Chapter 8
Memetic Algorithms in Continuous Optimization

Carlos Cotta and Ferrante Neri

8.1 Introduction and Basic Concepts

Intuitively, a set is considered to be discrete if it is composed of isolated elements, whereas it is considered to be continuous if it is composed of infinite and contiguous elements and does not contain “holes”.

More formally, to introduce the concept of continuous optimization, some preliminary definitions are required. If we consider sub-sets of real numbers, where the partial order is obviously valid, a set S is said to be *dense* if

$$\forall x_1, x_2 \in S\colon \exists x_3\colon x_1 \leqslant x_3 \leqslant x_2. \tag{8.1}$$

If the property above is not satisfied for all the points, the set is said to be *discrete*. When the property is not satisfied for some of the points in D, the set is composed of multiple not interconnected dense sets.

It must be remarked that, since a set of infinite numbers cannot be represented in a machine, in computer science all the sets are in principle discrete. On the other hand, a set where the distance between each pair of consecutive points is not bigger than the machine precision ε can be considered as a dense set. In other words, the definition of a dense set in computer science can be modified in the following way. A set S is said to be *dense in computer science* if

$$\forall x_1, x_2 \in S, x_1 < x_2\colon \exists x_3 \in S\colon \left[(x_3 - x_1) \geqslant \varepsilon\right] \wedge \left[(x_2 - x_3) \geqslant \varepsilon\right]. \tag{8.2}$$

Carlos Cotta
Dept. de Lenguajes y Ciencias de la Computación. Universidad de Málaga,
Campus de Teatinos, 29071 Málaga, Spain
e-mail: ccottap@lcc.uma.es

Ferrante Neri
Department of Mathematical Information Technology, P.O. Box 35 (Agora),
40014 University of Jyväskylä, Finland
e-mail: ferrante.neri@jyu.fi

F. Neri et al. (Eds.): Handbook of Memetic Algorithms, SCI 379, pp. 121–134.
springerlink.com © Springer-Verlag Berlin Heidelberg 2012

A multidimensional set composed of the Cartesian product of multiple dense sets S is said to be decision space D. An optimization problem defined on a decision space D is said to be *continuous optimization problem*. More specifically, throughout this chapter we refer to the minimization problem of an objective function $f(x)$, where x is a vector of n design variables in a decision space D. In general, this optimization problem can be subject to a set of constraints. For the sake of simplicity, in this chapter we just consider the minimization within a hyper-rectangular space.

Although the concept of continuous optimization is strictly related to the concept of continuous functions, the two concepts should not be identified. According to the Cauchy definition, a continuous function is characterized by the following property: an infinitesimal change in the independent variable corresponds to an infinitesimal change of the dependent variable. The optimization of a continuous function is always a continuous optimization problem. The reverse statement is not not true. In computer science, even when the function displays discontinuity points still the resulting problem is continuous.

This chapter focuses on continuous optimization problems and on the application of Memetic Algorithms (MAs) in order to solve such problems. Section 8.2 highlights the difference between global and local optimization for continuous problems. Section 8.3 briefly illustrates a set of popular global optimizers which can be used as an evolutionary framework within a MA.

8.2 Global and Local Continuous Optimization

In discrete optimization, solutions are simply characterized by their fitness values. Thus, a solution can either be optimal or suboptimal. In continuous optimization, the situation is different as the position of each candidate solution within the decision space takes a high importance. It can intuitively be seen from the Cauchy definition of continuous function that for a given point its closest points are expected to have a similar performance with respect to the point. In this context, the concept of *neighborhood* is extremely important. For a given point, its neighborhood is that set of points characterized by distance ε from it. This concept is fundamental in continuous optimization because, unlike the discrete optimization case, it make sense to discuss about small movements and search directions. In other words, unlike what happens in the discrete case, in continuous optimization it makes sense to discuss about the *gradient* which can be redefined, along the generic variable x_i, for the "continuous discrete" case of computer science in the following way:

$$\frac{\partial f}{\partial x_i} = \frac{f(x_i + \varepsilon) - f(x_i)}{\varepsilon}. \quad (8.3)$$

If a gradient can be defined, it can be used from a starting point to select the most promising neighbor and thus to identify a promising search direction. The information derived from the knowledge of the gradient values can be obviously exploited within an optimizer. A major difference should be highlighted between the gradient defined above for continuous problems in computer science and the classical

gradient in mathematical analysis. While in mathematical analysis a null gradient corresponds to a critical point, i.e., a true local/global optimum, plateau or saddle point, in computer science a null gradient (according to the definition above) in a point means that the entire neighborhood of this point has the same fitness values; thus the point falls within a plateau. From this consideration, it is clear that the null gradient condition cannot be used in computer science to identify the true local/global optima (which is the goal of the optimization) but only plateaus. The detection of local optima should be performed in a different way: a point is a *local minimum*(maximum) if the objective function values of the neighborhood are all higher (lower) than that of the point.

Without a loss of generality, let us consider minimization problems. Usually optimization problems are multimodal, i.e., contain multiple local minima. However, the goal in optimization is to detect the global optimum, that is, in our case, the minimum exhibiting the lowest function value. All the methods that make an explicit or implicit use of gradient information tend to detect the closest local minimum. Thus, an efficient global optimizer should not be based only on gradient information but also on direct fitness comparisons among solutions regardless their position within the decision space. This approach guarantees an extensive search and hopefully allows that algorithms get stuck within local optima. In this context, it is important to define the concept of *basin of attraction*. Two definitions can be given in both a broad and restricted sense. In a broad sense, for a given search strategy, objective function, and starting point(s), a basin of attraction is the set of points which can be reached. However, when in a generic way computer scientists refer to the term basin of attraction without specifying the search strategy, it is meant that the specific search strategy is the classical deterministic hill-descender which perturbs separately each variable. Thus, a decision space can be mapped as a composition of basins of attraction and the goal of global optimization is to detect the globally optimum basin of attractions and avoid the local ones.

MAs in continuous optimization are thus thought as algorithmic structures which require both global and local search components whose coordination make the success of the computational paradigm. These structures are usually composed of an evolutionary framework which has the role of performing the global search and one or more local search algorithms activated within the generation cycle of the external framework.

8.3 Global Optimization Algorithms

While some local search algorithms have been previously illustrated, in this chapter some global search algorithms, which are shown as examples of evolutionary frameworks in MAs, are briefly presented in the following section.

8.3.1 Stochastic Global Search, Brute Force and Random Walk

The simplest (and often not so efficient) way to perform the global optimal search of a black box function is the progressive perturbation of one or more solutions in order to improve upon its performance. The search can be performed by various search rules, for example by generating a new solution within the decision space or by adding a randomized perturbation vector to a trial solution. This class of algorithms is often named Stochastic Global Search or simply Stochastic Search, see [838], and has the crucial importance of being the basic principle behind all the modern computational intelligence optimization algorithms. It must be observed that all the modern algorithms which take their inspiration on the most various natural sources, such as principles of the evolution or collective behavior of animals or even MAs, are at the end stochastic search algorithms which differ one from another on the trial solution generation mechanism or the strategy for retaining the solutions (and selecting the search directions).

In order to clarify this concept let us consider two classical global optimization algorithms which are based on completely opposite search logics. The first, namely brute force, consists of the construction of a regular grid within the decision space and the sample of the points in correspondence to the nodes. This algorithm has been taken into account in this context because it is a global search algorithm based on a fully deterministic generation of solutions. Another famous simple stochastic search is the random walk, see [337]. This algorithm perturbs each coordinate of a candidate solution by means of a Gaussian distribution. It can be immediately observed that the random walk is a highly randomized method as the trial search directions rely only on stochastic perturbations.

As an additional remark, although very different, these two methods are both plagued by the same problem: their performance highly depends on the parameter setting. In the brute force, the selection of step size, and thus amount of points to sample, must be carried out to avoid inefficient search or an unacceptable computational time. Likewise, in the random walk the success of the algorithm heavily depends on the mean value and standard deviation of the perturbation Gaussian. In other words, regardless the degree of randomization in the search logic, when there is no information on the objective function, the parameter setting becomes key point in the algorithmic performance.

8.3.2 Evolution Strategy and Real Coded Evolutionary Algorithms

In 70s, while Genetic Algorithms (GAs) were developed for discrete and combinatorial optimization problems [389], Evolution Strategy (ES) were developed for continuous optimization problems [760, 798]. In ES, each individual is a real-valued vector composed of its candidate solution representation x and a set of self-adaptive parameters σ:

$$(x,\sigma) = (x_1,\ldots,x_n,\sigma_1,\ldots,\sigma_n) \tag{8.4}$$

In many evolution strategy variants, a set of self-adaptive parameters of a second kind can be added to the solution encoding. At each generation cycle, parent selection relies on pseudo-randomly selecting some solutions to undergo recombination and mutation. In evolution strategies a big emphasis is placed on mutation while recombination sometimes plays a minor role (although it is not simply dismissed as in evolutionary programming) – see [65] for a in-depth treatment of these two operators in ES. The general mutation rule is defined, for the generic i^{th} design variable, by:

$$\sigma_i = \sigma_i e^{N(0,\tau')+N_i(0,\tau)} \tag{8.5}$$

and

$$x_i = N(x_i, \sigma_i) \tag{8.6}$$

where $N(\mu, \sigma)$ is normally a distributed random number with mean μ and standard deviation σ. The update of σ can be performed by means of several rules proposed in literature. The most famous are the 1/5 success rule [760], uncorrelated mutation with one step size, uncorrelated mutation with n step sizes and correlated mutation, for details see [239]. The method shown in Eq. (8.5) corresponds to uncorrelated mutations with n step sizes, and τ, τ' are two parameters (the local and the global learning rate respectively) that can be set as [32]:

$$\tau = 1/\sqrt{2\sqrt{n}} \tag{8.7}$$

$$\tau' = 1/\sqrt{2n} \tag{8.8}$$

The notation $N_i(0, \tau)$ is used to denoted a different random number for each parameter, whereas $N(0, \tau)$ is a common –solution-wise– random number. The general idea is that the solutions are mutated within their neighborhood based on a certain probabilistic criterion with the aim of generating new promising solutions.

The recombination can be discrete or intermediary: discrete recombination generates an offspring solution by pseudo-randomly choosing the genes from two parent solutions, while intermediary recombination generates an offspring whose genes are obtained by calculating a randomly weighted average of the corresponding genes of two parents (other methods are possible though – see Section 8.4).

The parent selection can be performed either in the genetic algorithm fashion by replacing the whole parent population with the best members of the offspring population or by merging parent and offspring populations and selecting the wanted number of individuals on the basis of their fitness values. These strategies are usually known as comma and plus strategy respectively.

In the 90s, a reorganization of the knowledge regarding evolution inspired metaheuristics was performed. This lead to the fact that GAs, ES, Evolutionary Programming and other branches of the field have all been seen as an expression of the same idea and named Evolutionary Algorithms (EAs). These algorithms, characterized by four phases, 1) parent selection, 2) crossover, 3) mutation, 4) survivor selection, can be implemented to both continuous and discrete optimization, by properly

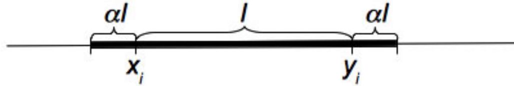

Fig. 8.1. Functioning of the BLX$-\alpha$ recombination operator. Offspring variable z_i is randomly sampled from the interval denoted by a thick line.

representing the solutions and their recombination. The most natural way to represent candidate solutions of a continuous optimization problem is simply to use them "as they are", i.e., have a representation of vectors of real numbers without any conversion (as in classical GAs where all the numbers were converted to binary).

A multitude of recombination strategies among pairs or small groups of solutions have been proposed in literature. The advantages of one strategy with respect to another are, in general, dependent on the problem. A very broadly used recombination strategy is the so called BLX$-\alpha$ crossover, see [246, 382]. For two given parent solutions x and y, their offspring z is generically calculated in the following way:

$$z_i = U[m_i - \alpha I, M_i + \alpha I] \tag{8.9}$$

where α is a parameter, $M_i = \max(x_i, y_i)$, $m_i = \min(x_i, y_i)$, $I = |x_i - y_i|$ and $U[a,b]$ is a uniformly distributed random number in the interval $[a,b]$. Parameter α is thus used to tune the explorative capability of crossover – see Fig. 8.1. A *parent centric* variant of BLX$-\alpha$ is also defined in [536] by sampling each offspring variable from a closed interval of radius $2\alpha I$ centered at any of the corresponding parental variables.

Precisely related to this exploration issue (or more generically to the avoidance of premature convergence), it is worth mentioning another EA variant that is commonly used as the population-based engine of continuous MA, namely the CHC (Cross generational elitist selection, Heterogeneous recombination, and Cataclysmic mutation) algorithm [246]. The main idea of this algorithm is to combine strong selective pressure with incest-prevention strategies and explorative recombination. The incest-prevention strategy amounts to avoiding that two very similar solutions are recombined (since this would likely produce very similar offspring as well, hence leading to diversity loss and potential premature convergence). To do so, a distance parameter δ is maintained, determining the minimal distance that must exist between two solutions if these are to be recombined. This parameter can change dynamically in order to cope with the progressive convergence of the population. As to the selection, it is typically done using the plus strategy of ES. Algorithm 16 shows the pseudocode of the CHC algorithm.

A final evolutionary approach for continuous optimization that deserves being mentioned is the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [362]. This algorithm falls within the class of estimation of distribution algorithms [534] (EDAs) and has been shown to be extremely efficient when solving continuous optimization benchmarks [28]. CMA-ES is based on generating solutions via a multivariate normal distribution whose mean and covariance matrix is adaptively

Algorithm 16. Pseudo-code of the CHC algorithm

```
1  begin
2      generate initial population P ← {p_1, ··· , p_μ};
3      initialize distance parameter δ;
4      while ¬ termination-condition do
5          create solutions pairs S ← (p_i, p_j);
6          P' ← ∅;
7          for (p, p') ∈ S do
8              d ←distance(p, p');
9              if d ⩽ δ then
10                 p'' ←recombine(p, p');
11                 P' ← P' ∪ {p''};
12             endif
13         endfor
14         P ←plus-select(P, P');
15         if P' = ∅ then
16             decrease δ;
17             if δ < 0 then
18                 restart population P;
19                 initialize distance parameter δ;
20             endif
21         endif
22     endw
23 end
```

learnt as in EDAs, i.e., utilizing truncation selection to pick a subset of the best solutions generated in each step, and using these solutions to update the distribution parameters. CMA-ES has a solid theoretical background and several desirable properties such as invariance to several transformations of the objective function and a relatively low number of parameters. Furthermore, it can not only serve as a population-based engine but also as a local searcher if adequately parameterized, e.g., $(1+1)$-CMA-ES [605], We refer to [360] for further details and source code of the CMA-ES algorithm.

8.3.3 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a population-based optimization metaheuristic introduced in [458], and then developed in various variants for test problems and applications. The main metaphor employed in PSO is that a group of particles makes use of their “personal” and “social” experience in order to explore a decision space and detect solutions with a high performance. More specifically, a population of candidate solutions is randomly sampled within the decision space. Subsequently, the fitness value of each candidate solution is computed and the solutions are ranked on the basis of their performance. The solution associated to the best fitness value detected overall is named global best x^{gb}. At the first generation, each solution x_i

is identified with the corresponding local best solution x_i^{lb}, i.e., the most successful value taken in the history of each solution. At each generation, each solution x_i is perturbed by means of the following rule:

$$x_i = x_i + v_i \tag{8.10}$$

where the velocity vector v_i is a perturbation vector generated in the following way:

$$v_i = \omega v_i + \alpha_1 (x_i^{lb} - x_i) + \alpha_2 (x^{gb} - x_i) \tag{8.11}$$

where ω is the so-called inertia parameter (the higher this parameter, the longer it takes the particle to change direction), and α_1, α_2 are two parameters that control the attraction the particle feels towards the best-known local/global solutions. These are typically set uniformly at random –within the interval $(0,1)$, i.e., 0 excluded and 1 included– in each step; we denote as $U(0,1)$ as such a uniform distribution. The fitness value of the newly generated x_i is calculated and if it outperforms the previous local best value the value of x_i^{lb} is updated. Similarly, if the newly generated solution outperforms the global best solution, a replacement occurs. At the end of the optimization process, the final global best detected is the estimate of the global optimum returned by the particle swarm algorithm. It is important to remark that in PSO, there is a population of particles which has the role of exploring the decision space and a population of local best solutions (the global best is the local best with the highest performance) to keep track of the successful movements.

In order to better understand the metaphor and thus the algorithmic philosophy behind PSO, the population can be seen as a group of individuals which search for the global optimum by combining the exploration along two components: the former is the memory and thus learning due to successful and unsuccessful moves (personal experience) while the latter is a partial imitation of the successful move of the most promising individual (social experience). In other words, as shown in the formula above, the perturbation is obtained by the vectorial sum of a move in the direction of the best overall solution and a move in the direction of the best success achieved by a single particle. These directions in modern PSO algorithms are weighted by means of random scale factors, since the choice has to turn out to be beneficial in terms of diversity maintenance and prevention of premature convergence.

Many versions and variants of PSO have been proposed in literature in order to attempt to enhance its performance. In order to give a flavor of possible PSO modifications, two examples are here reported. A popular variant is the linearly variable weight factor ω proposed in [809]:

$$\omega = \omega_{max} - (\omega_{max} - \omega_{min}) \frac{g}{G} \tag{8.12}$$

where g is the current generation and G is the generation budget. Parameters ω_{max} and ω_{min} are usually set equal to 0.9 and 0.4, respectively.

Algorithm 17. PSO pseudo-code

```
1  begin
2      generate N_p particles and N_p velocities pseudo-randomly;
3      copy the population of particles into the set of local bests: ∀i, x_{i-lb} = x_i ;
4      while budget condition do
5          for i = 1 : N_p do
6              compute f(x_i);
7          endfor
8          for i = 1 : N_p do
               // ** Velocity Update **
9              generate a vector of random numbers U(0,1);
10             v_i = ωv_i + U(0,1)(x_i^{lb} − x_i) + U(0,1)(x^{gb} − x_i);
               // ** Position Update **
11             x_i = x_i + v_i;
               // ** Survivor Selection **
12             if f(x_i) ⩽ f(x_{i-lb}) then
13                 x_i^{lb} = x_i;
14                 if f(x_i) ⩽ f(x^{gb}) then
15                     x^{gb} = x_i;
16                 endif
17             endif
18         endfor
19     endw
20 end
```

Another variant is the constriction factor proposed in [129]. Within such a scheme the velocity update is:

$$v_i = \chi v_i + c_1 U(0,1)\left(x_i^{lb} - x_i\right) + c_2 U(0,1)\left(x_i^{nb} - x_i\right) \tag{8.13}$$

where x_i^{nb} is the best within the neighborhood (see for details [129]). The constriction factor χ is defined as:

$$\chi = \frac{2}{\left|2 - \phi - \sqrt{\phi^2 - 4\phi}\right|} \tag{8.14}$$

where $\phi = c_1 + c_2 = 4.1$ and $c_1 = c_2 = 2.05$, see [129]. A pseudo-code showing the main features of the basic PSO is given in Algorithm 17.

8.3.4 Differential Evolution

Differential Evolution (DE) is an interesting optimizer for continuous problems which shares some properties of evolutionary algorithms (e.g., the crossover) and some others of swarm intelligence algorithms (the one-to-one replacement). According to its original definition given in [853], DE consists of the following steps.

An initial sampling of S_{pop} individuals is performed pseudo-randomly with a uniform distribution function within the decision space D. At each generation, for each individual x_i from the S_{pop} in the population, three mutually distinct individuals x_r, x_s and x_t are pseudo-randomly extracted from the population. According to DE logic, a provisional offspring x'_{off} is generated by mutation as:

$$x'_{off} = x_t + F(x_r - x_s) \tag{8.15}$$

where $F \in [0, 1^+[$ is a scale factor which controls the length of the exploration vector $(x_r - x_s)$ and thus determines how far from point x_i the offspring should be generated. With $F \in [0, 1^+[$, it is meant here that the scale factor should be a positive value which cannot be much greater than 1, see [733]. While there is no theoretical upper limit for F, effective values are rarely greater than 1.0. The mutation scheme shown in Eq. (8.15) is also known as DE/rand/1. Other variants of the mutation rule have been subsequently proposed in literature, see [745]:

- DE/best/1: $x'_{off} = x_{best} + F(x_s - x_t)$
- DE/cur-to-best/1: $x'_{off} = x_i + F(x_{best} - x_i) + F(x_s - x_t)$
- DE/best/2: $x'_{off} = x_{best} + F(x_s - x_t) + F(x_u - x_v)$
- DE/rand/2: $x'_{off} = x_r + F(x_s - x_t) + F(x_u - x_v)$
- DE/rand-to-best/2: $x'_{off} = x_r + F(x_{best} - x_i) + F(x_r - x_s) + F(x_u - x_v)$

where x_{best} is the solution with the best performance among individuals of the population, x_u and x_v are two additional pseudo-randomly selected individuals. It is worthwhile to mention the rotation invariant mutation shown in [732]:

- DE/current-to-rand/1 $x_{off} = x_i + K(x_t - x_i) + F'(x_r - x_s)$

where K is is the combination coefficient, which as suggested in [732] should be chosen with a uniform random distribution from $[0, 1]$ and $F' = K \cdot F$. For this special mutation the mutated solution does not undergo the crossover operation (since it already contains the crossover), described below.

Recently, in [733], a new mutation strategy has been defined. This strategy, namely DE/rand/1/either-or, consists of the following:

$$x'_{off} = \begin{cases} x_t + F(x_r - x_s) & \text{if } U(0,1) < p_F \\ x_t + K(x_r + x_s - 2x_t) & \text{otherwise} \end{cases} \tag{8.16}$$

where for a given value of F, the parameter K is set equal to $0.5(F+1)$.

When the provisional offspring has been generated by mutation, each gene of the individual x'_{off} is exchanged with the corresponding gene of x_i with a uniform probability and the final offspring x_{off} is generated:

$$x_{off,j} = \begin{cases} x_{i,j} & \text{if } U(0,1) < CR \\ x'_{off,j} & otherwise \end{cases} \tag{8.17}$$

Algorithm 18. DE/rand/1/bin pseudo-code

```
1  begin
2      generate N_p individuals of the initial population pseudo-randomly;
3      while budget condition do
4          for k = 1 : N_p do
5              compute f(x_k);
6          endfor
7          for k = 1 : N_p do
               // ** Mutation **
8              select three individuals x_r, x_s, and x_t;
9              compute x'_off = x_t + F(x_r - x_s);
               // ** Crossover **
10             x_off = x'_off;
11             for i = 1 : n do
12                 generate U(0,1);
13                 if U(0,1) > Cr then
14                     x_off[i] = x_k[i];
15                 endif
16             endfor
               // ** Survivor Selection **
17             if f(x_off) <= f(x_k) then
18                 save index for replacement x_k = x_off;
19             endif
20         endfor
21         perform replacements;
22     endw
23 end
```

where $U(0,1)$ is a random number between 0 and 1; j is the index of the gene under examination. This crossover strategy is well-known as binary crossover and indicated as “bin”. For the sake of completeness, we mention that there exist a few other crossover strategies, for example the exponential strategy see [733]. However in this paper we focus on the bin strategy since it is the most commonly used and often the most promising.

The resulting offspring x_{off} is evaluated and, according to a one-to-one spawning strategy, it replaces x_i if and only if $f(x_{off}) \leqslant f(x_i)$; otherwise no replacement occurs. For sake of clarity, the pseudo-code highlighting the working principles of DE is shown in Algorithm 18.

8.4 Particularities of Memetic Approaches for Continuous Optimization

In principle the deployment of memetic algorithms on continuous domains can be done using the generic algorithmic template presented in Chapter 4, much like it is

done for combinatorial problems – see Chapter 6. This said, continuous optimization problems have several distinctive features that must be considered in order to come up with efficient memetic solvers. Two of the most relevant ones are:

- *The cost of local search:* in many combinatorial domains it is frequently possible to compute the fitness of a perturbed solution incrementally, e.g., let x be a solution and let $x' \in \mathcal{N}(x)$ be a neighboring solution; then the fitness $f(x')$ can be often computed as $f(x') = f(x) + \Delta f(x,x')$, where $\Delta f(x,x')$ is a term that depends on the particular perturbation done on x and is typically efficient to compute (much more efficiently that a full fitness computation). For example, in the context of the traveling salesman problem and the 2-opt neighborhood, the fitness of a perturbed solution can be computed in constant time by calculating the difference between the weights of the two edges added and the two edges removed. This is much more difficult in the context of continuous optimization problems, which are often non-linear and hard to decompose as the sum of linearly-coupled terms. Hence local search usually has to resort to full fitness computations.
- *The underlying search landscape:* the interplay among the different search operators used in memetic algorithms (or even in simple evolutionary algorithms) is a crucial issue for achieving good performance in any optimization domain. When tackling a combinatorial problem, this interplay is a complex topic since each operator may be based on a different search landscape. It is then essential to understand these different landscape structures and how they are navigated – the "one operator, one landscape" view [434]. In the continuous domain the situation is somewhat simpler, in the sense that there exists a natural underlying landscape in D^n (typically $D = \mathbb{R}$), that is induced by distance measures such as Euclidian distance. In other words, neighborhood structures are defined by closed spheres of radius ε in the case of unary operators, and by solid hypercubes in the case of recombination (recall for example the BLX$-\alpha$ operator). The intuitive imagery of local optima and basins of attraction naturally fits here, and allows the designer to exert some control on the search dynamics by carefully adjusting the intensification/diversification properties of the operators used.

These two issues mentioned above have been dealt in the literature on memetic algorithms for continuous optimization in different ways. Starting with the first one (the cost of local search), it emphasizes the need for carefully selecting when and how local search is applied (obviously this is a general issue, also relevant in combinatorial problems, but definitely crucial in continuous ones). Needless to say, this decision-making is very hard in general [494, 857], see also Chapter 5, but some strategies have been put forward in previous works. A rather simple one is to resort to partial Lamarckianism [396] by randomly applying local search with probability $p_{LS} < 1$. Obviously, the application frequency is not the only parameter that can be adjusted to tune the computational cost of local search: the intensity of local

search (i.e., for how long is local improvement attempted on a particular solution) is another parameter to be tweaked. This adjustment can be done blindly (i.e., prefixing a constant value or a variation schedule across the run), or adaptively. For example, Molina et al. [605] define three different solution classes (on the basis of fitness) and associate a different set of local-search parameters for each of them. Related to this, Nguyen *et al.* [665] consider a stratified approach, in which the population is sorted and divided into n levels (n being the number of local search applications), and one individual per level is randomly selected. This is shown to provide better results than random selection. We refer to [40] for an in-depth empirical analysis of the time/quality tradeoffs when applying parameterized local search within memetic algorithms. This adaptive parameterization has been also exploited in so-called *local-search chains* [608], by saving the state of the local-search upon completion of a certain solution for later use if the same solution is selected again for local improvement. Let us finally note with respect to this parameterization issue that adaptive strategies can be taken one step further, entering into the realm of self-adaptation. An overview of the possibilities to this end is provided in Chapter 11.

As to what the exploitation/exploration balance regards, it is typically the case that the population-based component is used to navigate through the search space, providing interesting starting points to intensify the search via the local improvement operator. The diversification aspect of the population-based search can be strengthened in several ways, such as for example using multiple subpopulations [640], or diversity-oriented replacement strategies. The latter are common in scatter search [320] (SS), an optimization paradigm closely related to memetic algorithms in which the population (or reference set in the SS jargon) is divided in tiers: entrance to them is gained by solution on the basis of fitness in one case, or diversity in the other case. Additionally, SS often incorporated restarting mechanisms to introduce fresh information in the population upon convergence of the latter. Diversification can be also introduced via selective mating, as it is done in CHC (see Sect. 8.3.2). A related strategy was proposed by Lozano et al. [536] via the use of negative assortative mating: after picking a solution for recombination, a collection of potential mates is selected and the most diverse one is used. Other strategies range from the use of clustering [806] (to detect solutions likely within the same basin of attraction upon which it may not be fruitful to apply local search), or the use of standard diversity preservation techniques in multimodal contexts such as sharing or crowding. It should also be mentioned that sometimes the intensification component of the memetic algorithm is strongly imbricated in the population-based engine, without resorting to a separate local search component. This is for example the case of the so-called *crossover hill climbing* [432], a procedure which essentially amount to using a hill climbing procedure on states composed of a collection of solutions, using crossover as move operator (i.e., introducing a newly generated solution in the collection –substituting the worst one– if the former is better than the latter). This strategy was used in the context of real-coded memetic algorithms in [536]. A different intensifying strategy was used by [161], by considering an exact procedure for finding the best combination of variable values from the parents (a so-called

optimal discrete recombination). This obviously requires that the objective function is amenable to the application of an efficient procedure for exploring the dynastic potential (set of possible children) of the solutions being recombined – see also Chapter 12. We refer to [535] for a detailed analysis of diversification/intensification strategies in hybrid metaheuristics (in particular in memetic algorithms).

Acknowledgements. C. Cotta is partially supported by Spanish MICINN under project NEMESIS (TIN2008-05941) and by Junta de Andalucía under project TIC-6083. This research is supported by the Academy of Finland, Akatemiatutkija 130600, Algorithmic Design Issues in Memetic Computing.