On Meta-heuristics in Optimization and Data Analysis. Application to Geosciences

Henri Luchian, Mihaela Elena Breaban and Andrei Bautu

Abstract This chapter presents popular meta-heuristics inspired from nature focusing on evolutionary computation (EC). The first section, as an elevator pitch, briefly walks through problem solving, touching upon notions such as *optimization problems*, *meta-heuristics*, *constraint handling*, *hybridization*, and the *No Free Lunch Theorem for optimization*, and also giving very short introductions into several most popular meta-heuristics. The next two sections are dedicated to evolutionary algorithms and swarm intelligence (SI), two of the main areas of EC. Three particular optimization methods illustrating these two areas are presented in more detail: genetic algorithms (GAs), differential evolution (DE), and particle swarm optimization (PSO). For a better understanding of these algorithms, references to R packages implementing the algorithms and code samples to solve numerical and combinatorial problems are given. The fourth section is dedicated to the use of EC techniques in data analysis. Optimization of the hyper-parameters of conventional machine learning techniques is illustrated by a case study. The last section reviews applications of meta-heuristics in geosciences.

H. Luchian · M.E. Breaban (✉)
Faculty of Computer Science, Alexandru Ioan Cuza University of Iasi, Iasi, Romania
e-mail: pmihaela@infoiasi.ro

A. Bautu
Faculty of Navigation and Naval Management, Romanian Naval Academy,
Constanta, Romania

1 A Painless Introduction

A particular characteristic of problem solving becomes evident if computers are used for searching solutions to problems. Namely, when asked to solve a given problem, one is simultaneously, if implicitly, asked to solve the meta-problem of finding the best method to solve the problem. Best may refer to saving resources most often, time in the process of finding a solution; it may also point to the required accuracy/precision of the solution or to the set of instances of the problem which must be solved, or to a threshold for positive/negative errors, etc. In many cases, simply finding a method which can successfully look for a solution to the given problem is not sufficient; the method should comply with requirements such as those enumerated above, and moreover, it should do this in the best possible way. Therefore, irrespective of what best means, in order to deal with the companion meta-problem, one needs to be acquainted with a comprehensive set of methods for solving problems: the larger the set of methods one chooses from, the better the proposed method should be.

   This may be the reason why, along with the ever increasing use of computers for solving problems, a wealth of new approaches to problem solving has been proposed.

1.1 Briefly, on Problems and Methods to Solve Them

How many problem-solving methods does one need to master? Indeed, many new methods for solving problems were invented (some may say discovered) lately. As opposed to exact deterministic algorithms, many of these new methods are weak methods; a weak method is not rigidly related to one specific problem, but rather it can be applied for solving various problems. At times, one or another such problem-solving technique appears to be most fashionable. To an outsider, genetic algorithms (GAs), artificial neural networks, particle swarm optimization, and support vector machines to name just a few seemed to successively take by storm the proscenium over the last decades. Is each new method better than the previous ones and, consequently, is the choice of the method to solve ones specific problem a matter of keeping pace with fashion? Is there one particular method that solves best, among all existing methods and all problems? A positive answer to either question would mean that we actually have a free lunch when trying to solve a given problem: we could spare the time needed to identify the best method for finding solutions to the problem. However, a theorem proven in 1995 by Wolpert and McReady (1997), called the *No Free Lunch Theorem for optimization*, shows that the answer to both questions above is negative. Informally (and leaving aside details and nuances of the theorem), the *NFLTO* states that, averaging overall problems, all solving methods have the same performance, no matter what indicator of performance is used. Obviously, the common average is obtained from various

sets of values of the performance indicators for each method and various levels of each method's performance when applied to each specific problem. This means that in general, two different methods perform at their respective best on different problems, and consequently, each of them has a poorer performance on remaining problems. It follows that there is no problem-solving method which is the "best" method to solve all problems (indeed, if a method M would have equally good performances on all problems, then this would be Ms average performance; then, any method with scattered values of the performance indicator would outperform M on some problems). Therefore, for each problem-solving method, there is a subset of all problems for which it is the best solving method in some cases, and the subset may consist of only one problem or even zero problems. Conversely, given a problem to be solved, one has to find a particular method that works best for that problem which proves that the *meta-problem* mentioned above is non-trivial. Actually, it may be a very difficult problem; similar to the way some problem-solving methods are widely used even if they are not guaranteed to provide the exact solution, an approximate but acceptably good solution to the meta-problem may be useful.

Optimization problems There is an informal conjecture stating that anything we are doing, we optimize something; or, as Clerc put it in (2006), iterative optimization is as old as life itself. While each of these two statements may be the subject of subtle philosophical debates, it is true that many problems can be stated as optimization problems. Finding the average of n real numbers is an optimization problem (*find the number a which minimizes the sum of its distances absolute values of the differences to each of the given numbers*); the same goes for decision-making problems, for machine learning ones, and many others.

An optimization problem asks to find—if it exists—an extreme value (either minimum or maximum) of a given function. Finding the required solution is, in fact, a search process performed in the space of all candidate solutions; this is why the terms *optimization method* and *search method* are sometimes loosely used as synonyms, although the term *optimization* refers to the values of the function, while *search* (through the set of candidate solutions) usually points to values of the variables of the respective function. Several simple taxonomies of optimization problems are useful when studying meta-heuristics: optimization of functions of continuous variable/discrete variable; optimization with/without constraints; optimization with a fixed/moving optimum; single objective/multiple objective optimization. Here are some examples:

- constraint optimization raises the critical problem of handling constraints;
- continuous/discrete variables point to specific meta-heuristics that originally specialize in one the two types of optimization (e.g., GAs for discrete variables; differential evolution (DE) for continuous variables);
- self-adapting meta-heuristics are recommended for solving problems with a moving optimum;

- particular variants of existing meta-heuristics have been defined for multi-objective optimization (e.g., in DE).

Meta-heuristics, described below, are seen as optimization methods (i.e., methods for solving optimization problems). While meta-heuristics can also be used for solving, for example, complex-system-design problems or machine learning problems, such problems can also be stated as optimization problems.

Meta-heuristics Any problem-solving method belongs to one of three categories: exact deterministic methods, approximate deterministic methods, and non-deterministic methods. This chapter is concerned with the second and third categories, which flourished over the last few decades.

A *heuristic* is a problem-solving method which is able to find approximate solutions to the given problem either in a (significantly) shorter time than an exact algorithm or simply when no exact algorithm can find a solution. Approximate solutions may be acceptable in various situations; Simon (1969) argues that humans tend to *satisfice* (use an acceptable approximate solution obtained reasonably quickly) when it comes to complex situations/domains.

Meta-heuristic is a relatively recent term, introduced by Glover in 1986. Various definitions and taxonomies of *meta-heuristics* were subsequently proposed; the works mentioned below discuss these in detail. It is generally accepted that meta-heuristics are problem-independent high-level strategies which guide the process of finding (approximate) solutions to given problems. However, problem-independent methods (also called *weak methods*) may well be fine-tuned by incorporating in the search procedure some problem-specific knowledge; an early paper on this is (Grefenstette 1987).

Among several existing taxonomies of meta-heuristics, the most interesting one for our discussion is the classification concerned with the number of current solutions. A *trajectory* or *single-point meta-heuristic* works with only one current solution; the current solution is iteratively subject to conditional change. Local search meta-heuristics, such as Tabu Search, Iterated Local Search, and Variable Neighborhood Search (Blum and Roli 2003), fall into this category. A *population-based meta-heuristic* iteratively change a set of candidate solutions collectively called *population*; genetic algorithm (GA) or particle swarm Optimization, among others, belong in this category.

This section briefly discusses two trajectory-based methods: iterated hill climbing and simulated annealing.

Hill climbing Hill climbing is a *weak* optimization heuristic: In order to be applied for solving a given problem, the only properties that are required are that the function to be optimized takes on values which can always be compared against each other (a totally ordered set of values such as the real numbers or the natural numbers) and that it allows for a step-by-step improvement of candidate solutions (i.e., the problem is not akin to finding the needle in the haystack). Hill climbing does not use any other properties of the function to be optimized and does not

organize the search for the optimum following a tree structure—or any other structure. Therefore, it requires little computer memory. Hill climbing starts with an initial candidate solution and iteratively aims at improving the current candidate solution by replacing it with any (or the best) neighbor solution which is better than the current one; when there are no more possible improvements, the search stops. The neighborhood can be considered either in the set over which the function is defined (a neighbor can be obtained through a slight modification of a number which is a component of the candidate solution) or in the set of computer representations of candidate solutions (a neighbor there is reached by flipping one bit).

While the procedure sketched above is very effective for any mono-modal function (informally, a function whose graph has only one hilltop), it may get stuck in local optima if the function is multi-modal. In the latter case, the graph of the function will also have a second-highest hill, a third highest one, etc.; one run of the hill-climbing procedure having the initial solution at the shoulder of the second-highest hill will find the second-highest hilltop (a local optimum), but then, it will get stuck there, since no improvement is possible anymore in the neighborhood. This is why for multi-modal functions iterated hill climbing is used instead of one-iteration hill climbing: The method is applied several times in a row, with different initial candidate solutions, thus increasing the chance that one run of the method will start at the foot of the hill which contains the global optimum.

Simulated Annealing The problem described above—optimization methods getting stuck in local optima—was actually impairing potential advances in optimization methods. A breakthrough has been the Metropolis algorithm (Metropolis et al. 1953). The new idea was to occasionally allow for candidate solutions which are worse than the current one to replace the current solution. This is compatible with the hill-climbing metaphor: Indeed, when one wanders through a hilly landscape aiming at reaching the top of the highest hill, he/she may have to occasionally climb down a hill in order to reach a higher one.

The idea of expanding the exploration capabilities of the optimization method at the expense of the quality of the current solution proved to be very productive. Nevertheless, a better idea is to also keep under some kind of control the ratio between the number of steps when the current solution is actually improved and the number of steps when the current solution is worsened. This is where simulated annealing comes into scene. Beings of nature have not been the only inspiration for problem-solving researchers; non-living-world processes are also a rich source for metaphors and simulations in problem solving. One celebrating example is annealing: Cooled gradually, a metal can gain most desirable physical properties (e.g., ductility and flexibility), while sudden cooling of a metal hardens it.

Kirckpatrick et al. (1983) proposed a simulation of annealing which uses a parameter (the temperature) for controlling the improvement/worsening ratio mentioned above: The lower the temperature, the fewer steps which worsen the current solution are allowed. Analogously to what happens in the physical–chemical process

of annealing, the temperature starts at a (relatively) high value and decreases at each iteration of the current-solution-changing process. Simulated annealing has been successfully applied to solve many discrete and continuous optimization problems, including optimal design.

The rest of this chapter and Chapter "Genetic Programming Techniques with Applications in the Oil and Gas Industry" present several population-based meta-heuristics: GAs and genetic programing, DE, and particle swarm optimization. We briefly introduce each of them in the following paragraphs. Four particular topics of interest, in particular for the meta-heuristics under discussion, are then briefly touched upon.

Many more meta-heuristics have been proposed and new ones continue to appear. Monographs and surveys on meta-heuristics such as Glover (1986); Talbi (2009); Voß (2001) give comprehensive insights into the topic. The *International Journal of Meta-heuristics* publishes both theoretical and application papers on methods including: neighborhood search algorithms, evolutionary algorithms, ant systems, particle swarms, variable neighborhood search, artificial neural networks, and artificial immune systems. Those interested in approaches to solving the *meta-problem* above may wish to read about *hyper-heuristics*—a term coined by Burke; a survey is provided in Burke et al. (2013).

1.2 What Will the Rest of This Chapter and the Next One Elaborate On?

We introduce briefly the main topics of the two chapters.

Genetic Algorithms Ingo Rechenberg, a professor with the Technical University of Berlin and a parent of evolution strategies, made a statement which supports the use of evolutionary techniques for problem solving: "Natural evolution is, or comprises, a very efficient optimization process, which, by simulation, can conduct to solving difficult optimization processes" Rechenberg (1973). The statement is empirically supported by many successful applications of evolutionary techniques for solving various optimization problems. The field of evolutionary computing now includes various techniques; the pioneering ones have been the GAs (Holland 1975), the evolution programs Fogel et al. (1966), and the evolution strategies Rechenberg (1973; Schwefel 1993). Excellent textbooks on GAs are widely used: Michalewicz (1992; Mitchell 1996), or a more general one, on evolutionary computing (Jong 2006).

As the title of the groundbreaking book by Holland suggests, *adaptation* has been the core idea that led to GAs; *self-adapting* techniques became ever since more and more popular. Trying to reach the optimum starting from initial guesses as candidate solutions, such techniques self-adapt their search using properties of the search space of (the instance of) the problem.

GAs simulate a few basic factors of natural evolution: mutation, crossover, and selection. The implementation of each of these simulated factors involves generating random numbers: like all evolutionary methods, GAs are non-deterministic. Adaptation, which is instrumental in natural evolution, is simulated by calculating values of a function (the environment) and, on this basis, making candidate solutions compete for survival for the next generation. The evolution of the population of solutions can be seen as a learning process where candidate solutions learn collectively.

More sophisticated variants of GAs simulate further factors of natural evolution, such as the integrated evolution of two species [coevolution (Hillis 1990) the host–parasite model].

One particular feature of GAs is that the whole computation process takes place in two dual spaces: the space of candidate solutions to the given problem (where the evaluation and the subsequent selection for survival take place the *phenotype*) and the space of the representations of such solutions (where genetic operators such as mutation and crossover are applied the *genotype*). This characteristic is also borrowed from natural evolution, where the genetic code and the actual being evolved from that code are instantiations of the two-space paradigm: In natural evolution, the genetic code is altered through mutations and through crossover between parents; subsequently, the being evolved from the genetic code is evaluated with respect to its adaptation to the environment.

The genetic code in GAs is actually the way candidate solutions are represented in the computer. The standard GAs (Michalewicz 1992) works with chromosomes (representations of candidate solutions) which are strings of bits. When applied to solve real-world problems, GAs evolved toward sophisticated representations of candidate solutions, including varying-length chromosomes and multi-dimensional chromosomes. One particular representation of candidate solutions has been groundbreaking: trees from graph theory.

Genetic Programing   emerged as a distinct area of GA. In his seminal book (Koza 1992), Koza uses a particular definition for the solution to a problem: A solution is a computer program which solves the problem. Adding to this the idea that such computer programs can be developed automatically, in particular through genetic programing, a flourishing field of research and applications emerged. As Poli et. al. put it, genetic programing automatically solves problems without requiring the user to know or specify the form or structure of the solution in advance (Poli et al. 2008).

A tree can be seen as representing a calculation, in particular, a computer program. In genetic programing, computer programs evolve in an automated manner through self-adaptation of a population of trees each tree representing a candidate program. Evaluation of candidate solutions is carried out using a set of instances of the problem to be solved for which the actual solutions are known beforehand. Specific operators have been introduced to cope with peculiarities of the evolution of trees as abstract representations.

Spectacular results have been obtained using genetic programing, including patentable inventions.

Differential Evolution   Since 1996, when it was publicly proposed by Price and Storn (1997), DE became a popular optimization technique. It is a population-based method designed for minimizing multi-dimensional real-valued functions through vector processing; the function needs not be continuous (let alone differentiable) and, even if it is differentiable, no information on the gradient is used.

DE follows the general steps of an evolutionary scheme: initialisation, applying specific operators (see the one described below), evaluation, and selection; this sequence being iterated from the second step until a halting condition is met. The basic operation in DE is to add the weighted difference of two vectors in the population to a third one. Thus, the candidate solutions learn from each other; the computation is a self-adapting process.

From its early days, DE proved to be a powerful optimization technique: It won the general-purpose algorithms competition in the First International Contest on Evolutionary Optimization, 1996 (at the IEEE International Conference on Evolutionary Computation). As was the case with other evolutionary techniques, DE evolved to incorporate new elements such as elitism or coevolution. Pareto-based approaches have been proposed for tackling multiple objective optimization problems using DE (Madavan 2002).

Particle Swarm Optimization *Collective intelligence* (Nguyen and Kowalczyk 2012) is a rich source of inspiration for designing meta-heuristics through simulation. Particularly, successful among such meta-heuristics are *Ant Colony Optimization* (Dorigo and Stützle 2004) and *Particle Swarm Optimization*.

The seminal paper for the latter meta-heuristic is (Kennedy and Eberhart 1995); a textbook dedicated to PSO is (Clerc 2006). Bird flocking or fish schooling can be considered as being the inspiring metaphors from nature. The core idea is that at each iteration, each *particle* (candidate solution) moves through the search space according to a (linear) combination of the particles current move, of the best personal previous position, and of the best previous position of the neighbors (what neighbors means, is a parameter of the procedure). This powerful combination of the backtracking flavor (keeping track somehow of the previous personal best) and collective learning (partly aiming at the regional/global previous best) makes PSO well suited for optimization problems with a moving optimum.

1.3 Short Comments on Four Transversal Issues

Parameter Control   A key element for the successful design of any meta-heuristic is a proper adjustment of its parameters. Suffices it to think of the number of candidate GAs one has to select from when designing a GA for a given problem:

Mutation rates and crossover rates can, at least theoretically, take on any value between 0 and 1; there are tens of choices for the population size; the selection procedure can be any of at least ten popular ones (new ones can be invented), etc. This makes a search space for properly designing a GAs for a given problem in the range of at least hundreds of thousands candidate GAs; of these, only a few will probably have a good performance and finding these among all possible GAs for that problem is a non-trivial task.

In the design phase of a meta-heuristic, parameters can be set by hand or automatically. For example, for GAs, a supervisor GAs have been proposed (Grefenstette 1986) which can be used for off-line improvement of the parameters of a given GAs such as the population size, the mutation, and crossover rates.

If one chooses to have dynamic parameter values during the run of the algorithm, this can be done automatically, for example, upon automatedly checking whether or not any change of the best-so-far solution happened during a given number of iterations.

Constraint Handling   When the problem to be solved belongs to the *constraint optimization* class, a major concern along the iterative solution-improving process is that of preserving the *feasability* of candidate solutions, i.e., keeping only solutions which satisfy all the constraints. The way a feasible solution is obtained in the first place is beyond the scope of this paragraph—this may happen, for example, by applying a heuristic which ends up with a feasible but, very likely, non-optimal solution. Subsequently, the iterative solution-improvement process successively changes the current solution; every such change may turn a current solution which is feasible into one which is not. When unfeasible solutions (candidate solutions which do not satisfy the problem constraints) are obtained, the optimization method should address this.

There are three main ways of tackling unfeasible solutions. A first approach is to penalize unfeasible solutions and otherwise let them continue to be part of the search process. In this way, an unfeasible solution becomes even less competitive than it actually is with respect to the search-for-the-optimum process (see *fitness function* in the GAs section of this chapter). A second approach is to *repair* the new solution in case it is unfeasible (repairing means changing the solution in such a way that it becomes feasible); the fact that repairing may have the same complexity as the original given problem makes this approach least recommendable. The best approach seems to be that of including the constraints (or at least some of them) into the representation of solutions. This idea is convincingly illustrated for numerical problems in Michalewicz (1992) where bit string representations are used: Any bit string is decoded into a feasible solution. This approach has the decisive advantage that there is no need to check whether or not candidate solutions obtained from existing ones are feasible. When including the problem constraints into the codification of candidate solutions, one actually uses *hybridisation with the problem*, which is mentioned in the next paragraph.

Hybridisation According to one of the definitions in Blum and Roli (2003), a basic idea of meta-heuristics in general is to combine two or more heuristics in one problem-tailored procedure and use it as a specific meta-heuristic. *Hybridisation* has even more subtle aspects. Hybridisation happens when inserting an element from one meta-heuristic into another meta-heuristic (e.g., using crossover, a defining operator for GA, in an evolution strategy which, in its standard form, uses only mutations). Another form of hybridisation could be called hybridisation with the problem: Problem-specific properties can be used for defining particular operators in a meta-heuristic. An example can be found in Michalewicz (1992): For the transportation problem, a feasible solution remains feasible after applying on it a certain transformation; this transformation is then used to define the mutation operator. An example of hybridisation is illustrated in this book, in the chapter on genetic programing.

Hybridisation is recommended, in general, for improving the problem-solving method. This could be called intended hybridisation, and it has proven its beneficial effects in countless successful applications.

There also exists an unintended hybridisation which one should be aware of. For example, when trying to optimize a Royal Road function (Mitchell et al. 1992), the search in a large plateau (while a substring of 8 bits does not yet contain only 1s) is akin to a blind search, even though we run a GA for solving the problem. Indeed, the probability field constructed for the selection has, for the whole plateau, equal probabilities, and consequently, the selection is not biased toward solutions closer to the optimum—it is rather a random selection. This way, the GA designed to solve the Royal Road problem is (unwillingly) hybridised with random search which takes over temporarily while walking the plateau.

Experiments Non-deterministic methods are used in a way which differs from that of deterministic ones. The latter will always provide the same output for a given input, while the former may give different results when run repeatedly with the same input. This behavior leads to the need of assessing the quality of a non-deterministic algorithm by repeatedly running it with the same input. Various statistics can be used —usually, the average of the respective best solutions and their standard deviation, over a number of runs. Therefore, the proper use of non-deterministic methods requires at least basic knowledge of probabilities and statistics, in particular *Experiment design*. Testing statistical hypothesis gives substance to the study of the performance of (non-deterministic) meta-heuristics.

1.4 Going into Practice: Two Running Examples

In order to illustrate the optimization process conducted within the methods described in this chapter, two optimization problems are formulated here. Sample code in R (including the output) invoking the algorithms under consideration is listed in the next sections in an attempt to familiarize the reader with some available, easy-to-use software.

The first optimization problem, known as Six Hump Camel Back, is commonly used as a benchmark function to assess the performance of optimization algorithms to which its multi-modal complex landscape imposes serious difficulties. It is formulated as a minimization problem over two continuous variables. The problem is defined as follows:

$$\text{Minimize} \quad f(x_1, x_2) = (4 - 2.1x_1^2 + x_1^4/3)x_1^2 + x_1x_2 + (-4 + 4x_2^2)x_2^2 \\ \text{where} \quad -3 \le x_1 \le 3, \\ -2 \le x_2 \le 2. \tag{1}$$

The landscape of the function is illustrated in Fig. 1 with the aid of perspective and contour plots in R.

Visible on the plots above, the function has six local minima and two global minima. The two global minima lie at locations $(x_1, x_2) = (-0.0898, 0.7126)$ and $(x_1, x_2) = (0.0898, -0.7126)$; the value returned at these locations corresponds to $f(x_1, x_2) = -1.0316$.

The R code defining the Six Hump Camel Back function is shown below.

```
> SixHump <- function (x1, x2)
{
   (4-2.1*x1^2+x1^4/3)*x1^2+x1*x2+(-4+4*x2^2)*x2^2
}
```

An equivalent function can be implemented in R using as argument a vector. This formulation is more appropriate for our goals because, this general form which does not impose restrictions on the size of the input, can be further called by other R routines implementing the meta-heuristics presented in this chapter.

```
> SixHumpV <- function (x)
{
   (4-2.1*x[1]^2+x[1]^4/3)*x[1]^2+x[1]*x[2]+(-4+4*x[2]^2)*x[2]^2
}
```

We also illustrate the use of meta-heuristics on a constrained optimization problem with discrete variables, frequently arising in the oil and gas industry: portfolio selection. While this problem may be found under various formulations, we tackle here the variant presented in Shakhsi-Niaei et al. (2013). Given a firm with a budget b, n projects, with the net value of the ith project denoted by f_i and the cost of the ith project denoted by c_i, one must find the combination of projects that maximizes the total utility for the firm, as computed in Eq. 2:

$$\text{Maximize } z = \sum_{i=1}^{n} f_i x_i, \tag{2}$$

(a)

(b)
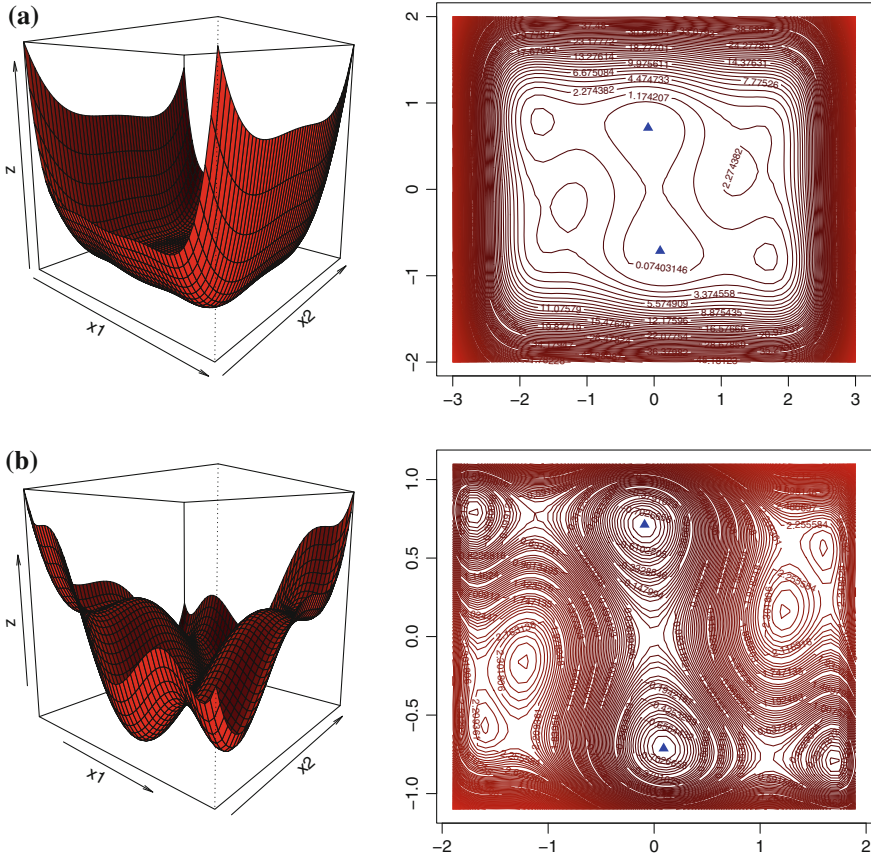
Fig. 1 Perspective and contour plots for Six Hump Camel Back: **a** for the entire domain of definition: $x_1 \in [-3, 3]$, $x_2 \in [-2, 2]$, **b** restricted to $x_1 \in [-1.9, 1.9]$, $x_2 \in [-1.1, 1.1]$. The two global optima are illustrated as *blue triangles* at locations $(x_1, x_2) = (-0.0898, 0.7126)$ and $(x_1, x_2) = (0.0898, -0.7126)$

$$\text{Subject to: } \sum_{i=1}^{n} c_i x_i \le b, \tag{3}$$

$$x_i \in 0, 1, \quad i = \overline{1, n}. \tag{4}$$

$$x_1 + x_2 \le 1, \tag{5}$$

$$x_5 + x_3 \le 1. \tag{6}$$

$$x_5 + x_3 + x_4 \le 2. \tag{7}$$

The variables x_i represent the decision to select project i ($x_i = 0$ means the project is not selected, whereas $x_i = 1$ means the project gets selected for implementation)—

constraint expressed by Eq. 4. The total budget of the firm must not be exceeded by the total costs of the projects selected (Eq. 3). Other constraints may be imposed on the problem (especially in a real-world context), such as Eq. 5 expresses the condition that either project 1 or project 2 gets implemented; Eq. 6 expresses the condition that either project 3 or project 5 gets implemented; Eq. 7 expresses the condition that at most 2 out of the 3 projects (3, 4, and 5) may get implemented.

2 Evolutionary Algorithms

Evolutionary algorithms (EAs) are simplified computational models of the evolutionary processes that occur in nature. They are search methods implementing principles of natural selection and genetics. Parts of this section follow closely the text in (Breaban 2011).

2.1 Terminology

Evolutionary algorithms use a vocabulary borrowed from genetics. They simulate the evolution across a sequence of *generations* (iterations within an iterative process) of a *population* (set) of candidate solutions. A candidate solution is internally represented as a string of *genes* and is called *chromosome* or *individual*. The position of a gene in a chromosome is called *locus,* and all the possible values for the gene form the set of *alleles* of the respective gene. The internal representation (encoding) of a candidate solution in an evolutionary algorithm form the *genotype*; this information is processed by the evolutionary algorithm. Each chromosome corresponds to a candidate solution in the search space of the problem which represents its *phenotype*. A decoding function is necessary to translate the genotype into phenotype. If the search space is finite, it is desirable that this function should satisfy the bijection property in order to avoid redundancy in chromosomes encoding (which would slow down the convergence) and to ensure the coverage of the entire search space.

The population maintained by an evolutionary algorithm evolves with the aid of genetic operators that simulate the fundamental elements in genetics: *Mutation* consists in a random perturbation of a gene, while *crossover* aims at exchanging genetic information among several chromosomes. The chromosome subjected to a genetic operator is called *parent* and the resulted chromosome is called *offspring*.

A process called *selection* involving some degree of randomness selects the individuals to breed and create offsprings, mainly based on individual merit. The individual merit is measured using a **fitness function** which quantifies how fitted the candidate solution encoded by the chromosome is for the problem being solved. The fitness function is formulated based on the mathematical function to be optimized.

The solution returned by an evolutionary algorithm is usually the most fitted chromosome in the last generation.

2.2 Directions in Evolutionary Algorithms

First efforts to develop computational models of evolutionary systems date back to 1950s (Bremermann 1958; Fraser 1957). Several distinct interpretations, which are widely used nowadays, were independently developed later. The main differences between these classes of evolutionary algorithms consist in solution encoding, operators implementation, and selection schemes.

Evolutionary programing crystallized in 1963 in the USA at San Diego University, when Fogel (1966) generated simple programs as simple finite-state machines; this technique was developed further by his son David Fogel. A random mutation operator was applied on state transition diagrams, and the best chromosome was selected for survival.

Evolutionary strategies (ES) were introduced in 1960s when Hans-Paul Schwefel and Ingo Rechenberg, working on a problem from mechanics involving shape optimization, designed a new optimization technique because existing mathematical methods were unable to provide a solution. The first ES algorithm was initially proposed by Schwefel in 1965 and developed further by Rechenberg (1973). Their method was designed to solve optimization problems with continuous variables; it used one candidate solution and applied random mutations followed by the selection of the fittest. ES were later strongly promoted by Back (1996) who incorporated the idea of population of solutions.

GAs were developed by John Henry Holland in 1973 after years of study of the idea of simulating the natural evolution. These algorithms model the genetic inheritance and the Darwinian competition for survival. GAs are described in more detail in Sect. 2.3.

Genetic programing is a specialized form of a GA. The specialization consists in manipulating a very specific type of encoding and, consequently, in using modified versions of the genetic operators. GP was introduced by Koza in 1992 in an attempt to perform automatic programing. GP manipulates directly phenotypes, which are computer programs (hierarchical structures) expressed as trees. It is currently intensively used to solve symbolic regression problems. Genetic programing and one important variation—gene expression programing—are described in Chapter "Genetic Programming Techniques with Applications in the Oil and Gas Industry" of this book.

DE (Storn and Price 1997) is a more recent class of evolutionary algorithms whose operators are specifically designed for numerical optimization. DE is described in detail in Sect. 2.4.

An in-depth analysis under a unified view of these distinct directions in evolutionary algorithms is presented in De Jong (2006).

Fig. 2 A generic genetic algorithm

```
t := 0
Initialize P_0
Evaluate P_0
while halting condition not met do
        t := t + 1
        select P_t from P_{t-1}
        apply crossover and mutation in P_t
        evaluate P_t
end while
```

2.3 Genetic Algorithms

GAs (Holland 1998) are the most well known and the most intensively used class of evolutionary algorithms.

A GA performs a multi-dimensional search by means of a population of candidate solutions which exchange information and evolve during an iterative process. The process is illustrated by the pseudo-code in Fig. 2.

In order to solve a problem with a GA, one must define the following elements:

- an encoding for candidate solutions (the genotype);
- an initialization procedure to generate the initial population of candidate solutions;
- a fitness function which defines the environment and measures the quality of the candidate solutions;
- a selection scheme;
- genetic operators (mutation and crossover);
- numerical parameters.

The encoding is considered to be the main factor determining the success or failure of a GA.

The standard **encoding** in GAs consists in binary strings of fixed length. The main advantage of this encoding is offered by the existence of a theoretical model (the Schema theorem) explaining the search process until convergence. Another advantage shown by Holland is the high implicit parallelism in the GA. A widely used extension to the binary encoding is gray coding.

Unfortunately, for many problems, this encoding is not a natural one and it is difficult to be adapted. However, GAs themselves evolved and the encoding extended to strings of integer and real numbers, permutations, trees, and multi-dimensional structures. Decoding the chromosome onto a candidate solution to the problem sometimes necessitates problem-specific heuristics.

Important factors that need to be analyzed with regard to the encoding are the size of the search space induced by a representation and the coverage of the phenotype space: Whether the phenotype space is entirely covered and/or reachable, whether the mapping from genotype to phenotype is injective, or "degenerate," and

whether particular (groups of) phenotypes are over-represented (Radcliffe et al. 1995). Also, the "heritability" and "locality" of the representation under crossover and mutation need to be studied (Raidl and Gottlieb 2005).

The **initialization** of the population is usually performed randomly. There exist approaches which make use of greedy strategies to construct some initial good solutions or other specific methods depending on the problem.

The **fitness function** is constructed based on the mathematical function to be optimized. For more complex problems, the fitness function may involve very complex computations and increase the intrinsic polynomial complexity of the GA.

Several probabilistic procedures based on the fitness distribution in population can be used to select the individuals to survive in the next generations and produce offsprings; this phase of the algorithm is known as **selection for variation**. All these procedures encourage to some degree the survival of the fittest individuals, allowing at the same time that the worst adapted individual survive and contribute with local information (short-length substrings) to the structure of the optimal solution. The most essential feature which differentiates them is the selection pressure: the degree to which the better individuals are favored; the higher the selection pressure, the more the better individuals are favored. The selection pressure has a great impact on the diversity in population and consequently on the convergence of GAs. If the selection pressure is too high, the algorithm will suffer from insufficient exploration of the search space and premature convergence occurs, resulting in sub-optimal solutions. On the contrary, if the selection pressure is too low, the algorithm will unnecessarily take longer time to reach the optimal solution. Various selection schemes were proposed and studied from this perspective. They can be grouped into two classes: proportionate-based selection and ordinal-based selection. Proportionate-based selection takes into account the absolute values of the fitness. The most known procedures in this class are as follows: roulette wheel (Holland 1975) and stochastic universal sampling (Baker 1987).

Because of its wide use and popularity among all GAs flavors, we describe, in the following, roulette wheel selection. For this procedure, each individual is assigned a probability of being selected proportional with its fitness value. The sum of these probability values over the set of all the individuals in a generation is 1. Let f_i be the fitness of the ith individual of the current population, then p_i is the probability of the individual for being selected:

$$p_i = \frac{f_i}{\sum\limits_{j=1}^{N} f_j},$$

where N is the number of individuals in the population (see, for a simple example, Fig. 3 which assumes a population of 5 individuals). On each application of the selection scheme, a random number is generated $r \in [0, 1)$ and the individual i with the highest cumulative frequency smaller than this random r is selected to survive to the next generation:
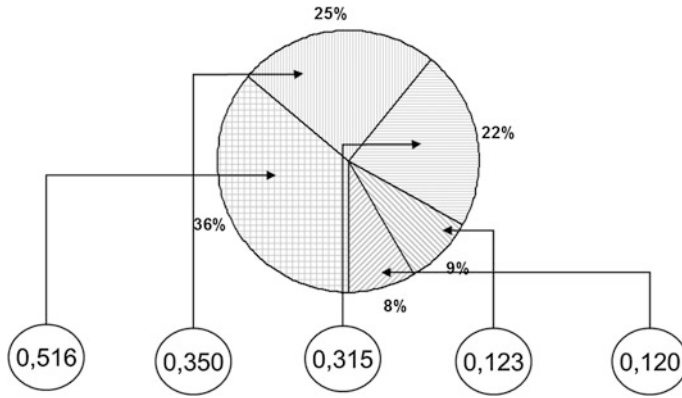
Fig. 3 Fitness values in a population of 5 individuals. The *bottom row* contains the fitness values of the individuals. Their associated probabilities are the labels of the circular sectors

$$i = k = \overset{\min}{1\ldots n}\{k| \sum_{j=1}^{k} \geq r\}.$$

Ordinal-based selection takes into account only the relative order of individuals according to their fitness values. The most used procedures of this kind are the linear ranking selection (Baker 1985) and the tournament selection (Goldberg 1989).

New individuals are created in population with the aid of two genetic operators: crossover and mutation. The classical **crossover** operator aims at exchanging genetic material between two chromosomes in two steps: A locus is chosen randomly to play the role of a cut point and splits each of the two chromosomes in two segments; then, two new chromosomes are generated by merging the first segment from the first chromosome with the second segment from the second chromosome and vice versa. This operator is called in literature one-point crossover and is illustrated in Fig. 4. Generalizations exist to three or more cut points. Uniform crossover builds sequentially the offspring by copying at each locus the allele randomly chosen from one of the two parents.

Various constraints imposed by real-world problems led to various encodings for candidate solutions; these problem-specific encodings subsequently necessitate the redefinition of crossover. Thus, algebraic operators are implied for the case of numerical optimization with real encoding; an impressive number of papers focused on permutation-based encodings proposing various operators and performing comparative studies. It is now a common procedure to wrap a problem-specific heuristic within the crossover operator in Ionita et al. (2006), the authors propose new operators for constraint satisfaction; (Luchian et al. 1994) presents new operators in the context of clustering]. Crossover in GAs stands at the moment for any procedure which combines the information encoded within two or several chromosomes to create new and hopefully better individuals.
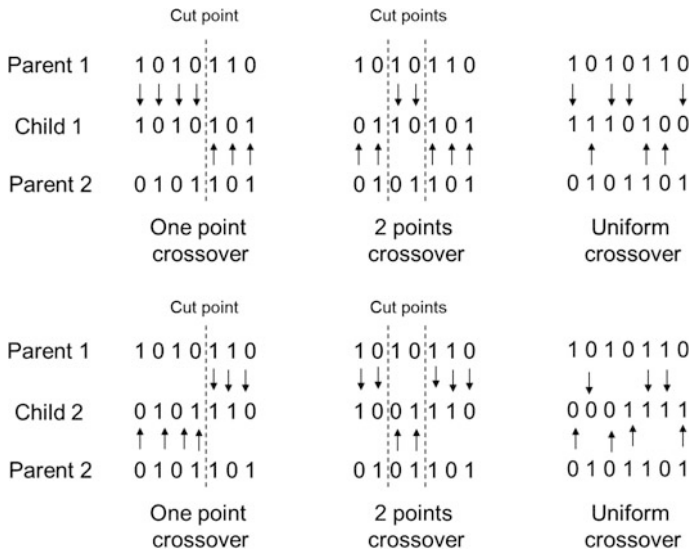
Fig. 4 Crossover operators in bit string GA

Mutation is a unary operator designed to introduce variability in population. In the case of binary GAs, the mutation operator modifies each gene (from 0 to 1 or from 1 to 0) with a given probability. As in the case of crossover, mutation takes various forms depending on the problem and the encoding used (see Fig. 5 for examples of how mutation works for different chromosome representations).

When designing a GA, decisions have to be made with regard to several **parameters**: population size, crossover and mutation rate, and a halting criterion. Except some general considerations (i.e., high mutation rate in first iterations, decreasing during the run, combined with a complementary evolution for crossover), finding the optimum parameter values comes more to empiricism than to abstract studies.

In the following, we illustrate the search process conducted by a GA using the package called "GA" (Scrucca 2013) in R to minimize the Six Hump Camel Back function, previously defined in Sect. 1.4.
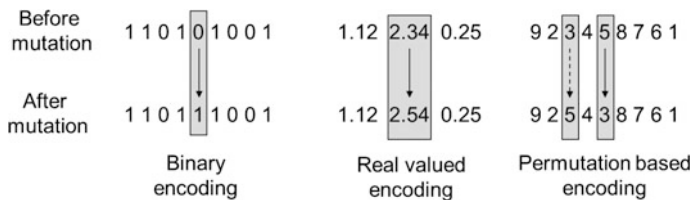

Fig. 5 The behavior of the mutation operator for different encodings

Because this is a problem with a continuous bi-dimensional search space, a real encoding and arithmetical operators are a natural choice. Moreover, empirical studies have reported that these settings obtain better performance compared to natural encoding and standard operators in the case of numerical optimization problems. The initialization scheme consists in randomly generating points (candidate solutions, chromosomes) in the bi-dimensional search space defined by the problem. We have to define further the fitness function that should be used to measure the quality of the chromosomes. Naturally, this is based on the objective function of our problem, but requires some minimal modifications: the GAs necessitate that the fitness function is designed for maximization: The higher the fitness value is, the better the candidate solution for our problem is. Because the problem we tackle is defined for minimization, low values of our objective function (previously defined in R as *SixHumpV*) correspond to better solutions, while high values to worse ones. Therefore, we need to build a new function playing as fitness in the GA, simply by multiplying our objective function with (-1):

```
SixHumpMax <- function(x)
+ {
+ -SixHumpV(x)
+ }
```

The lines of code below call the **ga** function to execute a GA which maximizes our newly defined function with a population of 20 chromosomes using real encoding and arithmetic operators for 50 iterations:

```
> library("GA")
> GA.sols <- ga(type = "real-valued", fitness = SixHumpMax,
+   min = c(-3, -2), max = c(3, 2), maxiter=50, popSize=20)
Iter = 1  | Mean = -20.10513  | Best = 0.3900806
Iter = 2  | Mean = -8.679598  | Best = 0.3900806
Iter = 3  | Mean = -1.909435  | Best = 0.3900806
Iter = 4  | Mean = -0.7739577 | Best = 0.521566
Iter = 5  | Mean = -0.4207289 | Best = 0.521566
...
Iter = 50 | Mean = 0.9275536  | Best = 1.020383
```

During its execution, the **ga** function prints at each iteration the mean of the fitness in population and the best fitness value. To show the final results, we call the **summary** function:

```
> summary(GA.sols)
+----------------------------------+
|          Genetic Algorithm       |
+----------------------------------+

GA settings:
Type                  = real-valued
Population size       = 20
Number of generations = 50
Elitism               = 1
Crossover probability = 0.8
Mutation probability  = 0.1
Search domain
     x1 x2
Min -3 -2
Max  3  2

GA results:

Iterations            = 50
Fitness function value = 1.020383
Solution              =
              x1            x2
[1,] -0.1262185 0.6870156
```
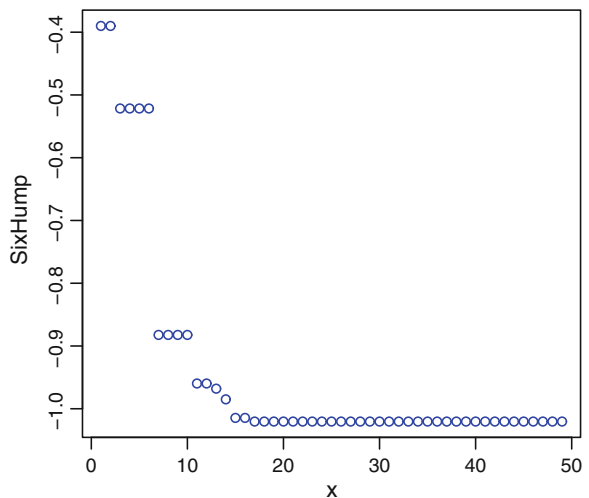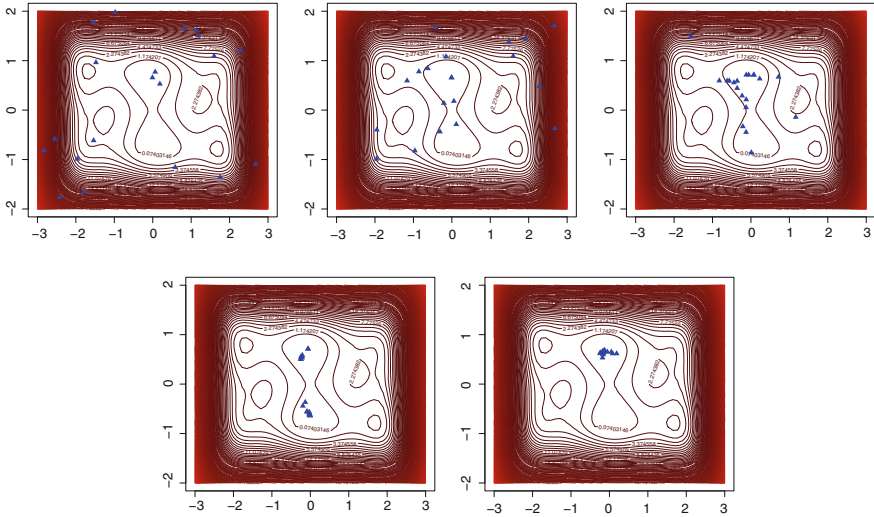
The best solution obtained over 50 iterations corresponds to Six Hump $(-0.1262185, 0.6870156) = -1.020383$. The evolution of the best value of the objective function in population during the run is illustrated in Fig. 6.

Fig. 6 The evolution of the best objective value in one run of the GA

Fig. 7 The evolution of the population in GA during one run of the algorithm: the distribution of the candidate solutions at iterations 1, 2, 5, 10, and 15

Figure 7 illustrates the distribution of the individuals in population during one run of the GA, at iterations 1, 2, 5, 10, and 50. The GA shows a very quick convergence toward the regions containing the global minima. The evolution of the fitness for the run illustrated here shows that the GA is able to locate in only a few number of iterations the promising area in the search space due to its good exploration abilities. However, by comparing the final solution to the minimum of the objective function (-1.020383 vs. -1.0316), we may conclude that in this run, the GA is deficient at exploitation: Even if very close to the global optima, starting at iteration number 17, the algorithm stopped improving the best solution achieved so far.

By illustrating only one run of the GA, a general conclusion on its convergence cannot be drawn on this basis due to the stochastic nature of the algorithm. To study its performance, 30 runs are performed with the same settings and for each run, the objective value corresponding to the solution returned is collected. In this manner, we obtain a sample of 30 values with mean -1.030361—which is closer to the optimum than the particular run reported previously, and standard deviation 0.0037 —which indicates that the algorithm is stable, returning each time solutions very close to the optimum. The confidence interval for the mean supports these conclusions: The mean of the objective values returned by the GA is less than -1.028960 (we are interested in the minimum) with probability 0.95. In the code below, "fitness" is a vector with 30 values corresponding to the objective values returned in 30 runs:

```
> t.test(fitness)

        One Sample t-test

data:  fitness
t = -1503.688, df = 29, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 -1.031762 -1.028960
sample estimates:
mean of x
-1.030361
```

Although the reported results are satisfactory, the GAs are usually enhanced in practice by hybridizing them with local search algorithms.

With a standard binary encoding, GAs are the most appropriate candidates when attempting to solve the portfolio selection problem by means of meta-heuristics.

In order to illustrate such an approach, we consider the problem defined in Sect. 1.1 with the following instantiation: the number of projects $n = 6$, the budget of the firm $b = 1000$, and the costs and the utilities of the projects as in Table 1. An optimal solution to this problem involves the selection of projects 1, 4, 5, and 6; it has total cost 850 and utility 1700.

One way to deal, within a GA, with the constraints imposed by the problem, is to encourage the search in the feasible region of the search space by penalizing the unfeasible candidate solutions. Under this approach, any solution that violates a constraint gets a lower fitness. Identifying the most appropriate scheme to penalize solutions is, by itself, an optimization problem. The code below implements one possible fitness function for our problem:

```
> portfolio <- function(x){
+   cost <- c(250,350,100,200,300,100)
+   utility <- c(500,400,150,300,600,300)
+   totalUtility <- sum (utility*x)
+   totalCost <- sum (cost*x)
+   penalty <- 0
+   if (totalCost > 1000)
+     penalty <- totalCost #penalty for exceeding the budget
+   p=sum(cost)
+   if (x[1]+x[2] > 1) penalty <- penalty+p #violating constraint 5)
+   if (x[3]+x[5] > 1) penalty <- penalty+p #violating constraint 6)
+   if (x[3]+x[4]+x[5] > 2) penalty <- penalty+p #violating constraint 7)
+   totalUtility - penalty
+ }
```

Table 1 Cost and utility of projects

Project	1	2	3	4	5	6
Cost	250	350	100	200	300	100
Utility	500	400	150	300	600	300

A GA with binary encoding is called to solve this problem instance:

```
> GA <- ga(type = "binary", fitness = portfolio, nBits = 6,
+  maxiter = 50, popSize = 10)
Iter = 1  | Mean = 270   | Best = 1300
Iter = 2  | Mean = 850   | Best = 1400
Iter = 3  | Mean = 1225  | Best = 1700
Iter = 4  | Mean = 1160  | Best = 1700
I...
Iter = 49  | Mean = 832.5  | Best = 1700

> summary(GA)
+---------------------------------+
|          Genetic Algorithm      |
+---------------------------------+

GA settings:
Type                  =  binary
Population size       =  20
Number of generations =  50
Elitism               =  1
Crossover probability =  0.8
Mutation probability  =  0.1

GA results:
Iterations            = 50
Fitness function value = 1700
Solution              =
     x1 x2 x3 x4 x5 x6
[1,]  1  0  0  1  1  1
```

2.4 Differential Evolution

Adhering by design to the area of evolutionary algorithms, but targeting in particular the field of numerical optimization, a method called DE was developed by Ken Price and Rainer Storn during 1994–1996 (Storn and Price 1997). The results

Fig. 8 The differential
evolution algorithm

1. $t := 0$
2. Initialize population $P_0 = \{x_1^{(0)}, x_2^{(0)}, ..., x_m^{(0)}\}$ of size m
3. Evaluate P_0
4. **while** *halting condition not met* **do**
5. $t := t + 1$
6.   **for** i=1 **to** m **do**
7. $y_i = \text{generateMutant}(P_{t-1})$
8. $z_i = \text{crossover}(x_i^{(t-1)}, y_i)$
9.   Evaluate z_i
10.   **if** z_i is better than $x_i^{(t-1)}$ **then**
11. $x_i^{(t)} = z_i$
12.   **else**
13. $x_i^{(t)} = x_i^{(t-1)}$
14.   **end if**
15.   **end for**
16. **end while**

in their seminal paper show that DE outperforms GAs in numerical optimization and this hypothesis was subsequently confirmed in competitions dedicated to real-valued function minimization.

DE makes use of the same terminology as GAs: A population of candidate solutions evolves by means of selection, mutation, and crossover. The differences occur at several levels: the encoding of the candidate solutions, the definition of the genetic operators, and the selection scheme.

Designed for numerical optimization, the internal **encoding** of the candidate solution (the genotype) is identical to the phenotype: A string of real values that correspond to the decision variables defined by the problem.

The **selection for variation** is replaced in DE by a simple pass through the entire population: Each chromosome is participating in the variation phase to create a new offspring by means of genetic operators. However, DE implements selection at replacement: The offspring is introduced in the new population only if it is better than its parent with regard to the fitness function. The pseudo-code of the DE algorithm is illustrated in Fig. 8.

There are several versions of the **mutation** operator (line 7 of the algorithm). However, they all share a mechanism that is a distinctive feature of DE within the EA framework: The perturbation term is obtained as the difference between some randomly selected chromosomes. This perturbation mechanism, particular to DE, suggestively gives the name of this method. The general formula creating one mutant y_i at time t is given below:

$$y_i = \lambda x_*^{(t-1)} + (1 - \lambda)x_{I_i}^{(t-1)} + \sum_{l=1}^{L} F_l(x_{J_{il}}^{(t-1)} - x_{K_{il}}^{(t-1)}) \tag{8}$$

where λ is a numerical value in range [0,1] controlling the influence of the best element in the current population, which is $x_*^{(t-1)}$. $x_{I_i}^{(t-1)}$ is a chromosome from the

current population, chosen at random ($I_i \in \{1, 2, \ldots m\}$). $L > = 1$ is an integer value specifying the number of pairs of chromosomes of the form $(x_{J_{il}}^{(t-1)}, x_{K_{il}}^{(t-1)})$ randomly chosen from the current population ($J_{il}, K_{il} \in \{1, 2, \ldots m\}$, $J_{il} \neq K_{il}$) and which are used in the perturbation mechanism. $F_l > 0, l = \overline{1 \ldots m}$ are scaling factors decisive for the influence of each difference.

Different settings of the numerical parameters λ and L lead to distinct DE algorithms. In order to specify, in a concise manner, the DE variant, a simple notation, was introduced based on three variables: DE/a/L/c where a depends on the value of λ, L is the number of vector differences used, and c is the type of crossover. The most popular versions of the DE algorithm are DE/best/1/$*$ and DE/rand/1/$*$. Both versions correspond to the case when only one difference is used to compute the mutant. The first case corresponds to $\lambda = 1$, respectively, to

$$y_i = x_*^{(t-1)} + F(x_{J_i}^{(t-1)} - x_{K_i}^{(t-1)}) \tag{9}$$

while the second case corresponds to $\lambda = 0$, respectively, to

$$y_i = x_{I_i}^{(t-1)} + F(x_{J_i}^{(t-1)} - x_{K_i}^{(t-1)}) \tag{10}$$

It must be noted that the mutation mechanism described above does not alter the current/selected chromosome x_i. It is the role of crossover to build an offspring of the current chromosome, by combining its genetic material with the one encoded by the mutant chromosome. From this perspective, DE is not entirely compliant with the general specifications of the two genetic operators.

Two versions of **crossover** are proposed in DE. A first one, called binomial crossover, is similar to the uniform crossover in GAs: It is a binary operator that mixes the components of the two chromosomes based on a given probability CR:

$$z_{i,d} = \begin{cases} y_{i,d} & \text{if } r_d < \text{CR or } d = d_0 \\ x_{i,d} & \text{otherwise} \end{cases} \quad d = \overline{1 \ldots D} \tag{11}$$

where r_d is a random number uniformly distributed in [0,1] and $d_0 \in [1, D]$ is a random position in the chromosome guaranteeing that the offspring contains at least one element from the mutant. D denotes the dimensionality of the problem, i.e., the length of the string representing a chromosome.

The second variant of the crossover operator is called exponential crossover and can be expressed by the following formulation:

$$z_{i,d} = \begin{cases} y_{i,d} & \text{for } d \in \Theta \\ x_{i,d} & \text{otherwise} \end{cases} \tag{12}$$

where Θ is a series of size at most D of consecutive circular numbers in range 1,2, … D, starting with a value d_0 and continuing with $(d_0 + 1) \div D$, $(d_0 + 2) \div D, \ldots$, $(d_0 + k) \div D$ where $a \div b$ expresses the modulus operator returning the remainder

of the division of a to b; k is the first trial that satisfies that a random uniformly generated number in [0,1] is higher than CR, thus following a truncated geometric distribution. For example, considering $d_0 = 6$ and $D = 10$, Θ could be the series 6, 7, 8 or 6, 7, 8, 9, 10, 1, 2, depending on the parameter CR; these two examples clearly illustrate the similarity of the exponential crossover in DE with the 2-point crossover in GAs.

In both versions of the crossover operator, CR is a parameter deciding the influence of the mutant on the structure of the offspring. A theoretical analysis of the two crossover variants and their influence on the sensitivity of DE to different values of CR are presented in Zaharie (2007).

An **elitist replacement strategy** guarantees survival of the fittest chromosome among the parent and the offspring.

To simulate a run of the DE algorithm on our minimization problem, we use the R package called DEoptim (Mullen et al. 2011).[1] The following code calls the DEoptim function which executes the DE/rand/1/bin algorithm (the variant implementing mutation based on a random candidate and one difference, and binary crossover) to minimize the SixHump function with a population consisting of 20 candidate solutions over 50 iterations; with the trace parameter set on TRUE, the best candidate solution (its value for the objective function and its components) in each iteration is shown during the run:

```
> library("DEoptim")
> DE.sols <- DEoptim(SixHumpV, lower = c(-3, -2), upper = c(3, 2),
+  control = list(strategy = 1, NP=20, itermax=50, storepopfrom = 1,
+  trace = TRUE))
Iteration: 1 bestvalit: -0.343676 bestmemit:    0.424858   -0.515384
Iteration: 2 bestvalit: -0.343676 bestmemit:    0.424858   -0.515384
Iteration: 3 bestvalit: -0.343676 bestmemit:    0.424858   -0.515384
Iteration: 4 bestvalit: -0.722848 bestmemit:   -0.090842    0.885970
Iteration: 5 bestvalit: -0.811161 bestmemit:    0.138414    0.742059
...
```

The performance of DE is highly dependent on the values of the numerical parameters. The authors of DE recommend setting CR to 0.9 and selecting F from the interval [0.5, 1.0]. The run illustrated here uses the default values in DEoptim: CR = 0.9 and $F = 0.8$.

The following lines of code list the best solution in the last iteration and output two plots: One representing the evolution of the best value of the objective function (the minimum) in the population and one representing the distribution of the candidate solutions during the run. The resulting plots are illustrated in Fig. 9.

---

[1]The package can be freely downloaded from http://cran.r-project.org/web/packages/DEoptim/index.html.
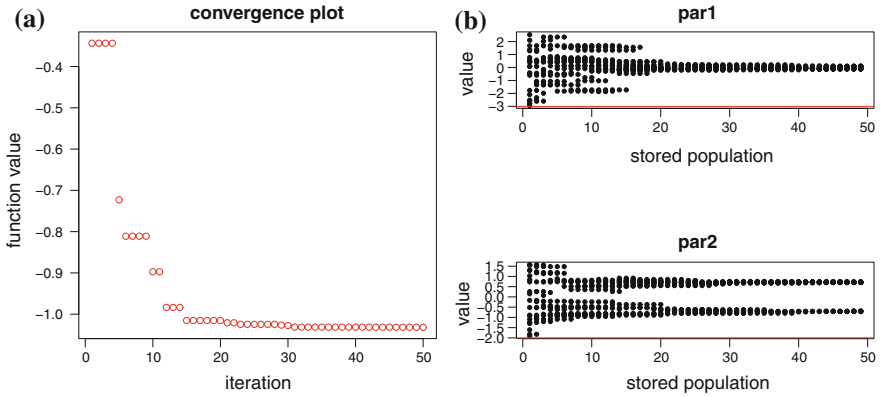
Fig. 9 The evolution of the population in DE during one run of the algorithm: **a** the evolution of the best fitness value in population and **b** the distribution of the candidate solutions (the genotype)

```
> DE.sols$optim
$bestmem
         par1          par2
 0.08984226 -0.71265649

$bestval
[1] -1.031628
...
> plot(DE.sols, plot.type = "bestvalit", col="red", pch=1)
> plot(DE.sols, plot.type = "storepop")
```

Figure 9 clearly illustrates the convergence toward the optimal solution in DE. In our run, the optimum is found after 31 iterations, as indicated by Fig. 9a. The diversity in population decreases significantly during the run according to Fig. 9b which presents in two distinct plots the distribution of the values in each iteration for each parameter of the objective function. This plot indicates an interesting behavior: convergence toward two distinct regions in the search space.

In order to get more insight into the dynamics of the population within DE, Fig. 10 illustrates the candidate solutions in the population at distinct moments during the run distributed over the contour plot illustrating the landscape of the objective function. The series (a) of plots show the distribution of the candidate solutions at iterations 1, 5, 10, and 15. The series (b) offers a zoomed-in perspective of the landscape (restricted to $x_1 \in [-1.9, 1.9]$ and $x_2 \in [-1.1, 1.1]$) showing the distribution of the candidate solutions at iterations 15, 20, 30, and 50. In the first iteration of the algorithm, the population is spread at random in the search space. At iteration number 10 (Fig. 10a)-3rd plot), groups of individuals were formed around local and global optima. Toward the end of our run, all the candidate solutions migrate in the regions corresponding to the two global optima.

(a)

(b)
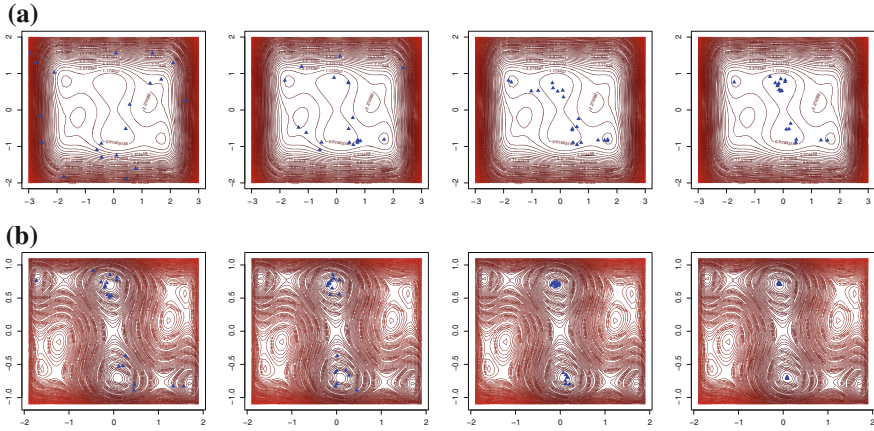
Fig. 10 The evolution of the population in DE during one run of the algorithm: **a** the distribution of the candidate solutions at iterations 1, 5, 10, and 15 and **b** a zoomed-in landscape showing the distribution of the candidate solutions at iterations 15, 20, 30, and 50

The mean of the objective values after 30 runs is −1.031615, with a standard deviation of 3.74e−05.

2.5 Extensions of EAs for Multi-modal and Multi-objective Problems

Variations were brought to the classical EAs not only at the encoding and operators level. In order to face the challenges imposed by real-world problems, modifications are also recorded in the general scheme of the algorithms.

EAs are generally preferred to trajectory-based meta-heuristics (i.e., hill climbing, simulated annealing, Tabu Search) in **multi-modal environments**, mostly due to their increased exploration capabilities. However, a standard EA still can be trapped in a local optimum due to premature attraction of the entire population into its basin of attraction. Therefore, the main concern of EAs for multi-modal optimization is to maintain diversity for a longer time in order to detect multiple (local) optima. To discover the global optima, the EA must be able to intensify the search in several promising regions and eventually encourage simultaneous convergence toward several local optima. This strategy is called *niching*: The algorithm forces the population to preserve subpopulations, each subpopulation corresponding to a niche in the search space, and different niches represent different (local) optimal regions.

Several strategies exist in the literature to introduce niching capabilities into evolutionary algorithms. Deb and Goldberg (1989) propose *fitness sharing*: The

fitness of each individual is modified by taking into account the number and fitness of its closely ranged individuals. This strategy determine the number of individuals in the attraction basin of an optimum to be dependent on the height of that peak.

Another widely used strategy is to arrange the candidate solutions into groups of individuals that can only interact between themselves. The *island model* evolves independently several populations of candidate solutions; after a number of generations, individuals in neighboring populations migrates between the islands (Whitley et al. 1998).

There are techniques, which divide the population, based on the distances between individuals (the so-called radii-based multi-modal search GAs). Genetic chromodynamics (Dumitrescu 2000) introduces a set of restrictions with regard to the way selection is applied or the way recombination takes place. A merging operator is introduced which merges very similar individuals after perturbation takes place. In Stoean et al. (2010), best successive local individuals are conserved, while sub-populations are topological separated.

De Jong introduced a new scheme of inserting the descendants into the population, called the *crowding* method (Kenneth 1975). To preserve diversity, the offspring replace only similar individuals in the population.

A field of intensive research within the evolutionary computation (EC) community is **multi-objective optimization**. Most real-world problems necessitate the optimization of several, often conflicting objectives. Population-based optimization methods offer an elegant and very efficient approach to this kind of problems: With small modifications of the basic algorithmic scheme, they are able to offer an approximation of the Pareto optimal solution set. While moving from one Pareto solution to another, there is always a certain amount of sacrifice in one objective(s) to achieve a certain amount of gain in the other(s). Pareto optimal solution sets are often preferred to single solutions in practice, because the trade-off between objectives can be analyzed and optimal decisions can be made on the specific problem instance.

Zitzler et al. (2000) formulate three goals to be achieved by multi-objective search algorithms:

- the Pareto solution set should be as close as possible to the true Pareto front,
- the Pareto solution set should be uniformly distributed and diverse over of the Pareto front in order to provide the decision maker a true picture of trade-offs,
- the set of solutions should capture the whole spectrum of the Pareto front. This requires investigating solutions at the extreme ends of the objective function space.

GAs have been the most popular heuristic approach to multi-objective design and optimization problems mostly because of their ability to simultaneously search different regions of a solution space and find a diverse set of solutions. The

crossover operator may exploit structures of good solutions with respect to different objectives to create new non-dominated solutions in unexplored parts of the Pareto front. In addition, most multi-objective GAs do not require the user to prioritize, scale, or weigh objectives. There are many variations of multi-objective GAs in the literature and several comparative studies. As in multi-modal environments, the main concern in multi-objective GAs optimization is to maintain diversity throughout the search in order to cover the whole Pareto front. Konak et al. (2006) provide a survey on the most known multi-objective GAs, describing common techniques used in multi-objective GA to attain the three above-mentioned goals.

3 Swarm Intelligence

Swarm intelligence (SI) is a computational paradigm inspired from the collective behavior in auto-organized decentralized systems. It stipulates that problem solving can emerge at the level of a collection of agents which are not aware of the problem itself, but collective interactions lead to the solution. SI systems are typically made up of a population of simple autonomous agents interacting locally with one another and with their environment. Although there is no centralized control, the local interactions between agents lead to the emergence of global behavior. Examples of systems like this can be found in nature, including ant colonies, bird flocking, animal herding, bacteria molding, and fish schooling.

The most successful SI techniques are ant colony optimization (ACO) and particle swarm optimization (PSO). In ACO (Dorigo and Stützle 2004), artificial ants build solutions walking in the graph of the problem and (simulating real ants) leaving artificial pheromone so that other ants will be able to build better solutions. ACO was successfully applied to an impressive number of optimization problems. PSO is an optimization method initially designed for continuous optimization; however, it was further adapted to solve various combinatorial problems. PSO is presented in more detail in the next section.

3.1 Particle Swarm Optimization

The PSO model was introduced in 1995 by Kennedy and Eberhart (1995), being discovered through simulation of a simplified social model such as fish schooling or bird flocking. It was originally conceived as a method for optimization of continuous nonlinear functions. Latter studies showed that PSO can be successfully adapted to solve combinatorial problems.

The evolutionary cultural model proposed by (Boyd and Richerson 1985) stands as the basic principle of PSO. According to this model, individuals of a society have two learning sources: individual learning and cultural transmission. Individual learning is efficient only in homogenous environments: The patterns acquired

through local interactions with the environment are generally applicable. For heterogenous environments, social learning—the essential feature of cultural transmission—is necessary.

In line with the evolutionary cultural model, the PSO algorithm uses a set of simple agents which collaborate in order to find solutions of a given optimization problem.

In the PSO paradigm, the environment corresponds to the search space of the optimization problem to be solved. A swarm of particles is placed in this environment. The location of each particle corresponds therefore to a candidate solution to the problem. A fitness function is formulated in accordance with the optimization criterion to measure the quality of each location. The particles move in their environment collecting information on the quality of the solutions they visit and share this information to the neighboring particles in the swarm. Each particle is endowed with memory to store the information gathered by individual interactions with the environment, simulating thus *individual learning*. The information acquired from neighboring particles corresponds to the *social learning* component. Eventually, the swarm is likely to move toward "more" optimum locations of the search space, similar to a flock of birds that collectively forage for food.

Unlike GAs, in PSO, there exist no evolution operators and no competition for survival; all particles survive and share information for the welfare of the swarm. The driving force is the emergent SI and attained by the sharing of local information between particles in order to produce global knowledge. It is important to note that problem solving is a population-wide phenomenon, because a particle by itself is probably incapable of solving even simple problems (Poli et al. 2007).

Usually, the swarm is composed of particles that share the same structural and behavioral features. Each particle is characterized by its current position in the search space, its velocity, and one or more of its best positions in the past (usually, only one position). Each particle uses the objective (fitness) function so that it can find out how good its current status is. The particles use a communication channel in order to exchange information with (some) of its peers. The topology of the swarm's social network is defined by the structure of the communication channel, where cliques of interconnected particles form neighborhoods.

In the classical PSO algorithm, the position of a particle in the search space is updated in each iteration depending on the position and velocity of the particle in the previous iteration. The formulas used to update the particles and the procedures are inspired from and conceived for continuous spaces. Therefore, each particle is represented by a vector x of length n indicating the position in the n-dimensional search space and has a velocity vector v used to update the current position. The velocity vector is computed following the rules:

- every particle tends to keep its current direction (an inertia term);
- every particle is attracted to the best position p it has achieved so far (implements the individual learning component);
- every particle is attracted to the best particle g in the neighborhood (implements the social learning component).

The velocity vector is computed as a weighted sum of the three terms above. Two random multipliers r_1, r_2 are used to gain stochastic exploration capability, while w, c_1, c_2 are weights usually empirically determined. The formulae used to update each of the individuals in the population at iteration t + 1 are as follows:

$$v_i^t = w \cdot v_i^{t-1} + c_1 \cdot r_1 \cdot (p_i^{t-1} - x_i^{t-1}) + c_2 \cdot r_2 \cdot (g \cdot i^{t-1} - x_i^{t-1}) \tag{13a}$$

$$x_i^t = x_i^{t-1} + v_i^t \tag{13b}$$

As a side effect of these changes, the velocity of the particle could enter a divergence process, throwing the particle further, and further away form p. To prevent this behavior, Kennedy and Eberhart clamped the amplitude of the velocity to a maximum value, denoted by $v_{\max}$:

$$v_i^t = \min(v_{\max}, \max(-v_{\max}, v_i^t)). \tag{14}$$

Equation 13b generates a new position in the search space (corresponding to a candidate solution). It can be associated to some extent to the mutation operator in evolutionary programing. However, in PSO, this mutation is guided by the past experience of both the particle and other members of the swarm. In other words, "PSO performs mutation with a conscience" (Jong 2006). Considering the best visited solutions stored in the personal memory of each individual as additional members of the population, PSO implements a weak form of selection (Angeline 1998).

The shape of the search space is unknown; hence, there exists no known optimum combination of the two learning sources (i.e., individual learning and cultural transmission). The classical PSO algorithm compensates this lack of information with random values for learning factors $c_1 \cdot r_1$ and $c_2 \cdot r_2$, which change in each iteration in order to weigh differently the learning sources. The velocity change produced by each term depends on the distance between the compared positions (i.e., the particle will move faster if values are larger) and the random learning factors. This allows PSO to simulate, during a single run, various search strategies. The solution that the algorithm outputs at the end of the run is obtained from the information stored in the memory of each particle after the last iteration is completed.

The search for the optimal solution in PSO is described by the iterative procedure in Fig. 11. The fitness function is denoted by f and is formulated for maximization.

Particle p_i is chosen in the basic version of the algorithm to be the best position in the problem space visited by particle i. However, the best position is not always dependent only on the fitness function. Constraints can be applied in order to adapt PSO to various problems, without slowing down the convergence of the algorithm. In constrained nonlinear optimization, the particles store only feasible solutions and ignore the infeasible ones (Hu and Eberhart 2002). In multi-objective optimization, only the Pareto-dominant solutions are stored (Coello and Lechunga 2002; Hu and

Fig. 11 Basic PSO

1. $t := 0$
2. Initialize $x_i^t, i = \overline{1..n}$
3. Initialize $v_i^t, i = \overline{1..n}$
4. Store personal best $p_i^t = x_i^t, \ i = \overline{1..n}$
5. Find neighborhood best $g_i^t = argmax_{y \in Nx_i^t}(f(y)), \ i = \overline{1..n}$
6. **while** *halting condition not met* **do**
7. $t := t + 1$
8.   Update $v_i^t, \ i = \overline{1..n}$ using equation 13a
9.   Update $x_i^t, \ i = \overline{1..n}$ using equation 13b
10.   Update personal best $p_i^t = argmax(f(p_i^{t-1}), f(x_i^t))$
11.   Find neighborhood best $g_i^t = argmax_{y \in Nx_i^t}(f(y))$
12. **end while**

Eberhart 2002). In dynamic environments, particle p is reset to the current position if a change in the environment is detected (Hu and Eberhart 2001).

The selection of particle g_i is performed in two steps: neighborhood selection followed by particle selection. The size of the neighborhood has a great impact on the convergence of the algorithm. It is generally accepted that a large neighborhood speeds-up the convergence, while small neighborhoods prevent the algorithm from premature convergence. Various neighborhood topologies were investigated with regard to their impact on the performance of the algorithm (Kennedy 2002; Kennedy and Mendes 2003); however, as expected, there is no free lunch: Different topologies are appropriate to different problems.

A major problem investigated in the PSO literature is the **premature convergence** of the algorithm in multi-modal optimization. This problem has been addressed in several papers and solutions include addition of a queen particle (Clerc 1999), alternation of the neighborhood topology (Kennedy 1999), introduction of subpopulations (Lïvbjerg et al. 2001), giving the particles a physical extension (Krink et al. 2002), alternation between phases of attraction and repulsion (Riget and Vesterstrøm 2002), giving different temporary search goals to groups of particles (Al-kazemi and Mohan 2002), giving particles quantum behavior (Sun et al. 2004), and the use of specific swarm-inspired operators (Breaban and Luchian 2005).

Another crucial problem is **parameter control**. The values and choices for some of these parameters may have significant impact on the efficiency and reliability of the PSO. There are several papers that address this problem; in most of them, the values for parameters are established through repeated experiments but there also exist attempts to adjust them dynamically, using EC algorithms.

The role played by the inertia weight was compared to that of the temperature parameter in simulated annealing (Shi and Eberhart 1998). A large inertia weight facilitates a global search, while a small inertia weight facilitates a local search. The parameters c_1 and c_2 are called generically learning factors; because of their distinct roles, c_1 was named the cognitive parameter (it gives the magnitude of the information gathered by each individual) and c_2 the social parameter (it weights the cooperation between particles). Another parameter used in PSO is the maximum

velocity which determines the maximum change each particle can take during one
iteration. This parameter is usually proportional with the search domain.

One run of the PSO algorithm can be illustrated using package pso built for R
which is consistent with standard PSO, as described in Bratton and Kennedy
(2007):

```
> library(pso)
> PSO.sols <- psoptim(rep(NA,2),SixHumpV,lower=c(-3,-2),upper=c(3,2),
   control=list( maxit=50, s=20, trace=1, REPORT=1))
S=20, K=3, p=0.1426, w0=0.7213, w1=0.7213, c.p=1.193, c.g=1.193
v.max=NA, d=7.211, vectorize=FALSE, hybrid=off
It 1: fitness=-0.3635
It 2: fitness=-0.8261
It 3: fitness=-0.8261
It 4: fitness=-0.8623
It 5: fitness=-0.9337
...
```

The final solution obtained in 50 iterations with a population of 20 individuals
reaches the global optima:

```
> show(PSO.sols)
$par
[1]  0.09041749 -0.71296641

$value
[1] -1.031627
```

The algorithm reaches quickly the global optima, as shown in Fig. 12.

Figure 13 illustrates the distribution of the individuals in population during one
run, at iterations 1, 2, 5, 10, 20, and 50.

3.2 PSO on Binary Domains

Although PSO was conceived for continuous optimization, an effort was done to
adapt the algorithm in order to be used for solving a wide range of combinatorial
and binary optimization problems. A short discussion of the binary version of PSO
is presented in this section, following the presentation from (Bautu 2010).

Kennedy and Eberhart (1997) introduced a first variant of binary PSO, com-
bining the evolutionary cultural model with the reasoned action model. According
to the latter, the action performed by an individual is the stochastic result of the
intention to do that action. The strength of the intention results from the interaction
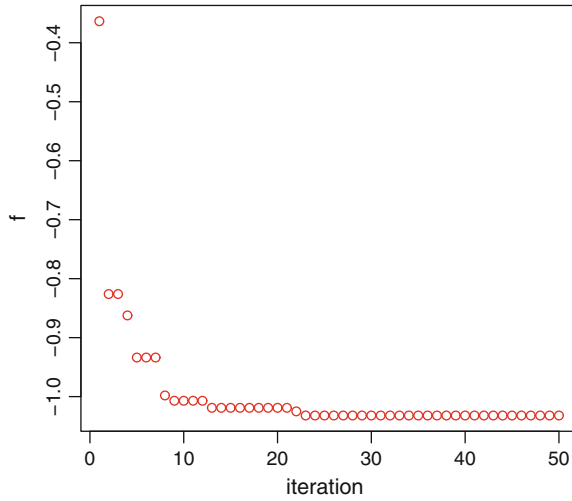of the personal attitude and the social attitude on the matter (Hale et al. 2002).

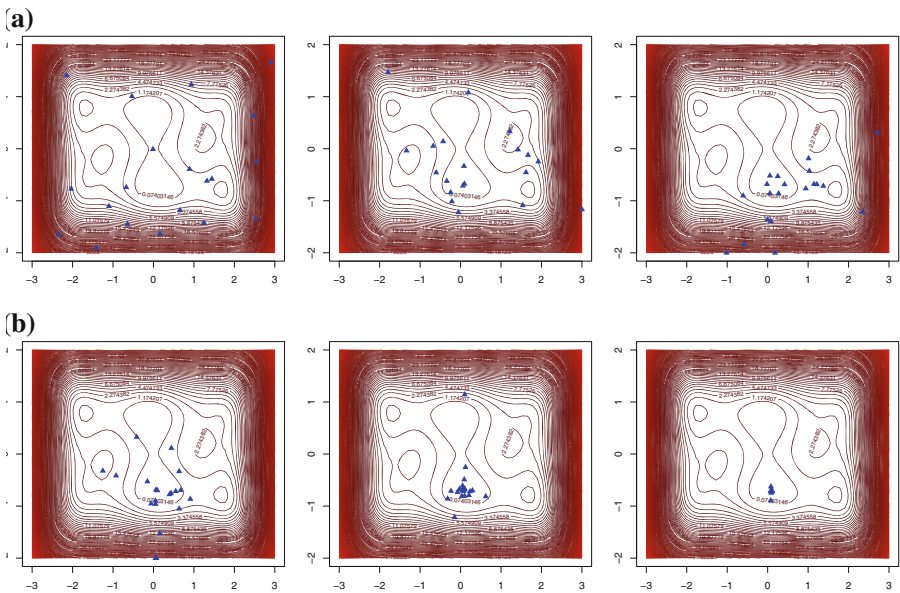**Fig. 12** The evolution of the best value of the objective function for one run of PSO

Fig. 13 The evolution of the population in PSO during one run of the algorithm: the distribution of the candidate solutions at iterations 1, 2, 5, 10, 20, and 15

The PSO algorithm for real-valued optimization updates the positions of particles based on a function that depends (indirectly) of various personal and social factors. In the binary domain, the intention of particles to move between the two allowed positions: 0 and 1 is modeled in a similar manner. The *probability* that the particle will move to position 1 is computed by:

$$P(p^t = 1) = f'(p^{t-1}, v^{t-1}, p_i^{t-1}, g_g^{t-1}). \tag{15}$$

The individual learning factor and the social learning factor act as personal and social attitudes that help to select one of the two binary options.

In particular, with respect to classical PSO, in binary PSO:

- the domain of particle positions in the context of binary optimization problems is $P = \{0, 1\}^n$;
- the cost function that describes the optimization problem is hence defined $c : \{0, 1\}^n \rightarrow \mathbb{R}$;
- the position of a particle consists in the responses of the particle to the n binary queries of the problem. The position in the search space is updated during each iteration depending on its velocity.

Let $p^t \in P$ and $v^t \in \mathbb{R}$ denote the position and the velocity of a particle at iteration t. The update equation for the particle's position in binary PSO is as follows:

$$p = \begin{cases} 1, & \text{if } \phi_3 < (1 + \exp(-v))^{-1} \\ 0, & \text{otherwise} \end{cases}, \tag{16}$$

where ϕ_3 is a random uniformly distributed variable in $[0, 1)$. It results that higher velocity induces higher probabilities for the particle to choose 1. The equation for the particle ensures that the particle stays within the search space domain; hence, no relocation procedure is required.

The velocity of the particle is updated using the same equation as in classical PSO. The semantics of each term in (13a) for binary PSO are special cases of their original meaning. For example, if the best position of the particle (p_i^t) is 1 and the current position (p^t) is 0, then $p_i^t - p^t = 1$. In this case, the second term in (13a) will increase the value of v^t; hence, the probability that the particle with choose 1 will also increase. Similarly, the velocity will decrease if $p_i^t = 0$ and $p^t = 1$. If the two positions are the same, the individual learning term will not change the velocity in order to try to maintain the current choice. The same is true for the velocity updates produced by the social learning term. The position of the particle may change due to the stochastic nature of (16), even if the velocity does not change between iterations.

The complete PSO algorithm for binary optimization problems is presented in vector form in (Fig. 14).

Require: c - the objective function
Ensure: S - the position that encodes the best solution
1. $t = 0$
2. Initialize particle positions (p^t)
3. Initialize particle velocities (v^t)
4. Store particle best solutions ($g_i^t = p^t$)
5. **while** searching allowed **do**
6. $t = t + 1$
7.   Update positions using equation (16)
8.   Find neighborhood best solutions with neighborhood operator N
           ($g_g^t = \mathrm{argmin}_{x \in \{b_i^t | N\}} c(x)$)
9.   Update velocity using equation (13a)
10.   Limit velocity using equation (14)
11. **end while**
12. Retrieve the solution.
13. **return** S

Fig. 14 The particle swarm optimization algorithm for binary optimization

Other PSO variants can also be successfully used on binary spaces. In Wang et al. (2008), the authors propose the outcome of the binary queries to be established randomly based on the position of the particle within a continuous space. Khanesar et al. (2009) present a variation of the binary PSO in which the particle toggles its binary position with probability depending its velocity.

4 Integrating Meta-heuristics with Conventional Methods in Data Analysis: A Practical Example

Meta-heuristics stand as basis for the design of efficient algorithms for various data analysis tasks. Such approaches are extensions of conventional techniques, obtained as hybridizations with meta-heuristics, or evolved as new self-contained data analysis methods.

There is a large variety of approaches for data clustering based on GAs (Breaban et al. 2012; Hruschka et al. 2009; Luchian et al. 1994), DE (Zaharie 2005), PSO (Breaban and Luchian 2011; Rana et al. 2011), and ACO (Shelokar et al. 2004). Learning Classifier Systems (Lanzi et al. 2000) are one of the major families of techniques that apply EC to machine learning; these systems evolve a set of condition–action rules able to solve classification problems. Decision trees (Turney 1995) and support vector machines (Stoean et al. 2009, 2011) are also evolved with GAs. The representative application example of EAs in regression analysis is the use of genetic programing for symbolic regression, topic covered in detail in Chapter "Genetic Programming Techniques with Applications in the Oil and Gas Industry" of this book. Many algorithms based on meta-heuristics tackle feature selection and feature extraction.

We restrict the discussion in this section to one particular application: Optimization of the parameters of machine learning algorithms used in data analysis. The performance of several machine learning algorithms depends heavily on some parameters involved in their design; such parameters are often called meta-parameters or hyper-parameters. The problem of choosing the best settings for these parameters is also known as model selection.

Examples may vary from simple algorithms such as k-nearest neighbors where k is such a hyper-parameter, to more complex algorithms. In the case of artificial neural networks, the structure of the network (the number of hidden layers, the number of neurons in each layer, the activation function) has a high impact on the accuracy of the results in classification or regression analysis; the degree of complexity of the network is a critical factor in the trade-off between overfitting the model to the training data and underfitting, and the right balance can be achieved only with extensive experiments. In the definition of support vector machines (SVMs), two numerical parameters play important roles: a constant C called regularization parameter and a constant ε corresponding to the width of the ϵ-insensitive zone, influence the number of support vectors used in the model, controlling the trade-off between two goals, fitting the training set well, and avoiding overfitting; parameters characterizing various kernel functions are also involved.

We illustrate here a simple model selection scheme by means of EAs for regression analysis. A small dataset called "rock," included in R, is used with this purpose. It consists of 48 rock samples from a petroleum reservoir characterized by the area of pores, total perimeter of pores, shape, and permeability.

```
> show(rock)
    area      peri     shape    perm
1   4990 2791.900 0.0903296    6.3
2   7002 3892.600 0.1486220    6.3
3   7558 3930.660 0.1833120    6.3
4   7352 3869.320 0.1170630    6.3
...
48  9718 1485.580 0.2004470  580.0
```

We illustrate regression analysis by training a support vector machine to learn a model able to predict permeability. The quality of the regression model is usually measured by the mean squared error, as defined below.

```
> MSE <- function(x,y)
+ {
+   mean((x-y)^2)
+ }
```

Support vector regression is implemented in R under package "e1071." The results obtained using radial kernel are shown below:

```
> library(e1071)
> svr <- svm(perm ~ area+peri+shape, data=rock,
+ type="eps-regression", kernel = "radial")
> predicted <-predict(svr,newdata=rock,type="response")
> MSE(predicted, rock$perm)
[1] 35316.21
> cor(predicted, rock$perm)
[1] 0.9040716
> plot(predicted, rock$perm)
```

The default settings of the three hyper-parameters used in the run above can be inspected next: **Cost** is the regularization parameter, **gamma** is a parameter of the kernel function, and epsilon is the size of the insensitive tube.

```
> summary(svr)
Parameters:
   SVM-Type:  eps-regression
 SVM-Kernel:  radial
       cost:  1
      gamma:  0.3333333
    epsilon:  0.1
```

These numerical parameters can be optimized in order to minimize the prediction error measured by MSE. We formulate this task as a numerical optimization problem defined over three numerical parameters (cost, gamma, and epsilon), aiming to minimize the MSE of the predictions obtained with support vector regression under the given settings:

```
> trainingError <- function(params)
+ {
+    svr <- svm(perm ~ area+peri+shape, data=rock, type="eps-regression",
+      kernel = "radial", gamma=params[1], cost = params[2], epsilon = params[3])
+    predicted <-predict(svr,newdata=rock,type="response")
+    MSE(predicted, rock$perm)
+ }
```

Any of the meta-heuristics presented in this chapter can be used to tackle this minimization problem. We illustrate here the use of DE:

```
> DEparams <- DEoptim(trainingError, lower = c(0, 0, 0), upper = c(4, 4, 1),
+ control = list(strategy = 1,NP=20, itermax=20, trace = TRUE))
Iteration: 1 bestvalit: 5937.692186 bestmemit:    1.929174    2.872409    0.012022
Iteration: 2 bestvalit: 5630.575260 bestmemit:    3.110530    3.717773    0.166768
Iteration: 3 bestvalit: 3623.210268 bestmemit:    2.818071    3.682759    0.077892
...
Iteration: 20 bestvalit: 1473.135923 bestmemit:    3.983884    3.812688    0.046011
```

The solution obtained by DE is stored next in the vector **params** and is used to train a new SVM.
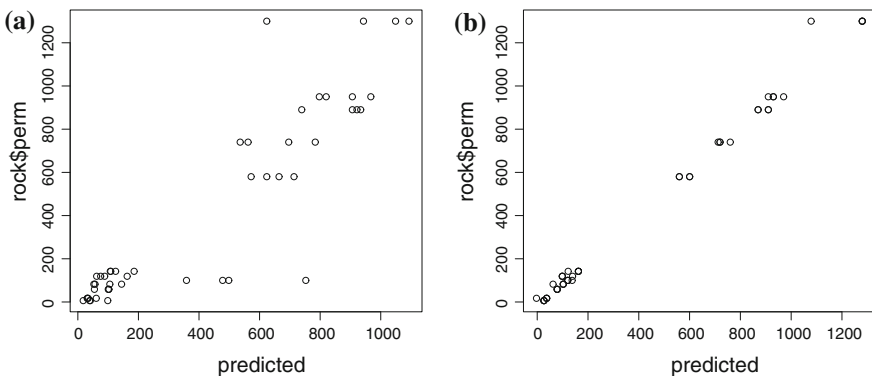
```
> params <- DEparams$optim$bestmem
> svr <- svm(perm ~ area+peri+shape, data=rock, scale = TRUE, type="eps-regression",
+ kernel = "radial", gamma=params[1], cost = params[2], epsilon = params[3])
> predicted <-predict(svr,newdata=rock,type="response")
> MSE(predicted, rock$perm)
[1] 1473.136
> cor(predicted, rock$perm)
[1] 0.9968882
```

Figure 15 illustrates the predicted values compared to real values for the case of SVR with default settings (a) and for the case of SVR with optimized hyper-parameters (b).

Nevertheless, the optimized model gives much better results with regard to the error of predictions, but is prone to overfitting: A single dataset was used both for training and testing; in this situation, the model is highly adapted to the dataset and may suffer from poor generalization power. We can avoid overfitting by using distinct sets for training and testing. The new function to be optimized should be formulated as shown below. Very similar with the previous version regarding its definition, this function is significantly different in behavior: It invokes a "training" dataset in the learning phase but computes the prediction error on a "testing" dataset:

Fig. 15 Predicted over expected values in regression analysis with SVR using: **a** default hyper-parameters settings and **b** optimized settings

```
> testingError <- function(params)
+ {
+     svr <- svm(perm ~ area+peri+shape, data=training, type="eps-regression",
+        kernel = "radial", gamma=params[1], cost = params[2], epsilon = params[3])
+     predicted <-predict(svr,newdata=testing,type="response")
+     MSE(predicted, rock$perm)
+ }
```

The validation of the regression model obtained with the optimized hyper-parameters requires in this case a third dataset called validation set. This phase closes the analysis which, as recommended in the case of any supervised learning task, is composed of three phases: training, testing, and validation. If the accuracy/error obtained in the validation phase is satisfactory, the model can be used in production.

5 Applications of Meta-Heuristics in Geosciences

Evolutionary algorithms have been used in solving geophysics optimization problems in two main directions: either by performing the optimization, or by optimizing parameters of other methods (e.g., neural networks) used in specific problems.

Evolutionary methods are compared to PSO in a study on optimization of reservoir models to match past petroleum production data in Yasin Hajizadeh et al. (2011). ACO, DE, PSO, and the neighborhood algorithm are integrated in a Bayesian framework in order to measure the uncertainty of the predictions obtained by each algorithm, in a case study involving two petroleum reservoirs. Ahmadi et al. (2013) perform the task of predicting reservoir permeability using a soft sensor implemented on the basis of a feed-forward artificial neural network, which was then optimized using a hybrid GA and PSO method. History matching is also the research topic in Park et al. (2014). A multi-objective evolutionary algorithm identifies optimal solutions and outperforms a traditional weighted-sum approach.

GAs are acknowledged as important tools for successful neural network data-driven models with applications in the oil and gas industry (Mohaghegh 2005; Shahab et al. 2005). Intelligent software tools used in the industry integrate hard (statistical) and soft (intelligent) computing techniques, such as fuzzy cluster analysis, genetic optimization, or neural computing (Shahab et al. 2005).

Direct use of a GA helps to evaluate hydrocarbon resource in a field dataset from North Cambay basin, India (Thander et al. 2014). Several parameters are required for resource estimation (e.g., areal extent, net pay thickness, oil saturation, etc.), yet a limited set is recorded in the exploration phase. Also, recordings are done with uncertainty, due to reservoir heterogeneity. GA copes well with the uncertainty in data and delivers estimations of the oil reserve using a real dataset.

An oil production planning problem that appears in the context of oil wells with insufficient oil pressure and which consists in identifying the amount of gas that should be injected in a well in order to maximize the amount of oil extracted from that well is solved by an evolutionary algorithm in Singh et al. (2013). The problem is more difficult since it is constrained by the total amount of gas available daily. The authors propose a multi-objective approach to the problem and also formulate a single objective version, focused on the maximization of profit, instead of the oil quantity. The problem of gas allocation among oil wells is also tackled in Ghaedi et al. (2013), by means of a hybrid GA, and in Abdel Rasoul et al. (2014). The problem of gas allocation among oil wells is also tackled in Ghaedi et al. (2013), by means of a hybrid GA, and in Abdel Rasoul et al. (2014).

The optimal well type and location are determined with PSO in (Onwunalu and Durlofsky 2010), in a study involving vertical, deviated, and dual-lateral wells. Comparisons with a GA over multiple runs of both algorithms show that PSO outperforms, on average, the GA, yet the advantages of using PSO over GA are varied among the cases surveyed. Driven by the goal of maximizing the total hydrocarbon recovery, an well placement problem is tackled in Nwankwor et al. (2013) with a hybrid PSO-DE algorithm is proposed for the problem. The hybrid is compared to basic variants of PSO and DE on three problem cases concerning the placement of vertical wells in 2D and 3D reservoir models. Optimal well placement under uncertainty is tackled in a two-stage approach in Lyons and Nasrabadi (2013). First, an ensemble Kalman filter is used to perform history matching on the reservoir data. Then, well placement is solved by a GA combined with pseudohistory matching.

Carbon dioxide (CO_2) sequestration is of great interest for oil engineers. In recent years, the idea of storing CO_2 in deep geological formations, such as depleted oil and gas reservoirs (with impermeable rocks), gained a lot of focus from the community as a solution for greenhouse gas mitigation by avoiding CO_2 from emission into the atmosphere. The CO_2 sequestration also helps by enhancing methods for oil or gas recovery (Zangeneh et al. 2013). Evolutionary algorithms are used in order to identify carbon dioxide seepage areas in Cortis et al. (2008). In Zangeneh et al. (2013), the parameters of a CO_2 storage model are optimized using a GA. A multi-objective GA (NSGA) is implemented for optimizing gas storage alongside oil recovery in Safarzadeh and Motahhari (2014). Based on the results from the GA, the authors are able to propose some production scenarios.

In (Fichter et al. 2000), a portfolio optimization problem for the oil and gas industry is tackled by means of a GA. GAs are chosen for this task both due to their scalability to extremely large portfolios and because they allow the analysis of portfolios from the point of view of value and risk measures.

GA and PSO are used to find the optimal parameters of a linear and an exponential model for the demand of oil in Iran in Assareh et al. (2010). The models use as input variables the population, the gross domestic product, import, and export data; they are used to forecast demand of oil up to 2030.

PSO emerged as a powerful algorithm for geophysical inverse problems when compared to GAs and simulated annealing in Martnez et al. (2010), Shaw, and

Srivastava (2007). Other applications include inversion of seismic refraction data Poormirzaee et al. (2014), crosshole traveltime tomography Tronicke et al. (2012), or reservoir characterization Fernández Martìnez et al. (2012).

A large number of meta-heuristics are compared with respect to training an artificial neural network for the task of forecasting the water temperature of a natural river in Piotrowski et al. (2014). The study involves a comparison of several versions of PSO, DE, direct search to the levenberg–Marquardt (LM) algorithm for ANN training. The study concludes that only the DE algorithm obtains results competitive to the LM algorithm. A similar optimization idea is described in Ahmadi and Ebadi (2014), where a hybrid combination between an artificial neural network and PSO, extended with dew point pressure data, leads to a better understanding of reservoir fluid behavior.

References

Abdel Rasoul RR, Daoud A, El Tayeb ESA (2014) Production allocation in multi-layers gas producing wells using temperature measurements with the application of a genetic algorithm. Pet Sci Technol 32(3):363–370

Ahmadi MA, Ebadi M (2014) Robust intelligent tool for estimation dew point pressure in retrograded condensate gas reservoirs: application of particle swarm optimization. J Pet Sci Eng 123:7–19

Ahmadi MA, Zendehboudi S, Lohi A, Elkamel A, Chatzis I (2013) Reservoir permeability prediction by neural networks combined with hybrid genetic algorithm and particle swarm optimization. Geophys Prospect 61(3):582–598

Al-kazemi B, Mohan CK (2002) Multi-phase generalization of the particle swarm optimization algorithm. In: Proceedings of the IEEE congress on evolutionary computation. IEEE Press

Angeline PJ (1998) Using selection to improve particle swarm optimization. In: Proceedings of the IEEE international conference on evolutionary computation. IEEE Press, pp 84–89. ISBN 0-7803-4869-9

Assareh E, Behrang MA, Assari MR, Ghanbarzadeh A (2010) Application of PSO (particle swarm optimization) and GA (genetic algorithm) techniques on demand estimation of oil in iran. Energy 35(12):5223–5229

Back T (1996) Evolutionary algorithms in theory and practice. Oxford University Press, New York

Baker JD (1985) Adaptive selection methods for genetic algorithms. In: Proceedings of an International Conference on Genetic Algorithms and their applications. Hillsdale, New Jersey, pp 101–111

Baker JD (1987) Reducing bias and inefficiency in the selection algorithm. In: Proceedings of the second international conference on genetic algorithms. pp 14–21

Bautu A (2010) Generalizations of Particle Swarm Optimization: applications of particle swarm algorithms to statistical physics and bioinformatics problems. PhD Thesis, Department of Computer Science, Al. I. Cuza University, Lambert Academic Publishing. ISBN 978-3848417315

Blum C, Roli A (2003) Metaheuristics in combinatorial optimization: overview and conceptual comparison. ACM Comput Surv 35(3):268–308. ISSN 0360-0300. doi:http://doi.acm.org/10.1145/937503.937505

Boyd R, Richerson PJ (1985) Culture and the evolutionary process. The University of Chicago Press, Chicago

Bratton D, Kennedy J (2007) Defining a standard for particle swarm optimization. In: Swarm intelligence symposium, 2007. SIS 2007, IEEE, pp 120–127

Breaban M (2011) Clustering: evolutionary approaches. PhD Thesis, Department of Computer Science, Al. I. Cuza University

Breaban M, Luchian H (2005) PSO under an adaptive scheme. In: Proceedings of the IEEE congress on evolutionary computation. IEEE Press, pp 1212–1217

Breaban ME, Luchian H (2011) PSO aided k-means clustering: introducing connectivity in k-means. In: Proceedings of the 13th annual conference on Genetic and evolutionary computation. ACM, pp 1227–1234

Breaban ME, Luchian H, Simovici D (2012) A genetic clustering algorithm by monomial projection pursuit. In Symbolic and numeric algorithms for scientific computing (SYNASC), 14th international symposium on 2012. IEEE, pp 214–219

Bremermann HJ (1958) The evolution of intelligence: the nervous system as a model of its environment. Technical Report No. 1, Department of Mathematics, University of Washington, Seattle

Burke EK, Gendreau M, Hyde M, Kendall G, Ochoa G, Özcan E, Qu R (2013) Hyper-heuristics: a survey of the state of the art. J Oper Res Soc 64(12):1695–1724

Clerc M (1999) The swarm and the queen: towards a deterministic and adaptive particle swarm optimization. In: Proceedings of the IEEE congress on evolutionary computation, vol 3, pp 1951–1957. doi:10.1109/CEC.1999.785513

Clerc M (2006) Particle swarm optimization. Hermes Sci, London. ISBN 1905209045

Coello CAC, Lechunga MS (2002) Mopso: a proposal for multiple objective particle swarm optimization. In Proceedings of the IEEE congress on evolutionary computation. IEEE Press, pp 1051–1056

Cortis A, Oldenburg CM, Benson SM (2008) The role of optimality in characterizing CO_2 seepage from geologic carbon sequestration sites. Int J Greenh Gas Control 2(4):640–652

De Jong KA (2006) Evolutionary computation. A unified approach. MIT Press, Cambridge

Deb K, Goldberg DE (1989) An investigation of niche and species formation in genetic function optimization. In: Proceedings of the 3rd international conference on genetic algorithms, San Francisco. Morgan Kaufmann Publishers Inc., pp 42–50, ISBN 1-55860-066-3. http://portal. acm.org/citation.cfm?id=645512.657099

Dorigo M, Stützle T (2004) Ant colony optimization. Bradford Company, Scituate. ISBN 0262042193

Dumitrescu D (2000) Genetic chromodynamics. Studia Universitatis Babes-Bolyai Cluj-Napoca, Ser. Informatica 45:39–50

Fernández Martnez JL, Mukerji T, Garca Gonzalo E, Suman A (2012) Reservoir characterization and inversion uncertainty via a family of particle swarm optimizers. Geophysics 77(1): M1–M16

Fichter DP et al (2000) Application of genetic algorithms in portfolio optimization for the oil and gas industry. In: SPE annual technical conference and exhibition. Society of Petroleum Engineers

Fogel LJ, Owens AJ, Walsh MJ (1966) Artifficial intelligence through simulated evolution. Wiley, New York

Fraser AS (1957) Simulations of genetic systems by automatic digital computers. Aust J Biol Sci 10:492–499

Ghaedi M, Ghotbi C, Aminshahidy B (2013) Optimization of gas allocation to a group of wells in gas lift in one of the iranian oil fields using an efficient hybrid genetic algorithm (HGA). Pet Sci Technol 31(9):949–959

Glover F (1986) Future paths for integer programming and links to artificial intelligence. Comput Oper Res 13(5):533–549. ISSN 0305-0548. doi:10.1016/0305-0548(86)90048-1

Grefenstette JJ (1986) Optimization of control parameters for genetic algorithms. IEEE Trans Syst Man Cybern 16(1): 122–128

Grefenstette JJ (1987) Incorporating problem specific knowledge into genetic algorithms. Genet Algorithms Simul Annealing 4:42–60

Hajizadeh Y, Demyanov V, Mohamed L, Christie M (2011) Comparison of evolutionary and swarm intelligence methods for history matching and uncertainty quantification in petroleum

reservoir models. In: Intelligent computational optimization in engineering. Springer, Berlin, pp 209–240

Hale JL, Householder BJ, Greene KL (2002) The theory of reasoned action. Sage Publications, Thousand Oaks, pp 259–286

Hillis WD (1990) Co-evolving parasites improve simulated evolution as an optimization procedure. Phys D Nonlinear Phenom 42(1):228–234

Holland JH (1975) Adaptation in natural and artificial systems. University of Michigan Press, Ann Arbor

Holland JH (1998) Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control and artificial intelligence. MIT Press, Cambridge. ISBN 0-262-58111

Hruschka ER, Campello RJGB., Freitas AA, De Carvalho APLF (2009) A survey of evolutionary algorithms for clustering. IEEE Trans Syst Man Cybern Part C Appl Rev 39(2):133–155

Hu X, Eberhart RC (2001) Tracking dynamic systems with PSO: where's the cheese? In Proceedings of the workshop on particle swarm optimization, pp 80–83

Hu X, Eberhart RC (2002) Multiobjective optimization using dynamic neighborhood particle swarm optimization. In: Proceedings of the IEEE congress on evolutionary computation. IEEE Press, pp 1677–1681

Hu X, Eberhart RC (2002) Solving constrained nonlinear optimization problems with particle swarm optimization. In: Proceedings of the sixth world multiconference on systemics, cybernetics and informatics

Ionita M, Croitoru C, Breaban M (2006) Incorporating inference into evolutionary algorithms for max-csp. In: 3rd international workshop on hybrid metaheuristics, LNCS 4030. Springer, Berlin, pp 139–149

Jong KD (2006) Evolutionary computation: a unified approach. MIT Press. ISBN 0-262-04194

Kennedy J (1999) Small worlds and mega-minds: effects of neighborhood topology on particle swarm performance. In: Proceedings of the IEEE congress of evolutionary computation, vol 3. IEEE Press, pp 931–1938. doi:10.1109/CEC.1999.785513

Kennedy J (2002) Population structure and particle swarm performance. In: Proceedings of the congress on evolutionary computation (CEC 2002). IEEE Press, pp 1671–1676

Kennedy J, Eberhart RC (1995) Particle swarm optimization. In: Proceedings of the 1995 IEEE international conference on neural networks, vol 4. IEEE Press, pp 1942–1948

Kennedy J, Eberhart RC (1995) Particle swarm optimization. In: Proceedings of IEEE international conference on neural networks, pp 1942–1948

Kennedy J, Eberhart RC (1997) A discrete binary version of the particle swarm algorithm. In: Proceedings of the world multiconference on systemics, cybernetics and informatics, vol 5, Piscataway. IEEE Press, pp 4104–4109

Kennedy J, Mendes R (2003) Neighborhood topologies in fully-informed and best-of-neighborhood particle swarms. In: Proceedings of the 2003 IEEE SMC workshop on soft computing in industrial applications (SMCia03). IEEE Computer Society, pp 45–50

Kenneth ADJ (1975) An analysis of the behavior of a class of genetic adaptive systems. PhD thesis, University of Michigan, Dissertation Abstracts International, vol 36, no 10, Ann Arbor, AAI7609381

Khanesar MA, Tavakoli H, Teshnehlab M, Shoorehdeli MA (2009) Novel binary particle swarm optimization. In: Tech Education and Publishing, pp 1–10. ISBN 978-953-7619-48-0

Kirkpatrick S, Gelatt CD, Vecchi MP et al (1983) Optimization by simmulated annealing. Science 220(4598):671–680

Konak A, Coit DW, Smith AE (2006) Multi-objective optimization using genetic algorithms: a tutorial. Reliab Eng Syst Safety 910(9):992–1007. http://www.sciencedirect.com/science/article/B6V4T-4J0NY2F-2/2/97db869c46fc43f457f3d509adaa15b5

Koza J (1992) Genetic programming: on the programming of computers by means of natural selection. MIT Press, Cambridge

Krink T, Vesterstrom JS, Riget J (2002) Particle swarm optimisation with spatial particle extension. In: Proceedings of the evolutionary computation on 2002. CEC'02. Proceedings of

the 2002 Congress—vol 02, CEC'02. IEEE Computer Society, Washington, pp 1474–1479. ISBN 0-7803-7282-4. http://portal.acm.org/citation.cfm?id=1251972.1252447

Lanzi PL, Stolzmann W, Wilson SW (2000) Learning classifier systems: from foundations to applications (No. 1813). Springer, Berlin

Lïvbjerg M, Rasmussen TK, Krink T (2001) Hybrid particle swarm optimiser with breeding and subpopulations. In: Proceedings of the genetic and evolutionary computation conference (GECCO-2001). Morgan Kaufmann, pp 469–476

Luchian S, Luchian H, Petriuc M (1994) Evolutionary automated classification. In: Proceedings of 1st congress on evolutionary computation, pp 585–588

Lyons J, Nasrabadi H (2013) Well placement optimization under time-dependent uncertainty using an ensemble kalman filter and a genetic algorithm. J Petrol Sci Eng 109:70–79

Martnez JLF, Gonzalo EG, Álvarez JPF, Kuzma HA, Pérez COM (2010) PSO: A powerful algorithm to solve geophysical inverse problems: Application to a 1D-DC resistivity case. J Appl Geophys 710(1):13–25

Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller AH, Teller E (1953) Equation of state calculations by fast computing machines. J Chem Phys 21(6):1087–1092

Michalewicz Z (1992) Genetic algorithms + data structures = evolution programs (3rd edn). Springer, Berlin. ISBN 3-540-60676-9

Mitchell M (1996) An introduction to genetic algorithms. MIT Press, Cambridge. ISBN 0-262-13316-4

Mitchell M, Forrest S, Holland JH (1992) The royal road for genetic algorithms: fitness landscapes and ga performance. In: Proceedings of the first European conference on artificial life, pp 245–254. The MIT Press, Cambridge

Mohaghegh SD (2005) A new methodology for the identification of best practices in the oil and gas industry, using intelligent systems. J Pet Sci Eng 49(3):239–260

Mohaghegh SD et al (2005) Recent developments in application of artificial intelligence in petroleum engineering. J Pet Technol 57(4):86–91

Mullen KM, Ardia D, Gil DL, Windover D, Cline J (2011) DEoptim: an R package for global optimization by differential evolution. J Stat Softw 40(6):1–26

Nateri K Madavan (2002) Multiobjective optimization using a pareto differential evolution approach. In: Proceedings of the world on congress on computational intelligence, vol 2. IEEE, pp 1145–1150

Nguyen NT, Kowalczyk R (2012) Transactions on computational collective intelligence III. Springer, Berlin

Nwankwor E, Nagar AK, Reid DC (2013) Hybrid differential evolution and particle swarm optimization for optimal well placement. Comput Geosci 17(2):249–268

Onwunalu JE, Durlofsky LJ (2010) Application of a particle swarm optimization algorithm for determining optimum well location and type. Comput Geosci 14(1):183–198

Park H-Y, Datta-Gupta A, King MJ (2014) Handling conflicting multiple objectives using pareto-based evolutionary algorithm during history matching of reservoir performance. J Pet Sci Eng

Piotrowski AP, Osuch M, Napiorkowski MJ, Rowinski PM, Napiorkowski JJ (2014) Comparing large number of metaheuristics for artificial neural networks training to predict water temperature in a natural river. Comput Geosci 64:136–151

Poli R, Kennedy J, Blackwell T (2007) Particle swarm optimization. Swarm Intell 1(1):33–57

Poli R, Langdon WB, McPhee NF (2008) A field guide to genetic programming. http://www.gp-field-guide.org.uk. (With contributions by JR Koza)

Poormirzaee R, Moghadam RH, Zarean A (2014) Inversion seismic refraction data using particle swarm optimization: a case study of Tabriz, Iran. Arab J Geosci 1–9

Radcliffe NJ, Surry PD, Jz E (1995) Fitness variance of formae and performance prediction. In: Foundations of genetic algorithms, pp 51–72

Raidl GR, Gottlieb J (2005) Empirical analysis of locality, heritability and heuristic bias in evolutionary algorithms: a case study for the multidimensional knapsack problem. Evol Comput 13(4):441–475

Rana S, Jasola S, Kumar R (2011) A review on particle swarm optimization algorithms and their applications to data clustering. Artif Intell Rev 35(3):211–222

Rechenberg I (1973) Evolutionsstrategie: optimierung technischer systeme nach prinzipien der biologischen evolution. In: Frommann-Holzboog

Rechenberg I (1973) Evolutionstrategie: optimierung Technisher Systeme nach Prinzipien der Biologischen Evolution. Frommann-Holzboog Verlag, Stuttgart

Riget J, Vesterstrøm JS (2002) A diversity-guided particle swarm optimizer-the ARPSO. Department of Computer Science, University of Aarhus, Aarhus, Denmark, Technical Report, vol 2. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.2929

Safarzadeh MA, Motahhari SM (2014) Co-optimization of carbon dioxide storage and enhanced oil recovery in oil reservoirs using a multi-objective genetic algorithm (NSGA-II). Pet Sci 11 (3):460–468

Schwefel H-PP (1993) Evolution and optimum seeking. Wiley, Hoboken

Scrucca L (2013) GA: a package for genetic algorithms in R. J Stat Softw 53(4):1–37. http://www.jstatsoft.org/v53/i04/

Shakhsi-Niaei M, Iranmanesh SH, Torabi SA (2013) A review of mathematical optimization applications in oil-and-gas upstream & midstream management. Int J Energy Stat 1 (02):143–154

Shaw R, Srivastava S (2007) Particle swarm optimization: a new tool to invert geophysical data. Geophysics 72(2):F75–F83

Shelokar PS, Jayaraman VK, Kulkarni BD (2004) An ant colony approach for clustering. Analytica Chimica Acta 509(2):187–195

Shi Y, Eberhart RC (1998) Parameter selection in particle swarm optimization. In: EP'98: proceedings of the 7th international conference on evolutionary programming VII. Springer, London, pp 591–600. ISBN 3540648917

Simon HA (1969) The sciences of the artificial, vol 136. MIT Press, Cambridge

Singh HK, Ray T, Sarker R (2013) Optimum oil production planning using infeasibility driven evolutionary algorithm. Evolut Comput 21(1):65–82

Stoean R, Preuss M, Stoean C, El-Darzi E, Dumitrescu D (2009) Support vector machine learning with an evolutionary engine. J Oper Res Soc 60(8):1116–1122

Stoean C, Preuss M, Stoean R, Dumitrescu D (2010) Multimodal optimization by means of a topological species conservation algorithm. IEEE Trans Evolut Comput 14(6):842–864

Stoean R, Stoean C, Lupsor M, Stefanescu H, Badea R (2011) Evolutionary-driven support vector machines for determining the degree of liver fibrosis in chronic hepatitis C. Artif Intell Med 51:53–65. ISSN 0933-3657

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. ISSN 09255001. doi:10.1023/A:1008202821328

Sun J, Feng B, Xu W (2004) Particle swarm optimization with particles having quantum behavior. In Proceedings of the IEEE congress on evolutionary computation. IEEE Press, pp 325–331

Talbi E-G (2009) Metaheuristics: from design to implementation, vol 74. Wiley, Hoboken

Thander B, Sircar A, Karmakar GP (2014) Hydrocarbon resource estimation: a stochastic approach. J Pet Explor Prod Technol 1–8

Tronicke J, Paasche H, Böniger U (2012) Crosshole traveltime tomography using particle swarm optimization: a near-surface field example. Geophysics 77(1):R19–R32

Turney P (1995) Cost-sensitive classification: empirical evaluation of a hybrid genetic decision tree induction algorithm. J Artif Intell Res 2:369–409

Voß S (2001) Meta-heuristics: the state of the art. In: Local search for planning and scheduling. Springer, Berlin, pp 1–23

Wang L, Wang X, Fu J, Zhen L (2008) A novel probability binary particle swarm optimization algorithm and its application. J Softw 3(9):28–35

Whitley Darrell, Rana Soraya, Heckendorn Robert B (1998) The island model genetic algorithm: on separability, population size and convergence. J Comput Inf Technol 7:33–47

Wolpert DH, Macready WG (1997) No free lunch theorems for optimization. IEEE Trans Comput 1(1):67–82

Zaharie D (2005) Density based clustering with crowding differential evolution. In: International symposium on symbolic and numeric algorithms for scientific computing, pp 343–350

Zaharie D (2007) A comparative analysis of crossover variants in differential evolution. In: Proceedings of IMCSIT 2007, pp 171–181

Zangeneh H, Jamshidi S, Soltanieh M (2013) Coupled optimization of enhanced gas recovery and carbon dioxide sequestration in natural gas reservoirs: case study in a real gas field in the south of Iran. Int J Greenhouse Gas Control 17:515–522

Zitzler E, Deb K, Thiele L (2000) Comparison of multiobjective evolutionary algorithms: empirical results. Evol Comput 8:173–195