The classical crossover works by replacing a random subtree from a parent solution by another random subtree taken from a second parent. Figure 5.10 shows an example of the random subtree crossover and includes four examples of tree representations for mathematical functions.

This section approaches the mathematical function discovery goal using the `rgp` package and the intention is to show how non-numerical representations can be handled by a modern optimization technique. Only a brief explanation of `rgp` features is provided, since the package includes a large range of functions. If needed, further details can be obtained in Flasch (2013) (`> vignette("rgp_introduction")`).

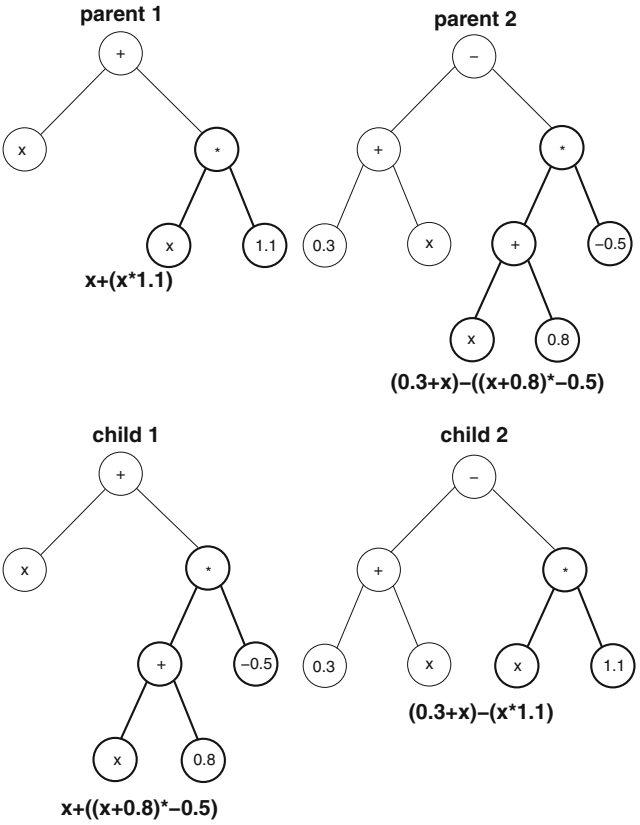

Fig. 5.10 Example of a genetic programming random subtree crossover

The first step is to define the symbolic expression search space. In `rgp`, solutions are represented as R functions, which are constructed in terms of three sets: input variables (function arguments), constants, and function symbols. Constants are created in terms of factory functions, which typically are stochastic and are called each time a new constant is needed. The function symbols usually include arithmetic operators, such as addition (`+`) or subtraction (`-`). Other R mathematical functions can also be included (e.g., `exp`, `log`), but some care is needed to avoid invalid expressions (e.g., `log(-1)` returns a `NaN`). These sets (whose members are known as building blocks) can be set using the `rgp` functions:

- `inputVariableSet`—arguments are the names (strings) of input variables;
- `constantFactorySet`—argument is a factory function that often includes a random number generator (e.g., `rnorm`, `runif`); and
- `functionSet`—arguments are strings that define the set of mathematical symbols.

Next, an evaluation function needs to be defined. The `rgp` assumes a a minimization goal. The final step consists in selecting the genetic programming parameters and running the algorithm. This is achieved using the `geneticProgramming` function, which includes parameters such as:

- `fitnessFunction`—evaluation function that includes one argument (the expression);
- `stopCondition`—termination criteria (usually based on maximum runtime, in seconds; `?makeStepsStopCondition` shows other stopping options and full details);
- `population`—initial population, if missing it is created using a random growth;
- `populationSize`—size of the population (N_P, the default is 100);
- `eliteSize`—number of individuals to keep (*elitism*, defaults to `ceiling (0.1 * populationSize)`);
- `functionSet`—set of mathematical symbols;
- `inputVariables`—set of input variables;
- `constantSet`—set of constant factory functions;
- `crossoverFunction`—crossover operator (defaults to `crossover`, which is the classical random subtree crossover, see Fig. 5.10);
- `mutationFunction`—mutation function (defaults to `NULL`, check `?mutateFunc` for `rgp` mutation possibilities);
- `progressMonitor`—function called every generation and that shows the progress of the algorithm; and
- `verbose`—if progress should be printed.

The result is a list with several components, including: `$population`, the last population; and `$fitnessValues`, the evaluation values of such population.

To show the `rgp` capabilities, the synthetic **rastrigin** function ($D = 2$) is adopted (check Sect. 7.3 for a real-world problem demonstration) and approximated with a polynomial function. Thus, the variable set includes two inputs (x_1 and x_2) and set of function symbols is defined as {"*", "+", "-"}. In this example, the set of constants is generated using a normal distribution. Moreover, the evaluation function is not the **rastrigin** function itself, since the goal is to approximate this function. Rather, the evaluation function is set as the error between the rastrigin and candidate expression outputs for an input domain. The mean squared error (MSE) is the adopted error metric. Widely used in statistics, the metric is defined as $MSE = \sum_{i=1}^{N}(y_i - \hat{y}_i)^2/N$, where y_i is the target value for input $\mathbf{x}_i$, $\hat{y}_i$ is the estimated value, and N is the number of input examples. MSE penalizes higher individual errors and the lower the metric, the better is the approximation. The genetic programming is set with a population size of $N_P = 50$, a random subtree mutation (with maximum subtree depth of 4, using the `mutateSubtree rgp` function) and it is stopped after 50 s. The implemented R code is:

```
### gp-rastrigin.R ###

library(rgp) # load rgp

# auxiliary functions:
rastrigin=function(x) 10*length(x)+sum(x^2-10*cos(2*pi*x))
fwrapper=function(x,f) f(x[1],x[2])

# configuration of the genetic programming:
ST=inputVariableSet("x1","x2")
cF1=constantFactorySet(function() rnorm(1)) # mean=0, sd=1
FS=functionSet("+","*","-")
# set the input samples (grid^2 data points):
grid=10 # size of the grid used
domain=matrix(ncol=2,nrow=grid^2) # 2D domain grid
domain[,1]=rep(seq(-5.2,5.2,length.out=grid),each=grid)
domain[,2]=rep(seq(-5.2,5.2,length.out=grid),times=grid)
eval=function(f) # evaluation function
{ mse(apply(domain,1,rastrigin),apply(domain,1,fwrapper,f)) }

# run the genetic programming:
set.seed(12345) # set for replicability
mut=function(func) # set the mutation function
{ mutateSubtree(func,funcset=FS,inset=ST, conset=cF1,
                mutatesubtreeprob=0.1,maxsubtreedepth=4) }
gp=geneticProgramming(functionSet=FS,inputVariables=ST,
                      constantSet=cF1,populationSize=50,
                      fitnessFunction=eval,
                      stopCondition=makeTimeStopCondition(50),
                      mutationFunction=mut,verbose=TRUE)
# show the results:
b=gp$population[[which.min(gp$fitnessValues)]]
cat("best solution (f=",eval(b),"):\n")
print(b)
# create approximation plot:
L1=apply(domain,1,rastrigin);L2=apply(domain,1,fwrapper,b)
MIN=min(L1,L2);MAX=max(L1,L2)
pdf("gp-function.pdf",width=7,height=7,paper="special")
plot(L1,ylim=c(MIN,MAX),type="l",lwd=2,lty=1,
     xlab="points",ylab="function values")
lines(L2,type="l",lwd=2,lty=2)
legend("bottomright",leg=c("rastrigin","GP function"),lwd=2,
       lty=1:2)
dev.off()
```

In this example, the two input variables are named `"x1"` and `"x2,"` while the input domain is created as a two dimensional grid, where each input is varied within the range $[-5.2, 5.2]$, with a total of `grid*grid`=100 samples. The `domain` matrix is created using the `rep` and `seq` R functions (type `> print(domain)` to check the matrix values). Such matrix is used by the `eval` function, which uses the `apply`

function at the row level to generate first the rastrigin and expression (`f`) outputs and then computes the MSE for all domain samples. The `mse` function computes the MSE and it is defined in the `rgp` package. The auxiliary `fwrapper` function was created for an easier use of `apply` over `f`, since `f` receives two arguments while a row from `domain` is one vector with two elements. After running the genetic programming, the best solution is presented. Also, a PDF file is created, related with a two dimensional plot, where the x-axis denotes all 100 points and the y-axis the rastrigin and genetic programming best solution output values. The result of running the demonstration file (`gp-rastrigin.R`) is:[1]

```
> source("gp-rastrigin.R")
STARTING genetic programming evolution run (Age/Fitness/
    Complexity Pareto GP search-heuristic) ...
evolution step 100, fitness evaluations: 1980, best fitness:
    1459.126753, time elapsed: 3.37 seconds
evolution step 200, fitness evaluations: 3980, best fitness:
   317.616080, time elapsed: 7.14 seconds
evolution step 300, fitness evaluations: 5980, best fitness:
   205.121919, time elapsed: 12.09 seconds
evolution step 400, fitness evaluations: 7980, best fitness:
   98.718003, time elapsed: 18.1 seconds
evolution step 500, fitness evaluations: 9980, best fitness:
   87.140058, time elapsed: 23.73 seconds
evolution step 600, fitness evaluations: 11980, best fitness:
  87.140058, time elapsed: 29.62 seconds
evolution step 700, fitness evaluations: 13980, best fitness:
  87.140058, time elapsed: 35.31 seconds
evolution step 800, fitness evaluations: 15980, best fitness:
  87.140058, time elapsed: 41.28 seconds
evolution step 900, fitness evaluations: 17980, best fitness:
  87.074739, time elapsed: 46.98 seconds
Genetic programming evolution run FINISHED after 954 evolution
    steps, 19060 fitness evaluations and 50.05 seconds.
best solution (f= 87.07474 ):
function (x1, x2)
x2 * x2 + (1.3647488967524 + x1 * x1 + -0.82968488587336 +
    1.3647488967524 + 1.3647488967524 + 1.3647488967524 +
    1.3647488967524 + 1.3647488967524 + 1.59224941702801 +
    1.3647488967524 + 1.3647488967524 + 1.3647488967524 +
    1.3647488967524 + 1.3647488967524)
```

After 50 s, the best obtained solution is $x_2^2 + x_1^2 + 15.7748$, corresponding to an MSE value of 87. Figure 5.11 shows the created plot, revealing an interesting fit. It should be noted that the final solution ($x_2^2 + x_1^2 + 15.7748$) was obtained by performing a manual "cleaning" of the returned symbolic expression. Such post-processing (using manual or automatic techniques) is a common task when human understandable knowledge is required.

[1]These results were achieved with `rgp` version 0.3-4 and later `rgp` versions might produce different results.

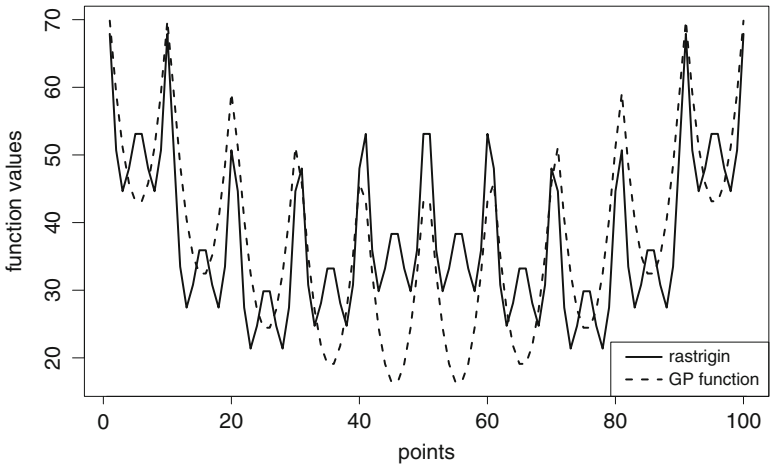

Fig. 5.11 Comparison of **rastrigin** function and best solution given by the genetic programming

5.9 Command Summary

`CEDA()`	Implement EDAs based on multivariate copulas (package `copulaedas`)
`constantFactorySet()`	Genetic programming set of constants (package `rgp`)
`copulaedas`	Package for EDAs based on copulas
`DEoptim`	Package for differential evolution
`DEoptim()`	Differential evolution algorithm (package `DEoptim`)
`DEoptim.control()`	Differential evolution control parameters (package `DEoptim`)
`edaRun()`	EDA optimization algorithm (package `copulaedas`)
`functionSet()`	Genetic programming set of functions (package `rgp`)
`genalg`	Package for genetic and evolutionary algorithms
`geneticProgramming`	Genetic programming algorithm (package `rgp`)
`gray()`	Returns a vector of gray colors from a vector of gray levels
`inputVariableSet()`	Genetic programming set of variables (package `rgp`)
`mse()`	Mean squared error (package `rgp`)
`mutateSubtree()`	Random subtree mutation (package `rgp`)
`plot.DEoptim()`	Plot differential evolution result (package `DEoptim`)
`plot.rbga()`	Plot genetic/evolutionary algorithm result (package `genalg`)
`pso`	Package for particle swarm optimization
`pso()`	Particle swarm optimization algorithm (package `pso`)
`rbga()`	Evolutionary algorithm (package `genalg`)
`rbga.bin()`	Genetic algorithm (package `genalg`)
`rgp`	Package for genetic programming
`show()`	Show an object

`summary.DEoptim()`	Summarize differential evolution result (package `DEoptim`)
`summary.rbga()`	Summarize genetic/evolutionary algorithm result (package `genalg`)
`suppressWarnings()`	Evaluates its expression and ignores all warnings
`try`	Runs an expression that might fail and handles error-recovery
`VEDA()`	Implement EDAs based on vines (package `copulaedas`)
`vignette()`	View a particular vignette or list available ones

5.10 Exercises

5.1. Apply a genetic algorithm to optimize the binary **max sin** task with $D = 16$ (from Exercise 4.2), using a population size of 20, elitism of 1, and maximum of 100 iterations. Show the best solution and fitness value.

5.2. Consider the **eggholder** function ($D = 2$):

$$f = -(x_2 + 47)\sin\left(\sqrt{|x_2 + x_1/2 + 47|}\right) - x_1 \sin\left(\sqrt{|x_1 - x_2 + 47|}\right) \tag{5.3}$$

Adapt the code of file `compare2.R` (Sect. 5.6) such that three methods are compared to minimize the **eggholder** task: Monte Carlo search (Sect. 3.4), particle swarm optimization (SPSO 2011), and EDA (DVEDA). Use ten runs for each method, with a maximum number of evaluations set to `MAXFN=1000` and solutions searched within the range [−512,512]. Consider the percentage of successes below −950. For the population based methods, use a population size of $N_P = 20$ and maximum number of iterations of $maxit = 50$.

5.3. Consider the original **bag prices** task ($D = 5$, Sect. 1.7) with a new hard constraint: $x_1 > x_2 > x_3 > x_4 > x_5$. Adapt the code of Sect. 5.7 in order to compare death penalty and repair constraint handling strategies using an EDA of type UMDA. Hint: consider a simple repair solution that reorders each infeasible solution into a feasible one.

5.4. Approximate the **eggholder** function of Exercise 5.2 using a genetic programming method with a population size of 100 and other default parameters. The genetic programming building blocks should be defined as:

- function symbols—use the same functions/operators that appear at the **eggholder** equation;
- constants—use a random sampling over the **eggholder** constants {2,47}; and
- variables—two inputs (x_1 and x_2).

Set the domain input with 500 samples randomly generated within the range $[-512, 512]$ and stop the algorithm after 20 s.

Chapter 6
Multi-Objective Optimization

6.1 Introduction

In previous chapters, only single objective tasks were addressed. However, multiple goals are common in real-world domains. For instance, a company typically desires to increase sales while reducing production costs. Within its marketing department, the goal might include maximizing target audiences while minimizing the marketing budget. Also, within the production department, the same company might want to maximize the manufactured items, in terms of both quality and production numbers, while minimizing production time, costs, and waste of material. Often, the various objectives can conflict, where gaining in one goal involves losing in another one. Thus, there is a need to set the right trade-offs.

To handle multi-objective tasks, there are three main approaches (Freitas 2004): weighted-formula, lexicographic and Pareto front, whose R implementation details are discussed in the next sections, after presenting the demonstrative tasks selected for this chapter.

6.2 Multi-Objective Demonstrative Problems

This section includes three examples of simple multi-objective tasks that were selected to demonstrate the methods presented in this chapter. Given an D-dimensional variable vector $\mathbf{x} = \{x_1, \ldots, x_D\}$ the goal is to optimize a set of m objective functions $\{f_1(x_1, \ldots, x_D), \ldots, f_m(x_1, \ldots, x_D)\}$. To simplify the demonstrations, only two $m = 2$ objective functions are adopted for each task (an $m = 3$ example is shown in Sect. 7.6).

The binary multi-objective goal consists in maximizing both functions of the set $\{f_{\text{sum of bits}}(x_1, \ldots, x_D), f_{\text{max sin}}(x_1, \ldots, x_D)\}$, where $x_i \in \{0, 1\}$ and the functions are defined in Eqs. (1.1) and (1.2). As explained in Sect. 1.7 (see also Fig. 1.3),

P. Cortez, *Modern Optimization with R*, Use R!, DOI 10.1007/978-3-319-08263-9_6

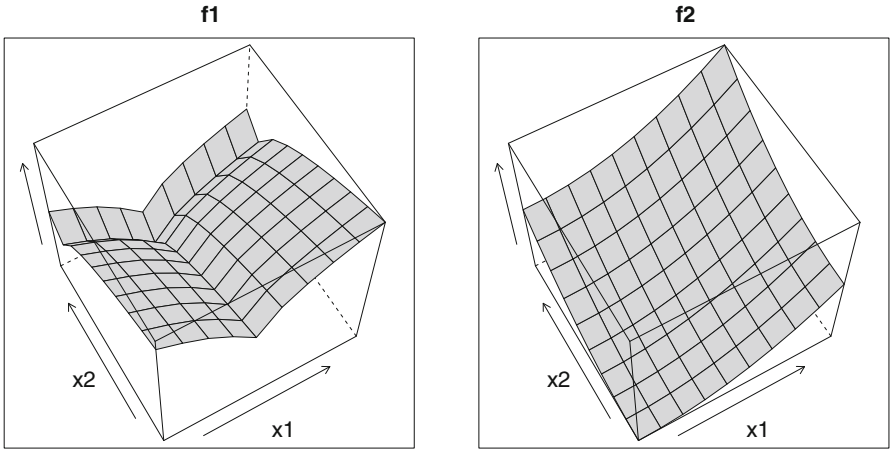

Fig. 6.1 Example of the FES1 f_1 (*left*) and f_2 (*right*) task landscapes ($D = 2$)

when $D = 8$ the optimum solutions are set at different points of the search space (**x**=(1,1,1,1,1,1,1,1) for f_1 and **x**=(1,0,0,0,0,0,0,0) for f_2), thus a trade-off is needed.

The **bag prices** integer multi-objective goal is set by maximizing f_1 and minimizing f_2, where $f_1 = f_{\text{bag prices}}$ and $f_2 = \sum_{i=1}^{D} sales(x_i)$, i.e., the number of bags that the factory will produce (see Sect. 1.7).

Finally, the real value multi-objective goal is defined in terms of the **FES1** benchmark (Huband et al. 2006), which involves minimizing both functions of the set:

$$\{f_1 = \sum_{i=1}^{D} |x_i - \exp\left((i/D)^2\right)/3|^{0.5}, f_2 = \sum_{i=1}^{D} (x_i - 0.5\cos(10\pi i/D) - 0.5)^2\} \tag{6.1}$$

where $x_i \in [0, 1]$. As shown in Fig. 6.1, the minimum solutions are set at distinct points of the search space.

The R code related with the three multi-optimization tasks is presented in file `mo-tasks.R`:

```
### mo-tasks.R file ###

# binary multi-optimization goal:
sumbin=function(x) (sum(x))
intbin=function(x) sum(2^(which(rev(x==1))-1))
maxsin=function(x) # max sin (explained in Chapter 3)
{ D=length(x);x=intbin(x)
  return(sin(pi*(as.numeric(x))/(2^D))) }

# integer multi-optimization goal:
profit=function(x)     # x - a vector of prices
{ x=round(x,digits=0) # convert x into integer
```

```
  s=sales(x)           # get the expected sales
  c=cost(s)            # get the expected cost
  profit=sum(s*x-c)    # compute the profit
  return(profit)
}
cost=function(units,A=100,cpu=35-5*(1:length(units)))
{ return(A+cpu*units) }
sales=function(x,A=1000,B=200,C=141,
               m=seq(2,length.out=length(x),by=-0.25))
{ return(round(m*(A/log(x+B)-C),digits=0))}
produced=function(x) sum(sales(round(x)))

# real value FES1 benchmark:
fes1=function(x)
{ D=length(x);f1=0;f2=0
  for(i in 1:D)
    { f1=f1+abs(x[i]-exp((i/D)^2)/3)^0.5
      f2=f2+(x[i]-0.5*cos(10*pi/D)-0.5)^2
    }
  return(c(f1,f2))
}
```

6.3 Weighted-Formula Approach

The weighted-formula approach, also known as priori approach, has the advantage of being the simplest multi-objective solution, thus it is more easy to implement. This approach involves first assigning weights to each goal and then optimizing a quality Q measure that is typically set using an additive or multiplicative formula:

$$\begin{aligned} Q &= w_1 \times g_1 + w_2 \times g_2 + \ldots + w_n \times g_n \\ Q &= g_1^{w_1} \times g_1^{w_2} \times \ldots \times g_n^{w_n} \end{aligned} \tag{6.2}$$

where $g_1, g_2, \ldots, g_n$ denote the distinct goals and $w_1, w_2, \ldots, w_n$ the assigned weights.

As discussed in Freitas (2004) and Konak et al. (2006), there are several disadvantages with the weighted-formula approach. First, setting the ideal weights is often difficult and thus weights tend to be set ad-hoc (e.g., based on intuition). Second, solving a problem to optimize Q for a particular vector $\mathbf{w}$ yields a single solution. This means that optimizing with a different combination of weights requires the execution of a new optimization procedure. Third, even if the weights are correctly defined, the search will miss trade-offs that might be interesting for the user. In particular, the linear combination of weights (as in the additive formula) limits the search for solutions in a non-convex region of the Pareto front (Fig. 6.2).

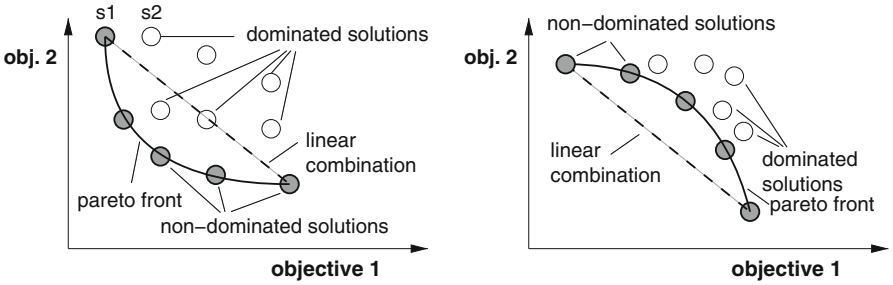

Fig. 6.2 Examples of convex (*left*) and non-convex (*right*) Pareto fronts, where the goal is to minimize both objectives 1 and 2

To solve the first two weighted-formula limitations, enhanced optimization variants have been proposed. One interesting example is the weight-based genetic algorithm (WBGA), which encodes a different weight vector into each solution of the genetic population (Konak et al. 2006).

In this section, a pure weighted-formula approach is adopted for the three tasks presented in Sect. 6.2. Five additive weight combinations are tested: $\mathbf{w}_1 = (1.00, 0.00)$, $\mathbf{w}_2 = (0.75, 0.25)$, $\mathbf{w}_3 = (0.50, 0.50)$, $\mathbf{w}_4 = (0.75, 0.25)$, and $\mathbf{w}_5 = (0.00, 1.00)$. It should be noted that in all three tasks, there are different scales for each of the objectives (e.g., [0,8] range for $f_{\text{sum of bits}}$ and [0,1] for $f_{\text{max sin}}$). Thus, the optimization method will tend to improve more the objective associated with the largest scale, unless more differentiated weights are used. Nevertheless, for the sake of simplicity, the same weight combinations are used for all three benchmarks.

As the search engine, genetic and evolutionary algorithms are adopted, as implemented in the `genalg` package. The advantage of this package is that it can handle both binary (`rbga.bin` function) and real value (`rbga` function) representations. A distinct run of the optimization algorithm is executed for each of the five weight combinations. The population size is set to 20 individuals for **bag prices** and **FES1** multi-objective tasks, while a smaller population size of 12 is used for the simpler binary multi-objective problem. The weighted-formula R code is presented in file `wf-test.R`:

```
### wf-test.R file ###

source("mo-tasks.R") # load multi-optimization tasks
library(genalg) # load genalg package

set.seed(12345) # set for replicability

step=5 # number of weight combinations
w=matrix(ncol=2,nrow=step) # weight combinations
w[,1]=seq(1,0,length.out=step)
w[,2]=1-w[,1]

print("Weight combinations:")
print(w)
```

```
# --- binary task:
D=8 # 8 bits
eval=function(x) return(W[1]*sumbin(x)+W[2]*maxsin(x))
cat("binary task:\n")
for(i in 1:step)
{
 W= -w[i,] # rbga.bin minimization goal: max. f1 and max. f2
 G=rbga.bin(size=D,popSize=12,iters=100,zeroToOneRatio=1,
            evalFunc=eval,elitism=1)
 b=G$population[which.min(G$evaluations),] # best individual
 cat("w",i,"best:",b)
 cat(" f=(",sumbin(b),",",round(maxsin(b),2),")","\n",sep="")
}

# --- integer task:
D=5 # 5 bag prices
eval=function(x) return(W[1]*profit(x)+W[2]*produced(x))
cat("integer task:\n")
res=matrix(nrow=nrow(w),ncol=ncol(w)) # for CSV files
for(i in 1:step)
{
 W=c(-w[i,1],w[i,2]) # rbga min. goal: max. f1 and min. f2
 G=rbga(evalFunc=eval,stringMin=rep(1,D),stringMax=rep(1000,D),
        popSize=20,iters=100)
 b=round(G$population[which.min(G$evaluations),]) # best
 cat("w",i,"best:",b)
 cat(" f=(",profit(b),",",produced(b),")","\n",sep="")
 res[i,]=c(profit(b),produced(b))
}
write.table(res,"wf-bag.csv",
            row.names=FALSE,col.names=FALSE,sep=" ")
# --- real value task:
D=8 # dimension
eval=function(x) return(sum(W*fes1(x)))
cat("real value task:\n")
for(i in 1:step)
{
 W=w[i,] # rbga minimization goal
 G=rbga(evalFunc=eval,stringMin=rep(0,D),stringMax=rep(1,D),
        popSize=20,iters=100)
 b=G$population[which.min(G$evaluations),] # best solution
 cat("w",i,"best:",round(b,2))
 cat(" f=(",round(fes1(b)[1],2),",",round(fes1(b)[2],2),")",
     "\n",sep="")
 res[i,]=fes1(b)
}
write.table(res,"wf-fes1.csv",
            row.names=FALSE,col.names=FALSE,sep=" ")
```

The distinct weight combinations are stored in matrix `w`. Given that the `genalg` package performs a minimization, the $f'(s) = -f(s)$ transformation (Sect. 1.4) is adopted when the objective requires a maximization and thus the auxiliary `W` vector is used to multiple the weight values by -1 when needed. After executing

each optimization run, the code displays the best evolved solution and also the two objective evaluation values. For comparison with other multi-objective approaches, the best evaluation values are stored into CSV files (using the `write.table` function) for the last two tasks. The execution result is:

```
> source("wf-test.R")
[1] "Weight combinations:"
     [,1] [,2]
[1,] 1.00 0.00
[2,] 0.75 0.25
[3,] 0.50 0.50
[4,] 0.25 0.75
[5,] 0.00 1.00
binary task:
w 1 best: 1 1 1 1 1 1 1 1 f=(8,0.01)
w 2 best: 1 1 1 1 1 1 1 1 f=(8,0.01)
w 3 best: 1 1 1 1 1 1 1 1 f=(8,0.01)
w 4 best: 0 1 1 1 1 1 1 1 f=(7,1)
w 5 best: 0 1 1 1 1 1 1 1 f=(7,1)
integer task:
w 1 best: 420 362 419 367 415 f=(43165,117)
w 2 best: 425 433 408 390 410 f=(43579,112)
w 3 best: 412 390 407 305 446 f=(43435,120)
w 4 best: 399 418 438 405 372 f=(43499,114)
w 5 best: 986 969 969 913 991 f=(4145,5)
real value task:
w 1 best: 0.32 0.35 0.37 0.42 0.51 0.57 0.74 0.91 f=(0.92,1.44)
w 2 best: 0.36 0.33 0.38 0.43 0.51 0.58 0.68 0.91 f=(0.87,1.41)
w 3 best: 0.36 0.34 0.39 0.43 0.5 0.58 0.49 0.89 f=(1.11,1.21)
w 4 best: 0.33 0.35 0.39 0.42 0.49 0.36 0.29 0.25 f=(2.3,0.4)
w 5 best: 0.18 0.15 0.16 0.23 0.16 0.16 0.15 0.17 f=(4.61,0.01)
```

As expected, the obtained results show that in general the genetic and evolutionary algorithms manage to get the best f_1 values for the $\mathbf{w}_1 = (1.00, 0.00)$ weight combination and best f_2 values for the $\mathbf{w}_5 = (0.00, 1.00)$ vector of weights. The quality of the remaining task evolved solutions (i.e., for **bag prices** and **FES1**) will be discussed in Sect. 6.5.

6.4 Lexicographic Approach

Under the lexicographic approach, different priorities are assigned to different objectives, such that the objectives are optimized in their priority order (Freitas 2004). When two solutions are compared, first the evaluation measure for the highest-priority objective is compared. If the first solution is significantly better (e.g., using a given tolerance value) than the second solution, then the former is chosen. Else, the comparison is set using the second highest-priority objective. The process is repeated until a clear winner is found. If there is no clear winner, then the solution with the best highest-priority objective can be selected.

In Freitas (2004), the advantages and disadvantages of the lexicographic approach are highlighted. When compared with the weighted-formula, the lexicographic approach has the advantage of avoiding the problem of mixing non-commensurable criteria in the same formula, as it treats each criterion separately. Also, if the intention is to just to compare several solutions, then the lexicographic approach is easier when compared with the Pareto approach. However, the lexicographic approach requires the user to a priori define the criteria priorities and tolerance thresholds, which similarly to the weighted-formula are set ad-hoc.

Given that in previous section an evolutionary/genetic algorithm was adopted, the presented lexicographic implementation also adopts the same base algorithm. In particular, the `rbga.bin()` function code is adapted by replacing the probabilistic (roulette wheel) selection with a tournament selection. This operator works by randomly sampling k individuals (solutions) from the population and then selects the best n individuals (Goldberg and Deb 1991). The advantage of using tournament is that there is no need for a single fitness value, since the selection of what is the "best" can be performed under a lexicographic comparison with the k solutions. It should be noted that the same tournament function could be used to get other multi-objective optimization adaptations. For instance, a lexicographic hill climbing could easily be achieved by setting the *best* function of Algorithm 2 as the same tournament operator (in this case by comparing $k = 2$ solutions).

The lexicographic genetic algorithm R code is provided in the file `lg-ga.R`:

```
### lg-ga.R file ###

# lexicographic comparison of several solutions:
#    x - is a matrix with several objectives at each column
#        and each row is related with a solution
lexibest=function(x) # assumes LEXI is defined
{
 size=nrow(x); m=ncol(x)
 candidates=1:size
 stop=FALSE; i=1
 while(!stop)
   {
    F=x[candidates,i] # i-th goal
    minFID=which.min(F) # minimization goal is assumed
    minF=F[minFID]
    # compute tolerance value
    if(minF>-1 && minF<1) tolerance=LEXI[i]
    else tolerance=abs(LEXI[i]*minF)
    I=which((F-minF)<=tolerance)
    if(length(I)>0) # at least one candidate
      candidates=candidates[I] # update candidates
    else stop=TRUE
    if(!stop && i==m) stop=TRUE
    else i=i+1
   }
 if(length(candidates)>1)
   { # return highest priority goal if no clear winner:
     stop=FALSE; i=1
```

```
     while(!stop)
       {
        minF=min(x[candidates,i])
        I=which(x[candidates,i]==minF)
        candidates=candidates[I]
        if(length(candidates)==1||i==m) stop=TRUE
        else i=i+1
       }
     # remove (any) extra duplicate individuals:
     candidates=candidates[1]
   }
 # return lexibest:
 return(candidates)
}

# compare k randomly selected solutions from Population:
#    returns n best indexes of Population (decreasing order)
#    m is the number of objectives
tournament=function(Population,evalFunc,k,n,m=2)
{
 popSize=nrow(Population)
 PID=sample(1:popSize,k) # select k random tournament solutions
 E=matrix(nrow=k,ncol=m) # evaluations of tournament solutions
 for(i in 1:k) # evaluate tournament
    E[i,]=evalFunc(Population[PID[i],])

 # return best n individuals:
 B=lexibest(E); i=1; res=PID[B] # best individual
 while(i<n) # other best individuals
   {
     E=E[-B,];PID=PID[-B] # all except B
     if(is.matrix(E)) B=lexibest(E)
     else B=1 # only 1 row
     res=c(res,PID[B])
     i=i+1
   }
 return(res)
}

# lexicographic adapted version of rbga.bin:
#    this function is almost identical to rbga.bin except that
#    the code was simplified and a lexicographic tournament is
    used
#    instead of roulette wheel selection
lrbga.bin=function(size=10, suggestions=NULL, popSize=200,
                   iters=100, mutationChance=NA, elitism=NA,
                   zeroToOneRatio=10,evalFunc=NULL)
{
 vars=size
 if(is.na(mutationChance)) { mutationChance=1/(vars + 1) }
 if(is.na(elitism)) { elitism=floor(popSize/5)}
 if(!is.null(suggestions))
   {
```

```
    population=matrix(nrow=popSize, ncol=vars)
    suggestionCount=dim(suggestions)[1]
    for(i in 1:suggestionCount)
      population[i, ]=suggestions[i, ]
    for(child in (suggestionCount + 1):popSize)
      {
       population[child, ]=sample(c(rep(0, zeroToOneRatio),1),
           vars,rep=TRUE)
       while(sum(population[child, ])==0)
          population[child, ]=sample(c(rep(0, zeroToOneRatio),
              1),vars,rep=TRUE)
      }
  }
else
  {
   population=matrix(nrow=popSize, ncol=vars)
   for(child in 1:popSize)
     {
      population[child,]=sample(c(rep(0, zeroToOneRatio),1),
          vars,rep=TRUE)
      while (sum(population[child, ]) == 0)
        population[child, ]=sample(c(rep(0, zeroToOneRatio),1),
            vars,rep=TRUE)
     }
  }
# main GA cycle:
for(iter in 1:iters)
  {
   newPopulation=matrix(nrow=popSize, ncol=vars)
   if(elitism>0) # applying elitism:
     {
      elitismID=tournament(population,evalFunc,k=popSize,n=
          elitism)
      newPopulation[1:elitism,]=population[elitismID,]
     }
   #  applying crossover:
   for(child in (elitism + 1):popSize)
     {
      ### very new code inserted here : ###
      pID1=tournament(population,evalFunc=evalFunc,k=2,n=1)
      pID2=tournament(population,evalFunc=evalFunc,k=2,n=1)
      parents=population[c(pID1,pID2),]
      ### end of very new code          ###
      crossOverPoint=sample(0:vars, 1)
      if(crossOverPoint == 0)
        newPopulation[child,]=parents[2,]
      else if(crossOverPoint == vars)
        newPopulation[child, ]=parents[1, ]
      else
        {
         newPopulation[child,]=c(parents[1,][1:crossOverPoint],
             parents[2,][(crossOverPoint+1):vars])
         while(sum(newPopulation[child,])==0)
```

```
            newPopulation[child, ]=sample(c(rep(0,zeroToOneRatio
                ),1),vars,rep=TRUE)
          }
      }
    population=newPopulation # store new population
    if(mutationChance>0) # applying mutations:
      {
       mutationCount=0
       for(object in (elitism+1):popSize)
         {
          for(var in 1:vars)
            {
             if(runif(1)< mutationChance)
               {
                population[object, var]=sample(c(rep(0,
                    zeroToOneRatio),1),1)
                mutationCount=mutationCount+1
               }
            }
         }
      }
   } # end of GA main cycle
 result=list(type="binary chromosome",size=size,popSize=popSize,
        iters=iters,suggestions=suggestions,
        population=population,elitism=elitism,
        mutationChance=mutationChance)
 return(result)
}
```

The new `lrbga.bin()` is a simplified version of the `rbga.bin()` function (which is accessible by simple typing `> rbga.bin` in the R console), where all verbose and monitoring code has been removed. The most important change is that before applying the crossover a tournament with k=2 individuals is set to select the two parents. It should be noted that $k = 2$ is the most popular tournament strategy (Michalewicz and Fogel 2004). The same tournament operator is also used to select the elitism individuals from the population (in this case with: $k = N_p$ – population size; and $n = E$ – elitism). The `tournament()` function returns the n best individuals, according to a lexicographic comparison. Under this implementation, the evaluation function needs to return vector with the fitness values for all m objectives.

The `tournament()` assumes the first objective as the highest priority function, the second objective is considered the second highest priority function, and so on. It should be noted that `tournament()` uses the `is.matrix()` R function, which returns true if `x` is a matrix object. The lexicographic comparison is only executed when there are two or more solutions (which occurs when `x` object is a matrix). Function `lexibest()` implements the lexicographic comparison, returning the best index of the tournament population. This function assumes that the tolerance thresholds are defined in object `LEXI`. Also, these thresholds are interpreted as percentages if $-1 < f_i < 1$ for the i-th objective, else absolute values are used. Working from the highest priority to the smallest one, the tournament

population is reduced on a step by step basis, such that on each iteration only the best solutions within the tolerance range for the i-th objective are selected. If there is no clear winner, `lexibest()` selects the best solution, as evaluated from the highest to the smallest priority objective.

The optimization of the binary multi-objective goal is coded in file `lg-test.R`, using a tolerance of 20 % for both objectives and the same other parameters that were used in Sect. 6.3:

```
### lg-test.R file ###

source("mo-tasks.R") # load multi-optimization tasks
source("lg-ga.R") # load lrgba.bin
set.seed(12345) # set for replicability

LEXI=c(0.2,0.2) # tolerance 20% for each goal
cat("tolerance thresholds:",LEXI,"\n")

# --- binary task:
D=8 # 8 bits
# eval: transform binary objectives into minimization goal
#       returns a vector with 2 values, one per objective:
eval=function(x) return(c(-sumbin(x),-maxsin(x)))
popSize=12
G=lrbga.bin(size=D,popSize=popSize,iters=100,zeroToOneRatio=1,
            evalFunc=eval,elitism=1)
print("Ranking of last population:")
B=tournament(G$population,eval,k=popSize,n=popSize,m=2)
for(i in 1:popSize)
{
 x=G$population[B[i],]
 cat(x," f=(",sumbin(x),",",round(maxsin(x),2),")","\n",sep="")
}
```

Given that there is not a single best solution, after executing the lexicographic genetic algorithm, the code shows a ranking (according to the lexicographic criterion) of all individuals from last population:

```
> source("lg-test.R")
tolerance thresholds: 0.2 0.2
[1] "Ranking of last population:"
01111111 f=(7,1)
01111111 f=(7,1)
01111111 f=(7,1)
01111111 f=(7,1)
01111111 f=(7,1)
01111111 f=(7,1)
01111111 f=(7,1)
01110111 f=(6,0.99)
01011111 f=(6,0.92)
10101111 f=(6,0.84)
10101111 f=(6,0.84)
01010111 f=(5,0.88)
```

With a single run, the lexicographic algorithm is capable of finding the same $(f_1, f_2) = (7, 1)$ solution that belongs to the Pareto front (see next section).

6.5 Pareto Approach

A solution s_1 dominates (in the Pareto sense) a solution s_2 if s_1 is better than s_2 in one objective and as least as good as s_2 in all other objectives. A solution s_i is non-dominated when there is no solution s_j that dominates s_i and the Pareto front contains all non-dominated solutions (Luke 2012). An example situation is shown in the left of Fig. 6.1, where s_1 is a non-dominated solution and part of the Pareto front, while s_2 is a dominated one (both solutions have the same f_2 value but s_1 presents a better f_1). Assuming this concept, Pareto multi-objective optimization methods return a set of non-dominated solutions (from the Pareto front), rather than just a single solution.

When compared with previous approaches (weighted-formula and lexicographic), the Pareto multi-objective optimization presents several advantages (Freitas 2004). It is a more natural method, since a "true" multi-objective approach is executed, providing to the user an interesting set of distinct solutions and letting the user (a posteriori) to decide which one is best. Moreover, under a single execution, the method optimizes the distinct objectives, thus no multiple runs are required to get the Pareto front points. In addition, there is no need to set ad-hoc weights or tolerance values. The drawback of the Pareto approach is that a larger search space needs to be explored and tracked, thus Pareto based methods tend to be more complex than single-objective counterparts.

Considering that Pareto based methods need to keep track of a population of solutions, evolutionary algorithms have become a natural and popular solution to generate Pareto optimal solutions. Examples of standard evolutionary multi-objective approaches include the strength Pareto evolutionary algorithm 2 (SPEA-2) and non-dominated sorting genetic algorithm-II (NSGA-II) (Deb 2001). Multi-objective evolutionary algorithms (MOEA) often use Pareto-based ranking schemes, where individuals in the Pareto front are rank 1, then the front solutions are removed and the individuals from the new front are rank 2, and so on.

The NSGA-II is implemented in the `mco` package and thus it is adopted in this section. NSGA-II is an evolutionary algorithm variant specifically designed for multi-objective optimization and that uses three useful concepts: Pareto front ranking, elitism, and sparsity. The full NSGA-II algorithmic details can be found in Deb (2001) and Luke (2012), although the skeleton of the algorithm is similar to Algorithm 5. The initial population P is randomly generated and then a cycle is executed until a termination criterion is met. Within each cycle, NSGA-II uses a Pareto ranking scheme to assign a ranking number to each individual of the population. An elitism scheme is also adopted, storing an archive (P_E) of the best individuals. The elitism individuals are selected taking into account their rank number and also their sparsity. An individual is in a sparse region if the neighbor

individuals are not too close to it. To measure how close two points are, a distance metric is used, such as Manhattan distance, which is defined by the sum of all m objective differences between the two points. Then, a new population ($Children \leftarrow breed(P_E)$) is created, often by using a tournament selection (e.g., with $k = 2$), crossover, and mutation operators. The next population is set as the union of the archive ($P \leftarrow Children \cup P_E$) and a new cycle is executed over P.

In the `mco` package, NSGA-II is implemented with the `nsga2` function. The package also contains other functions, such as related with multi-objective benchmarks (e.g., `belegundu()`) and Pareto front (e.g., `paretoSet()`). The useful `nsga2()` function performs a minimization of vectors of real numbers and includes the main parameters:

- `fn`—function to be minimized (should return a vector with the several objective values);
- `idim`—input dimension (D);
- `odim`—output dimension (number of objective functions, m);
- `...`—extra arguments to be passed `fn`;
- `lower.bounds`, `upper.bounds`—lower and upper bounds;
- `popsize`—population size (N_P, default is 100);
- `generations`—number of generations ($maxit$, default is 100) or a vector;
- `cprob`—crossover probability (default is 0.7); and
- `mprob`—mutation probability (default is 0.2).

The function returns a list with the final population (if `generations` is a number), with components:

- `$par`—the population values;
- `$value`—matrix with the best objective values (in columns) for the last population individuals (in rows); and
- `$pareto.optimal`—a boolean vector that indicates which individuals from the last generation belong to the Pareto front.

When `generations` is a vector, a vector list is returned where the i-th element contains the population after `generations[i]` iterations (an R code example is shown in the next presented code).

File `ngsa2-test.R` codes the optimization of the three multi-objective tutorial tasks under the NSGA-II algorithm:

```
### nsga2-test.R file ###

source("mo-tasks.R") # load multi-optimization tasks
library(mco) # load mco package

set.seed(12345) # set for replicability
m=2 # two objectives

# --- binary task:
D=8 # 8 bits
# eval: transform binary objectives into minimization goal
```

```
#       round(x) is used to convert real number to 0 or 1 values
eval=function(x) c(-sumbin(round(x)),-maxsin(round(x)))
cat("binary task:\n")
G=nsga2(fn=eval,idim=D,odim=m,
        lower.bounds=rep(0,D),upper.bounds=rep(1,D),
        popsize=12,generations=100)
# show last Pareto front
I=which(G$pareto.optimal)
for(i in I)
{
 x=round(G$par[i,])
 cat(x," f=(",sumbin(x),",",round(maxsin(x),2),")","\n",sep="")
}

# --- integer task:
D=5 # 5 bag prices
# eval: transform objectives into minimization goal
eval=function(x) c(-profit(x),produced(x))
cat("integer task:\n")
G=nsga2(fn=eval,idim=5,odim=m,
        lower.bounds=rep(1,D),upper.bounds=rep(1000,D),
        popsize=20,generations=1:100)
# show best individuals:
I=which(G[[100]]$pareto.optimal)
for(i in I)
{
 x=round(G[[100]]$par[i,])
 cat(x," f=(",profit(x),",",produced(x),")","\n",sep=" ")
}
# create PDF with Pareto front evolution:
pdf(file="nsga-bag.pdf",paper="special",height=5,width=5)
par(mar=c(4.0,4.0,0.1,0.1))
I=1:100
for(i in I)
{ P=G[[i]]$value # objectives f1 and f2
  P[,1]=-1*P[,1] # show positive f1 values
  # color from light gray (75) to dark (1):
  COL=paste("gray",round(76-i*0.75),sep="")
  if(i==1) plot(P,xlim=c(-500,44000),ylim=c(0,140),
                xlab="f1",ylab="f2",cex=0.5,col=COL)
  Pareto=P[G[[i]]$pareto.optimal,]
  # sort Pareto according to x axis:
  I=sort.int(Pareto[,1],index.return=TRUE)
  Pareto=Pareto[I$ix,]
  points(P,type="p",pch=1,cex=0.5,col=COL)
  lines(Pareto,type="l",cex=0.5,col=COL)
}
dev.off()

# create PDF comparing NSGA-II with WF:
pdf(file="nsga-bag2.pdf",paper="special",height=5,width=5)
par(mar=c(4.0,4.0,0.1,0.1))
# NSGA-II best results:
```

```
P=G[[100]]$value # objectives f1 and f2
P[,1]=-1*P[,1] # show positive f1 values
Pareto=P[G[[100]]$pareto.optimal,]
# sort Pareto according to x axis:
I=sort.int(Pareto[,1],index.return=TRUE)
plot(Pareto[I$ix,],xlim=c(-500,44000),ylim=c(0,140),
     xlab="f1",ylab="f2",type="b",lwd=2,lty=1,pch=1)
# weight-formula best results:
wf=read.table("wf-bag.csv",sep=" ")
I=sort.int(wf[,1],index.return=TRUE)
lines(wf[I$ix,],type="b",lty=2,lwd=2,pch=3)
legend("topleft",c("NSGA-II","weighted-formula"),
       lwd=2,lty=1:2,pch=c(1,3))
dev.off()

# --- real value task:
D=8 # dimension
cat("real value task:\n")
G=nsga2(fn=fes1,idim=D,odim=m,
        lower.bounds=rep(0,D),upper.bounds=rep(1,D),
        popsize=20,generations=1:100)
# show best individuals:
I=which(G[[100]]$pareto.optimal)
for(i in I)
{
 x=round(G[[100]]$par[i,],digits=2); cat(x)
 cat(" f=(",round(fes1(x)[1],2),",",round(fes1(x)[2],2),")",
     "\n",sep="")
}
# create PDF with Pareto front evolution:
pdf(file="nsga-fes1.pdf",paper="special",height=5,width=5)
par(mar=c(4.0,4.0,0.1,0.1))
I=1:100
for(i in I)
{ P=G[[i]]$value # objectives f1 and f2
  # color from light gray (75) to dark (1):
  COL=paste("gray",round(76-i*0.75),sep="")
  if(i==1) plot(P,xlim=c(0.5,5.0),ylim=c(0,2.0),
                xlab="f1",ylab="f2",cex=0.5,col=COL)
  Pareto=P[G[[i]]$pareto.optimal,]
  # sort Pareto according to x axis:
  I=sort.int(Pareto[,1],index.return=TRUE)
  Pareto=Pareto[I$ix,]
  points(P,type="p",pch=1,cex=0.5,col=COL)
  lines(Pareto,type="l",cex=0.5,col=COL)
}
dev.off()

# create PDF comparing NSGA-II with WF:
pdf(file="nsga-fes1-2.pdf",paper="special",height=5,width=5)
par(mar=c(4.0,4.0,0.1,0.1))
# NSGA-II best results:
P=G[[100]]$value # objectives f1 and f2
```

```
Pareto=P[G[[100]]$pareto.optimal,]
# sort Pareto according to x axis:
I=sort.int(Pareto[,1],index.return=TRUE)
plot(Pareto[I$ix,],xlim=c(0.5,5.0),ylim=c(0,2.0),
     xlab="f1",ylab="f2",type="b",lwd=2,pch=1)
# weight-formula best results:
wf=read.table("wf-fes1.csv",sep=" ")
I=sort.int(wf[,1],index.return=TRUE)
lines(wf[I$ix,],type="b",lty=2,lwd=2,pch=3)
legend("top",c("NSGA-II","weighted-formula"),
       lwd=2,lty=1:2,pch=c(1,3))
dev.off()
```

The execution of function `nsga2` is straightforward taking into account the weight-formula and lexicographic examples. For the binary task, each solution parameter ($\in [0, 1]$) is first rounded in order to transform it into a binary number, since `nsga2()` only works with real values. After calling the algorithm, the code shows all Pareto front solutions from the last generation. For each of the **bag prices** and **FES1** tasks, the code also creates two PDF files. The first PDF contains the search evolution in terms of the f_1 (x-axis) and f_2 (y-axis) objectives, where individual solutions are represented by small circle points and the Pareto front solutions are connected with lines. Also, a varying color scheme is adopted to plot the points and lines, ranging from light gray (first generation) to black (last generation). The second PDF compares the best Pareto front optimized by NSGA-II with the five solutions obtained by the five runs (with different weight combinations) executed for the weighted-formula approach. The execution result is:

```
> source("nsga2-test.R")
binary task:
11111111 f=(8,0.01)
01111111 f=(7,1)
11111111 f=(8,0.01)
01111111 f=(7,1)
11111111 f=(8,0.01)
01111111 f=(7,1)
11111111 f=(8,0.01)
11111111 f=(8,0.01)
11111111 f=(8,0.01)
11111111 f=(8,0.01)
11111111 f=(8,0.01)
01111111 f=(7,1)
integer task:
414 403 406 431 394  f=( 43736 , 114 )
1000 996 993 989 988  f=( -500 , 0 )
752 944 929 871 999  f=( 12944 , 17 )
813 649 872 971 791  f=( 19729 , 28 )
1000 934 979 942 996  f=( 3204 , 4 )
803 967 645 627 745  f=( 22523 , 34 )
414 403 406 503 473  f=( 42955 , 107 )
554 629 591 443 563  f=( 38665 , 74 )
775 721 510 621 782  f=( 30643 , 50 )
436 494 494 614 565  f=( 40789 , 87 )
```

```
900 934 979 942 996  f=( 6684 , 8 )
807 498 506 641 707  f=( 32477 , 59 )
790 749 595 877 789  f=( 25273 , 37 )
979 882 957 938 794  f=( 8873 , 11 )
997 787 634 985 728  f=( 15792 , 24 )
775 634 725 602 782  f=( 29307 , 45 )
997 788 647 991 728  f=( 15328 , 23 )
432 494 494 433 563  f=( 42191 , 95 )
620 654 680 393 608  f=( 36061 , 67 )
946 672 668 622 638  f=( 26670 , 42 )
real value task:
0.15 0.12 0.12 0.11 0.16 0.15 0.16 0.15 f=(4.85,0)
0.34 0.35 0.38 0.43 0.49 0.59 0.72 0.91 f=(0.45,1.44)
0.34 0.35 0.38 0.43 0.49 0.59 0.35 0.5 f=(1.58,0.7)
0.34 0.35 0.39 0.25 0.2 0.13 0.14 0.13 f=(3.46,0.15)
0.22 0.18 0.39 0.27 0.2 0.12 0.13 0.13 f=(4.11,0.09)
0.34 0.35 0.39 0.43 0.3 0.59 0.34 0.46 f=(2.02,0.57)
0.34 0.35 0.38 0.43 0.49 0.5 0.72 0.91 f=(0.67,1.37)
0.34 0.35 0.38 0.43 0.49 0.59 0.72 0.72 f=(0.82,1.19)
0.19 0.36 0.12 0.11 0.16 0.19 0.16 0.15 f=(4.36,0.05)
0.34 0.35 0.39 0.43 0.2 0.13 0.07 0.14 f=(3.13,0.23)
0.34 0.35 0.38 0.42 0.49 0.59 0.72 0.67 f=(0.92,1.13)
0.34 0.35 0.39 0.41 0.24 0.59 0.23 0.48 f=(2.24,0.53)
0.34 0.35 0.39 0.43 0.11 0.35 0.45 0.48 f=(2.5,0.46)
0.34 0.35 0.38 0.43 0.49 0.59 0.72 0.5 f=(1.03,0.98)
0.33 0.35 0.39 0.43 0.39 0.4 0.13 0.33 f=(2.56,0.37)
0.33 0.35 0.39 0.43 0.3 0.4 0.13 0.3 f=(2.7,0.33)
0.22 0.35 0.39 0.42 0.48 0.12 0.13 0.16 f=(3.01,0.29)
0.29 0.35 0.38 0.43 0.49 0.59 0.72 0.49 f=(1.22,0.96)
0.34 0.35 0.38 0.43 0.47 0.59 0.69 0.43 f=(1.29,0.89)
0.34 0.35 0.38 0.42 0.5 0.58 0.72 0.16 f=(1.33,0.85)
```

For the sake of simplicity, the comparison of NSGA-II results with other approaches is performed using only a single NGSA-II run. It should be noted that for a more robust comparison, several runs should be applied, as shown in Sects. 4.5 and 5.6.

The binary multi-objective task is quite simple given that the optimum Pareto front only contains two solutions (found by NSGA-II): $(f_1, f_2) = (7, 1)$ and $(f_1, f_2) = (8, 0.01)$. When compared with the weight-formula, the same best solutions were obtained, although NSGA-II gets both solutions under the same run.

As shown in the left two graphs of Fig. 6.3, the **bag prices** and **FES1** are more complex multi-objective tasks when compared with the binary **sum of bits/max sin** problem. Figure 6.3 reveals that the optimum Pareto fronts seem to have a convex shape for both integer and real value optimization tasks, although the final shape of such Pareto curve is only achieved during the last generations of NSGA-II. The right plots of Fig. 6.3 confirm that NSGA-II optimizes within a single run a more interesting Pareto front when compared with the results obtained by the weighted-formula approach (and that requires executing 5 runs). For **bag prices**, the weighted approach misses several interesting solutions outside the linear combination (dashed line) extreme trade-offs. For **FES1**, the weighted approach is not able to get quality f_1 values when compared with NSGA-II results.

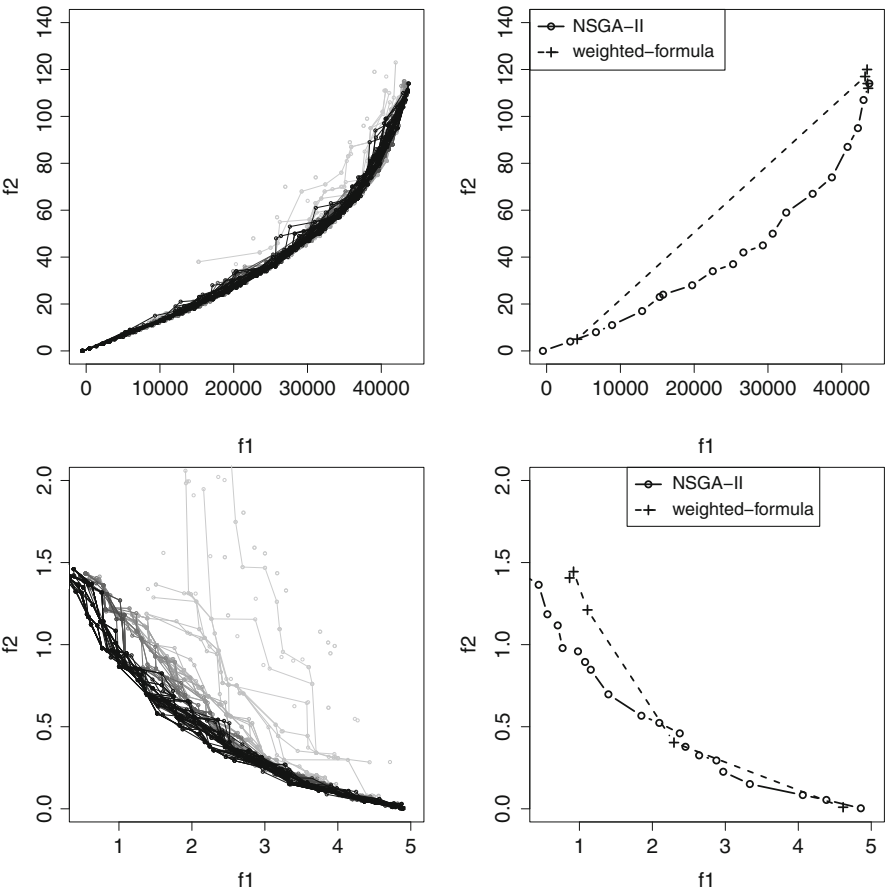

Fig. 6.3 NSGA-II results for **bag prices** (*top graphs*) and **FES1** (*bottom graphs*) tasks (*left graphs* show the Pareto front evolution, while *right graphs* compare the best Pareto front with the weighted-formula results)

6.6 Command Summary

`belegundu()`	multi-objective belegundu test problem (package `mco`)
`is.matrix()`	returns true if argument is a matrix
`lrbga.bin`	lexicographic genetic algorithm (chapter file `"lg-ga.R"`)
`mco`	package for multi-criteria optimization algorithms
`nsga2()`	NSGA-II algorithm (package `mco`)
`paretoSet()`	returns the Pareto front from a `mco` result object (package `mco`)
`tournament()`	tournament under a lexicographic approach (chapter file `"lg-ga.R"`)

6.7 Exercises

6.1. Encode the lexicographic hill climbing function `lhclimbing()`, which uses a lexicographic tournament with $k = 2$ to select the best solutions. Also, demonstrate the usefulness of `lhclimbing()` to optimize the **bag prices** multi-objective task under the tolerance vector $(0.1, 0.1)$, where the first priority is f_1 (profit). Show the best solution.

6.2. Consider the **FES2** task (Huband et al. 2006), where the goal is to minimize three functions under the range $x_i \in [0, 1]$:

$$\begin{aligned} f_1 &= \sum_{i=1}^{D} (x_i - 0.5\cos(10\pi i/D) - 0.5)^2 \\ f_2 &= \sum_{i=1}^{D} |x_i - \sin^2(i-1)\cos^2(i-1)|^{0.5} \\ f_2 &= \sum_{i=1}^{D} |x_i - 0.25\cos(i-1)\cos(2i-2) - 0.5|^{0.5} \end{aligned} \tag{6.3}$$

Using the `scatterplot3d()` function (package `scatterplot3d`), compare the final Pareto front solutions when exploring two multi-optimization approaches:

1. a WBGA, which encodes a different weight vector into each solution of the genetic population (use function `rgba`); and
2. NSGA-II algorithm.

For both approaches, use a population size of 20,100 generations and a dimension of $D = 8$.

Chapter 7
Applications

7.1 Introduction

Previous chapters have approached demonstrative optimization tasks that were synthetically generated. The intention was to present a tutorial perspective and thus more simpler tasks were approached. As a complement, this chapter addresses real-world applications, where the data available is taken from a physical phenomena. Exemplifying the optimization of real-world data in R is interesting for two main reasons. First, physical phenomena may contain surprising or unknown features. Second, it provides additional code examples of how to load real-world data into R.

This chapter addresses three real-world tasks that are discussed in the next sections: traveling salesman problem (TSP), time series forecasting (TSF), and wine quality classification.

7.2 Traveling Salesman Problem

The TSP is a classical combinatorial optimization. The goal is to find the cheapest way of visiting all cities (just once) of a given map, starting in one city and ending in the same city (Reinelt 1994). The complete traveling is known as Hamiltonian cycle or TSP tour. The standard TSP assumes symmetric costs, where the cost of visiting from city A to B is the same as visiting from B to A. This problem is non-deterministic polynomial (NP)-complete, which means that there is no algorithm capable of reaching the optimum solution in a polynomial time (in respect to n) for all possible TSP instances of size n. While being a complex task, TSP has been heavily studied in the last decades and a substantial improvement has been reached. For example, in 1954 an instance with 49 cities was the maximum TSP size solved, while in 2004 the record was established in 24,978 cities (Applegate et al. 2011).

Due to its importance, several optimization methods were devised to specifically address the TSP, such as 2-opt and concorde. The former method is a local search

P. Cortez, *Modern Optimization with R*, Use R!, DOI 10.1007/978-3-319-08263-9_7

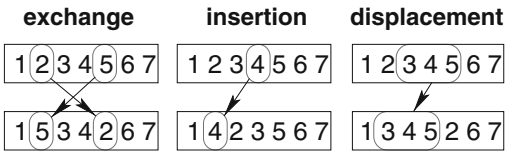

Fig. 7.1 Example of three order mutation operators

algorithm that is tested in this section and starts with a random tour and then exchanges two cities in order to avoid any two crossing paths until no improvements are possible (Croes 1958). The latter is a recent advanced exact TSP solver for symmetric TSPs and that is based on branch-and-cut approach (Applegate et al. 2001). Rather than competing with these TSP specific methods, in this section the TSP is used as an application domain to show how ordered representations can be handled by modern optimization methods. Also, it is used as example to demonstrate how a Lamarckian evolution works.

An ordered representation is a natural choice for TSP since it assumes that solutions can be generated as permutations of the symbols from an alphabet. Without losing generality, the alphabet can be defined by the integers in the set $\{1, 2, \ldots, n\}$ (Rocha et al. 2001). Under this representation, the search space is $n!$ for a particular TSP instance of size n. The adaption of modern optimization methods to this representation type requires assuring that generated solutions (e.g., created in the *initialization* and *change* functions of Algorithm 1) are feasible, avoiding missing integers and repeated values.

For single-state methods, several mutation operators can be adopted to change solutions, such as (Michalewicz and Fogel 2004): exchange, insertion, and displacement. The first operator swaps two randomly selected cities, the second operator inserts a city into a random position and the third operator inserts a random subtour into another position. Figure 7.1 shows examples of these order mutations.

For evolutionary approaches, there are several crossover methods that preserve order, such as partially matched crossover (PMX), order crossover (OX), and cycle crossover (CX) (Rocha et al. 2001). PMX first selects two cutting points and the corresponding matching section is exchanged between the two parents, through position-to-position exchange operations. The OX also exchanges two sections between the parents but keeping an emphasis on the relative order of the genes from both parents. Both operators are shown in Fig. 7.2, while CX is described in Exercise 7.1.

In this section, two modern optimization methods are adapted to TSP: simulated annealing and evolutionary algorithm (under two variants). Simulated annealing uses a mutation operator to change solutions, while the evolutionary variants use specialized crossover and mutation operators. Given that the `genalg` package is not flexible in terms of accepting new genetic operators, the `rbga.bin` code was adapted to accept user defined operators under the new `oea` function. The two evolutionary variants include the standard algorithm and a Lamarckian evolution. The latter employs a greedy approach, where a local learning procedure is used

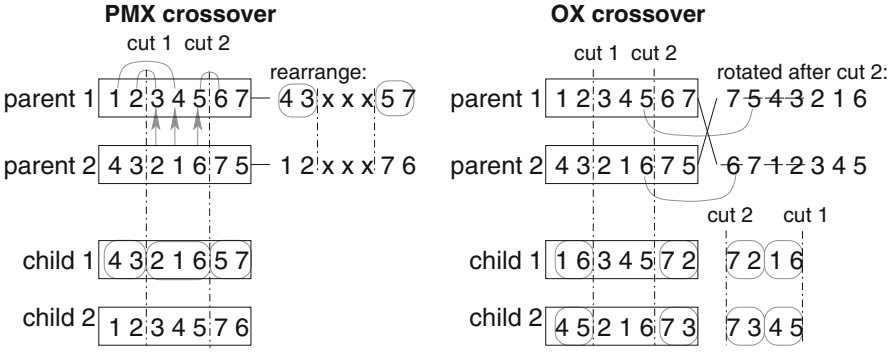

Fig. 7.2 Example of PMX and OX crossover operators

first to improve a solution and then the improved solution replaces the population original solution. The R code that implements the ordered representation operators and Lamarckian evolution option is presented in file `oea.R`:

```
### oea.R file ###

### mutation operators:
exchange=function(s,N=length(s))
{ p=sample(1:N,2) # select two positions
  temp=s[p[1]] # swap values
  s[p[1]]=s[p[2]]
  s[p[2]]=temp
  return(s)
}

insertion=function(s,N=length(s),p=NA,i=NA)
{ if(is.na(p)) p=sample(1:N,1) # select a position
  I=setdiff(1:N,p) # ALL except p
  if(is.na(i)) i=sample(I,1) # select random place
  if(i>p) i=i+1 # need to produce a change
  I1=which(I<i) # first part
  I2=which(I>=i) # last part
  s=s[c(I[I1],p,I[I2])] # new solution
  return(s)
}

displacement=function(s,N=length(s))
{ p=c(1,N)
  # select random tour different than s
  while(p[1]==1&&p[2]==N) p=sort(sample(1:N,2))
  I=setdiff(1:N,p[1]:p[2]) # ALL except p
  i=sample(I,1) # select random place
  I1=which(I<i) # first part
  I2=which(I>=i) # last part
  s=s[c(I[I1],p[1]:p[2],I[I2])]
  return(s)
```

```
}

### crossover operators:
# partially matched crossover (PMX) operator:
# m is a matrix with 2 parent x ordered solutions
pmx=function(m)
{
 N=ncol(m)
 p=sample(1:N,2) # two cutting points
 c=m # children
 for(i in p[1]:p[2])
   { # rearrange:
     c[1,which(c[1,]==m[2,i])]=c[1,i]
     # crossed section:
     c[1,i]=m[2,i]
     # rearrange:
     c[2,which(c[2,]==m[1,i])]=c[2,i]
     # crossed section:
     c[2,i]=m[1,i]
   }
 return(c)
}

# order crossover (OX) operator:
# m is a matrix with 2 parent x ordered solutions
ox=function(m)
{
 N=ncol(m)
 p=sort(sample(1:N,2)) # two cutting points
 c=matrix(rep(NA,N*2),ncol=N)
 # keep selected section:
 c[,p[1]:p[2]]=m[,p[1]:p[2]]
 # rotate after cut 2 (p[2]):
 I=((p[2]+1):(p[2]+N))
 I=ifelse(I<=N,I,I-N)
 a=m[,I]
 # fill remaining genes:
 a1=setdiff(a[2,],c[1,p[1]:p[2]])
 a2=setdiff(a[1,],c[2,p[1]:p[2]])
 I2=setdiff(I,p[1]:p[2])
 c[,I2]=rbind(a1,a2)
 return(c)
}

### order (representation) evolutionary algorithm:
# adapted version of rbga.bin that works with ordered vectors,
# accepts used defined mutation and crossover operators and
# accepts a Lamarckian evolution if evalFunc returns a list
# note: assumes solution with values from the range 1,2,...,size
oea=function(size=10,suggestions=NULL,popSize=200,iters=100,
    mutationChance=NA,
              elitism=NA,evalFunc=NULL,
              crossfunc=NULL,mutfunc=mutfunc,REPORT=0)
```

```
{
 if(is.na(mutationChance)) { mutationChance=0.5 }
 if(is.na(elitism)) { elitism=floor(popSize/5)}

 # population initialization:
 population=matrix(nrow=popSize,ncol=size)
 if(!is.null(suggestions))
   {
    suggestionCount=dim(suggestions)[1]
    for(i in 1:suggestionCount)
       population[i, ] = suggestions[i, ]
    I=(suggestionCount+1):popSize ### new code
   }
 else I=1:popSize ### new code
 for(child in I) ### new code
    population[child,]=sample(1:size,size) ### new code

 # evaluate population:
 evalVals = rep(NA, popSize)
 # main GA cycle:
 for(iter in 1:iters)
   {
    # evaluate population
    for(object in 1:popSize)
      {### new code
       EF = evalFunc(population[object,])
       if(is.list(EF)) # Lamarckian change of solution
         { population[object,]=EF$solution
           evalVals[object] = EF$eval
         }
       else evalVals[object]=EF
       ### end of new code
      }
    sortedEvaluations=sort(evalVals,index=TRUE)
    if(REPORT>0 && (iter%%REPORT==0||iter==1))
       cat(iter,"best:",sortedEvaluations$x[1],"mean:",mean(
           sortedEvaluations$x),"\n")
    sortedPopulation=matrix(population[sortedEvaluations$ix,],
        ncol=size)

    # check elitism:
    newPopulation=matrix(nrow=popSize,ncol=size)
    if(elitism>0) # applying elitism:
       newPopulation[1:elitism,]=sortedPopulation[1:elitism,]

    ### very new code inserted here : ###
    # roulette wheel selection of remaining individuals
    others=popSize-elitism
    prob=(max(sortedEvaluations$x)-sortedEvaluations$x+1)
    prob=prob/sum(prob) # such that sum(prob)==1

    # crossover with half of the population (if !is.null)
    if(!is.null(crossfunc)) # need to crossover
```

```
      half=round(others/2)
    else half=0 # no crossover
    if(!is.null(crossfunc))
      {
       for(child in seq(1,half,by=2))
         {
          selIDs=sample(1:popSize,2,prob=prob)
          parents=sortedPopulation[selIDs, ]
          if(child==half)
             newPopulation[elitism+child,]=crossfunc(parents)
                 [1,] # 1st child
          else
            newPopulation[elitism+child:(child+1),]=crossfunc(
                parents) # two children
         }
      }
    # mutation with remaining individuals
    for(child in (half+1):others)
      {
       selIDs=sample(1:popSize,1,prob=prob)
       newPopulation[elitism+child,]=mutfunc(sortedPopulation[
           selIDs,])
      }
    ### end of very new code           ###
    population=newPopulation # store new population

   } # end of GA main cycle
 result=list(type="ordered chromosome",size=size,
 popSize=popSize, iters=iters,population=population,
 elitism=elitism, mutationChance=mutationChance,
 evaluations=evalVals)return(result)
}
```

File `oea.R` implements all mentioned order operations. Also, it defines the new order evolutionary algorithm in `oea()`. This function uses the $\{1, 2, \ldots, n\}$ alphabet, where `size`=n and the first population is randomly generated using the `sample(1:size,size)` command (except for what is included in object `suggestions`). The `oea` code adopts a slight different approach for changing individuals when compared with the `genalg` package, which uses crossover to create new individuals and then mutates the created individuals (Algorithm 5). In the order representation case, a mutation is applied to the whole solution and thus it does not make sense to define a mutation probability for a particular gene (as in `genalg`). Following approach similar to what is described in Rocha et al. (2001), in `oea()` a pure roulette wheel selection is used to select the fittest individuals that are used to generate new solutions. Then, half of the fittest individuals are crossed and the remaining half are mutated (unless there is no crossover function, which in this case all fittest individuals are mutated). The new arguments (when compared with `rbga.bin`) are:

- `crossfunc`—crossover function (optional); if defined, half of the selected individuals (except `elitism`) are generated by this operator;
- `mutfunc`—mutation function that changes the remaining (non crossed) selected population individuals; and
- `REPORT`—shows the best and mean population fitness values every `REPORT` generations.

The `oea` function introduces another useful feature when compared with `genalg`. The evaluation function (`evalFunc`) can return the fitness value or a list with two components: `$eval`—the fitness value; and `$solution`—the original or an improved solution. When a list is returned, then it is assumed that the population individual will be changed after its evaluation, under a Lamarckian evolution scheme (explained in Sect. 1.6).

To demonstrate the simulated annealing and order evolutionary algorithm, the Qatar TSP was selected and that includes 194 cities. Other national TSPs are available at http://www.math.uwaterloo.ca/tsp/world/countries.html. The R code is presented in file `tsp.R`:

```
### tsp.R file ###

library(TSP) # load TSP package
library(RCurl) # load RCurl package
source("oea.R") # load ordered evolutionary algorithm

# get Qatar - 194 cities TSP instance:
txt=getURL("http://www.math.uwaterloo.ca/tsp/world/qa194.tsp")
# simple parse of txt object, removing header and last line:
txt=strsplit(txt,"NODE_COORD_SECTION") # split text into 2 parts
txt=txt[[1]][2] # get second text part
txt=strsplit(txt,"EOF") # split text into 2 parts
txt=txt[[1]][1] # get first text part
# save data into a simple .csv file, sep=" ":
cat(txt,file="qa194.csv")
# read the TSP format into Data
# (first row is empty, thus header=TRUE)
# get city Cartesian coordinates

Data=read.table("qa194.csv",sep=" ")
Data=Data[,3:2] # longitude and latitude
names(Data)=c("x","y") # x and y labels
N=nrow(Data) # number of cities

# distance between two cities (EUC_2D-norm)
# Eulidean distance rounded to whole number
D=dist(Data,upper=TRUE)
D[1:length(D)]=round(D[1:length(D)])
# create TSP object from D:
TD=TSP(D)

set.seed(12345) # for replicability
cat("2-opt run:\n")
```

```
PTM=proc.time() # start clock
R1=solve_TSP(TD,method="2-opt")
sec=(proc.time()-PTM)[3] # get seconds elapsed
print(R1) # show optimum
cat("time elapsed:",sec,"\n")

MAXIT=100000
Methods=c("SANN","EA","LEA") # comparison of 3 methods
RES=matrix(nrow=MAXIT,ncol=length(Methods))
MD=as.matrix(D)

# overall distance of a tour (evaluation function):
tour=function(s)
{ # compute tour length:
  EV<<-EV+1 # increase evaluations
  s=c(s,s[1]) # start city is also end city
  res=0
  for(i in 2:length(s)) res=res+MD[s[i],s[i-1]]
  # store memory with best values:
  if(res<BEST) BEST<<-res
  if(EV<=MAXIT) F[EV]<<-BEST
  # only for hybrid method:
  # return tour
  return(res)
}

# move city index according to dir
mindex=function(i,dir,s=NULL,N=length(s))
{ res=i+dir #positive or negative jump
  if(res<1) res=N+res else if(res>N) res=res-N
  return(res)
}

# local improvement and evaluation:
#   first tries to improve a solution with a
#   local search that uses domain knowledge (MD)
#   returns best solution and evaluation value
local_imp_tour=function(s,p=NA)
{ # local search
 N=length(s); ALL=1:N
 if(is.na(p)) p=sample(ALL,1) # select random position
 I=setdiff(ALL,p)

 # current distance: p to neighbors
 pprev=mindex(p,-1,N=N); pnext=mindex(p,1,N=N)
 dpcur=MD[s[pprev],s[p]]+MD[s[p],s[pnext]]
 # new distance if p is remove to another position:
 dpnew=MD[s[pprev],s[pnext]]

 # search for best insertion position for p:
 ibest=0;best=-Inf
 for(i in I) # extra cycle that increases computation
 {
```

```
  inext=mindex(i,1,N=N);iprev=mindex(i,-1,N=N)
  if(inext==p) inext=pnext
  if(iprev==p) iprev=pprev
  # dinew: new distance p to neighbors if p inserted:
  # current i distance without p:
  if(i<p) {dinew=MD[s[iprev],s[p]]+MD[s[p],s[i]]
           dicur=MD[s[iprev],s[i]]
          }
  else
    { dinew=MD[s[i],s[p]]+MD[s[p],s[inext]]
      dicur=MD[s[i],s[inext]]
    }
  # difference between current tour and new one:
  dif=(dicur+dpcur)-(dinew+dpnew)

  if(dif>0 && dif>best) # improved solution
    {
      best=dif
      ibest=i
    }
 }

 if(ibest>0) # insert p in i
   s=insertion(s,p=p,i=ibest)
 return(list(eval=tour(s),solution=s))
}

# SANN:
cat("SANN run:\n")
set.seed(12345) # for replicability
s=sample(1:N,N) # initial solution
EV=0; BEST=Inf; F=rep(NA,MAXIT) # reset these vars.
C=list(maxit=MAXIT,temp=2000,trace=TRUE,REPORT=MAXIT)
PTM=proc.time() # start clock
SANN=optim(s,fn=tour,gr=insertion,method="SANN",control=C)
sec=(proc.time()-PTM)[3] # get seconds elapsed
cat("time elapsed:",sec,"\n")
RES[,1]=F

# EA:
cat("EA run:\n")
set.seed(12345) # for replicability
EV=0; BEST=Inf; F=rep(NA,MAXIT) # reset these vars.
pSize=30;iters=ceiling((MAXIT-pSize)/(pSize-1))
PTM=proc.time() # start clock
OEA=oea(size=N,popSize=pSize,iters=iters,evalFunc=tour,crossfunc
    =ox,mutfunc=insertion,REPORT=iters,elitism=1)
sec=(proc.time()-PTM)[3] # get seconds elapsed
cat("time elapsed:",sec,"\n")
RES[,2]=F

# Lamarckian EA (LEA):
cat("LEA run:\n")
```

```
set.seed(12345) # for replicability
EV=0; BEST=Inf; F=rep(NA,MAXIT) # reset these vars.
pSize=30;iters=ceiling((MAXIT-pSize)/(pSize-1))
PTM=proc.time() # start clock
LEA=oea(size=N,popSize=pSize,iters=iters,evalFunc=local_imp
  _tour,crossfunc=ox,mutfunc=insertion,REPORT=iters,elitism=1)
sec=(proc.time()-PTM)[3] # get seconds elapsed
cat("time elapsed:",sec,"\n")
RES[,3]=F

# create PDF with comparison:
pdf("qa194-opt.pdf",paper="special")
par(mar=c(4.0,4.0,0.1,0.1))
X=seq(1,MAXIT,length.out=200)
ylim=c(min(RES)-50,max(RES))
plot(X,RES[X,1],ylim=ylim,type="l",lty=3,lwd=2,xlab="evaluations
    ",ylab="tour distance")
lines(X,RES[X,2],type="l",lty=2,lwd=2)
lines(X,RES[X,3],type="l",lty=1,lwd=2)
legend("topright",Methods,lwd=2,lty=3:1)
dev.off()

# create 3 PDF files with best tours:
pdf("qa194-2-opt.pdf",paper="special")
par(mar=c(0.0,0.0,0.0,0.0))
plot(Data[c(R1[1:N],R1[1]),],type="l",xaxt="n",yaxt="n")
dev.off()
pdf("qa194-ea.pdf",paper="special")
par(mar=c(0.0,0.0,0.0,0.0))
b=OEA$population[which.min(OEA$evaluations),]
plot(Data[c(b,b[1]),],type="l",xaxt="n",yaxt="n")
dev.off()
pdf("qa194-lea.pdf",paper="special")
par(mar=c(0.0,0.0,0.0,0.0))
b=LEA$population[which.min(LEA$evaluations),]
plot(Data[c(b,b[1]),],type="l",xaxt="n",yaxt="n")
dev.off()
```

The code starts by reading the Qatar TSP instance from the Web by using the `getURL` function of the `RCurl` package. The data is originally in the TSPLIB Format (extension `.tsp`) and thus some parsing (e.g., remove the header part until `NODE_COORD_SECTION`) is necessary to convert it into a CSV format. The national TSPs assume a traveling cost that is defined by the Euclidean distance rounded to the nearest whole number (TSPLIB EUC_2D-norm). This is easily computed by using the R `dist` function, which returns a distance matrix between all rows of a data matrix.

The code tests four methods to solve the Qatar instance: 2-opt, simulated annealing, an order evolutionary algorithm, and an evolutionary Lamarckian variant. The first method is executed using the the `TSP` package, which is specifically addressed to handle the TSP and includes two useful functions: `TSP`—generates a TSP object from a distance matrix; and `solve_TSP`—solves a TSP instance under

several `method` options (e.g., `"2-opt"` and `"concorde"`). To simplify the code and analysis, the remaining optimization methods are only compared under a single run, although a proper comparison would require the use of several runs, as shown in Sect. 4.5. The simulated annealing and evolutionary algorithms are executed under the same conditions. Similarly to the code presented in Sect. 4.5, the global `EV`, `BEST`, and `F` are used to trace the evolution of optimization according to the number of function evaluations. The method parameters were fixed into a temperature of $T = 2{,}000$ for the simulated annealing and population size of $L_P = 30$ for the evolutionary algorithm. The Lamarckian approach works as follows. When the evaluation function is called, a local search that uses domain knowledge is applied. This local search works by randomly selecting a city and then moving such city to the best possible position when analyzing only the city to direct neighbors (previous and next city) distances.

The `tour()` (evaluation function) uses an already defined distance matrix (`MD` object) to save computational effort (i.e., the Euclidean distance is only calculated once). The `local_imp_tour` evaluates and returns the solution improved by the domain knowledge local search method and it is used by the Lamarckian evolutionary algorithm. The same initialization seed is used and both simulated annealing and evolutionary methods are traced up to `MAXIT=100000` function evaluations. The insertion operator is used to change a solution, and the OX operator is adopted to cross half of the individuals in the evolutionary algorithm. For each method, the code shows the length of the tour and time elapsed (in seconds). The code also generates three PDF files with a comparison of simulated annealing and evolutionary approaches and two optimized tours (for 2-opt and evolutionary algorithm methods). The execution result of file `tsp.R` is:

```
2-opt run:
object of class "TOUR"
result of method "2-opt" for 194 cities
tour length: 10279
time elapsed: 0.151
SANN run:
sann objective function values
initial       value 91112.000000
final         value 40687.000000
sann stopped after 99999 iterations
time elapsed: 74.469
EA run:
1 best: 87620 mean: 93128
3448 best: 22702 mean: 25067.03
time elapsed: 88.026
LEA run:
1 best: 87002 mean: 92674.93
3448 best: 12130 mean: 14462.53
time elapsed: 546.005
```

The execution times of simulated annealing and pure evolutionary algorithm are quite similar and are much longer when compared with 2-opt approach. The Lamarckian evolution method requires around six times more computation when

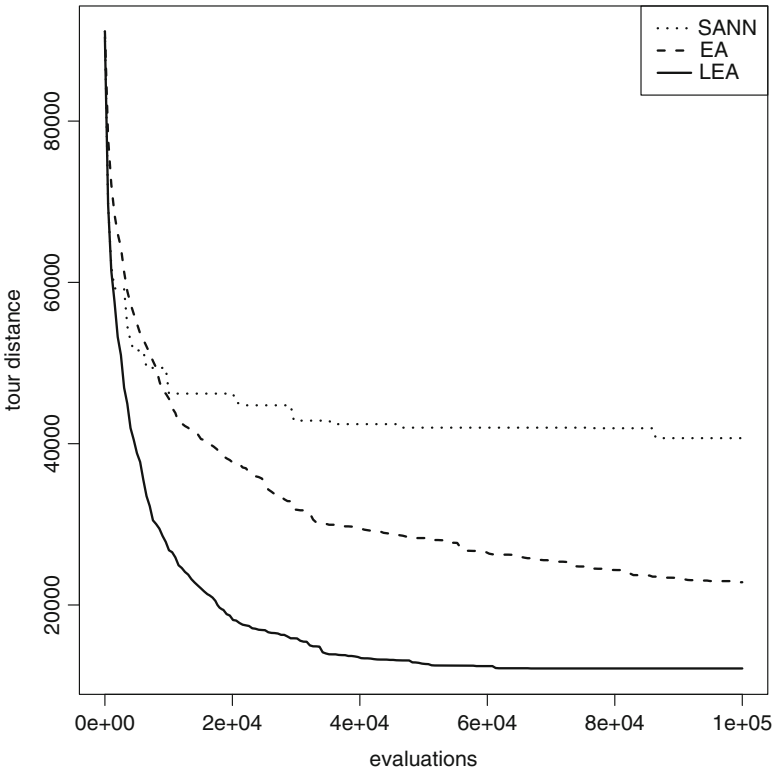

Fig. 7.3 Comparison of simulated annealing (SANN) and evolutionary algorithm (EA) approaches for the Qatar TSP

compared with the evolutionary algorithm. The extra computation is explained by the use of the local search, given that all evaluations require the execution of an extra linear cycle.

The comparison between the simulated annealing and evolutionary methods is shown in Fig. 7.3. Under the experimental setup conditions, the simulated annealing initially performs similarly to the two evolutionary algorithm methods. However, after around 10,000 evaluations the simulated annealing improvement gets slower when compared with the standard evolutionary algorithm and after around 50,000 evaluations, the convergence is rather flat, reaching a tour value of 40,687. The pure evolutionary algorithm performs better than the simulated annealing, getting an average distance decrease of 10,184 after 50,000 evaluations, when compared with the simulated annealing, and obtaining a final tour of 22,702. The Lamarckian method performs much better than the pure evolutionary algorithm, presenting an average tour improvement of 17,117 after 50,000 evaluations and reaching a final value of 12,130. The best solution is produced by the 2-opt method (which is also the fastest method), with a tour length of 10,279 and that is 1,851 points better than

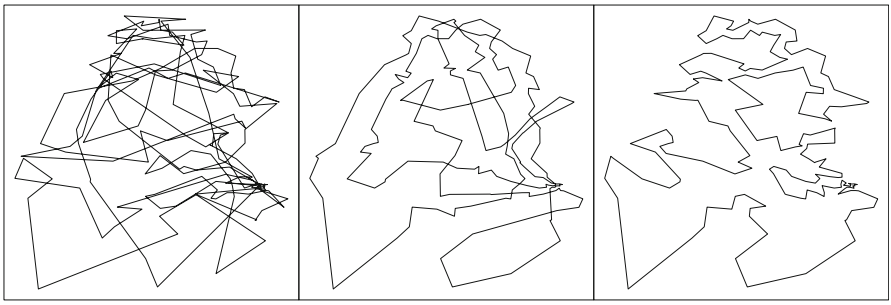

Fig. 7.4 Optimized tour obtained using evolutionary algorithm (*left*), Lamarckian evolution (*middle*), and 2-opt (*right*) approaches for the Qatar TSP

the Lamarckian evolved tour. A visual comparison of the evolutionary algorithm, Lamarckian evolution, and 2-opt optimized tours is shown in Fig. 7.4, revealing a clear improvement in the quality of the solutions when moving from left to right.

In this demonstration, 2-opt provided the best results, followed by the Lamarckian approach. However, 2-opt was specifically proposed for the symmetrical and standard TSP and thus performs a massive and clever use of the distance matrix (domain knowledge) to solve the task. The Lamarckian method also uses the distance matrix, although with a much simpler approach when compared with 2-opt. As explained in Chap. 1.1, the simulated annealing and evolutionary algorithms are general purpose methods that only have an indirect access to the domain knowledge through the received evaluation function values. This means that the same modern optimization algorithms could be easily applied (by adjusting the evaluation function) to other TSP variants (e.g., with constrains) or combinatorial problems (e.g., job shop scheduling) while 2-opt (or even the Lamarckian method) could not.

To demonstrate the previous point, a TSP variant is now addressed, where the goal is set in terms of searching for the minimum area of the tour (and not tour length). This new variant cannot be directly optimized using the 2-opt method. However, the adaptation to a modern optimization method is straightforward and just requires changing the evaluation function. To show this, the same Qatar instance and evolutionary ordered representation optimization is adopted. The new R code is presented in file `tsp2.R`:

```
### tsp2.R file ###
# this file assumes that tsp.R has already been executed

library(rgeos) # get gArea function

poly=function(data)
{ poly="";sep=", "
  for(i in 1:nrow(data))
  { if(i==nrow(data)) sep=""
    poly=paste(poly,paste(data[i,],collapse=" "),sep,sep="")
  }
```

```
  poly=paste("POLYGON((",poly,"))",collapse="")
  poly=readWKT(poly) # WKT format to polygon
}

# new evaluation function: area of polygon
area=function(s) return( gArea(poly(Data[c(s,s[1]),])) )

cat("area of 2-opt TSP tour:",area(R1),"\n")

# plot area of 2-opt:
pdf("qa-2opt-area.pdf",paper="special")
par(mar=c(0.0,0.0,0.0,0.0))
PR1=poly(Data[c(R1,R1[1]),])
plot(PR1,col="gray")
dev.off()

# EA:
cat("EA run for TSP area:\n")
set.seed(12345) # for replicability
pSize=30;iters=20
PTM=proc.time() # start clock
OEA=oea(size=N,popSize=pSize,iters=iters,evalFunc=area,crossfunc
    =ox,mutfunc=insertion,REPORT=iters,elitism=1)
sec=(proc.time()-PTM)[3] # get seconds elapsed
bi=which.min(OEA$evaluations)
b=OEA$population[which.min(OEA$evaluations),]
cat("best fitness:",OEA$evaluations[1],"time elapsed:",sec,"\n")

# plot area of EA best solution:
pdf("qa-ea-area.pdf",paper="special")
par(mar=c(0.0,0.0,0.0,0.0))
PEA=poly(Data[c(b,b[1]),])
plot(PEA,col="gray")
lines(Data[c(b,b[1]),],lwd=2)
dev.off()
```

The evaluation function uses the `gArea()` function of the `rgeos` package to compute the area of a polygon. Before calculating the area, the function first converts the selected solution into a polygon object by calling the `poly` auxiliary function. The latter function first encodes the tour under the Well Known Text (WKT) format (see http://en.wikipedia.org/wiki/Well-known_text) and then uses `readWKT()` (from the `rgeos` package) function to create the polygon (`sp geometry` object used by the `rgeos` package). For comparison purposes, the area is first computed for the tour optimized by the 2-opt method. Then, the evolutionary optimization is executed and stopped after 20 iterations. The code also produces two PDF files with area plots related to the best solutions optimized by the evolutionary algorithm and 2-opt methods.

The result of executing file `tsp2.R` is:

```
> source("tsp2.R")
area of 2-opt TSP tour: 465571.6
EA run for TSP area:
```

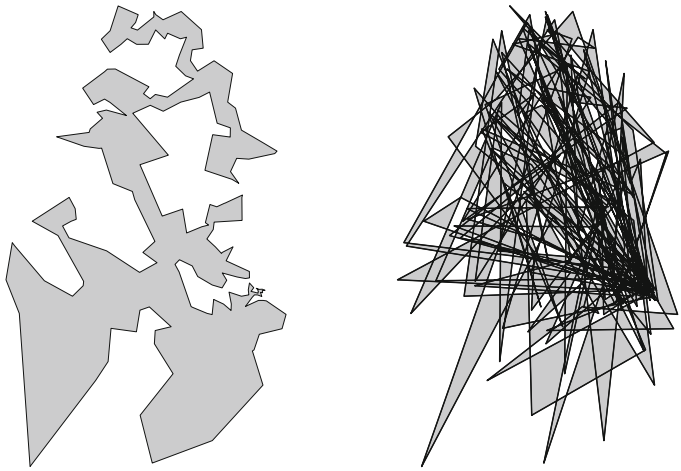

Fig. 7.5 Area of Qatar tour given by 2-opt (*left*) and optimized by the evolutionary approach (*right*)

```
1 best: 103488.9 mean: 562663.5
20 best: 616.7817 mean: 201368.5
best fitness: 616.7817 time elapsed: 23.383
```

Now the evolutionary approach achieves a value that is much lower when compared with 2-opt (difference of 464954.8). The area of each optimized solution is shown in Fig. 7.5. As shown by the plots, the evolutionary algorithm best solution contains a huge number of crossing paths, which clearly reduces the area of the optimized tour. In contrast, the 2-opt solution does not contain crossing paths. In effect, 2-opt intentionally avoids crossing paths and such strategy is very good for reducing the path length but not the tour area.

7.3 Time Series Forecasting

A univariate time series is a collection of timely ordered observations related with an event $(y_1, y_2, \ldots, y_t)$ and the goal of TSF is to model a complex system as a black-box, predicting its behavior based on historical data (Makridakis et al. 1998). Past values (called in-samples) are first used to fit the model and then forecasts are estimated $(\hat{y}_t)$ for future values (called out-of-samples). The TSF task is to determine a function f such that $\hat{y}_t = f(y_{t-1}, y_{t-2}, \ldots, y_{t-k})$, where k denotes the maximum time lag used by the model. Under one-step ahead forecasting, the errors (or residuals) are given by $e_i = y_i - \hat{y}_i$, where $i \in \{T+1, T+2, \ldots, T+h\}$, T is the current time, h is the horizon (or number of predictions), and the errors are to be minimized according to an accuracy metric, such as the mean absolute

error ($MAE = \frac{\sum_i |e_i|}{h}$) (Stepnicka et al. 2013). TSF is highly relevant in distinct domains, such as Agriculture, Finance, Sales, and Production. TSF forecasts can be used to support individual and organizational decision making (e.g., for setting early production plans).

Due to its importance, several statistical TSF methods were proposed, such as the autoregressive integrated moving-average (ARIMA) methodology, which was proposed in 1976 and is widely used in practice (Makridakis et al. 1998). The methodology assumes three main steps: model identification, parameter estimation, and model validation. The ARIMA base model is set in terms of a linear combination of past values (AR component of order p) and errors (MA component of order q). The definition assumed by the R `arima` function is:

$$\hat{x_t} = a_1 x_{t_1} + \ldots + a_p x_{t_p} + e_t + b_1 e_{t_1} + \ldots + b_q e_{t_q} \tag{7.1}$$

where $x_t = y_t - m$ and $m, a_1, \ldots, a_p$ and $b_1, \ldots, b_q$ are coefficients that are estimated using an optimization method. The `arima()` default is to use a conditional sum of squares search to find starting values and then apply a maximum likelihood optimization. To identify and estimate the best ARIMA model, the `auto.arima` function is adopted from the `forecast` package.

In this section, genetic programming and `rgb` package is adopted to fit a time series using a simple function approximation that uses arithmetic operators (`+`, `-`, `*` and `/`). As explained in Sect. 5.8, the advantage of genetic programming is that it can find explicit solutions that are easy to interpret by humans. The selected series is the sunspot numbers (also known as Wolf number), which measures the yearly number of dark spots present at the surface of the sun. Forecasting this series is relevant due to several reasons (Kaboudan 2003): the sunspots data generation process is unknown; sunspots are often used to estimate solar activity levels; accurate prediction of sunspot numbers is a key issue for weather forecasting and for making decisions about satellite orbits and space missions. The data range from 1700 to 2012. In this demonstration, data from the years 1700–1980 are used as in-samples and the test period is 1981–2012 (out-of-samples). Forecasting accuracy is measured using the MAE metric and one-step ahead forecasts. The respective R code is presented in file `tsf.R`:

```
### tsf.R file ###
library(RCurl) # load RCurl package

# get sunspot series
txt=getURL("http://sidc.oma.be/silso/DATA/yearssn.dat")
# consider 1700-2012 years (remove 2013 * row that is provisory
    in 2014)
series=strsplit(txt,"\n")[[1]][1:(2012-1700+1)]
cat(series,sep="\n",file="sunspots.dat") # save to file
series=read.table("sunspots.dat")[,2] # read from file

L=length(series) # series length
forecasts=32 # number of 1-ahead forecasts
```

```
outsamples=series[(L-forecasts+1):L] # out-of-samples
sunspots=series[1:(L-forecasts)] # in-samples

# mean absolute error of residuals
maeres=function(residuals) mean(abs(residuals))

# fit best ARIMA model:
INIT=10 # initialization period (no error computed before)
library(forecast) # load forecast package
arima=auto.arima(sunspots) # detected order is AR=2, MA=1
print(arima) # show ARIMA model
cat("arima fit MAE=",
    maeres(arima$residuals[INIT:length(sunspots)]),"\n")
# one-step ahead forecasts:
# (this code is needed because forecast function
#  only issues h-ahead forecasts)
LIN=length(sunspots) # length of in-samples
f1=rep(NA,forecasts)
for(h in 1:forecasts)
  { # execute arima with fixed coefficients but with more
      in-samples:
   arima1=arima(series[1:(LIN+h-1)],order=arima$arma[c(1,3,2)],
       fixed=arima$coef)
   f1[h]=forecast(arima1,h=1)$mean[1]
  }
e1=maeres(outsamples-f1)
text1=paste("arima (MAE=",round(e1,digits=1),")",sep="")

# fit genetic programming arithmetic model:
library(rgp) # load rgp
ST=inputVariableSet("x1","x2")#same order of AR arima component
cF1=constantFactorySet(function() rnorm(1)) # mean=0, sd=1
FS=functionSet("+","*","-","/") # arithmetic

# genetic programming time series function
#   receives function f
#   if(h>0) then returns 1-ahead forecasts
#   else returns MAE over fitting period (in-samples)
gpts=function(f,h=0)
{
  if(h>0) TS=series
  else TS=series[1:LIN]
  LTS=length(TS)
  F=rep(0,LTS) # forecasts
  E=rep(0,LTS) # residuals
  if(h>0) I=(LTS-h+1):LTS # h forecasts
  else I=INIT:LTS # fit to in-samples
  for(i in I)
    {
     F[i]=f(TS[i-1],TS[i-2])
     if(is.nan(F[i])) F[i]=0 # deal with NaN
     E[i]=TS[i]-F[i]
    }
```

```
  if(h>0) return (F[I]) # forecasts
  else return(maeres(E[I])) # MAE on fit
}

# mutation function
mut=function(func)
{ mutateSubtree(func,funcset=FS,inset=ST,conset=cF1,
                mutatesubtreeprob=0.3,maxsubtreedepth=4)}

set.seed(12345) # set for replicability
gp=geneticProgramming(functionSet=FS,inputVariables=ST,
                      constantSet=cF1,
                      populationSize=100,
                      fitnessFunction=gpts,
                      stopCondition=makeStepsStopCondition(1000),
                      mutationFunction=mut,
                      verbose=TRUE)
f2=gpts(gp$population[[which.min(gp$fitnessValues)]],
 h=forecasts)
e2=maeres(outsamples-f2)

text2=paste("gp (MAE=",round(e2,digits=1),")",sep="")
cat("best solution:\n")
print(gp$population[[which.min(gp$fitnessValues)]])
cat("gp fit MAE=",min(gp$fitnessValues),"\n")

# show quality of one-step ahead forecasts:
ymin=min(c(outsamples,f1,f2))
ymax=max(c(outsamples,f1,f2))
pdf("fsunspots.pdf")
par(mar=c(4.0,4.0,0.1,0.1))
plot(outsamples,ylim=c(ymin,ymax),type="b",pch=1,
     xlab="time (years after 1980)",ylab="values",cex=0.8)
lines(f1,lty=2,type="b",pch=3,cex=0.5)
lines(f2,lty=3,type="b",pch=5,cex=0.5)
legend("topright",c("sunspots",text1,text2),lty=1:3,
       pch=c(1,3,5))
dev.off()
```

The ARIMA model is automatically found using the `auto.arima` function, which receives as inputs the in-samples. For this example, the identified model is an $ARIMA(2,0,1)$, with $p = 2$ and $q = 1$. The `forecast` function (from package `forecast`) executes multi-step ahead predictions. Thus, one-step ahead forecasts are built by using an iterative call to the function, where in each iteration the ARIMA model is computed with one extra in-sample value. For comparison purposes, the genetic programming method uses the same p order and thus the input variables are `x1`$= y_{t-1}$ and `x2`$= y_{t-2}$. In order to save code, the `gpts` function is used under two execution goals: fitness function, computing the MAE over all in-samples except for the first `INIT` values (when `h=0`); and estimation of forecasts, returning `h` one-step ahead forecasts. Since the `/` operator can generate `NaN` values (e.g., `0/0`), any `NaN` value is transformed into 0. To simplify the demonstration, only

one run is used, with a fixed seed. The genetic programming is stopped after 1,000 generations and then a PDF file is created, comparing the forecasts with the sunspot values. The result of executing file `tsf.R` is:[1]

```
> source("tsf.R")
Series: sunspots
ARIMA(2,0,1) with non-zero mean

Coefficients:
         ar1      ar2      ma1  intercept
      1.4565  -0.7493  -0.1315    48.0511
s.e.  0.0552   0.0502   0.0782     2.8761

sigma^2 estimated as 263.6:  log likelihood=-1183.17
AIC=2376.33   AICc=2376.55   BIC=2394.52
arima fit MAE= 12.22482
STARTING genetic programming evolution run (Age/Fitness/
    Complexity Pareto GP search-heuristic) ...
evolution step 100, fitness evaluations: 1980, best fitness:
    17.475042, time elapsed: 7.63 seconds
evolution step 200, fitness evaluations: 3980, best fitness:
    17.474204, time elapsed: 13.18 seconds
evolution step 300, fitness evaluations: 5980, best fitness:
    14.099732, time elapsed: 19.82 seconds
evolution step 400, fitness evaluations: 7980, best fitness:
    12.690703, time elapsed: 27.49 seconds
evolution step 500, fitness evaluations: 9980, best fitness:
    11.802043, time elapsed: 36.59 seconds
evolution step 600, fitness evaluations: 11980, best fitness:
    11.791989, time elapsed: 48.06 seconds
evolution step 700, fitness evaluations: 13980, best fitness:
    11.784837, time elapsed: 58.44 seconds
evolution step 800, fitness evaluations: 15980, best fitness:
    11.768817, time elapsed: 1 minute, 8.75 seconds
evolution step 900, fitness evaluations: 17980, best fitness:
    11.768817, time elapsed: 1 minute, 18.74 seconds
evolution step 1000, fitness evaluations: 19980, best fitness:
    11.768817, time elapsed: 1 minute, 28.74 seconds
Genetic programming evolution run FINISHED after 1000 evolution
    steps, 19980 fitness evaluations and 1 minute, 28.74
    seconds.
best solution:
function (x1, x2)
x1/(x2 + x1) * (x1 + x1/(x2 + x1) * (x1 + x1 - x1/
  (1.3647488967524 +1.3647488967524)) - x1/(x2 + x1) * x1/
  (1.3647488967524 +(1.3647488967524 + 1.3647488967524/x2)))
gp fit MAE= 11.76882
```

The $ARIMA(2,0,1)$ model fits the in-samples with an MAE of 12.2 and the genetic programming method best fitness is slightly better ($MAE = 11.8$).

[1]These results were achieved with `rgp` version 0.3-4 and later `rgp` versions might produce different results.

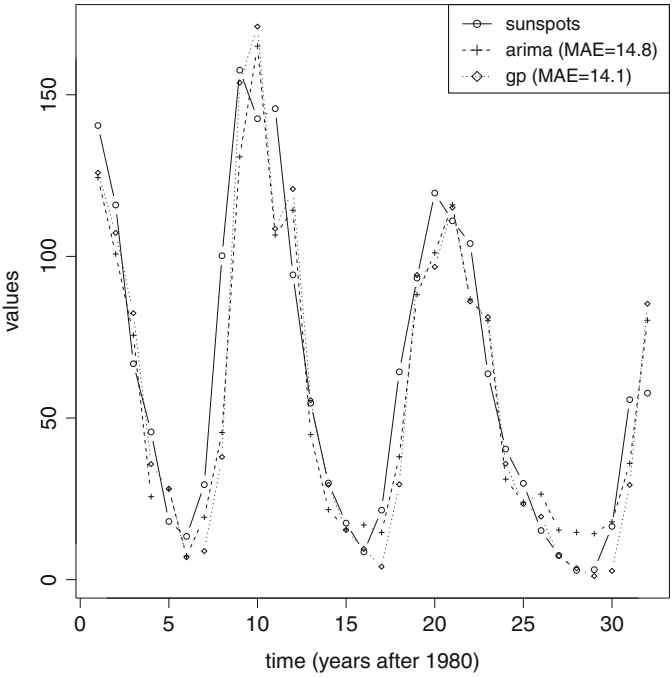

Fig. 7.6 Sunspot one-step ahead forecasts using ARIMA and genetic programming (gp) methods

The genetic programming representation does not include the MA terms (i.e., past errors) of ARIMA but the evolved solution is nonlinear (due to the $\star$ operator). The quality of the one-step ahead forecasts is shown in Fig. 7.6. Both ARIMA and genetic programming predictions are close to the true sunspot values. Overall, the genetic programming solution produces slightly better forecasts with an improvement of 0.7 when compared with the ARIMA method in terms of MAE measured over the out-of-samples. This is an interesting result, since ARIMA methodology was specifically designed for TSF while genetic programming is a much more generic optimization method.

7.4 Wine Quality Classification

Classification is an important data mining/machine learning task, where the goal is to build a data-driven model (i.e., model fit using a dataset) that is capable of predicting a class label (output target) given several input variables that characterize an item (Cortez 2012). For example, a classification model can estimate the type of credit client, "good" or "bad," given the status of her/his bank account, credit purpose, and amount.

Often, it is possible to assign probabilities ($p \in [0, 1]$) for a class label when using a classifier. Under such scheme, the choice of a label is dependent on a

decision threshold D, such that the class is true if $p > D$. The receiver operating characteristic (ROC) curve shows the performance of a two class classifier across the range of possible threshold (D) values, plotting one minus the specificity (false positive rate—FPR; x-axis) versus the sensitivity (also known as true positive rate—TPR; y-axis) (Fawcett 2006). The overall accuracy is given by the area under the curve ($AUC = \int_0^1 ROC\, dD$), measuring the degree of discrimination that can be obtained from a given model. The ROC analysis is a popular and richer measure for evaluating classification discrimination capability. The main advantage of the ROC curve is that performance is not dependent on the class label distributions and error costs (Fawcett 2006). Since there is a trade-off between specificity and sensitivity errors, the option for setting the best D value can be left for the domain expert. The ideal classified should present an AUC of 1.0, while an AUC of 0.5 denotes a random classifier. Often, AUC values are read as: 0.7—good; 0.8—very good; and 0.9—excellent.

Given the interest in classification, several machine learning methods have been proposed, each one with its own purposes and advantages. In this section, the support vector machine (SVM) model is adopted. This is a powerful learning tool that is based on a statistical learning theory and was developed in the 1990s under the work of Vapnik and its collaborators (e.g., Cortes and Vapnik 1995). The model is popular due to its learning flexibility (i.e., no *a priori* restriction is imposed) and tendency for achieving high quality classification results. In effect, SVM was recently considered one of the most influential data mining algorithms due to its classification capabilities (Wu et al. 2008).

The basic idea of an SVM is to transform the input space into a higher feature space by using a nonlinear mapping that depends on a kernel function. Then, the algorithm finds the best linear separating hyperplane, related to a set of support vector points, in the feature space. The gaussian kernel is popular option and presents less hyperparameters and numerical difficulties than other kernels (e.g., polynomial or sigmoid). When using this kernel, SVM classification performance is affected by two hyperparameters: γ, the parameter of the kernel, and $C > 0$, a penalty parameter of the error term. Thus, model selection is a key issue when using SVM. When performing model selection, classification performance is often measured over a validation set, containing data samples that do not belong to the training set. This procedure is used to avoid overfitting, since SVM can easily fit every single sample, possibly including noise or outliers, and such model would have limited generalization capabilities. The validation set can be created by using a holdout or k-fold cross-validation method (Kohavi 1995) to split the training data into training and validation sets.

Classification performance is also affected by the input variables used to fit the model (this includes SVM and other models). Feature selection, i.e., the selection of the right inputs, is useful to discard irrelevant variables (also known as features), leading to simpler models that are easier to interpret and often presenting higher predictive accuracies (Guyon and Elisseeff 2003).

There is no optimal universal method for tuning a classifier. Thus, often trial-and-error or heuristics are used, normally executed using only one type of selection, such as backward selection for feature selection (Guyon and Elisseeff 2003) and grid

search for model selection (Hsu et al. 2003). However, ideally both selection types should be performed simultaneously. This section follows such approach, under a multi-objective optimization search. As explained in Freitas (2004), the multi-objective strategy is justified by the trade-off that exists between having less features and increasing the classifier performance. The use of a modern optimization method, such as the NSGAII algorithm adopted in this section, is particularly appealing to non-specialized data mining/machine learning users, given that the search is fully automatic and more exhaustive, thus tending to provide better performances when compared with the manual design.

The University California Irvine (UCI) machine learning repository (Bache and Lichman 2013) contains more than 280 datasets that are used by the data mining community to test algorithms and tools. Most datasets are related with classification tasks. In particular, this section explores the **wine quality** that was proposed and analyzed in Cortez et al. (2009). The goal is to model wine quality based on physicochemical tests. The output target (quality) was computed as the median of at least three sensory evaluations performed by wine experts, using a scale that ranges from 1 (very poor) to 10 (excellent). The physicochemical tests include 11 continuous variables (inputs), such as chlorides and alcohol (vol.%). As explained in Cortez et al. (2009), building a data-driven model, capable of predicting wine quality from physicochemical values, is important for the wine domain because the relationships between the laboratory tests and sensory analysis are complex and are still not fully understood. Thus, an accurate data-driven model can support the wine expert decisions, aiding the speed and quality of the evaluation performance. Also, such model could also be used to improve the training of oenology students.

This section exemplifies how the best classification model can be optimized by performing a simultaneous feature and model selection. Given that two objectives are defined, i.e., improving classification performance and reducing the number of features used by the model, a multi-objective approach is adopted. The example code uses the `mco` and `rminer` packages. The former is used to get the `nsga2` function (NSGAII algorithm). The latter library facilitates the use of data mining algorithms in classification and regression tasks by presenting a short and coherent set of functions (Cortez 2010). The `rminer` package is only briefly described here, for further details consult `help(package=rminer)`. The classification example for the white wine quality dataset is coded in file `wine-quality.R`:

```
### wine-quality.R file ###

library(rminer) # load rminer package
library(kernlab) # load svm functions used by rminer
library(mco) # load mco package

# load wine quality dataset directly from UCI repository:
file="http://archive.ics.uci.edu/ml/machine-learning-databases/
    wine-quality/winequality-white.csv"
d=read.table(file=file,sep=";",header=TRUE)

# convert the output variable into 3 classes of wine:
```

```
# "poor_or_average" <- 3,4,5 or 6;
# "good_or_excellent" <- 7, 8 or 9
d$quality=cut(d$quality,c(1,6,10),
              c("poor_or_average","good_or_excellent"))
output=ncol(d) # output target index (last column)
maxinputs=output-1 # number of maximum inputs

# to speed up the demonstration, select a smaller sample of
   data:
n=nrow(d) # total number of samples
ns=round(n*0.25) # select a quarter of the samples
set.seed(12345) # for replicability
ALL=sample(1:n,ns) # contains 25% of the index samples
# show a summary of the wine quality dataset (25%):
print(summary(d[ALL,]))
cat("output class distribuition (25% samples):\n")
print(table(d[ALL,]$quality)) # show distribution of classes

# holdout split:
# select training data (for fitting the model), 70%; and
# test data (for estimating generalization capabilities), 30%.
H=holdout(d[ALL,]$quality,ratio=0.7)
cat("nr. training samples:",length(H$tr),"\n")
cat("nr. test samples:",length(H$ts),"\n")

# evaluation function:
# x is in the form c(Gamma,C,b1,b2,...,b11)
eval=function(x)
{ n=length(x)
  gamma=2^x[1]
  C=2^x[2]
  features=round(x[3:n])
  inputs=which(features==1)
  attributes=c(inputs,output)
  # divert console:
  # sink is used to avoid kernlab ksvm messages in a few cases
  sink(file=textConnection("rval","w",local = TRUE))
  M=mining(quality~.,d[H$tr,attributes],method=c("kfold",3),
      model="svm",search=gamma,mpar=c(C,NA))
  sink(NULL) # restores console
  # AUC for the internal 3-fold cross-validation:
  auc=as.numeric(mmetric(M,metric="AUC"))
  auc1=1-auc # transform auc maximization into minimization goal
  return(c(auc1,length(inputs)))
}

# NSGAII multi-objective optimization:
cat("NSGAII optimization:\n")
m=2 # two objectives: AUC and number of features
lower=c(-15,-5,rep(0,maxinputs))
upper=c(3,15,rep(1,maxinputs))
PTM=proc.time() # start clock
```

```
G=nsga2(fn=eval,idim=length(lower),odim=m,lower.bounds=lower,
    upper.bounds=upper,popsize=12,generations=10)
sec=(proc.time()-PTM)[3] # get seconds elapsed
cat("time elapsed:",sec,"\n")

# show the Pareto front:
I=which(G$pareto.optimal)
for(i in I)
  { x=G$par[i,]
    n=length(x)
    gamma=2^x[1]
    C=2^x[2]
    features=round(x[3:n])
    inputs=which(features==1)
    cat("gamma:",gamma,"C:",C,"features:",inputs,"; f=(",
        1-G$value[i,1],G$value[i,2],")\n",sep=" ")
  }

# create PDF showing the Pareto front:
pdf(file="nsga-wine.pdf",paper="special",height=5,width=5)
par(mar=c(4.0,4.0,0.1,0.1))
SI=sort.int(G$value[I,1],index.return=TRUE)
plot(1-G$value[SI$ix,1],G$value[SI$ix,2],xlab="AUC",ylab="nr.
    features",type="b",lwd=2)
dev.off()

# selection of the SVM model with 4 inputs:
x=G$par[I[7],]
gamma=2^x[1]
C=2^x[2]
features=round(x[3:n])
inputs=which(features==1)
attributes=c(inputs,output)
# fit a SVM with the optimized parameters:
cat("fit SVM with nr features:",length(inputs),"nr samples:",
    length(H$tr),"gamma:",gamma,"C:",C,"\n")
cat("inputs:",names(d)[inputs],"\n")
M=fit(quality~.,d[H$tr,attributes],model="svm",
      search=gamma,mpar=c(C,NA))
# get SVM predictions for unseen data:
P=predict(M,d[H$ts,attributes])
# create PDF showing the ROC curve for unseen data:
auc=mmetric(d[H$ts,]$quality,P,metric="AUC")
main=paste("ROC curve for test data",
           " (AUC=",round(auc,digits=2),")",sep="")
mgraph(d[H$ts,]$quality,P,graph="ROC",PDF="roc-wine",main=main,
    baseline=TRUE,Grid=10,leg="SVM")
```

In this example, the `read.table` function is used to read the CSV file directly from the UCI repository. Originally, there are seven numeric values for the wine quality variable (range from 3 to 9). The example approaches a simple binary task, thus the `cut` R function is used to transform the numeric values into two classes. Also, the original dataset includes 4,898 samples, which is a large number for the SVM fit. To reduce the computational effort, in this demonstration 25 % of the samples are first selected. Given that classification performance should be

accessed over unseen data, not used for fitting, the popular holdout split validation procedure is adopted, where 70 % of the selected samples are used for the search of the best model, while the other 30 % of the samples are used as test set, for estimating the model true generalization capabilities. The `holdout` function creates a list with the training (`$tr`) and test (`$ts`) indexes related with a output target. The NSGAII chromosome is made in terms of real values and includes γ, C and 11 values related with feature selection. As advised in Hsu et al. (2003), the γ and C parameters are searched using exponentially growing sequences, where $\gamma \in [2^{-15}, 2^3]$ and $C \in [2^{-5}, 2^{15}]$. The feature selection values are interpreted as boolean numbers, where the respective input variable is included in the model if > 0.5.

The evaluation function is based on the powerful `mining` function, which trains and tests a classifier under several runs and a given validation method. In this case, the used function arguments were:

- `x=quality~.`—an R `formula` that means that the target is the quality attribute and that all other `data` attributes are used as inputs;
- `data=d[H$tr,attributes]`—dataset used (`data.frame`), in this case corresponds to the training set samples and variables defined by the solution `x` (features and output);
- `method=c("kfold",3)`—the estimation method used by the function (in this case a threefold cross-validation);
- `model="svm"`—model type name;
- `search=gamma`—hyperparameter to tune (in this case it is fixed to the value of `gamma`); and
- `mpar=c(C,NA)`—vector with extra model parameters (in this case it sets C).

For some *gamma* and C configurations, the SVM function produces some messages and thus the useful `sink` R function was adopted to discard these messages. At the end, the evaluation function returns the AUC value (computed using the `mmetric rminer` function) and number of features.

The NSGAII algorithm is executed using a small population size (12) and stopped after 10 generations, in order to reduce the computational effort of the demonstration. After the search, the Pareto front is shown in the console and also plotted into a PDF file. In the example, one particular model (with four inputs) is selected and fit using all training data (`fit` function from `rminer`). Then, predictions are executed for the test data (using the `predict rminer` function) and the respective ROC curve is saved into another PDF file (using the `mgraph rminer` function). The execution result of file `wine-quality.R` is:

```
> source("wine-quality.R")
 fixed.acidity    volatile.acidity  citric.acid
     residual.sugar
 Min.   : 3.800   Min.   :0.0800   Min.   :0.0000   Min.   :
     0.600
 1st Qu.: 6.300   1st Qu.:0.2100   1st Qu.:0.2600   1st Qu.:
     1.800
 Median : 6.800   Median :0.2600   Median :0.3100   Median :
       5.700
```

```
 Mean   : 6.813   Mean   :0.2788   Mean   :0.3253   Mean   :
     6.322
 3rd Qu.: 7.300   3rd Qu.:0.3200   3rd Qu.:0.3700   3rd Qu.:
       9.500
 Max.   :14.200   Max.   :0.8150   Max.   :1.2300   Max.   :
    26.050
   chlorides       free.sulfur.dioxide total.sulfur.dioxide
 Min.   :0.00900   Min.   :  3.0       Min.   : 21.0
 1st Qu.:0.03600   1st Qu.: 23.0       1st Qu.:108.0
 Median :0.04300   Median : 34.0       Median :134.0
 Mean   :0.04508   Mean   : 34.9       Mean   :138.4
 3rd Qu.:0.05000   3rd Qu.: 45.0       3rd Qu.:167.0
 Max.   :0.21100   Max.   :146.5       Max.   :366.5
    density             pH          sulphates         alcohol
 Min.   :0.9871   Min.   :2.790   Min.   :0.2200   Min.   : 8.50
 1st Qu.:0.9918   1st Qu.:3.090   1st Qu.:0.4100   1st Qu.: 9.50
 Median :0.9937   Median :3.180   Median :0.4700   Median :10.40
 Mean   :0.9939   Mean   :3.189   Mean   :0.4878   Mean   :10.54
 3rd Qu.:0.9959   3rd Qu.:3.283   3rd Qu.:0.5500   3rd Qu.:11.40
 Max.   :1.0030   Max.   :3.810   Max.   :0.9800   Max.   :14.00
             quality
 poor_or_average :950
 good_or_excellent:274



output class distribuition (25% samples):

 poor_or_average good_or_excellent
             950               274
nr. training samples: 856
nr. test samples: 368
NSGAII optimization:
time elapsed: 124.027
gamma: 0.09344539 C: 0.4146168 features: 1 2 4 5 6 7 9 10 11 ; f
    =( 0.8449871 9 )
gamma: 0.002701287 C: 48.64076 features: 6 11 ; f=( 0.8044546 2)
gamma: 4.332014e-05 C: 876.2796 features: 3 5 6 7 9 11 ;
  f=(0.827304 6)
gamma: 0.002422175 C: 56.40689 features: 11 ; f=( 0.7830263 1 )
gamma: 4.332014e-05 C: 948.0165 features: 5 6 7 9 11 ;
  f=( 0.8255598 5 )
gamma: 0.4768549 C: 0.0702998 features: 1 2 4 5 6 7 10 11 ;
 f=( 0.8399648 8 )
gamma: 0.0007962147 C: 948.0165 features: 5 6 7 11 ;
 f=( 0.8144563 4 )
fit SVM with nr features: 4 nr samples: 856 gamma: 0.0007962147
    C: 948.0165
inputs: chlorides free.sulfur.dioxide total.sulfur.dioxide
    alcohol
```

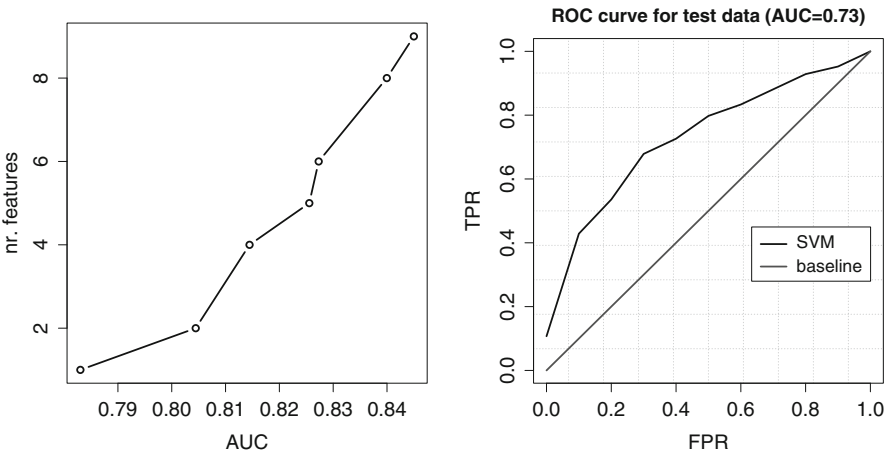

Fig. 7.7 The optimized Pareto front (*left*) and ROC curve for the SVM with four inputs (*right*) for the white wine quality task

The code starts by showing the summary of the selected dataset (with 25 % of the data). The output classes are biased, where the most (78 %) of the samples are related to poor or average wines. The NSGAII algorithm optimizes a Pareto front with seven solutions. The left of Fig. 7.7 shows such front, which is non-convex, and confirms the number of features versus classification performance trade-off. The example code selects a particular model, in this case the SVM that uses four inputs (AUC = 0.81). The right of Fig. 7.7 shows the ROC curve for such model, computed over the test set. The respective AUC value is also shown in the plot and it is slightly lower (difference of 0.07) than the one obtained using an internal threefold cross-validation. This result was expected given that test set metrics tend to be lower than validation metrics. The search algorithm has access to the validation samples and thus it can highly optimize this value. Yet, the test data is only used to measure the true generalization capabilities of the optimized model and after the search is completed. Nevertheless, it should be stressed that 0.73 corresponds to a good discrimination, while the model includes a very small number of input variables. Thus, this is an interesting classification model that was automatically found by the multi-objective search.

7.5 Command Summary

`arima`	Fit an ARIMA time series model
`auto.arima`	Automatic identification and estimation of an ARIMA model (package `forecast`)
`displacement`	Displacement operator (chapter file `"oea.R"`)

dist	Computes a distance matrix between rows of a data matrix
exchange	Exchange operator (chapter file "oea.R")
fit	Fit a supervised data mining model (package rminer)
forecast	Package for time series forecasting
forecast()	Generic function for forecasting from a time series model (package forecast)
gArea()	Compute the area of a polygon (package rgeos)
holdout	Returns indexes for holdout data split with training and test sets (package rminer)
insertion	Insertion operator (chapter file "oea.R")
mining	Trains and tests a model under several runs and a given validation method (package rminer)
mgraph	Plots a data mining result graph (package rminer)
mmetric	Compute classification or regression error metrics (package rminer)
ox	Order crossover (OX) operator (chapter file "oea.R")
oea	Order representation evolutionary algorithm (chapter file "oea.R")
ox	Order crossover (OX) operator (chapter file "oea.R")
plot	Plot function for geometry objects (package rgeos)
pmx	Partially matched crossover (PMX) operator (chapter file "oea.R")
predict	Predict function for fit objects (package rminer)
readWKT	Read WKT format into a geometry object (package rgeos)
rgeos	Package that interfaces to geometry engine—open source
rminer	Package for a simpler use of classification and regression data mining methods
TSP	Package for traveling salesman problems
TSP()	Creates a TSP instance (package TSP)

7.6 Exercises

7.1. Encode the cycle crossover (cx function) for order representations, which performs a number of cycles between two parent solutions: P_1 and P_2 (Rocha et al. 2001). Cycle 1 starts with the first value in P_1 (v_1) and analyzes the value at same position in P_2 (v). Then, it searches for v in P_1 and analyzes the corresponding value at position P_2 (new v). This procedure continues until the new v is equal to v_1, ending the cycle. All P_1 and P_2 genes that were found in this cycle are marked. Cycle 2 starts with the first value in P_1 that is not marked and ends as described in cycle 1. The whole process proceeds with similar cycles until all genes have been

marked. The genes marked in odd cycles are copied from P_1 to child 1 and from P_2 to child 2, while genes marked in even cycles are copied from P_1 to child 2 and from P_2 to child 1.

Show the children that result from applying the cycle crossover to the parents $P_1 = (1,2,3,4,5,6,7,8,9)$ and $P_2 = (9,8,1,2,3,4,5,6,7)$.

7.2. Encode the random mutation (`randomm`) and random crossover (`randomx`) operators that randomly select an ordered mutation (exchange, insertion, or displacement) or crossover (PMX, OX or CV). Optimize the same Qatar TSP instance using two simulated annealing and evolutionary algorithm variants that use the new `randomm` and `randomx` operators. Using the same setting of Sect. 7.2, show the total distance of the optimized simulated annealing and evolutionary algorithm tours.

7.3. Using the same sunspots TSF example (from Sect. 7.3), optimize coefficients of the $ARIMA(2,0,1)$ model using a particle swarm optimization method and compare the MAE one-step ahead forecasts with the method returned by `auto.arima` function. As lower and upper bounds for the particle swarm optimization use the $[-1,1]$ range for all coefficients of ARIMA except m, which should be searched around the sunspots average (within $\pm 10\,\%$ of the average value).

7.4. Change the wine classification code (from Sect. 7.4) such that three quality classes are defined: "bad"—3, 4 or 5; "average"—6; "good"—7, 8 or 9. To speed up the execution of this exercise, consider only 10 % of the original samples (randomly selected). Then, adapt the optimization to perform only model selection (search for γ and C; use of a fixed number of 11 inputs) and consider three objectives: the maximization of the AUC value for each class label (use the `metric="AUCCLASS"` argument for the `mmetric` function). Finally, show the Pareto front values in the console and also in a plot using the `scatterplot3d` function.

References

Applegate D, Bixby R, Chvátal V, Cook W (2001) TSP cuts which do not conform to the template paradigm. In: Computational combinatorial optimization. Springer, Berlin, pp 261–303

Applegate DL, Bixby RE, Chvatal V, Cook WJ (2011) The traveling salesman problem: a computational study. Princeton University Press, Princeton

Bache K, Lichman M (2013) UCI machine learning repository. http://archive.ics.uci.edu/ml

Bäck T, Schwefel HP (1993) An overview of evolutionary algorithms for parameter optimization. Evol Comput 1(1):1–23

Baluja S (1994) Population-based incremental learning: a method for integrating genetic search based function optimization and competitive learning. Tech. rep., DTIC Document

Banzhaf W, Nordin P, Keller R, Francone F (1998) Genetic programming. An introduction. Morgan Kaufmann, San Francisco

Bélisle CJ (1992) Convergence theorems for a class of simulated annealing algorithms on R d. J Appl Probab 29:885–895

Boyd S, Vandenberghe L (2004) Convex optimization. Cambridge University Press, Cambridge

Brownlee J (2011) Clever algorithms: nature-inspired programming recipes, Lulu

Caflisch RE (1998) Monte carlo and quasi-monte carlo methods. Acta Numer 1998:1–49

Chen WN, Zhang J, Chung HS, Zhong WL, Wu WG, Shi YH (2010) A novel set-based particle swarm optimization method for discrete optimization problems. IEEE Trans Evol Comput 14(2):278–300

Clerc M (2012) Standard particle swarm optimization. hal-00764996, version 1. http://hal.archives-ouvertes.fr/hal-00764996

Cortes C, Vapnik V (1995) Support vector networks. Mach Learn 20(3):273–297

Cortez P (2010) Data mining with neural networks and support vector machines using the R/rminer tool. In: Perner P (ed) Advances in data mining: applications and theoretical aspects. 10th industrial conference on data mining. Lecture notes in artificial intelligence, vol 6171. Springer, Berlin, pp 572–583

Cortez P (2012) Data mining with multilayer perceptrons and support vector machines. Springer, Berlin, pp 9–25 (Chap. 2)

Cortez P, Rocha M, Neves J (2004) Evolving time series forecasting ARMA models. J Heuristics 10(4):415–429

Cortez P, Cerdeira A, Almeida F, Matos T, Reis J (2009) Modeling wine preferences by data mining from physicochemical properties. Dec Support Syst 47(4):547–553

Croes G (1958) A method for solving traveling-salesman problems. Oper Res 6(6):791–812

Deb K (2001) Multi-objective optimization. In: Multi-objective optimization using evolutionary algorithms. Wiley, Chichester, pp 13–46

Eberhart R, Kennedy J, Shi Y (2001) Swarm intelligence. Morgan Kaufmann, San Francisco

P. Cortez, *Modern Optimization with R*, Use R!, DOI 10.1007/978-3-319-08263-9

Eberhart RC, Shi Y (2011) Computational intelligence: concepts to implementations. Morgan Kaufmann, San Francisco
Fawcett T (2006) An introduction to ROC analysis. Pattern Recognit Lett 27:861–874
Flasch O (2013) A friendly introduction to rgp. http://cran.r-project.org/web/packages/rgp/vignettes/rgp_introduction.pdf
Freitas AA (2004) A critical review of multi-objective optimization in data mining: a position paper. ACM SIGKDD Explor Newslett 6(2):77–86
Glover F (1986) Future paths for integer programming and links to artificial intelligence. Comput Oper Res 13(5):533–549
Glover F (1990) Tabu search: a tutorial. Interfaces 20(4):74–94
Glover F, Laguna M (1998) Tabu search. Springer, Heidelberg
Goldberg DE, Deb K (1991) A comparative analysis of selection schemes used in genetic algorithms, Urbana 51:61801–62996.
Gonzalez-Fernandez Y, Soto M (2012) copulaedas: an R package for estimation of distribution algorithms based on Copulas. arXiv preprint arXiv:12095429
Guyon I, Elisseeff A (2003) An introduction to variable and feature selection. J Mach Learn Res 3:1157–1182
Holland J (1975) Adaptation in natural and artificial systems. Ph.D. thesis, University of Michigan
Hsu CH, Chang CC, Lin CJ (2003) A practical guide to support vector classification. Tech. rep., National Taiwan University
Huang CM, Lee YJ, Lin DK, Huang SY (2007) Model selection for support vector machines via uniform design. Comput Stat Data Anal 52(1):335–346
Huband S, Hingston P, Barone L, While L (2006) A review of multiobjective test problems and a scalable test problem toolkit. IEEE Trans Evol Comput 10(5):477–506
Ihaka R, Gentleman R (1996) R: a language for data analysis and graphics. J Comput Graph Stat 5(3):299–314
Joe H (1997) Multivariate models and dependence concepts, vol 73. CRC Press, Boca Raton
Kaboudan MA (2003) Forecasting with computer-evolved model specifications: a genetic programming application. Comput Oper Res 30(11):1661–1681
Kennedy J, Eberhart R (1995) Particle swarm optimization. In: ICNN'95 - IEEE international conference on neural networks proceedings. IEEE Computer Society, Perth, pp 1942–1948
Kohavi R (1995) A study of cross-validation and bootstrap for accuracy estimation and model selection. In: Proceedings of the international joint conference on artificial intelligence (IJCAI), vol 2. Morgan Kaufmann, Montreal
Konak A, Coit DW, Smith AE (2006) Multi-objective optimization using genetic algorithms: a tutorial. Reliab Eng Syst Saf 91(9):992–1007
Larrañaga P, Lozano JA (2002) Estimation of distribution algorithms: a new tool for evolutionary computation, vol 2. Kluwer Academic, Boston
Lucasius CB, Kateman G (1993) Understanding and using genetic algorithms part 1. Concepts, properties and context. Chemom Intell Lab Syst 19(1):1–33
Luke S (2012) Essentials of metaheuristics. Lulu.com, online version at http://cs.gmu.edu/~sean/book/metaheuristics
Makridakis S, Weelwright S, Hyndman R (1998) Forecasting: methods and applications, 3rd edn. Wiley, New York
Mendes R (2004) Population topologies and their influence in particle swarm performance. Ph.D. thesis, Universidade do Minho
Mendes R, Cortez P, Rocha M, Neves J (2002) Particle swarms for feedforward neural network training. In: Proceedings of the 2002 international joint conference on neural networks (IJCNN 2002). IEEE Computer Society, Honolulu, pp 1895–1899
Michalewicz Z (1996) Genetic algorithms + data structures = evolution programs. Springer, Berlin
Michalewicz Z (2008) Adaptive Business Intelligence, Computer Science Course 7005 Handouts
Michalewicz Z, Fogel D (2004) How to solve it: modern heuristics. Springer, Berlin
Michalewicz Z, Schmidt M, Michalewicz M, Chiriac C (2006) Adaptive business intelligence. Springer, Berlin

Michalewicz Z, Schmidt M, Michalewicz M, Chiriac C (2007) Adaptive business intelligence: three case studies. In: Evolutionary computation in dynamic and uncertain environments. Springer, Berlin, pp 179–196

Muenchen RA (2013) The popularity of data analysis software. http://r4stats.com/articles/popularity/

Mühlenbein H (1997) The equation for response to selection and its use for prediction. Evol Comput 5(3):303–346

Mullen K, Ardia D, Gil D, Windover D, Cline J (2011) Deoptim: an r package for global optimization by differential evolution. J Stat Softw 40(6):1–26

Paradis E (2002) R for beginners. Montpellier (F): University of Montpellier. http://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf

Price KV, Storn RM, Lampinen JA (2005) Differential evolution a practical approach to global optimization. Springer, Berlin

R Core Team (2013) R: a language and environment for statistical computing. R Foundation for Statistical Computing, Vienna. http://www.R-project.org/

Reinelt G (1994) The traveling salesman: computational solutions for TSP applications. Springer, New York

Robert C, Casella G (2009) Introducing Monte Carlo methods with R. Springer, New York

Rocha M, Cortez P, Neves J (2000) The Relationship between learning and evolution in static and in dynamic environments. In: Fyfe C (ed) Proceedings of the 2nd ICSC symposium on engineering of intelligent systems (EIS'2000). ICSC Academic Press, Paisley, pp 377–383

Rocha M, Mendes R, Cortez P, Neves J (2001) Sitting guest at a wedding party: experiments on genetic and evolutionary constrained optimization. In: Proceedings of the 2001 congress on evolutionary computation (CEC2001), vol 1. IEEE Computer Society, Seoul, pp 671–678

Rocha M, Cortez P, Neves J (2007) Evolution of neural networks for classification and regression. Neurocomputing 70:2809–2816

Rocha M, Sousa P, Cortez P, Rio M (2011) Quality of service constrained routing optimization using evolutionary computation. Appl Soft Comput 11(1):356–364

Schrijver A (1998) Theory of linear and integer programming. Wiley, Chichester

Stepnicka M, Cortez P, Donate JP, Stepnicková L (2013) Forecasting seasonal time series with computational intelligence: on recent methods and the potential of their combinations. Expert Syst Appl 40(6):1981–1992

Storn R, Price K (1997) Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. J Glob Optim 11(4):341–359

Tang K, Li X, Suganthan P, Yang Z, Weise T (2009) Benchmark functions for the cec'2010 special session and competition on large-scale global optimization. Tech. rep., Technical report, University of Science and Technology of China

Vance A (2009) R You Ready for R? http://bits.blogs.nytimes.com/2009/01/08/r-you-ready-for-r/

Venables W, Smith D, R Core Team (2013) An introduction to R. http://cran.r-project.org/doc/manuals/R-intro.pdf

Wolpert DH, Macready WG (1997) No free lunch theorems for optimization. IEEE Trans Evol Comput 1(1):67–82

Wu X, Kumar V, Quinlan J, Gosh J, Yang Q, Motoda H, MacLachlan G, Ng A, Liu B, Yu P, Zhou Z, Steinbach M, Hand D, Steinberg D (2008) Top 10 algorithms in data mining. Knowl Inf Syst 14(1):1–37

Zuur A, Ieno E, Meesters E (2009) A beginner's guide to R. Springer, New York

Solutions

Exercises of Chap. 2

2.1

```
v=rep(0,10)    # same as: v=vector(length=10);v[]=0
v[c(3,7,9)]=1 # update values
print(v)       # show v
```

2.2

```
v=seq(2,50,by=2) # one way
print(v)
v=(1:25)*2 # other way
print(v)
```

2.3

```
m=matrix(nrow=3,ncol=4)
m[1,]=1:4
m[2,]=sqrt(m[1,])
m[3,]=sqrt(m[2,])
m[,4]=m[,3]^2 # m[,3]*m[,3]
print(round(m,digits=2))
cat("sums of rows:",round(apply(m,1,sum),digits=2),"\n")
cat("sums of columns:",round(apply(m,2,sum),digits=2),"\n")
```

2.4

```
# 1 - use of for ... if
counteven1=function(x)
{ r=0
  for(i in 1:length(x))
     { if(x[i]%%2==0) r=r+1 }
  return(r)
}
```


P. Cortez, *Modern Optimization with R*, Use R!, DOI 10.1007/978-3-319-08263-9

```
# 2 - use of sapply
# auxiliary function
ifeven=function(x) # x is a number
{ if(x%%2) return(TRUE) else return(FALSE)}

counteven2=function(x)
{ return(sum(sapply(x,ifeven))) }

# 3 - use of direct condition (easiest way)
counteven3=function(x)
{ return(sum(x%%2==0)) }

x=1:10
cat("counteven1:",counteven1(x),"\n")
cat("counteven2:",counteven2(x),"\n")
cat("counteven3:",counteven3(x),"\n")
```

2.5

```
DIR="" # change to other directory if needed
pdf(paste(DIR,"maxsin.pdf",sep=""),width=5,height=5) # create
    PDF
D=8 # number of binary digits, the dimension
x=0:(2^D-1);y=sin(pi*x/2^D)
plot(x,y,type="l",ylab="evaluation function",
     xlab="search space",lwd=2)
pmax=c(x[which.max(y)],max(y)) # set the maximum point
points(pmax[1],pmax[2],pch=19,lwd=2) # plot the maximum
legend("topright","optimum",pch=19,lwd=2) # add a legend
dev.off() # close the graphical device
```

2.6

```
# 1
# install.packages("RCurl") # if needed, install the package
library(RCurl)
# 2
fires=getURL("http://archive.ics.uci.edu/ml/
    machine-learning-databases/forest-fires/forestfires.csv")
write(fires,file="forestfires.csv") # write to working directory
# 3, read file:
fires=read.table("forestfires.csv",header=TRUE,sep=",")
# 4
aug=fires$temp[fires$month=="aug"]
cat("mean temperature in Aug.:",mean(aug),"\n")
# 5
feb=fires$temp[fires$month=="feb"]
jul=fires$temp[fires$month=="jul"]
sfeb=sample(feb,10)
sjul=sample(jul,10)
saug=sample(aug,10)
p1=t.test(saug,sfeb)$p.value
p2=t.test(saug,sjul)$p.value
p3=t.test(sjul,sfeb)$p.value
```

```
cat("p-values (Aug-Feb,Aug-Jul,Jul-Feb):",
    round(c(p1,p2,p3),digits=2),"\n")
# 6
aug100=fires[fires$month=="aug"&fires$area>100,]
print(aug100)
# 7
write.table(aug100,"aug100.csv",sep=",",row.names=FALSE)
```

Exercises of Chap. 3

3.1

```
source("blind.R") # load the blind search methods

binint=function(x,D)
{ x=rev(intToBits(x)[1:D]) # get D bits
  # remove extra 0s from raw type:
  as.numeric(unlist(strsplit(as.character(x),""))[(1:D)*2])
}
intbin=function(x) sum(2^(which(rev(x==1))-1))
maxsin=function(x,Dim) sin(pi*(intbin(x))/(2^Dim))

D=16 # number of dimensions

# blind search:
PTM=proc.time() # start clock
x=0:(2^D-1) # integer search space
search=t(sapply(x,binint,D=D))
S=fsearch(search,maxsin,"max",D) # full search
sec=(proc.time()-PTM)[3] # get seconds elapsed
cat("fsearch s:",S$sol,"f:",S$eval,"time:",sec,"s\n")

# adapted grid search:
N=1000
PTM=proc.time() # start clock
x=seq(0,2^D-1,length.out=N)
search=t(sapply(x,binint,D=D))
S=fsearch(search,maxsin,"max",D) # grid
sec=(proc.time()-PTM)[3] # get seconds elapsed
cat("gsearch s:",S$sol,"f:",S$eval,"time:",sec,"s\n")

# adapted monte carlo search:
PTM=proc.time() # start clock
x=sample(0:2^D-1,N)
search=t(sapply(x,binint,D=D))
S=fsearch(search,maxsin,"max",D) # grid
sec=(proc.time()-PTM)[3] # get seconds elapsed
cat("mcsearch s:",S$sol,"f:",S$eval,"time:",sec,"s\n")
```

3.2

```
source("blind.R") # load the blind search methods
source("grid.R") # load the grid search methods
source("functions.R") # load the profit function

D=5 # number of dimensions
# grid search code:
S1=gsearch(rep(11,D),rep(350,D),rep(450,D),profit,"max")
cat("gsearch s:",round(S$sol),"f:",S$eval,"\n")

# dfsearch code:
domain=vector("list",D)
for(i in 1:D) domain[[i]]=seq(350,450,by=11)
S=dfsearch(domain=domain,FUN=profit,type="max")
cat("dfsearch s:",round(S$sol),"f:",S$eval,"\n")
```

3.3

```
source("blind.R") # load the blind search methods
source("montecarlo.R") # load the monte carlo method

rastrigin=function(x) 10*length(x)+sum(x^2-10*cos(2*pi*x))

# experiment setup parameters:
D=30
Runs=30
N=10^c(2,3,4) # number of samples

# perform all monte carlo searches:
S=matrix(nrow=Runs,ncol=length(N))
for(j in 1:length(N)) # cycle all number of samples
for(i in 1:Runs) # cycle all runs
   S[i,j]=mcsearch(N[j],rep(-5.2,D),rep(5.2,D),
                   rastrigin,"min")$eval
# compare average results:
p21=t.test(S[,2],S[,1])$p.value
p31=t.test(S[,3],S[,2])$p.value
cat("N=",N,"\n")
cat("average f:",apply(S,2,mean),"\n")
cat("p-value (N=",N[2],"vs N=",N[1],")=",
    round(p21,digits=2),"\n")
cat("p-value (N=",N[3],"vs N=",N[2],")=",
    round(p31,digits=2),"\n")
boxplot(S[,1],S[,2],S[,3],names=paste("N=",N,sep=""))
```

Exercises of Chap. 4

4.1

```
# steepest ascent hill climbing method:
hclimbing=function(par,fn,change,lower,upper,control,
                   type="min",...)
```

```
{ fpar=fn(par,...)
  for(i in 1:control$maxit)
     {
      par1=change(par,lower,upper)
      fpar1=fn(par1,...)
      if(control$N>0) # steepest ascent code
      { for(j in 1:control$N-1)
         { cand=change(par,lower,upper)
           fcand=fn(cand,...)
           if(   (type=="min" && fcand<fpar1)
              || (type=="max" && fcand>fpar1))
              {par1=cand;fpar1=fcand}
         }
      }
      if(control$REPORT>0 &&(i==1||i%%control$REPORT==0))
        cat("i:",i,"s:",par,"f:",fpar,"s'",par1,"f:",fpar1,"\n")
      if(   (type=="min" && fpar1<fpar)
         || (type=="max" && fpar1>fpar)) { par=par1;fpar=fpar1 }
     }
  if(control$REPORT>=1) cat("best:",par,"f:",fpar,"\n")
  return(list(sol=par,eval=fpar))
}
```

4.2

```
source("hill.R") # load the hill climbing methods

intbin=function(x) sum(2^(which(rev(x==1))-1))
maxsin=function(x) sin(pi*(intbin(x))/(2^D))
D=16 # number of dimensions
s=rep(0,D) # initial search point

# hill climbing:
maxit=20
C=list(maxit=maxit,REPORT=0) # maximum of 10 iterations
ichange=function(par,lower,upper) # integer change
{hchange(par,lower,upper,rnorm,mean=0,sd=1) }
b=hclimbing(s,maxsin,change=ichange,lower=rep(0,D),
  upper=rep(1,D),
          control=C,type="max")
cat("hill b:",b$sol,"f:",b$eval,"\n")

# simulated annealing:
eval=function(x) -maxsin(x)
ichange2=function(par) # integer change
{D=length(par);hchange(par,lower=rep(0,D),upper=rep(1,D),rnorm,
    mean=0,sd=1)}
C=list(maxit=maxit)
b=optim(s,eval,method="SANN",gr=ichange2,control=C)
cat("sann b:",b$par,"f:",abs(b$value),"\n")

# tabu search:
b=tabuSearch(size=D,iters=maxit,objFunc=maxsin,config=s,neigh=4,
    listSize=8)
```

```
ib=which.max(b$eUtilityKeep) # best index
cat("tabu b:",b$configKeep[ib,],"f:",b$eUtilityKeep[ib],"\n")
```

4.3

```
library(tabuSearch) # get tabuSearch

rastrigin=function(x) f=10*length(x)+sum(x^2-10*cos(2*pi*x))
intbin=function(x) # convert binary to integer
{ sum(2^(which(rev(x==1))-1)) } # explained in Chapter 3
breal=function(x) # convert binary to D real values
{ # note: D and bits need to be set outside this function
  s=vector(length=D)
  for(i in 1:D) # convert x into s:
  { ini=(i-1)*bits+1;end=ini+bits-1
    n=intbin(x[ini:end])
    s[i]=lower+n*drange/2^bits
  }
  return(s)
}
# note: tabuSearch does not work well with negative evaluations
# to solve this drawback, a MAXIMUM constant is defined
MAXIMUM=10000
brastrigin=function(x) MAXIMUM-rastrigin(breal(x)) # max. goal

D=8;MAXIT=500
bits=8 # per dimension
size=D*bits
lower=-5.2;upper=5.2;drange=upper-lower
s=sample(0:1,size=size,replace=TRUE)
b=tabuSearch(size=size,iters=MAXIT,objFunc=brastrigin,config=s,
    neigh=bits,listSize=bits,nRestarts=1)
ib=which.max(b$eUtilityKeep) # best index
cat("b:",b$configKeep[ib,],"f:",MAXIMUM-b$eUtilityKeep[ib],"\n")
```

Exercises of Chap. 5

5.1

```
library(genalg) # get rba.bin

intbin=function(x) sum(2^(which(rev(x==1))-1))
maxsin=function(x) -sin(pi*(intbin(x))/(2^D))
D=16 # number of dimensions

# genetic algorithm:
GA=rbga.bin(size=D,popSize=20,iters=100,zeroToOneRatio=1,
            evalFunc=maxsin,elitism=1)
```

```
b=which.min(GA$evaluations) # best individual
cat("best:",GA$population[b,],"f:",-GA$evaluations[b],"\n")
```

5.2

```
library(pso)
library(copulaedas)
source("blind.R") # get fsearch
source("montecarlo.R") # get mcsearch

# evaluation function: -------------------------------------
eggholder=function(x) # length of x is 2
{ x=ifelse(x<lower[1],lower[1],x) # (only due to EDA):
  x=ifelse(x>upper[1],upper[1],x) # bound if needed
  f=(-(x[2]+47)*sin(sqrt(abs(x[2]+x[1]/2+47)))
     -x[1]*sin(sqrt(abs(x[1]-(x[2]+47))))
    )
  # global assignment code: <<-
  EV<<-EV+1 # increase evaluations
  if(f<BEST) BEST<<-f # minimum value
  if(EV<=MAXFN) F[EV]<<-BEST
  return(f)
}

# auxiliary functions: -------------------------------------
crun2=function(method,f,lower,upper,LP,maxit,MAXFN) # run a
    method
{ if(method=="MC")
   {
    s=runif(D,lower[1],upper[1]) # initial search point
    mcsearch(MAXFN,lower=lower,upper=upper,FUN=eggholder)
   }
  else if(method=="PSO")
     { C=list(maxit=maxit,s=LP,type="SPSO2011")
       psoptim(rep(NA,length(lower)),fn=f,
               lower=lower,upper=upper,control=C)
     }
  else if(method=="EDA")
     { setMethod("edaTerminate","EDA",edaTerminateMaxGen)
       DVEDA=VEDA(vine="DVine",indepTestSigLevel=0.01,
           copulas = c("normal"),margin = "norm")
       DVEDA@name="DVEDA"
       edaRun(DVEDA,f,lower,upper)
     }
}

successes=function(x,LIM,type="min") # number of successes
{ if(type=="min") return(sum(x<LIM)) else return(sum(x>LIM)) }

ctest2=function(Methods,f,lower,upper,type="min",Runs, # test
               D,MAXFN,maxit,LP,pdf,main,LIM) # all methods:
{ RES=vector("list",length(Methods)) # all results
  VAL=matrix(nrow=Runs,ncol=length(Methods)) # best values
  for(m in 1:length(Methods)) # initialize RES object
```

```
    RES[[m]]=matrix(nrow=MAXFN,ncol=Runs)

  for(R in 1:Runs) # cycle all runs
    for(m in 1:length(Methods))
      { EV<<-0; F<<-rep(NA,MAXFN) # reset EV and F
        if(type=="min") BEST<<-Inf else BEST<<- -Inf # reset BEST
        suppressWarnings(crun2(Methods[m],f,lower,upper,LP,maxit,
            MAXFN))
        RES[[m]][,R]=F # store all best values
        VAL[R,m]=F[MAXFN] # store best value at MAXFN
       }
   # compute average F result per method:
   AV=matrix(nrow=MAXFN,ncol=length(Methods))
   for(m in 1:length(Methods))
     for(i in 1:MAXFN)
       AV[i,m]=mean(RES[[m]][i,])
   # show results:
   cat(main,"\n",Methods,"\n")
   cat(round(apply(VAL,2,mean),digits=0)," (average best)\n")
   cat(round(100*apply(VAL,2,successes,LIM,type)/Runs,
             digits=0)," (%successes)\n")

  # create pdf file:
  pdf(paste(pdf,".pdf",sep=""),width=5,height=5,paper="special")
  par(mar=c(4.0,4.0,1.8,0.6)) # reduce default plot margin
  MIN=min(AV);MAX=max(AV)
  # use a grid to improve clarity:
  g1=seq(1,MAXFN,length.out=500) # grid for lines
  plot(g1,AV[g1,1],ylim=c(MIN,MAX),type="l",lwd=2,main=main,
       ylab="average best",xlab="number of evaluations")
  for(i in 2:length(Methods)) lines(g1,AV[g1,i],lwd=2,lty=i)
  if(type=="min") position="topright" else position="bottomright"
  legend(position,legend=Methods,lwd=2,lty=1:length(Methods))
  dev.off() # close the PDF device
}

# define EV, BEST and F:
MAXFN=1000
EV=0;BEST=Inf;F=rep(NA,MAXFN)
# define method labels:
Methods=c("MC","PSO","EDA")
# eggholder comparison: ----------------------------------
Runs=10; D=2; LP=20; maxit=50
lower=rep(-512,D);upper=rep(512,D)
ctest2(Methods,eggholder,lower,upper,"min",Runs,D,MAXFN,
 maxit,LP,
      "comp-eggholder","eggholder (D=2)",-950)
```

5.3

```
source("functions.R") # bag prices functions
library(copulaedas) # EDA

# auxiliary functions: -----------------------------------
```

```
# returns TRUE if prices are sorted in descending order
prices_ord=function(x)
{ d=diff(x) # d lagged differences x(i+1)-x(i)
  if(sum(d>=0)) return (FALSE) else return (TRUE)
}
ord_prices=function(x)
{ x=sort(x,decreasing=TRUE) # sort x
  # x is sorted but there can be ties:
  k=2          # remove ties by removing $1
  while(!prices_ord(x)) # at each iteration
    { if(x[k]==x[k-1]) x[k]=x[k]-1
      k=k+1
    }
  return(x)
}

# evaluation function: -----------------------------------
cprofit3=function(x) # bag prices with death penalty
{ x=round(x,digits=0) # convert x into integer
  x=ifelse(x<1,1,x)        # assure that x is within
  x=ifelse(x>1000,1000,x)  # the [1,1000] bounds
  if(!prices_ord(x)) res=Inf # if needed, death penalty!!!
  else
    {
     s=sales(x);c=cost(s);profit=sum(s*x-c)
     # if needed, store best value
     if(profit>BEST) { BEST<<-profit; B<<-x}
     res=-profit # minimization task!
    }
  EV<<-EV+1 # increase evaluations
  if(EV<=MAXFN) F[EV]<<-BEST
  return(res)
}
# example of a very simple and fast repair of a solution:
# sort the solution values!
localRepair2=function(eda, gen, pop, popEval, f, lower, upper)
{
 for(i in 1:nrow(pop))
 { x=pop[i,]
   x=round(x,digits=0) # convert x into integer
   x=ifelse(x<lower[1],lower[1],x) # assure x within
   x=ifelse(x>upper[1],upper[1],x) # bounds
   if(!prices_ord(x)) x=ord_prices(x) # order x
   pop[i,]=x;popEval[i]=f(x) # replace x in population
 }
 return(list(pop=pop,popEval=popEval))
}

# experiment: --------------------------------------------
MAXFN=5000
Runs=50; D=5; LP=50; maxit=100
lower=rep(1,D);upper=rep(1000,D)
```

```
Methods=c("Death","Repair")
setMethod("edaTerminate","EDA",edaTerminateMaxGen)
UMDA=CEDA(copula="indep",margin="norm"); UMDA@name="UMDA"

RES=vector("list",length(Methods)) # all results
VAL=matrix(nrow=Runs,ncol=length(Methods)) # best values
for(m in 1:length(Methods)) # initialize RES object
   RES[[m]]=matrix(nrow=MAXFN,ncol=Runs)
for(R in 1:Runs) # cycle all runs
  {
    B=NA;EV=0; F=rep(NA,MAXFN); BEST= -Inf # reset vars.
    setMethod("edaOptimize","EDA",edaOptimizeDisabled)
    setMethod("edaTerminate","EDA",edaTerminateMaxGen)
    suppressWarnings(edaRun(UMDA,cprofit3,lower,upper))
    RES[[1]][,R]=F # store all best values
    VAL[R,1]=F[MAXFN] # store best value at MAXFN

    B=NA;EV=0; F=rep(NA,MAXFN); BEST= -Inf # reset vars.
    # set local repair search method:
    setMethod("edaOptimize","EDA",localRepair2)
    # set additional termination criterion:
    setMethod("edaTerminate","EDA",
              edaTerminateCombined(edaTerminateMaxGen,
                  edaTerminateEvalStdDev))
    # this edaRun might produces warnings or errors:
    suppressWarnings(try(edaRun(UMDA,cprofit3,lower,upper),
        silent=TRUE))
    if(EV<MAXFN) # if stopped due to EvalStdDev
       F[(EV+1):MAXFN]=rep(F[EV],MAXFN-EV) # replace NAs
    RES[[2]][,R]=F # store all best values
    VAL[R,2]=F[MAXFN] # store best value at MAXFN
  }

# compute average F result per method:
MIN=Inf
AV=matrix(nrow=MAXFN,ncol=length(Methods))
for(m in 1:length(Methods))
  for(i in 1:MAXFN)
   {
    AV[i,m]=mean(RES[[m]][i,])
    # update MIN for plot (different than -Inf):
    if(AV[i,m]!=-Inf && AV[i,m]<MIN) MIN=AV[i,m]
   }
# show results:
cat(Methods,"\n")
cat(round(apply(VAL,2,mean),digits=0)," (average best)\n")
# Mann-Whitney non-parametric test:
p=wilcox.test(VAL[,1],VAL[,2],paired=TRUE)$p.value
cat("p-value:",round(p,digits=2),"(<0.05)\n")

# create PDF file:
pdf("comp-bagprices-constr2.pdf",width=5,height=5,
    paper="special")
```

```
par(mar=c(4.0,4.0,1.8,0.6)) # reduce default plot margin
# use a grid to improve clarity:
g1=seq(1,MAXFN,length.out=500) # grid for lines
MAX=max(AV)
plot(g1,AV[g1,2],ylim=c(MIN,MAX),type="l",lwd=2,
     main="bag prices with constraint 2",
     ylab="average best",xlab="number of evaluations")
lines(g1,AV[g1,1],lwd=2,lty=2)
legend("bottomright",legend=rev(Methods),lwd=2,lty=1:4)
dev.off() # close the PDF device
```

5.4

```
library(rgp) # load rgp

# auxiliary functions:
eggholder=function(x) # length of x is 2
  f=(-(x[2]+47)*sin(sqrt(abs(x[2]+x[1]/2+47)))
     -x[1]*sin(sqrt(abs(x[1]-(x[2]+47))))
    )
fwrapper=function(x,f)
{ res=suppressWarnings(f(x[1],x[2]))
  # if NaN is generated (e.g. sqrt(-1)) then
  if(is.nan(res)) res=Inf # replace by Inf
  return(res)
}

# configuration of the genetic programming:
ST=inputVariableSet("x1","x2")
cF1=constantFactorySet(function() sample(c(2,47),1) )
FS=functionSet("+","-","/","sin","sqrt","abs")
# set the input samples:
samples=500
domain=matrix(ncol=2,nrow=samples)
domain[]=runif(samples,-512,512)
eval=function(f) # evaluation function
  mse(apply(domain,1,eggholder),apply(domain,1,fwrapper,f))

# run the genetic programming:
gp=geneticProgramming(functionSet=FS,inputVariables=ST,
                      constantSet=cF1,populationSize=100,
                      fitnessFunction=eval,
                      stopCondition=makeTimeStopCondition(20),
                      verbose=TRUE)
# show the results:
b=gp$population[[which.min(gp$fitnessValues)]]
cat("best solution (f=",eval(b),"):\n")
print(b)
L1=apply(domain,1,eggholder)
L2=apply(domain,1,fwrapper,b)
# sort L1 and L2 (according to L1 indexes)
# for an easier comparison of both curves:
L1=sort.int(L1,index.return=TRUE)
L2=L2[L1$ix]
```

```
L1=L1$x
MIN=min(L1,L2);MAX=max(L1,L2)
plot(L1,ylim=c(MIN,MAX),type="l",lwd=2,lty=1,
     xlab="points",ylab="function values")
lines(L2,type="l",lwd=2,lty=2)
legend("bottomright",leg=c("eggholder","GP function"),lwd=2,lty
    =1:2)
# note: the fit is not perfect, but the search space is
#       too large
```

Exercises of Chap. 6

6.1

```
source("hill.R") # load the blind search methods
source("mo-tasks.R") # load MO bag prices task
source("lg-ga.R") # load tournament function

# lexicographic hill climbing, assumes minimization goal:
lhclimbing=function(par,fn,change,lower,upper,control,
                    ...)
{
  for(i in 1:control$maxit)
     {
      par1=change(par,lower,upper)
      if(control$REPORT>0 &&(i==1||i%%control$REPORT==0))
         cat("i:",i,"s:",par,"f:",eval(par),"s'",par1,"f:",
             eval(par1),"\n")
      pop=rbind(par,par1) # population with 2 solutions
      I=tournament(pop,fn,k=2,n=1,m=2)
      par=pop[I,]
     }
  if(control$REPORT>=1) cat("best:",par,"f:",eval(par),"\n")
  return(list(sol=par,eval=eval(par)))
}

# lexico. hill climbing for all bag prices, one run:
D=5; C=list(maxit=10000,REPORT=10000) # 10000 iterations
s=sample(1:1000,D,replace=TRUE) # initial search
ichange=function(par,lower,upper) # integer value change
{ hchange(par,lower,upper,rnorm,mean=0,sd=1) }
LEXI=c(0.1,0.1) # explicitly defined lexico. tolerances
eval=function(x) c(-profit(x),produced(x))
b=lhclimbing(s,fn=eval,change=ichange,lower=rep(1,D),
          upper=rep(1000,D),control=C)
cat("final ",b$sol,"f(",profit(b$sol),",",produced(b$sol),")\n")
```

6.2

```
library(genalg) # load rbga function
library(mco)  # load nsga2 function

set.seed(12345) # set for replicability

# real value FES2 benchmark:
fes2=function(x)
{ D=length(x);f=rep(0,3)
  for(i in 1:D)
     {
      f[1]=f[1]+(x[i]-0.5*cos(10*pi/D)-0.5)^2
      f[2]=f[2]+abs(x[i]-(sin(i-1))^2*(cos(i-1)^2))^0.5
      f[3]=f[3]+abs(x[i]-0.25*cos(i-1)*cos(2*i-2)-0.5)^0.5
     }
  return(f)
}

D=8;m=3

# WBGA execution:
# evaluation function for WBGA
# (also used to print and get last population fes2 values:
# WBGA chromosome used: x=(w1,w2,w3,v1,v2,v3,...,vD)
#   where w_i are the weights and v_j the values
eval=function(x,REPORT=FALSE)
{ D=length(x)/2
  # normalize weights, such that sum(w)=1
  w=x[1:m]/sum(x[1:m]);v=x[(m+1):length(x)];f=fes2(v)
  if(REPORT)
    { cat("w:",round(w,2),"v:",round(v,2),"f:",round(f,2),"\n")
      return(f)
    }
  else return(sum(w*f))
}
WBGA=rbga(evalFunc=eval,
          stringMin=rep(0,D*2),stringMax=rep(1,D*2),
          popSize=20,iters=100)
print("WBGA last population:")
# S1 contains last population fes2 values in individuals x
    objectives
S1=t(apply(WBGA$population,1,eval,REPORT=TRUE))
LS1=nrow(S1)

# NSGA-II execution:
NSGA2=nsga2(fn=fes2,idim=D,odim=m,
        lower.bounds=rep(0,D),upper.bounds=rep(1,D),
        popsize=20,generations=100)
S2=NSGA2$value[NSGA2$pareto.optimal,]
print("NSGA2 last Pareto front:")
print(S2)
LS2=nrow(S2)
```

```
# Comparison of results:
library(scatterplot3d)
S=data.frame(rbind(S1,S2))
names(S)=c("f1","f2","f3")
col=c(rep("gray",LS1),rep("black",LS2))
# nice scatterplot3d
#  WBGA points are in gray
#  NSGA2 points are in black
#  NSGA2 produces a more disperse and interesting
#    Pareto front when compared with WBGA
scatterplot3d(S,pch=16,color=col)
```

Exercises of Chap. 7

7.1

```
# cycle crossover (CX) operator:
# m is a matrix with 2 parent x ordered solutions
cx=function(m)
{
 N=ncol(m)
 c=matrix(rep(NA,N*2),ncol=N)
 stop=FALSE
 k=1
 ALL=1:N
 while(length(ALL)>0)
 {
  i=ALL[1]
  # perform a cycle:
  base=m[1,i];vi=m[2,i]
  I=i
  while(vi!=base)
   {
    i=which(m[1,]==m[2,i])
    vi=m[2,i]
    I=c(I,i)
   }
  ALL=setdiff(ALL,I)
  if(k%%2==1) c[,I]=m[,I] else c[,I]=m[2:1,I]
  k=k+1
 }
 return(c)
}

# example of CX operator:
m=matrix(ncol=9,nrow=2)
m[1,]=1:9
m[2,]=c(9,8,1,2,3,4,5,6,7)
print(m)
print("---")
print(cx(m))
```

7.2

```
# this solution assumes that file "tsp.R" has already been
    executed
source("oea.R") # load ordered evolutionary algorithm
source("s7-1.R") # get the cycle operator

# random mutation
randomm=function(s)
{ return(switch(sample(1:3,1),exchange(s),insertion(s),
    displacement(s))) }

# random crossover
randomx=function(m)
{ return(switch(sample(1:3,1),pmx(m),ox(m),cx(m))) }

Methods=c("new SANN","new EA")
# new SANN:
cat("new SANN run:\n")
set.seed(12345) # for replicability
s=sample(1:N,N) # initial solution
EV=0; BEST=Inf; F=rep(NA,MAXIT) # reset these vars.
C=list(maxit=MAXIT,temp=2000,trace=TRUE,REPORT=MAXIT)
PTM=proc.time() # start clock
SANN=optim(s,fn=tour,gr=randomm,method="SANN",control=C)
sec=(proc.time()-PTM)[3] # get seconds elapsed
cat("time elapsed:",sec,"\n")
RES[,1]=F
cat("tour distance:",tour(SANN$par),"\n")

# new EA:
cat("new EA run:\n")
set.seed(12345) # for replicability
EV=0; BEST=Inf; F=rep(NA,MAXIT) # reset these vars.
pSize=30;iters=ceiling((MAXIT-pSize)/(pSize-1))
PTM=proc.time() # start clock
OEA=oea(size=N,popSize=pSize,iters=iters,evalFunc=tour,crossfunc
    =randomx,mutfunc=randomm,REPORT=iters,elitism=1)
sec=(proc.time()-PTM)[3] # get seconds elapsed
cat("time elapsed:",sec,"\n")
RES[,2]=F
cat("tour distance:",tour(OEA$population[which.min(OEA$
    evaluations),]),"\n")

# there is no improvement when compared with "tsp.R" file
```

7.3

```
# this solution assumes that file "tsf.R" has already been
    executed

library(pso) # load pso
```

```
# evaluation function of arma coefficients:
evalarma=function(s)
{ a=suppressWarnings(arima(sunspots,order=c(AR,0,MA),fixed=s))
  R=a$residuals[INIT:length(sunspots)]
  R=maeres(R)
  if(is.nan(R)) R=Inf # death penalty
  return(maeres(R))
}

AR=2;MA=1
maxit=100; LP=50
meants=mean(sunspots);K=0.1*meants
lower=c(rep(-1,(AR+MA)),meants-K)
upper=c(rep(1,(AR+MA)),meants+K)
C=list(maxit=maxit,s=LP,trace=10,REPORT=10)
set.seed(12345) # set for replicability
PSO=psoptim(rep(NA,length(lower)),fn=evalarma,
               lower=lower,upper=upper,control=C)
arima2=arima(sunspots,order=c(AR,0,MA),fixed=PSO$par)
print(arima2)
cat("pso fit MAE=",PSO$value,"\n")

# one-step ahead predictions:
f3=rep(NA,forecasts)
for(h in 1:forecasts)
  { # execute arima with fixed coefficients but with more
      in-samples:
   arima1=arima(series[1:(LIN+h-1)],order=arima2$arma[c(1,3,2)],
       fixed=arima2$coef)
   f3[h]=forecast(arima1,h=1)$mean[1]
  }
e3=maeres(outsamples-f3)
text3=paste("pso arima (MAE=",round(e3,digits=1),")",sep="")

# show quality of one-step ahead forecasts:
ymin=min(c(outsamples,f1,f3))
ymax=max(c(outsamples,f1,f3))
par(mar=c(4.0,4.0,0.1,0.1))
plot(outsamples,ylim=c(ymin,ymax),type="b",pch=1,
     xlab="time (years after 1980)",ylab="values",cex=0.8)
lines(f1,lty=2,type="b",pch=3,cex=0.5)
lines(f3,lty=3,type="b",pch=5,cex=0.5)
legend("topright",c("sunspots",text1,text3),lty=1:3,
 pch=c(1,3,5))
```

7.4

```
# this solution assumes that file "wine-quality.R" has already
    been executed

# reload wine quality dataset since a new quality is defined:
file="http://archive.ics.uci.edu/ml/machine-learning-databases/
    wine-quality/winequality-white.csv"
d=read.table(file=file,sep=";",header=TRUE)
```

```
# convert the output variable into 3 classes of wine:
# "bad" <- 3,4,5
# "average" <- 6
# "good" <- 7, 8 or 9
d$quality=cut(d$quality,c(0,5.5,6.5,10),c("bad","average",
 "good"))

n=nrow(d) # total number of samples
ns=round(n*0.10) # select only 10% of the samples for a fast
    demonstration
set.seed(12345) # for replicability
ALL=sample(1:n,ns) # contains 10% of the index samples
# show a summary of the wine quality dataset (10%):
print(summary(d[ALL,]))
cat("output class distribuition (10% samples):\n")
print(table(d[ALL,]$quality)) # show distribution of classes

# holdout split:
# select training data (for fitting the model), 70%; and
# test data (for estimating generalization capabilities), 30%.
H=holdout(d[ALL,]$quality,ratio=0.7)
cat("nr. training samples:",length(H$tr),"\n")
cat("nr. test samples:",length(H$ts),"\n")

# new evaluation function:
# x is in the form c(Gamma,C)
eval=function(x)
{ n=length(x)
  gamma=2^x[1]
  C=2^x[2]
  inputs=1:maxinputs # use all inputs
  attributes=c(inputs,output)
  # divert console:
  # sink is used to avoid kernlab ksvm messages in a few cases
  sink(file=textConnection("rval","w",local = TRUE))
  M=mining(quality~.,d[H$tr,attributes],method=c("kfold",3),
      model="svm",search=gamma,mpar=c(C,NA))
  sink(NULL) # restores console
  # AUC for the internal 3-fold cross-validation:
  auc=as.numeric(mmetric(M,metric="AUCCLASS"))
  # auc now contains 3 values, the AUC for each class
  auc1=1-auc # transform auc maximization into minimization goal
  return(c(auc1))
}

# NSGAII multi-objective optimization:
cat("NSGAII optimization:\n")
m=3 # four objectives: AUC for each class and number of features
lower=c(-15,-5)
upper=c(3,15)
PTM=proc.time() # start clock
```

```
G=nsga2(fn=eval,idim=length(lower),odim=m,lower.bounds=lower,
    upper.bounds=upper,popsize=12,generations=10)
sec=(proc.time()-PTM)[3] # get seconds elapsed
cat("time elapsed:",sec,"\n")

# show the Pareto front:
I=which(G$pareto.optimal)
for(i in I)
  { x=G$par[i,]
    n=length(x)
    gamma=2^x[1]
    C=2^x[2]
    features=round(x[3:n])
    inputs=which(features==1)
    cat("gamma:",gamma,"C:",C,"; f=(",
        1-G$value[i,1:3],")\n",sep=" ")
  }

Pareto=1-G$value[I,] # AUC for each class
Pareto=data.frame(Pareto)
names(Pareto)=c("AUC bad","AUC average","AUC good")
# sort Pareto according to f1:
S=sort.int(Pareto[,1],index.return=TRUE)
Pareto=Pareto[S$ix,]

library(scatterplot3d) # get scatterplot3d function
scatterplot3d(Pareto,xlab="f1",ylab="f2",zlab="f3",
              pch=16,type="b")

# looking at the Pareto front, the wine expert could
# select the best model and then measure the performance
# of such model on the test set...
```

Index

P. Cortez, *Modern Optimization with R*, Use R!, DOI 10.1007/978-3-319-08263-9