Chapter 6
Memetic Algorithms in Discrete Optimization

Jin-Kao Hao

6.1 Introduction

Discrete optimization concerns in essence the search for a “best” configuration (optimal solution) among a set of *finite* candidate configurations according to a particular criterion. There are several ways to describe a discrete optimization problem. In its most general form, it can be defined as a collection of problem instances, each being specified by a pair (S, f) [704], where S is the set of finite candidate configurations, defining the *search space*; f is the *cost* or *objective function*, given by a mapping $f\colon S \rightarrow R^+$.

Solving the instance (S, f) is to find an $s^* \in S$ such that $f(s^*) \leqslant f(s)$ for all $s \in S$ (this minimization formulation can easily be transformed into a maximization problem). Such a configuration s^* is a globally optimal solution (or simply an optimal solution) to the given instance.

Given its generality, discrete optimization allows many problems of practical and theoretical importance to be conveniently formulated. Examples are the classical problems of general integer programming, permutation problems (e.g., traveling salesman problem, bandwidth minimization, linear arrangement), and constraint satisfaction and optimization problems (satisfiability problems in propositional logic, graph partitioning, k-coloring). Discrete optimization naturally covers practical problems of the environment, renewable energy, distribution, infrastructure design, communications and productivity in the manufacturing and service sectors.

However, discrete optimization problems are known to be difficult to solve in general. Most of them, in particular those of practical interest, belong to the class of NP-hard problems, and thus cannot be efficiently solved to optimality. Over the past decades, important efforts have been made to improve the solution methods and important progresses have been achieved in both exact and heuristic strategies in pursuit of optimal or near optimal solutions.

Jin-Kao Hao
LERIA, Université d’Angers, 2 Boulevard Lavoisier, 49045 Angers Cedex 01, France
e-mail: jin-kao.hao@univ-angers.fr

This chapter concerns the design of Memetic Algorithms (MAs) [615, 617] for finding optimal or high quality near optimal solutions to hard discrete optimization problems.

6.2 Survey of Memetic Algorithms for Discrete Optimization

6.2.1 Rationale

From a fundamental point of view, the task of searching for a best solution in a combinatorial space is all about a suitable balance between "exploitation" and "exploration" for an effective examination of the given search space. The dual concept of exploitation and exploration covers two fundamental and complementary aspects of any effective search procedure. This concept is also known under the term "intensification" and "diversification" introduced within the Tabu Search (TS) methodology [317].

Exploitation emphasizes the ability of a method to examine intensively and in depth specific search areas while exploration is the ability of a method to diversify the search in order to find promising new search areas. Consequently, if the search focuses solely on exploitation, it will confine itself in a limited area, fails to visit other areas of the search space, and may be trapped in poor optima. On the other hand, a method relying heavily on exploration and overlooking exploitation will lack capacity to examine in depth a given area and miss out solutions of good quality. To be effective, a search method thus needs to appropriately conciliate exploitation and exploration. Memetic Algorithms constitute a very interesting framework offering a variety of strategies and mechanisms to achieve this general objective.

MAs are hybrid search methods that are based on the population-based search framework [35, 239] and neighborhood-based local search framework (LS) [393]. Popular examples of population-based methods include Genetic Algorithms and other Evolutionary Algorithms while Tabu Search and Simulated Annealing (SA) are two prominent local search representatives. The basic rationale behind a MA is to combine these two different search methods in order to take advantage of their complementary search strategies. Indeed, it is generally believed that the population-based search framework offers more facilities for exploration while neighborhood search provides more capabilities for exploitation. If they are combined in a suitable way, the resulting hybrid method can then offer a good balance between exploitation and exploration, assuring a high search performance.

Like other metaheuristics, MAs are a general optimization framework that can potentially be applied to various discrete search or optimization problems. Nevertheless, it should be clear that a blind application of MAs (or any other metaheuristics) to a particular problem will not be able to lead to satisfactory solutions. To be effective, the MA framework must be carefully adapted to the given problem and integrate problem-specific knowledge within its search operators and strategies. This is the key point of a successful MA application in practice.

6.2.2 Memetic Algorithms in Overview

Memetic Algorithms [615, 617] are a population-based computational framework and share a number of features with methods like Evolutionary Algorithms [35, 239], and Scatter Search [320]. MAs operate on a set of candidate solutions and use these solutions to create new solutions by applying variation operators such as combinations and local improvements.

From a general perspective, a MA is composed of a number of basic components: a pool of candidate solutions (also called population of individuals) to sample the search space, a combination operator (crossover) to create new candidate solutions (offspring) by blending two or more existing solutions, an improvement operator to ameliorate offspring solutions, and a population management strategy. In addition to these elements, the MA also needs an evaluation or fitness function to assess the quality of each candidate solution as well as a selection mechanism to determine the candidate solutions that will survive and undergo variations.

From an operational perspective, a typical MA starts with an initial population (see §6.3.4) and then repeats cycles of evolution. Each cycle, also called a generation, consists of four sequential steps.

1. *Selection of parents*: Selection aims to determine the candidate solutions that will survive in the following generations and be used to create new solutions. Selection for reproduction often operates in relation with the fitness (quality) of the candidate solutions; high quality solutions have thus more chances to be chosen. Well-known examples of selection strategies include roulette-wheel and tournament. Selection can also be done according to other criteria such as diversity. In such a case, only "distanced" individuals are allowed to survive and reproduce. If the solutions of the population are sufficiently diversified, selection can also be carried out randomly. The selection strategy influences the diversity of the population (see also §6.3.3).

2. *Combination of parents for offspring generation*: Combination aims to create new *promising* candidate solutions by blending (suitably) existing solutions (parents), a solution being promising if it can potentially lead the optimization process to new search areas where better solutions may be found. To achieve this, the combination operator is often designed such that it captures the semantics of the targeted problem to ensure the heritage of good properties from parents to offspring. Additionally, the design of the combination operator should ideally take care of creating *diversified* offspring. From a perspective of exploration and exploitation, such a combination is intended to play a role of strategic diversification with a long term goal of reinforcing the intensification. A carefully designed combination operator constitutes a driving force of a successful MA.

3. *Local improvement of offspring*: The goal of local improvement is to improve the quality of an offspring as far as possible. For this purpose, local improvement takes an offspring as its input (current solution) and then iteratively

Algorithm 10. Memetic Algorithm Template

```
1 Input: $|P|$; // Size of population $P$
2 Output: $s^*$; // Best solution found
3 $P \leftarrow$ PopGeneration($|P|$); //
4 PopEvaluation($P$); // Fitness evaluation of each individual
5 $s^* \leftarrow best(P)$; // Record the best solution found so far
6 $f^* \leftarrow f(s^*)$; // Record the fitness of the best solution
7 while Stop Condition is not verified do
8     $(p_1 ... p_k) \leftarrow$ ParentsSelection($P$); // $k \geqslant 2$ parents are selected
9     $s' \leftarrow$ Recombination($p_1 ... p_k$); // Offspring generation
10    $s \leftarrow$ OffspringImprovement($s'$); // Improvement of offspring
          solution by local search
11    $P \leftarrow$ PopulationUpdate($s$,$P$); // Population update according
          to a quality-diversity rule
12    $(s^*,f^*) \leftarrow$ BestSolutionUpdate($s^*$,$f^*$,$P$); // Best solution and its
          fitness are always recorded
13 endw
14 return $s^*$
```

replaces the current solution by another solution taken from a given neighborhood. This process stops and returns the best solution found when a user-defined stop condition is met. Compared with the combination operator, local improvement plays essentially the role of intensifying the search by exploiting search paths delimited by the underlying neighborhood. Like combination, local improvement is another key component and driving force of a MA.

4. *Update of the population*: This step decides whether a new solution should become a member of the population and which existing solution of the population should be replaced. Often, these decisions are made according to criteria related to both quality and diversity. Such a strategy is commonly employed in methods like Scatter Search and many Evolutionary Algorithms. For instance, a basic quality-based updating rule would replace the worst solution of the population while a diversity-based rule would substitute for a similar solution according to a distance metric. Other criteria like recency (age) can also be considered. The policies employed for managing the population are essential to maintain an appropriate diversity of the population, to prevent the search process from premature convergence, and to help the algorithm to continually discover new promising search areas.

The general MA template is described in Algorithm 10 where special attention must be payed to the design of particular components. The stop condition can be a maximum number of cycles (generations), a maximum number of evaluations, a maximum number of cycles without improving the best solution, a solution quality to be reached or a lower-bounded threshold for the population diversity.

We deliberately leave out the mutation operator within this MA template. In some sense, local search can be viewed as a guided macro-mutation operator. However, mutation can also be applied to reinforce population diversity. As a lean design principle, only necessary components are included in a MA, any unjustified and superficial elements must be excluded.

6.2.3 Performance of Memetic Algorithms for Discrete Optimization

The computational performance of a MA depends first on the representation of the solution space (solution encoding) which should preferably be problem dependent and ease the design of efficient search operators.

The performance of a MA depends then on the design of its two key search components: Combination and local improvement operators. Their design should integrate useful problem-specific knowledge of the given problem in order to ensure aggressive exploitation and guided exploration.

The performance of a MA is also conditioned by the way the population is managed to promote and maintain a fertile diversity during the search process. Indeed, much like conventional Evolutionary Algorithms, premature convergence can easily occur if the population loses its diversity. Diversity management is particularly important with MAs because of the specific nature of their aggressive and intensified search strategies. Consequently, it is crucial for a MA to maintain with rigor a "good" population diversity as long as possible.

The interaction between the components of a MA can directly influence the behavior and the performance of the MA. A long or short local search phase after each combination could change the search trajectories. Similarly, a very effective local search procedure may weaken the role of the combination operator while a very strong combination operator may make it less critical to have a highly efficient local improvement procedure.

Finally, the runtime efficiency of a MA depends for a large part on the choice of the data structures employed to implement the different components of the MA. A typical example concerns local improvement procedures that explore the candidate solutions of a neighborhood and represent the most time-consuming part of a MA. In such a situation, it is critical to devise appropriate data structures to enable and streamline a fast neighborhood evaluation (see §6.3.1.3). Otherwise, the computational overheads will jeopardize the search power of the method.

6.3 Special Design Considerations

6.3.1 Design of Dedicated Local Search

Local improvement is one of the most important components of a MA and ensures essentially the role of intensive exploitation of the search space. This is typically achieved either by dedicated local search heuristics (see examples in [460, 523, 524]) or by tailored general neighborhood search methods. In this part, we focus our discussion on adaptation of local search metaheuristics [393], but a large part of the discussion applies to the design of local improvement procedures based on specific heuristics.

6.3.1.1 Local Search Template

Let (S, f) be our search problem where S and f are respectively the search space and optimization objective. A neighborhood N over S is any function that associates to each solution $s \in S$ some other solutions $N(s) \subset S$. Any solution $s' \in N(s)$ is called a neighboring solution or simply a neighbor of s. For a given neighborhood N, a solution s is a *local optimum* with respect to N if s is the best in terms of f among the solutions in $N(s)$.

The notion of neighborhood can be explained in terms of the *move* operator. Typically applying a move mv to a solution s changes s slightly and leads to a neighboring solution s'. This transition from a solution to a neighbor is denoted by $s' = s \oplus mv$. Let $\Gamma(s)$ be the set of all possible moves which can be applied to s, then the neighborhood $N(s)$ of s can be defined by: $N(s) = \{s \oplus mv | mv \in \Gamma(s)\}$.

A typical local search algorithm begins with an initial configuration s in S and proceeds iteratively to visit a series of configurations following the neighborhood. At each iteration, a particular neighbor $s' \in N(s)$ is sought to replace the current configuration and the choice of s' is determined by the underlying metaheuristic and by referring to the quality of the neighboring solution. For instance, a strict Descent algorithm always replaces the current solution s by a *better* neighbor s' while tabu search replaces the current solution by a *best* neighbor s' even if the latter is of inferior quality. Still with simulated annealing, the transition from s to a randomly selected neighbor s' is conditioned by a changing probability.

6.3.1.2 Neighborhood Design

The success of a LS algorithm depends strongly on its neighborhood. The neighborhood defines the subspace of the search problem to be explored by the method. For a given problem, the definition of the neighborhood should structure the search space such that it helps the search process to find its way to good solutions.

The choice of neighborhood is conditioned by the representation (genotype) used to encode the candidate solutions of the search space (phenotype). It may further depend on the structure and constraints of the problem on hand. Here we briefly review some neighborhoods associated to three conventional representations, which have a variety of applications.

- *Binary representation*: With this representation, each solution of the search space is coded by a binary string. Binary representation is very popular in discrete optimization due to the fact that many problems are naturally formulated with binary variables. Typical examples include SAT/Max-SAT, Knapsack, Unconstrained Quadratic Optimization, graph bi-partitioning etc. For these binary problems, two basic neighborhoods are defined by the $k\text{-}flip$ and *Swap* move operators. The $k\text{-}flip$ move changes the values of k ($k \geqslant 1$) variables. So any neighbor $s' \in N(s)$ has a Hamming distance of k to solution s. A larger k induces a larger (and stronger) neighborhood. Nevertheless, whether a larger neighborhood should be preferred in practice depends on the computational cost to evaluate the neighborhood. *Swap* exchanges the values of two variables that have different values. Note that *Swap* can be simulated by two 1-*flip* moves.

- *Permutation representation*: Here, each solution of the search space corresponds to a permutation $\pi : \{1..n\} \rightarrow \{1..n\}$. Permutation representation has a large range of applications in discrete optimization. Prominent examples include Traveling Salesman Problem, Flow-Shop/Job-Shop scheduling, Linear Arrangement, Bandwidth Minimization etc. Two basic neighborhoods for this representation are available using *Swap* and *Rotation* moves. Given a permutation (solution) π, The *Swap* move exchanges $\pi(i)$ and $\pi(j)$ for some i and j $(i \neq j)$. If π' is a neighbor of π by swapping i and j, then $\pi'(k) = \pi(k)$ for $k \neq i, j$, $\pi'(i) = \pi(j)$ and $\pi'(j) = \pi(i)$. The *Rotation* move rotates all the values between $\pi(i)$ and $\pi(j)$ for some $i < j$. Thus, if π' is a rotation neighbor of π obtained with $i < j$, then $\pi'(k) = \pi(k) + 1$ for $i \leqslant k < j$, $\pi'(j) = \pi(i)$, and $\pi'(k) = \pi(k)$ for all other k. Note that $Rotation(i, j)$ can be simulated by $j - i$ successive *Swap* moves starting with $Swap(i, i+1)$.

- *Integer representation*: With this representation, each solution of the search space corresponds to an integer vector whose values are taken from some discrete domains. Integer representation is very useful and convenient for many constraint satisfaction and optimization problems. A common neighborhood is defined by a "one-change" move that consists in replacing the current value of a single variable by a new domain value. The set of candidate variables under consideration for a value change can be identified with a number of rules specific to the problem at hand. For instance, if the search algorithm deals with unfeasible solutions, i.e. some variables are receiving conflicting values relative to some constraints, the set of candidate variables can be constituted of the subset of *conflicting* variables [289, 291, 672]. Such a neighborhood is typically employed in local search algorithms for solving Constraint Satisfaction Problems. More generally, candidate variables for a value change can be identified as those that are critical for improving the objective function or for reaching the feasibility.

These neighborhoods can be applied directly to a given problem if the problem fits well the required representation. A common practice is to adapt a conventional neighborhood with problem-specific knowledge. Moreover, in some situations, it is useful to investigate the possibility of multiple neighborhoods that can be applied at different stages of the search process (see §6.3.1.4 below).

6.3.1.3 Neighborhood Evaluation

Another design issue that arises is the evaluation of a given neighborhood. Indeed, a local search procedure moves iteratively from the current solution to a new solution chosen within the neighborhood. To make this choice, local search needs to know the cost variation (also called the *move value*) between the current solution s and a candidate neighbor $s' \in N(s)$. The move value indicates whether the neighbor s' is of better, worse or equal quality relative to s. Let $\Delta f = f(s')$ - $f(s)$ denote this move value.

- *Incremental evaluation*: Basically, there are two ways to obtain Δf for a neighbor. The trivial way is to calculate $f(s')$ from "scratch" using the objective function[1] f. Doing this way may be expensive if f needs to be evaluated very often or if the evaluation of f itself involves complex calculations. A more efficient alternative aims to derive the value of $f(s')$ from the value $f(s)$ by updating only what is strictly necessary. Indeed, if a neighbor s' is close to its initial solution s, which is true for many neighborhoods, then the evaluation of $f(s')$ can be carried out in this incremental manner. For a number of basic neighborhoods, like those shown previously, such an incremental evaluation is often possible.

- *Full search of neighborhood*: The incremental evaluation can be applied to *all the neighbors* of a given neighborhood relation. In this case, it is generally useful to investigate dedicated data structures (call it Δ-table) to store the move values for all the neighbors of the current solution. Δ-table provides a convenient way to know the quality of each neighbor and enables an efficient search of the full neighborhood. With such a Δ-table, the local search algorithm can decide easily at each iteration which neighbor to take according to its search strategy. For instance, a best-improvement descent algorithm will take the move that is identified by the most negative value in the Δ-table to minimize the objective function. After each move, the Δ-table (often only a portion of it) is updated accordingly using the incremental evaluation technique to propagate the effect of the move. Δ-table is a very useful technique for local search algorithms. This is particularly the case for descent-based methods like Tabu Search where a best neighbor needs to be identified (see examples in [393]).

- *Approximative evaluation*: The practical usefulness of Δ-table depends on both the complexity and the number of updates needed after each move transition. It may happen that, the move value can not be incrementally calculated or the Δ updates need to change a large portion of Δ-table. In this case, it would be useful to replace the initial evaluation function by a (fast) approximative evaluation function [424]. More generally, approximate evaluation is useful if the evaluation function is computationally expensive to calculate or if the function is ill-defined.

- *Order of evaluation*: If the neighborhood is not completely searched, one must decide the order in which the neighborhood is explored. For instance, the first-improvement descent technique moves to any improving neighbor. If there are several improving neighbors, the descent search picks the "first" one encountered

[1] For the reason of simplicity, the term "objective function" is used here. A more precise term is "evaluation function", see §6.3.4.

in the order the neighbors are examined. To allow such a method to increase its search diversity, a random order may be preferred [704].

6.3.1.4 Combination of Neighborhoods

Very often, different neighborhoods may be available, enabling alternative ways to explore the search space. In such a situation, it is interesting to consider combined use of multiple neighborhoods. For illustrative purpose, consider two neighborhoods N_1 and N_2. Then one can consider at least three ways to use them in a combined way.

First, *neighborhood union* $N_1 \cup N_2$ includes all the neighbors of the two underlying neighborhoods, so that any member of N_1 and N_2 is a member of $N_1 \cup N_2$. A local search algorithm using this combined neighborhood selects the next neighboring solution among all the solutions in both neighborhoods. This combination has no sense if one neighborhood is fully included in the other one.

With *Probabilistic neighborhood union* $N_1 \oslash N_2$, a neighbor solution in N_1 (or N_2) belongs to $N_1 \oslash N_2$ with probability p (resp. probability 1-p). A local search algorithm using this combined neighborhood selects at each iteration the next neighbor from N_1 with probability p and from N_2 with probability 1-p.

Token-ring combination $N_1 \rightarrow N_2$ is time-dependent and defined alternatively either by N_1 or N_2 according to some pre-defined conditions [209]. A local search algorithm using this combined neighborhood cycles through these neighborhoods. It typically starts with one neighborhood until the search stagnates, then changes to the other neighborhood until the search stagnates again to switch back to the first neighborhood and so on.

The advantage of combined neighborhood was already demonstrated a long time ago in [524] for solving the Traveling Salesman Problem. More generally, the issue of transitioning among alternative neighborhoods was discussed with the Tabu Search framework and strategic oscillation design in [312]. More recent examples of local search methods focusing on multiple neighborhoods include Variable Neighborhood Search [363], Neighborhood Portfolio Search [209] and Progressive Neighborhood Search [323]. Examples of studies on neighborhood combinations can be found in [353, 539].

6.3.2 Design of Semantic Combination Operator

6.3.2.1 Solution Combination

Combination is another key component of a MA and constitutes one leading force to explore the search space. The basic idea of combination is very appealing since it provides a very general way of generating new solutions by mixing existing solutions. Contrary to local changes of local improvement, combination can bring into new solutions more useful information, that may be beneficial for a healthy evolution of the search process.

As a first step, it would be tempting to consider the application of a blind (random) crossover operators for solution combinations. Doing this has the advantage of ease of application. However, one question should be asked before this approach is attempted: Is the crossover operator meaningful with respect to the optimization objective? If the answer is negative, the crossover operator is probably not appropriate and the sole role it would play in this case would be to introduce some random diversification in the search process.

In practice, instead of applying blind crossovers, it is often preferable to consider dedicated combination operators that have strong "semantics" with respect to the optimization objective. A semantic combination aims to pass intrinsic good properties from parents to offspring. The design of such a combination operator is far from trivial and in fact represents a challenging issue. Although there are some theoretical guidances, the discovery of such a semantic combination operator in practice relies basically on a deep analysis and understanding of the given problem. Compared with the design of local search procedures, the design of a meaningful combination operator constitutes probably one of the most creative parts of an effective MA.

6.3.2.2 Theoretical Foundations

The schemata theory [389] and the building block hypothesis [325] are often mentioned to explain (partially) the performance of Genetic Algorithms. Intuitively, building blocks are promising patterns of solutions that can be progressively assembled by crossover to get improved solutions. Given that this theory is defined for binary and simple Genetic Algorithm, it is not directly applicable in the context of MAs. Nevertheless, assembling building blocks to generate new solutions remains an appealing idea. In [750, 753], the concept of forma is introduced to generalize the schemata theory. A formal framework is even proposed to try to capture some fundamental aspects of MA in [752]. The forma theory suggests a set of general principles for the design of solution representations and recombination operators. According to this theory, a suitable recombination operator is required to fulfill two conditions called *respect* and *proper assortment*. Intuitively, the *respect* condition advocates the heritage of shared characteristics of parents to offspring, while *proper assortment* ensures the heritage of desirable characteristics of each parent by their offspring. This is in accordance with the general principle of conserving good features through inheritance and discarding bad features developed in Grouping Genetic Algorithms [248].

6.3.2.3 Design of Combination Operator

These abstract considerations only provide us with very general guidances for designing recombination operators. For a particular problem, it is still necessary to find out what are the building blocks (interesting patterns or characteristics) of solutions that can be assembled and inherited through the recombination process. Unfortunately, there is no short-cut to this quest and a fine analysis and deep understanding of the given problem is indispensable to find useful clues.

First, one can analyze the samples of optimal or high quality solutions to possibly identify regular patterns shared by these solutions. Indeed, if such a pattern exists, then the recombination operator can be constrained to conserve the pattern from the parent solutions and to avoid breaking the pattern. Alternatively, the recombination operator can also be encouraged to promote the emergence of favorable building blocks. For instance, such an analysis applied to the Traveling Salesman Problem shows that high quality local optima share sub-tours [523, 524]. This property has been used by several highly successful crossover operators which conserve common edges or sub-tours in offspring solutions [286, 636, 648, 720, 931]. Similarly, for the graph k-coloring problem, an analysis of coloring solutions discloses that some nodes are always grouped to the same color class (i.e. colored with the same color). This characteristic has helped to devise powerful combination operators, as shown in [217, 290] and in [292, 537, 549, 726] with multi-parents.

6.3.2.4 Multi-Parent Combination

Combination may operate with more than two parents. Multiple parent combination is even a general rule for the Scatter Search metaheuristic which uses, in its original form, linear combinations of several solutions to create new solutions [308]. Although there is no theoretical justifications, the practical advantage of multiple parent recombination was demonstrated in several occasions for discrete optimization. For instance, for the graph k-coloring problem, several recent and top-performing algorithms integrate multiple parent combination [292, 537, 549, 726], where color classes from different solutions are assembled to build offspring colorings. More generally, when multiple solutions are used for creating a new solution, one can define special rules to score the solution components of each parent solution and use strategic voting rules to combine components from different parents solutions.

A question that arises for multi-parent combination is how to determine the number of the parents. By using two parents, the offspring is expected to inherit 50% material from each parent. The contribution of each parent to the new solution descreases with an increasing number of parents. If the building blocks from different parents are independent from one another, taking more parents into account would be interesting to build good and diversified offspring. Otherwise, if a building block from a parent is epistatic with respect to the building blocks of other parents, blending more parents means more disruption, and thus should be avoided.

6.3.3 Population Diversity Management

Population diversity is another important issue that should be considered in the design of an effective MA [290, 726, 836]. If the population diversity is not properly managed, the population will converge prematurely and the search process stops with poor local optima. This is particularly true when a small population is used by the MA. In what follows, we first provide some precisions about the nature of diversity and explain how fertile diversity can be promoted and maintained within a population. Note however that diversity is not interesting per se within a MA. The ultimate goal of population diversity is to help the search process not only to avoid premature converge, but also to continually discover interesting new solutions in order to explore non-visited promising search areas. See also Chapter 10.

6.3.3.1 Diversity

Population diversity can be measured by a similarity (or distance) metric applied to the members of the population. The metric can be defined either on the solution representation level (genotype metric) or solution level (phenotype metric) [325]. For instance, pair-wise *Hamming distance* can be used as a genotype metric to measure population diversity. Diversity can also be measured in terms of *entropy* [267] or by the so-called *moment of inertia* [614]. Genotype metric is usually problem independent, and thus may or may not reflect the intrinsic diversity of a population with respect to the given optimization objective.

Population diversity can also be measured at the phenotype level over the solution space. For instance, for partition problems like graph k-coloring, the distance between two partitions can be measured by the so-called *transfer distance* which is the minimum number of elements that need to be moved between classes of one partition so that the resulting partition becomes the other partition [189, 763]. A phenotype metric is defined over the solution space and thus is more likely to measure the real diversity of a population.

In order to observe suitably the population diversity, it is useful to first determine the most appropriate distance or similarity metric with respect to the optimization objective of the given problem. Moreover, if the population diversity needs to be continually monitored, it becomes important to pay attention to the cost of computing the underlaying metric.

6.3.3.2 Promoting and Maintaining Useful Diversity

Population diversity can be promoted and managed at several levels of a MA. One evident possibility is to define specific selection rules to favor the selection of *distanced parents* for mating. Another possibility concerns the variation operators which can be designed in such a way that they favor the generation of diverse and varied offspring. For instance, the "Distance Preserving Crossover" introduced in [286, 588] is constrained to generate an offspring which is at the same distance from both parents. More generally, the path-relinking type of combinations typically construct offspring solutions by considering both the solution quality and its distance to its parent solutions [320] (see also [538] for an example).

Population diversity can also be controlled by the offspring acceptation and replacement strategies. Specifically, this can be done according to both solution diversity and quality. For instance, in [726] a minimum diversity-quality threshold is imposed between the solutions of the population. The acceptation of a new offspring is conditioned not only by its quality, but also by its distance to existing solutions. Similarly, diversity and quality are considered to select the victim solution to be replaced by the offspring.

Other useful ideas for diversity preservation can be found in the areas of Genetic Algorithms. Well-known examples include sharing [327] and crowding [204, 546].

6.3.4 Other Issues

In addition to the components mentioned until now, the design of an effective Memetic Algorithm should take into account a number of other considerations which are briefly discussed in this section.

- *Initial population*: There are basically two ways to obtain an initial population: Random generation and constructive elaboration. While random generation is easy to apply, it can hardly generate initial solutions of good quality. To improve the basic random generation method, a simple sampling technique can be applied. Let P be the population size, then one can generate $K > P$ solutions and then retain only the P "best" ones. Initial generation by construction can be used if some fast greedy heuristics are available for the given problem. Notice that, in this case, the greedy heuristics must be randomized such that each application leads to a different solution. Another issue that can be considered at the initialization stage is to take care of building a diversified population. This can be achieved by controlling the distance between each new solution and the existing solutions of the population. Only distant new solutions are allowed to join the population.

- *Distance*: At several places, MAs may need to measure the distance between two solutions or between a solution and a group of solutions. For instance, parents selection may operate in such a way that the selected parents are sufficiently distant. Similarly, a population management strategy may decide the acceptation or rejection of an offspring by considering its distance to the members of the population. When an operation refers to the notion of distance, it is preferable to employ an appropriate distance metric which is meaningful with respect to the given problem. For instance, for partition problems like graph coloring (see §6.4.1), Hamming distance is not a suitable metric to characterize the difference of two partitions. Instead, transfer distance between partitions should be preferred. Once again, the choice of the distance metric should ideally be correlated with the semantics of the problem on hand.

- *Rich evaluation function*: Evaluation function assesses the quality of a candidate solution with respect to the optimization objective and orients the search method to "navigate" through the search space. A good evaluation function is expected to be able to distinguish each solution from the other solutions and thus to effectively guide the search method to make the most appropriate choice at each iteration. Very often, the initial optimization objective f is directly used as evaluation function. However, such a function may not be sufficiently discriminant to distinguish different solutions. To improve the discriminating power, it is useful to incorporate in the evaluation function additional information, e.g. relative to the structure of the problem instance to be solved. Examples can be found in [248, 431, 772]. Moreover, when constrained optimization problems are considered, some constrains may be hard to satisfy, and thus are relaxed. Among various constraint relaxation techniques, a common practice is to integrate the relaxed constraints into the evaluation function as a (weighted) component or as a part of a multi-component evaluation function (see examples in [316, 902, 903]).

- *Constraints*: The constraints in the considered problem may influence the design of some MA components. For instance, suppose that the MA algorithm is expected to explore only feasible solutions. Then one must decide whether a combination operator is constrained to create only feasible solutions. If infeasible offspring is allowed, it is necessary to consider a dedicated mechanism to repair the broken constraints. Similarly, neighborhood design can take into consideration the constraints to identify eligible moves. For instance, in feasibility search problems, this is often done by identifying problem variables involving violated constraints and restricting the set of authorized moves to those defined on these conflicting variables. Finally, as previously stated, constraints that are difficult to solve can be used in the design of the evaluation function.

- *Connections with Scatter Search and Path Relinking*: As discussed in [311] and [317] (Chapter 9), the MA framework shares ideas with Scatter Search and Path Relinking [313, 320]. These latter methods provide unifying principles for joining solutions based on generalized path constructions (in both Euclidean and neighborhood spaces) and by using strategic design. Solution combination in Scatter Search originated historically from strategies for combining decision rules and combining constraints. In Scatter Search, dispersed new solutions are created from a set of reference solutions by weighted combinations of subsets of the reference solutions that are selected as elite solutions. With Path Relinking, offspring solutions are generated by exploring, within a neighborhood space, trajectories that connect two or more reference solutions. One notices that the reference solutions or subsets of them can be considered as parent solutions for combination while combination resorts to diverse strategies such as attribute voting and weighting.

6.4 Case Studies

In this section, we show two case studies of quite different nature with the purpose of showing how these issues can be effectively implemented in practice. We particularly focus on the design of combination and local search operators.

6.4.1 Graph Coloring Problems

6.4.1.1 Problem Description

Given an integer k and a undirected graph $G = (V,E)$ with a set V of vertices and a set E of edges, a legal k-coloring of G is a partition of V into k distinct color classes such that each color class is composed of pairwise non-adjacent vertices. The graph k-coloring problem (k-COLOR) aims at finding a legal k-coloring for a fixed k while the graph coloring problem (COLOR) determines the smallest k for a given graph G (its *chromatic number* χ_G) such that G has a legal k-coloring. Since COLOR can be handled by solving a series of k-COLOR with decreasing k values, we only consider here k-COLOR.

For a given k-COLOR instance, i.e. an integer k and graph $G = (V,E)$, let $s = \{C_1, C_2...C_k\}$ denote a partition of V into k distinct color classes such that each C_i ($i \in \{1,2...k\}$) contains all the vertices that are colored with color i. Let S denote all such partitions. For any $s \in S$, define its conflict number $f(s)$ to be the number of pairs of adjacent vertices x and y ($\{x,y\} \in E$) belonging to a same color class of s. Then k-COLOR can be solved by minimizing $f(s)$; $f(s)$=0 implies that s is a legal k-coloring, i.e. all its color classes C_i are conflict-free.

Notice that among the large number of existing heuristic algorithms for k-COLOR, Memetic Algorithms are certainly among the most powerful ones and provide the best results on the well-known DIMACS benchmark instances of this well-known NP-complete problem.

6.4.1.2 Partition Crossovers

In order to design a semantic combination operator, let us try to get an idea about the possible "building blocks" for our problem. The goal of k-COLOR is to determine a set of k *distinct* conflict-free color classes. In this context, color classes can be considered our basic "building blocks". If there are several "good" color classes among some candidate solutions, then these color classes can favorably be recombined to obtain new candidate solutions. This idea was first explored by the Greedy Partition Crossover (GPX) described in [290] and the Union of Independent Sets crossover in [217], which are also related to the design of grouping crossovers described in [248].

Operating with two parent k-colorings s_1 and s_2, GPX builds step by step the k classes $C_1^0, \ldots, C_k^0$ of the offspring s_0. At the first step, GPX creates C_1^0 by choosing a *largest* class from one parent and removes its vertices from both parents s_1 and s_2. GPX repeats then the same operations for the next k-1 steps, but alternates each time the parent considered. If some vertices remain unassigned at the end of these k steps, they are randomly assigned to one of the k color classes. The alternation between the parents aims at a balanced mixture of information from both parents and avoiding the dominance of one parent over the other one during the recombination.

Table 6.1 shows an example with 3 color classes ($k = 3$) and 10 vertices represented by capital letters A,B,$\cdots$,J.

Table 6.1. The Greedy Partition Crossover: An example from [290]

parent $s_1 \rightarrow$	A B C	$\boxed{\text{D E F G}}$	H I J	$C_1^0 := \{D,E,F,G\}$	A B C		H I J
parent s_2	C $\underline{\text{D E G}}$	A $\underline{\text{F}}$ I	B H J	remove D,E,F and G	C	A I	B H J
offspring s					D E F G		
parent s_1	A $\underline{\text{B}}$ C		$\underline{\text{H}}$ I $\underline{\text{J}}$	$C_2^0 := \{B,H,J\}$	A C		I
parent $s_2 \rightarrow$	C	A I	$\boxed{\text{B H J}}$	remove B,H and J	C	A I	
offspring s	D E F G				D E F G	B H J	
parent $s_1 \rightarrow$	$\boxed{\text{A C}}$		I	$C_3^0 := \{A,C\}$			I
parent s_2	$\underline{\text{C}}$	$\underline{\text{A}}$ I		remove A and C		I	
offspring s	D E F G	B H J			D E F G	B H J	A C

The basic idea underlying GPX was also explored with multiple parent combination operators [292, 352, 537, 549, 726]. Using multiple parents for combination is fertile for k-COLOR since this offers more possibilities to obtain good (large) color classes for each step of the recombination operation. By generalizing two parents to multiple parents, refined and additional strategies were also introduced to make the combination process as effective as possible. For instance with the AMaPX operator of [537], in order to favor the creation of *diversified offspring*, each time a color class from a parent is transmitted to the offspring, this parent's k-coloring will not be considered for the next few steps of offspring building. In [726], in order to measure the goodness of the color classes of the parent colorings, the combination operator takes into account the size of each color class, the number of conflicting vertices as well as the degrees of the vertices in the color class.

A question that arises when multiple parents are used is how to determine the number of parents. It is clear that by using more parents, fewer classes will be transmitted from each parent to the offspring and this also implies that the class blending from each parent is also more disrupted. An analysis of the relations between the number of vertices, the number of color classes and the number of parents permits to identify a heuristic rule to fix the right number of parents [726].

In [292], the combination operation is performed within a slightly different context. The algorithm maintains a pool of conflict-free color classes obtained during the search process. From time to time, these color classes are used to generate new k-colorings. Other combination operators using similar ideas are investigated in [217, 352, 549].

6.4.1.3 Local Improvement by Tabu Search

In memetic coloring algorithms, Tabu Search is frequently used for local improvement to ameliorate a new offspring created by the combination operator. For illustration purpose, we use the TS algorithm described in [290] as an example. It uses the constrained "one-change" move described in §6.3.1.2 such that a *neighbor* s' of a given configuration s is obtained by moving a single *conflicting* vertex v from a color class C_i to another color class C_j. When such a move $< v, i >$ is performed, the couple $< v,i >$ is classified tabu for the next tl iterations. Therefore, v cannot be reassigned to the class i during this period, unless moving v back to the color class i leads to a configuration better than the best configuration found so far (*aspiration criterion*). The tabu tenure tl for a move is variable and depends on the number nb_{CFL} of conflicting vertices in the current configuration: $tl = Random(A) + \alpha * nb_{CFL}$ where A and α are two parameters and the $Random(A)$ function returns a random number from $\{0,\cdots,A-1\}$. To implement the tabu list, it is sufficient to use a $|V| \times k$ table.

The algorithm memorizes and returns the *most recent* configuration s_* among the best configurations found: After each iteration, the current configuration s replaces s_* if $f(s) \leqslant f(s_*)$ (and not only if $f(s) < f(s_*)$). The rational to return the last best configuration is that we want to produce a solution which is as far away as possible from the initial solution in order to better preserve the diversity in the population.

6.4.2 Maximum Parsimony Phylogeny

6.4.2.1 Problem Description

Phylogenetics is the study of evolutionary relationships among various groups of organisms (for example, species or populations). These connections are represented graphically through phylogenetic trees. Computational phylogenetics aims to infer phylogenetic trees from molecular data such as protein or DNA sequences [256]. The main phylogenetic approaches include methods using a distance-matrix, the maximum likelihood or maximum parsimony criterion.

Maximum parsimony phylogeny generally takes as input a multiple sequence alignment which is a matrix M of characters composed of n lines (related to a set S of species, where $|S| = n$) and k columns which represent the characters of the sequences [255]. Each sequence is also called a taxon. Each character of the matrix belongs to an alphabet Σ. A phylogenetic tree T of the given input is a binary tree such that (1) the leaves of T are the set of n species, and (2) each internal node is induced by the sequence of parsimony of its two descendant sequences. Given two sequences $S_1 =< x_1,\cdots,x_k >$ and $S_2 =< y_1,\cdots,y_k >$ with $\forall i \in \{1..k\}, x_i, y_i$ belonging to the power set $\mathscr{P}(\Sigma = \{-,A,C,G,T\})$, the sequence of parsimony $P(S_1,S_2) =< z_1,\cdots,z_k >$ of S_1 and S_2 is given by ([264]) :

$$\forall i, 1 \leqslant i \leqslant k, z_i = \begin{cases} x_i \cup y_i, \text{if } x_i \cap y_i = \emptyset \\ x_i \cap y_i, \text{otherwise} \end{cases} \tag{6.1}$$

The score of the sequence of parsimony defines the “distance” separating its two descent sequences:

$$f_{P(S_1,S_2)} = \sum_{i=1}^{k} c_i \quad \text{where} \quad c_i = \begin{cases} 1, \text{if } x_i \cap y_i = \emptyset \\ 0, \text{otherwise} \end{cases} \tag{6.2}$$

Algorithm 11. The general DiBIP crossover scheme

```
Input: $T_1$, $T_2$, $\delta$, $\Delta$, $\oplus$, $\Lambda$
Output: A child tree $T^*$

1. Apply the tree-to-distance operator $\Delta$ to each parent tree $T_i$ ($i$=1,2) to obtain the
   corresponding distance matrix $D_i = \Delta(T_i)$;
2. Apply the matrix operator $\oplus$ to $D_1$ and $D_2$ to obtain $D^*$: $D^* \leftarrow D_1 \oplus D_2$;
3. Apply the distance-to-tree operator $\Lambda$ to $D^*$ to obtain a child tree: $T^* \leftarrow \Lambda(D^*)$.
```

Let T be a binary parsimony tree with n leafs or species. T has then $n-1$ sequences of parsimony (internal nodes). Let I denote the set of these internal nodes. The Fitch parsimony score $f(T)$ of T is defined as follows:

$$f(T) = \sum_{i \in I} f_i(T) \tag{6.3}$$

The aim of the Maximum Parsimony problem (MP) is then to find a most parsimonious phylogenetic tree T^* such that T^* minimizes the parsimony score. Since there are $\prod_{i=3}^{n}(2i-3)$ possible binary trees with n leafs, this problem is a highly combinatorial search problem. The MP problem is computationally difficult since its associated decision problem is equivalent to the NP-complete Steiner problem in a hypercube [277]. MP has been subject of many studies for many years. Among them, neighborhood-based local search and various hybrid algorithms are certainly the most popular solution methods. In what follows, we show a Memetic Algorithm called HYDRA [767], which combines a dedicated tree crossover called DiBIP [322] and a progressive neighborhood local search method [323].

6.4.2.2 Distance-Based Information Preservation Crossover

First, let us notice that conventional tree crossovers known in genetic programming are not suitable here. The Distance-Based Information Preservation crossover (DiBIP) is specifically designed for the MP problem. DiBIP is based on a topological distance between species (leafs) and aims to preserve common properties of parents in terms of this distance between species. For instance, two species that are close (or far) in both parents should stay close (resp. distant) in the offspring. Given two parents trees, the DiBIP crossover is realized in three steps: Calculate a distance matrix for each parent tree, then combine the two resulting matrices to get a third matrix and finally create a child tree from this last matrix.

The general DiBIP crossover scheme is described in Algorithm 11 where T_1 and T_2 denote two parents trees. δ is a distance metric to measure the distance of each pair of species of a tree T, Δ a tree-to-distance operator to obtain a distance matrix of a tree, $\oplus$ a matrix operator to combine 2 distance matrices to produce a new distance matrix, Λ a distance-to-tree operator to construct a tree from a given distance matrix.

A specific DiBIP crossover operator is obtained once δ, Δ, $\oplus$, and Λ are provided. The distance measure δ should be ideally correlated to the evolutionary changes between species. For instance, 2 species separated in the tree by a small number of evolutionary changes should have a smaller distance than 2 species separated by a large number of changes. The distance measure should additionally be tree-topology dependent. In this sense, the length of the elementary path between 2 species is a possible option while Hamming distance is not suitable here because this metric is totally independent of tree topologies.

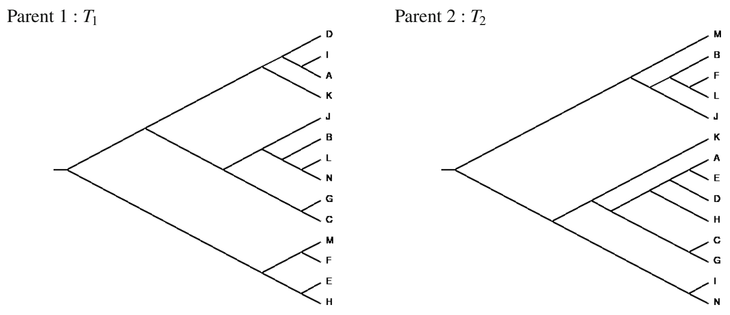
Moreover, since we want to preserve representative features of the parents during the crossover operation, a valid matrix operator $\oplus$ should favor such an inheritance from parents to offspring and meet some relation preservation property. For instance, if a pair of species (a,b) is closer than another pair (c,d) in both parents, then this relation should be conserved. Consider the operation $\oplus$ such that for a pair of species (i,j), $(D_1 \oplus D_2)(i,j) = \alpha.\min\{D_1(i,j), D_2(i,j)\} + (1-\alpha).\max\{D_1(i,j), D_2(i,j)\}$ with $\alpha \in [0,1]$. This indeed defines a valid $\oplus$ operator. Furthermore, this definition offers in fact many possibilities and seems particularly relevant to MP. For instance, the arithmetic average ($\alpha = 0.5$) and the max operator max ($\alpha = 0$) are 2 special cases. At last, let us mention that the arithmetic addition is another simple valid $\oplus$ operator.

We now show a concrete example. Given two species i and j, define their distance δ_{ij} to be the *topological distance*, i.e. the length of the elementary path between the respective ascendants of i and j, (minus 1 if the path contains the root of the tree T). The matrix operator $\oplus$ is the addition $+$ such that $D(i,j) = D_1(i,j) + D_2(i,j)$, which satisfies the relation preservation property previously mentioned. The distance-to-tree operator Λ is a non-deterministic variant of the well-known UPGMA (Unweighted Pair Group Method with Arithmetic Mean) method [833]. Figs. 6.1 and 6.2 show an application of this crossover operator. One observes that the closeness of species in both parents is conserved in the child. This observation applies equally to distant species.

6.4.2.3 Progressive Neighborhood Search

For local improvement, HYDRA uses *Progressive Neighborhood Search* (PNS) which operates with a variable-size neighborhoods [323]. Given a parsimony tree T, a neighboring tree T' is typically obtained by a move that consists in cutting a sub-tree from T and reinserting the sub-tree elsewhere in the initial tree. If a meaningful metric can be defined to measure the distance between the cutting and inserting points, then it would be possible to define neighborhoods of variable sizes. In [323], the topological distance δ shown in Section 6.4.2.2 is used for this purpose. A distance parameter d is introduced to constraint the distance between the pruned edge i and the edge j receiving the insertion such that $\delta_{ij} \leqslant d$.

So, setting $d = \infty$ leads to a large neighborhood where the pruned edge (with its subtree) can be reinserted anywhere in the tree. Consequently, the topological change can be important. This case corresponds in fact to the well-known *Subtree Pruning Regrafting* neighborhood [862] whose size equals $2(n-3)(2n-7)$ [12]. Reversely, setting $d = 1$ gives a small neighborhood where neighboring trees are close to the current tree. This case corresponds to another well-known neighborhood



$D_1 = \Delta(T_1)$

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
A	-	**B**												
B	6	-	**C**											
C	5	3	-	**D**										
D	1	5	4	-	**E**									
E	5	5	4	4	-	**F**								
F	5	5	4	4	2	-	**G**							
G	5	3	0	4	4	4	-	**H**						
H	5	5	4	4	0	2	4	-	**I**					
I	0	6	5	1	5	5	5	5	-	**J**				
J	5	1	2	4	4	4	2	4	5	-	**K**			
K	2	4	3	1	3	3	3	3	2	3	-	**L**		
L	7	1	4	6	6	6	4	6	7	2	5	-	**M**	
M	5	5	4	4	2	0	4	2	5	4	3	6	-	**N**
N	7	1	4	6	6	6	4	6	7	2	5	0	6	-

$D_2 = \Delta(T_2)$

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
A	-	**B**												
B	8	-	**C**											
C	4	6	-	**D**										
D	1	7	3	-	**E**									
E	0	8	4	1	-	**F**								
F	9	1	7	8	9	-	**G**							
G	4	6	0	3	4	7	-	**H**						
H	2	6	2	1	2	7	2	-	**I**					
I	6	4	4	5	6	5	4	4	-	**J**				
J	7	1	5	6	7	2	5	5	3	-	**K**			
K	4	4	2	3	4	5	2	2	2	3	-	**L**		
L	9	1	7	8	9	0	7	7	5	2	5	-	**M**	
M	6	2	4	5	6	3	4	4	2	1	2	3	-	**N**
N	6	4	4	5	6	5	4	4	0	3	2	5	2	-

Fig. 6.1. Application of the DiBIP Tree Crossover [322] – The parents

called *Nearest Neighbor Interchange* [922] which swaps two adjacent branches of the tree leading to $(2n-6)$ neighbors [770]. By varying the parameter d, one gets neighborhoods of intermediate sizes.

The *Progressive Neighborhood Search* is based on this parametric neighborhood and its neighborhood changes during the search process by varying the value of d. In the particular MP context, PNS carries out a descent search starting with a large neighborhood (i.e. with large d) and reduces progressively the neighborhood. Indeed, at the beginning of the search, it is possible to obtain strong quality

$D^* = D_1 \oplus D_2$

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
A	-	**B**												
B	14	-	**C**											
C	9	9	-	**D**										
D	2	12	7	-	**E**									
E	5	13	8	5	-	**F**								
F	14	6	11	12	11	-	**G**							
G	9	9	0	7	8	11	-	**H**						
H	7	11	6	5	2	9	6	-	**I**					
I	6	10	9	6	11	10	9	9	-	**J**				
J	12	2	7	10	11	6	7	9	8	-	**K**			
K	6	8	5	4	7	8	5	5	4	6	-	**L**		
L	16	2	11	14	15	6	11	13	12	4	10	-	**M**	
M	11	7	8	9	8	3	8	6	7	5	5	9	-	**N**
N	13	5	8	11	12	11	8	10	7	5	7	5	8	-

Child : $T^* = \Lambda(D^*)$

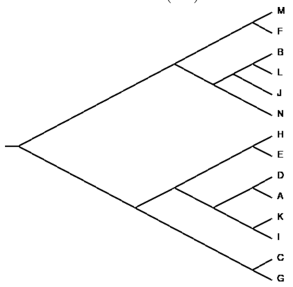


Fig. 6.2. Application of the DiBIP Tree Crossover [322] – The offspring

improvement by important topological modifications of the tree with large d. When the search progresses and the quality of the trees becomes better and better, only small improvements can be expected with small tree modifications. It is thus more judicious to switch to smaller and small neighborhoods to accelerate the search.

One notices that PNS shares some features with Variable Neighborhood Search (VNS) [363]. However, contrary to VNS, the neighborhoods explored by PNS are not systematically of increasing sizes. Within the context of our Maximum Parsimony problem, PNS even progressively reduces its neighborhood.

6.5 Conclusions

In this chapter we have presented the basic concepts of Memetic Algorithms for Discrete Optimization. Focus is given to the key design issues of an effective MA algorithm. We have explained the usefulness of a deep study and understanding of the optimization problem on hand. We have insisted on the importance of a careful adaptation of the general search strategies offered by the MA framework, a suitable incorporation of problem specific knowledge in different components of the MA as well as a logical integration of these components. The pursuit goal is clearly to build an effective MA algorithm that is able to ensure a balanced exploitation and exploration of the search space.

It should be clear that a blind MA application would have little chance to deliver good results for difficult optimization problems. High performance can only be possible by a disciplined and careful specialization of the general MA framework to the targeted problem. It is equally important to apply the "lean design" principle in order to avoid redundant or superficial algorithmic components.

The framework of Memetic Algorithms constitutes an interesting enrichment to the arsenal of existing discrete optimization methods and offers a valuable alternative for tackling hard discrete optimization problems.

Acknowledgements. This work was partially supported by "*Angers Loire Métropole*" and the Region of "*Pays de la Loire*" within the following projects: MILES, BIL, Radapop and LigeRO. I'm grateful to my research collaborators within and outside the Group "Metaheuristics, Optimization and Applications" of the LERIA Laboratory.