Chapter 2
Computational Intelligence

Abstract This chapter provides an introduction to Computational Intelligence (CI). Artificial Intelligence (AI) and CI are briefly compared. CI by itself is an umbrella term covering different branches of methods most of them following the paradigm “nature-inspired”. While CI and AI partly overlap, the methods applied in CI benefit from nature-inspired strategies and implement them as computational algorithms. Mathematical optimization is shortly explained. CI comprises five main branches: Evolutionary Computation (EC), Swarm Intelligence (SI), Neural Networks, Fuzzy Logic, and Artificial Immune Systems. A focus is laid on EC and SI as the most prominent CI methods used in logistics and supply chain management. EC is coupled with Evolutionary Algorithms (EA). Methods belonging to EC respectively EA are Evolution Strategy, Genetic Algorithm (GA), Genetic and Evolutionary Programming, the multiobjective variants Non-dominated Sorting GA (NSGA) and Strength Pareto EA (SPEA), Memetic Algorithms, and further methods. The most important methods belonging to SI are Particle Swarm Optimization (PSO), Discrete PSO and Ant Colony Optimization. EA and SI approaches are also attributed to the class of metaheuristics which use general problem-solving concepts for problem solution during their search for better solutions in a wide range of application domains.

2.1 Foundations of Computational Intelligence

For hundreds of millions of years, nature has continuously developed a bunch of techniques to overcome obstacles in the daily lives of plants, animals and humans and has found solutions to our problems within the real world.
2.1.1 Artificial and Computational Intelligence and Related Techniques

2.1.1.1 Artificial Intelligence

Artificial Intelligence (McCarthy 2007), abbreviated AI, was “scientifically born” in the middle of the last century. Around 1955, John McCarthy (McCarthy et al. 1955) was the first scientist to introduce AI as a technology in computer science, stating that AI was the basis for a “truly intelligent machine”. The interest in AI as a branch of computer science has been rapidly growing since. Today, many other disciplines such as engineering, biology, psychology, linguistics, etc. are involved in or concerned with AI in an interdisciplinary way.

While research in AI is still growing, definitions, classification and wording become blurred (Neapolitan and Jiang 2012): Originally, it was distinguished between Strong AI and standard AI (also called normal AI or Weak AI). Strong AI means general intelligence in the sense of the creation of human-like intelligence of machines. Thus, strong AI focusses on the development of intelligent machines (i.e. software or hardware robots) and the design of human cognition comprising creativity, consciousness and emotions to make the robots somehow human-like. The aim here is to match or even to exceed human intelligence. In contrast, weak AI is more dedicated to the solution of particular problems applying human-like approaches. Reasoning, adaptation, learning and social intelligence are key components, which are used in computer programs in order to solve particular problems (e.g. text or speech recognition).

In other words: Weak AI aims to provide intelligent algorithms somewhere stored inside the software, whereas strong AI yields rather an embodiment of intelligence in robot-like machines and makes them behave somehow intelligent. Today, the two directions of AI are mixing again: AI is understood as making machines intelligent as well as creating intelligent machines.

AI comprises various methods for the intelligent processing of information (e.g. perception, learning, planning, knowledge, reasoning, and communication) and provides—particularly in robotics—machines and computers with the ability to move in, interact with, and participate in the real world. Therefore, AI mostly uses symbolic techniques such as logic- and knowledge-based methods, decision trees, case-based reasoning, and stochastic automata. With the help of these techniques, logical conclusions or automata states are derived.

Many research directions in AI have been evolved over the last 60 years: Probabilistic methods are used for uncertain reasoning. Classifiers and statistical learning methods support the learning. Search and optimization methods perform mathematical optimization. Neural networks, logic and control theory have emerged. Programming languages for artificial intelligence (e.g. Lisp and PROLOG) were developed. And recently, the research in intelligent machines and robotics has been making big leaps.
But, in spite of all its success AI has a disadvantage, namely the lack of more general fault-tolerant mechanisms: AI is often “too logical” to robustly solve problems that have not been well-posed, as is the case with most real-world problems. But, reality is the perfect test bed. Reality is imperfect/not perfect, often uncertain, noisy and volatile—reality is real. This is how our world looks and behaves. This is where Computational Intelligence (CI) plays an important role.

2.1.1.2 Computational Intelligence

*Computational Intelligence* (Kruse et al. 2013) (abbreviated CI) is often mentioned in relation with AI. Many researchers state (Fulcher and Jain 2008), that they do not refer to CI as a sub-part of AI: but that CI and AI comprise partly the same or similar methods. However, CI and AI use different methodologies and approaches to describe the principles of their methods. The name CI emerged in the 1990s (Poole et al. 1998), and first focused on intelligent robots. Today, interestingly, CI seems to be more clearly defined as AI: CI mainly deals with “nature-inspired” computational methods for facilitating intelligent behavior in complex and possibly changing environments (Engelbrecht 2007).

Furthermore, CI differs from AI in the sense that CI uses (mostly) sub-symbolic techniques, that is to say techniques that go beyond a certain symbolic level. A state or a problem instance is expressed or represented by numerical numbers instead of a symbolic entity as generally done in AI. Depending on the application areas and the particular problems, the advantages of CI methods might be superior to AI: CI methods perform efficient approximations within reasonable computation time because they often use non-deterministic, stochastic components, heuristics and metaheuristics, which converge quickly and robustly to one or more solutions even in the presence of noise, uncertainty, moving targets and changing search spaces.

Other advantages of CI methods are that they often work efficiently only needing a rough model of the problem, that they are fault-tolerant, and that they are mostly well-suited for parallelization. On the other hand, the biggest disadvantage of CI, compared to deterministic methods, is that the optimal solution(s) can generally be neither proved nor guaranteed to be found.

2.1.1.3 Techniques Related to Artificial and Computational Intelligence

Further related areas of research are *Machine Learning* (Mitchell 1997) (ML), *Soft Computing* (SC), *Natural Computing* (Rozenberg et al. 2012) (or Natural Computation, NC), *Bionics* (or bionic engineering, BE), and many more.

ML (Machine Learning) is counted as a sub-category of AI and deals with learning from data. Its relation to CI is only given by the neural networks. Furthermore, the paradigms of ML do not necessarily focus on “nature-inspired”.

SC (Soft Computing) is an area of research which investigates methods to efficiently (but inexactely) find solutions to computational extensive and hard
problems in huge search spaces. NC (Natural Computing) combines the paradigm “nature-inspired” with the categories “computing natural phenomena” and often “employing natural materials”. Thus, NC focuses on the synthesis of nature by means of computing, on the development of novel nature-inspired hardware and on the understanding of natural phenomena as information processing.

CI, SC and NC comprise partly the same methods, such as neural networks, evolutionary computation, swarm intelligence and fuzzy logic. But the application focus of CI is to solve various real-world problems, whereas that of SC is to compute hard problems in huge search spaces, and the application focus of NC is to understand nature by computing and rebuilding it.

Another related discipline is BE (bionics engineering). In contrast to the other mentioned research areas, focusing on computer science aspects, bionics plays a role on the physical or technical level. In bionics, the aim is to imitate the biological structures or physical abilities of fauna and flora and to transfer these abilities into technical products. Nevertheless, bionics has contributed to the research of evolutionary computation, neural networks, and swarm intelligence by imitating mechanisms or processes found in nature. Examples of bionically engineered products are augmented material surfaces to reduce the drag in fluid dynamics (e.g. shark skin, bird wings) or to avoid the adherence of dirt (e.g. lotus effect).

2.1.1.4 Interest in Computational Intelligence

The potential of CI is big; the interest in CI is great and still growing. Increasingly more conferences deal with CI. The most important ones are probably the bi-annual IEEE World Congress on Computational Intelligence (IEEE WCCI) with the annual conferences IEEE Congress on Evolutionary Computation (IEEE CEC), the International Joint Conference on Neural Network (IJCNN), and the IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), as well as the annual IEEE Symposium Series on Computational Intelligence (SSCI). The largest CI organization is the IEEE Computational Intelligence Society, which initiates these conferences.

2.1.2 Properties of Computational Intelligence

The main paradigm of CI is “nature-inspired”. This comprises various facets: Biology and particularly flora and fauna provide different mechanisms in nature, which have been researched and transferred to algorithms mimicking these mechanisms on a computational level. Such mechanisms are for example the evolution of living things, which succeeded over many generations to adapt better to their environmental conditions and to be superior to the others. Further mechanisms are the simulation of the functionality of brains to learn and remember, or the
reaction to the intrusion of pathogens into the body, or the adaption to the behavior of swarms like birds, fish or ants do.

Most methods of CI are more or less derived from nature. Nature is providing particular problem-solving strategies in the real-world, particularly how to best live, survive, and reproduce in certain environments. While CI exclusively covers such methods, AI knows only some of the nature-inspired methods (e.g. neural networks as the most common method applied in both, AI and CI).

Investigating nature, the methods of CI are inspired by biological models or processes. They are cast into theoretical models and then coded in algorithms, often without adapting them specifically onto the application area. Typically, methods of CI can be characterized by these three aspects:

• They are nature-inspired computational methodologies.
• They solve complex problems without (much) problem-specific knowledge.
• They are well suited to solve real-world problems.

Today, CI is categorized within the following five main branches. We try to characterize each CI branch featuring three individual properties:

• **Evolutionary Computation (EC):**
  - EC applies biological mechanisms of evolution by using the principle of survival of the fittest.
  - A population, consisting of several individuals, will improve over generations by selection, crossover and mutation.
  - EC is powerful in solving optimization and search problems within non-uniform search spaces.

• **Swarm Intelligence (SI):**
  - SI is the collective behavior of self-organizing multi-agent systems.
  - The entirety of a population consisting of simple agents, who are interacting only locally, leads to an intelligent-like global behavior.
  - SI mostly helps to solve optimization and search problems, often belonging to a kind of distance minimization.

• **Neural Network (NN; respectively Artificial Neural Network, ANN)**
  - NNs are inspired by real biological neural networks of brains of animals or human beings.
  - Data are processed through a network of interconnected artificial neurons.
  - NNs identify relationships between input and output data, finding patterns and generating information.

• **Fuzzy Logic (FL):**
  - FL is a many-valued logic, similar to the human reasoning.
  - It allows the usage of approximate values and incomplete or ambiguous data.
  - FL provides approximate solutions or conclusions.
• Artificial Immune System (AIS):
  – AIS is inspired by the immune system of humans, animals and plants.
  – It adapts the characteristics and processes of their immune system for learning and memory.
  – AIS is applied in adaptive systems for problem solving.

Additionally, further methods are sometimes counted as branches of CI, sometimes not, such as Reinforcement Learning (Kramer 2009) and Simulated Annealing. Reinforcement Learning adapts the behavior of an artificial agent (e.g. a software agent or a robot) by rewarding wanted and penalizing unwanted behavior. Simulated Annealing uses a thermodynamic principle in physics: In annealing processes (i.e. metal cooling processes) the material tries to harden to a state of minimal energy. Simulating this effect on a computational level, simulated annealing is mainly used for minimization problems.

The methods of CI allow another kind of information processing as known from AI. CI methods adopt some simple principles of nature which allow them to search either for alternative solutions (e.g. one or more local minima or maxima) or global optima. Some of these methods are well suited for adaptation to changing environments or conditions—and are thus used for solving control problems.

Some of the CI methods are able to learn and to remember—consequently they are used in pattern recognition or classification. CI methods are more or less fault-tolerant against incomplete and noisy or indefinite inputs. Many CI methods allow parallel processing (e.g. EC and SI use sets of parallel available solution candidates). Often it is sufficient to apply the CI methods to an easy model. Thus, (almost) no problem-specific knowledge is needed. Overall, depending on the problem, CI often allows to apply more sophisticated, nature-inspired solution strategies than AI.

### 2.1.3 The Big Picture of Computational Intelligence

Figure 2.1 provides a systematic overview of CI—to be more specific a big picture of CI—with a focus on Evolutionary Computation (EC) and Swarm Intelligence (SI). In general, EC uses Evolutionary Algorithms (EA). Their main representative methods are Genetic Algorithms (GA), Evolution Strategies (ES), Genetic Programming (GP), and Evolutionary Programming (EP). Further methods belonging to EC and EA are Memetic Algorithms (MA), Learning Classifier Systems (LCS), Differential Evolution (DE), Harmony Search (HS), and some more.

In the past, Swarm Algorithms (SA) were often mentioned as a sub-category of EC as well. Nowadays, SAs are counted as representative methods of SI, which include, for instance, Ant Colony Optimization (ACO), Particle Swarm Optimization (PSO), Bees Algorithm (BA) and Cuckoo Search (CS). PSO knows further sub-methods such as binary, discrete or combinatorial PSO.
Today, the list of such nature-inspired algorithms has become quite long. For instance in the survey by Fister Jr et al. (2013) more than 70 methods are classified, most of them coming from the field of SI approaches. The search and optimization strategies from the field of EC and SI are also denoted as metaheuristics. But, some of them include other general problem-solving approaches, not necessarily referred to the paradigm “nature-inspired”.

The world of Neural Networks (NN) is even larger, where Perceptron Neural Networks (PNN) (also denoted as Feedforward Neural Networks or FNN) and Self-Organizing Maps (SOM) are the most prominent methods of NN. Fuzzy Logic (FL) and Artificial Immune Systems (AIS) are the other main CI branches. The branches NN, FL and AIS are only shortly mentioned. For, the main focus in our book is laid on EC and SI, because these respective methods are especially successful in logistics applications. At the end of this chapter we also discuss briefly a few further metaheuristics, which share similarities with EC and SI methods and also belong to the most frequently used approaches for logistics problems.
2.1.4 Application Areas of Computational Intelligence

2.1.4.1 Optimization and Search

A large, perhaps the largest, application area of CI is optimization in all its facets (Burke and Kendall 2006). In optimization, respectively mathematical optimization, the problem is—generally speaking—to find minima or maxima, which are possibly subject to constraints.

An optimization problem is mathematically defined as follows:

*Given*: A function \( f : A \rightarrow \mathbb{R} \) from some set \( A \) to the real numbers

*Sought*: An element \( x_0 \) in \( A \) such that \( f(x_0) \leq f(x) \) for all \( x \) in \( A \) (so-called minimization) or such that \( f(x_0) \leq f(x) \) for all \( x \) in \( A \) (so-called maximization).

Mathematical definitions are well suited to clearly and uniquely explain a mathematical aspect. However, for non-mathematicians, these definitions are often not self-explaining. Therefore, we explain optimization to our students as follows, using the illustration shown in Fig. 2.2.

An obviously identifiable continuous optimization problem is to find the highest peak in a mountain range: The borderline between the mountains and the sky is comparable to the mathematical function \( f(x) \) denoting the height of the mountains at every position \( x \). The domain \( A \) is the search space (also called choice set) where all possible positions \( x \) are forming the set of candidate solutions (feasible solutions). A candidate solution \( x \) is thus a member of the search space \( A \) of possible solutions to a given problem.

The function \( f(x) \) is the objective function, which is sometimes called cost function (in minimization problems) or utility function (in maximization problems) or fitness function if an evolutionary computation method is applied.

The optimization goal is to find the specific feasible solution \( x_0 \) that maximizes the objective function. This particular feasible solution \( x_0 \) with \( f(x_0) \geq f(x) \) for all

\[ f(x) = \text{objective function} = \text{utility function} = \text{cost function} \]

\[ f(x_0) = f(\text{optimal solution}) = \text{optimum} = \text{maximum} \]

![Fig. 2.2 An optimal solution \( x_0 \) and its maximum \( f(x_0) \) of the objective function \( f(x) \)]
x in A is then called the optimal solution. In minimization problems, the goal is to find \( f(x_0) \geq f(x) \).

Additionally, an optimization problem often involves constraints, which limit the search space. Mathematically, the candidate solutions have to additionally satisfy inequality and equality constraints, such as \( g(x) \leq 0 \) and \( h(x) = 0 \).

Optimization problems can comprise continuous variables \( x \) or discrete \( x \) or combined ones in the search space. The cost function may be continuous or discrete and might exist only implicitly. Some optimization problems are combinatorial optimization problems, where a certain sequence of possible entities has to be found. Different ways to deal with such optimization problems are shown later.

In physics or engineering, optimization is often related to the maximization of force, power, efficiency or stability, or to the minimization of energy consumption, loss, time or faults. In economics, optimization tends to maximize profit, return-on-investment (ROI), economical gain, whereas minimization yields the smallest amount of invested resources. E.g. in logistics, optimization often yields the shortest, fastest or cheapest transportation route.

Related application areas are search problems which sometimes cannot be described as optimization problems. In search problems, the objective is to find either an item with specified properties among a set of other items, or even to find a feasible solution, or to find a diverse set of good alternatives (e.g. sets of local minima or maxima). Therefore, additional add-ons to the algorithms (such as niching or sharing) may extend the classical optimization algorithms.

The methods of CI do not make a real distinction between optimization and search. Depending on the characteristics of the problem (e.g. NP-hard) and on the search space (e.g. discrete, continuous) some CI methods are more suitable than others.

In general, the methods of CI are heuristics or metaheuristics: An approach is called a heuristic, if it is used to obtain high-quality solutions, although optimality cannot be guaranteed without checking all the possible solutions. Metaheuristics are based on general heuristic concepts. They can usually be applied to a wide area of different search or optimization problems, especially in computationally hard domains, where exact optimization approaches sometimes appear as insufficient or too time-consuming. Although convergence verifications (i.e. proofs for obtaining optimal solutions under suitable conditions) for some of the well-known metaheuristics are available, the notion of “metaheuristics” has been kept.

Some search methods or algorithms perform a deterministic search. This means that they always return the same answer, given exactly the same input and starting conditions. But, CI methods are (mostly) non-deterministic search methods or algorithms. This means that they may follow different optimization paths and return different answers given exactly the same problem. However, the different answers should (but need not) converge in the course of time to equally good solutions.

Furthermore, CI methods can do more than just mathematically search and optimize some variables: Some of these methods provide the ability to learn and classify, e.g. as used in data mining (e.g. particularly NN, AIS). Some are used to recognize patterns, in printed or handwritten texts as well as in spoken language.
(e.g. NN). Others are used to adapt to changing environments or conditions, a quality which is needed in designing controllers (e.g. EC, FL). Some help model and simulate e.g. biological evolution processes, brain functionality or technical or economical processes (e.g. EC, SI, NN). Some enable evolutionary design and arts or the composition of music (e.g. EC). Some are used in computer games or in agent based systems (e.g. SI, EC, NN), to model and simulate experiments which cannot be tested in reality, to process big data and so on.

2.1.4.2 Multiobjective Optimization

If more than one objective function has to be optimized mutually, the problem becomes a multiobjective optimization (MOO) problem. Multiobjective optimization, or multicriteria or multiattribute optimization, is the process of simultaneously optimizing two or more conflicting objectives, subject to certain constraints. Generally, in such multiobjective problems one cannot identify a single solution that simultaneously optimizes each objective. While searching for solutions, the attempt to improve an objective further must lead to downgrade another objective. The result is a set of solutions, which are called non-dominated, Pareto optimal, or Pareto efficient. They are characterized by the fact that they cannot be neglected by replacing them with other solutions which improve one objective function without worsening another. Solving a multiobjective optimization problem means to find such non-dominated solutions, the Pareto optimal solutions.

In the section Multiobjective Evolutionary Algorithms, two evolutionary algorithms which perform a so-called Pareto optimization are presented.

2.2 Methods of Computational Intelligence

2.2.1 Evolutionary Computation

Evolutionary Computation (EC) goes back to the 1950s and 1960s, where new schemes of optimization via self-adaptation were investigated. At that time, EC was counted as a sub-field of AI. Today, EC is noted as a sub-field of CI as well, because the methods belonging to EC are adapting biological evolutionary mechanisms. Many expressions of the biological evolution have been used to describe the computational algorithms of EC:

- In general, the EC methods work with a set of solution candidates. Such a set is called a population. The solution candidates are referred to as individuals. The individuals are often encoded by so-called chromosomes, just like in nature.
- The EC methods are iterative methods. Within one loop, the set of individuals is called a generation. Thus, in EC the solution is computed by using a population of changing individuals, each of them representing a solution candidate.
• The EC methods iteratively improve the “quality” of the individuals. This quality is called fitness and is measured by the objective function, called the fitness function. While fitness often implies that the fitness has to be maximized, in minimization problems the objective function might be named as cost function.

• The iterative steps are generations where—from one generation to the next—some parent individuals pass on their good features and traits to the child individuals using mechanisms of reproduction such as crossover and recombination, mutation and selection.

The methods of EC are well suited to solve optimization and search problems just like evolution continuously optimizes the living beings in specific environments. These real-life environments are the search spaces in the mathematical optimization. If the search spaces change over time, the optimization process has to follow the moving targets. EC does so automatically due to its iterative character. This can be used in the development of controllers (e.g. driverless autonomous cars or control of the combustion process in burners (Büche et al. 2002)). If we understand the optimization process of EC as a design process, many applications can be found in design (e.g. development of gas turbines (Dornberger et al. 2000), the Mars Rover antenna, and computer circuits). Beyond mathematical optimization, natural sciences and engineering, the methods of EC are used in economics, social sciences and art.

2.2.2 Evolutionary Algorithms

Evolutionary Algorithms (EA) are either considered a sub-category of EC, or a synonym of EC. In general, EAs describe a set of stochastic, generic population-based, (meta)heuristic optimization algorithms. All the algorithms belonging to EA are characterized by reproduction applying selection, recombination, and mutation. Applying these operators leads to an artificial evolution process on a computational level mimicking the real evolution in particular aspects.

The most important methods that belong to EA are Genetic Algorithms (GA), Evolution Strategies (ES), Genetic Programming (GP), and Evolutionary Programming (EP). Surprisingly, Swarm Algorithms (SA) are sometimes also considered belonging to EA; but we link the SA to the other independent big branch of CI named Swarm Intelligence (SI). Additionally, all these methods provide different variants which result in many further metaheuristics: Differential Evolution (DE), Learning Classifier Systems (LCS), Harmony Search (HS) etc.

Hybrid metaheuristics result by merging two or more methods. Memetic algorithms (MA) were developed by combining EA methods with local search methods. Metaheuristics, which are hybridized with approaches from mathematical programming (such as branch-and-bound, the Simplex algorithm or Newton gradient methods), are usually denoted as matheuristics.
The algorithms defer by the representation of the problem, i.e. the coding of the solution candidates, by different working modes of the EA operators and by the order of the execution of these operators. Here, recombination does not necessarily need to take place, because one parent changed by mutations might be sufficient.

The principles of the EA methods are simple. The following list shows how narrow the relation between certain aspects in nature or biology and the computational algorithms of EA are:

- The paradigm taken from evolution in nature is Darwin’s principle “survival of the fittest”.
- In nature, biology encodes the properties of an organism, called phenotype, using four amino acids. On the computational level, this limitation does not exist; various, totally different encodings of the candidate solutions are possible on the level of the individuals. In general, the coding takes place with binary, integer or real values or with a set of given objects carrying the hereditary information, called genotype.
- Every individual, in the EA every encoded candidate solution, obtains a particular quality (called fitness) showing how good it is in solving the problem—in relation to the other candidate solutions.
- Mimicking the natural selection process, the better individuals are selected with higher probability in order to produce the children (new candidate solutions) for the next generation (next iteration).
- The selected individuals are recombined, where mutation additionally adds genetic variations.

In computer science, these principles are implemented into an algorithm. The sequence and importance of the single principles might differ from case to case. But, the general flowchart of an EA can be noted as follows and shown in Fig. 2.3 (left):

1. A random initial set of individuals as zero (initial) generation is generated.
2. The fitness of each individual is computed.
3. While no termination criterion is fulfilled, a next new generation is generated.
4. Therefore, particular individuals, depending on their fitness, are selected in order to form the new set of individuals in the new generation.
5. (Within a certain probability) they are recombined using crossover.
6. (Within a certain probability) they are mutated.
7. Their new fitness is computed.

The class diagram of an object-oriented programming (OOP) implementation of an EA is shown in Fig. 2.3 (right):

- The class Population contains 1-to-many Generation(s). All Generations belong to this Population.
- Each Generation consists of 1-to-many Individual(s), where these Individuals belong to this Generation.
- Each Individual has exactly 1 Fitness value.
The selection defines which individuals form the population of the next generation. It can be used in different ways: Either those parent individuals are selected which are further used for recombination and mutation. Or a selection is used to take out the best individuals of all newly generated individuals and the existing old ones to form the next generation. Or a selection defines which individuals are eliminated and will not be used for the offspring. Hence, different selection operators are possible:

- The standard selection takes the \( n \) fittest, ranked individuals as parents, where \( n \) is the number of individuals per generation.

- Roulette-wheel selection allows a fitness-proportionate selection, where the possibility of choosing an individual depends on the magnitude of its fitness value in relation to the fitness values of the other individuals.

- Tournament selection takes the best individual of two randomly chosen parent individuals.

**Fig. 2.3** General flowchart of an EA (left) and the class diagram of an object-oriented programming (OOP) implementation of an EA (right)
• *Truncation selection* takes a certain portion of the best individuals.
• *Elitism selection* (shortly called *elitism*) always copies the best individual from the old generation into the new generation in order to guarantee that no setback in the evolution process happens.

Different selection operators can be combined. Particularly, elitism is used frequently in order to always preserve the last best solution.

The advantages of the EA methods are that they are (almost) problem-independent and very robust. Thus, EA can be used as black-box optimization and search algorithms. (Almost) no knowledge about the problem which should be solved and only a slight knowledge about the search space are necessary. Various classes of optimization problems can be solved, because the problems must neither be continuous nor differentiable. The EA is a stochastic exploration of the search space for local and global optimization, and—in the ideal case—tremendously faster than an exhaustive search.

### 2.2.2.1 Evolution Strategy

In the 1960s and 1970s, Ingo Rechenberg (1994) introduced new methods for optimizing engineering designs, such as aerofoils or airplane structures. He called this approach *Evolution Strategy* (Rechenberg 1973) (ES), because a kind of artificial evolution was performed: The features of the better designs were combined, stochastically slightly changed and passed on to a next generation of modified designs. Although this approach was carried out on hardware designs, it is obvious that principles belonging to evolutionary computation were performed. Rechenberg combined this approach with bionics, transferring mechanisms from nature to technical respectively engineering products.

While Rechenberg at the beginning applied the ES rather in experimental optimization on hardware designs, Hans-Paul Schwefel (1977) extended the ideas more to computational optimization, and encoded the technical designs (phenotypes) using vectors with the real numbers from $\mathbb{R}^n$ (genotypes).

The principles of the ES for computational optimization are the following: A set of vectors consisting of $n$ real-valued components represents a set of different solution candidates. Each vector component describes a particular feature and can be mutated by adding a normally distributed random value. The magnitude of this random value is called the mutation strength and is determined by certain adaptation mechanisms. Each vector is tested to solve the problem, and ranked depending on its fitness value.

Different main types of the ES with particular $(\mu, \lambda)$-notation or $(\mu + \lambda)$-notation are possible. $\mu$ stands for the number of recent candidate solutions in the parent population, and $\lambda$ for the number of additional candidate solutions generated from the parent population:

• A $(1 + 1)$-ES is the simplest ES. This configuration consists of one parent vector (candidate solution) and one modified vector. The better candidate solution is
taken as parent for the next generation. The simplest ES corresponds to the greedy hill climbing algorithm: In each iteration, the greedy algorithm takes the actual best candidate solution out of a set of at least two alternatives.

• More common is the \((1 + \lambda)\)-ES, which consists of one parent vector and \(\lambda\) modified ones. Here, the \(\lambda\) modified candidate solutions plus the recent parent solution compete to be selected for the next generation.

• In the \((1, \lambda)\)-ES, only the \(\lambda\) modified vectors compete to be selected for the next generation.

• In order to overcome local optima, the \((\mu, \lambda)\)-ES proposes a generation which consists of \(\mu\) parent vectors and \(\lambda\) modified ones, where (if \(\lambda > \mu\)) the \(\mu\) best candidate solutions of the new \(\lambda\) ones are taken for the next generation.

• In the \((\mu + \lambda)\)-ES, all \(\mu\) parent and \(\lambda\) modified vectors compete in the selection process for the next generation.

• Often, a recombination between the parent vectors is additionally applied, denoted by \(\rho\) and \((\mu/\rho, \lambda)\)-ES. \(\rho\) is the number of parents who contribute to the next generation using particular recombination operators.

In ES, mutation is the driving operator to modify the vectors representing the candidate solutions. However, to find the optimal magnitude of mutation, called the mutation strength, it is an optimization problem by itself. Rechenberg introduced the so-called 1/5-rule, which proposes to adjust the mutation strength in such a way that in each generation around 20% of the generated candidate solutions are better than the original parent ones.

Another method to find a suitable mutation strength is self-adaptation, which integrates an additional component into the vector containing the mutation strength value. Thus, the mutation strength is optimizing itself.

A further extension of the original ES is the controlled mutation of vector components by a so-called covariance matrix adaptation (CMA). This kind of ES is called the CMA-ES (Hansen and Ostermeier 2001), which guides the mutation strength in the direction of the possibly highest increase of the fitness value. Although ES are stochastic derivative-free methods, the CMA-ES works similarly to the usage of derivative information in gradient methods.

2.2.2.2 Genetic Algorithm

In the 1970s, John H. Holland (1975) developed a method for solving search and optimization problems, applying the principles of EA. He called the method Genetic Algorithm (GA) and the candidate solutions individuals. These individuals are encoded by chromosomes, which are typically implemented as vectors or strings. Each component of the string (or vector) is called a gene. The main difference to the real-valued ES is that Holland introduced the simple binary coding \(\{0, 1\}^N\) for encoding a solution instance.

While all variants of ES work rather on phenotype level, the variants of GA fully optimize on genotype level which is mapped from the real problem instances down
To move to this level of encoded individuals, a unique mapping from the representation of the search space (e.g. real valued or integer discrete) is necessary. This is called a genotype-phenotype mapping. Holland’s basic idea was that a representation with binary digits would be the easiest representation for a computer to handle.

Furthermore, the two values (0 and 1) of GAs are similar to the four values of the nucleic acid in nature (base-4 chromosomes of the DNA). However, the advantages of the original GAs with a binary coding diminished when computational power was increasing. Nowadays, various encoding schemes are used in GAs. A variable length coding of the chromosomes is also possible.

In a GA the genetic operators and the computation of the fitness work in the following way:

- **Recombination/crossover**: Recombination was originally introduced as a simple crossover, where two parent individuals exchange their genes at a certain crossover position, as shown in Fig. 2.4. Besides a one-point-crossover, an n-point-crossover exchanges sequences of genes between two individuals. In combinatorial problems, special types of crossover, such as partially mapped crossover (PMX), allow to recombine individuals’ information without producing (too much) child individuals with invalid coding.

- **Mutation**: Using the binary coding, a mutation is obvious, because the bit of the particular gene has only to be inverted (Fig. 2.4). With the other coding schemes, a mutation has to be defined as how the value of a gene is changing to which other value.

- **Fitness**: The computation of the fitness value needs a remapping from the genotype representation to the phenotype level, because the problem that must be solved is often not represented in a binary search space. Furthermore, GAs can be interactive in the sense that the evaluation of the fitness value happens outside the GAs—perhaps interactively by the user or within a technical unit.

Holland and Goldberg (Goldberg 1989) describe the ability of GAs to converge to optima with the building block hypothesis. It says that a GA is able to collect positive traits stored in short sequences of genes, called building blocks, which are inherited and summed up over generations of improving individuals.
2.2.2.3 Genetic Programming

Genetic Programming (GP) was elaborated by John R. Koza (1992) in the 1980s to be applied for function approximation and for the automatic generation of computer code and programs. GP is based on GA and uses a genotype representation of the problem, e.g. of the mathematical formulae or of the entire source code. While GA mostly uses fixed-length individuals, GP allows individuals of non-fixed length.

In the standard GP, the chromosomes are tree-like representations of the mathematical functions or code expressions, as shown in Fig. 2.5. Using such trees, GP aims to optimize the entire structure of the tree rather than to optimize solely the values or operators of certain nodes.

The initial idea of GP prefers programming languages which naturally use tree-based internal data structures such as Lisp. Each chromosome is a code candidate representing a tree. Then, the nodes—and thus all branches depending on that node—are free to be exchanged, recombined and mutated. This happens not only between different chromosomes, but also within each particular chromosome. A side effect is that many code candidate solutions with an invalid syntax are generated, and have to be sorted out. The fitness is evaluated by the fulfilment of the syntax rules, the requirements of the developed code, and its “elegance”, or rather its “parsimony”.

There are many extensions of the original GP: In order to also allow other programming languages (beyond tree-based languages), such as the assembly language, or imperative ones, GP has been extended to Linear Genetic Programming (LGP). LGP codes the instructions used in the programming languages in the chromosomes as a sequence of instructions. The standard GA operators are then applied to improve the code over time. In Grammatical Evolution (GE), the entire computer program is encoded in an integer string using a particular grammar: Each possible code entity is a certain integer value, and GE applies the GA operators to the string (chromosome). GE allows the optimization on the abstract genotype of the problem, while the standard GP works rather on the phenotype level on the direct code. The Gene Expression Programming (GEP) uses chromosomes of fixed length, which are coded as trees. Modified genetic operators avoid the creation of an invalid code. Meta-genetic Programming (MGP) gives more freedom to evolve how and where the genetic operators (recombination and mutation) can modify the chromosomes. MGP allows not only the optimization of the code, but also the optimization of the genetic optimization process by itself.

Fig. 2.5 Tree-structure representation of the function $5x^2 + 2y$
2.2.2.4 Evolutionary Programming

*Evolutionary Programming* (EP) was introduced in the 1960s by Lawrence J. Fogel, Alvin J. Owens, and M. J. Walsh (Fogel 1999). EP is a special kind of an ES, originally used to optimize the parameters of finite state machines for deriving computer programs or logic circuits. EP fully works on the level of phenotype and does not need a genotype representation. Thus, in contradiction to GP, EP aims to directly optimize the numerical parameters of a given problem (e.g. computer program), but not the structure of the source code itself.

EP mainly applies self-adaptive mutation to a set (i.e. the population) of specific individuals. In general, each parent individual is taken to generate a child individual. No classic crossover is applied, but a variation operator allows the combination of the information of different parents for the offspring. The selection is in reverse to the selection of EA: This EP selection, called the survivor selection, removes the individuals which will not to be taken into the next generation.

2.2.2.5 Multiobjective Evolutionary Algorithms

*Non-dominated Sorting Genetic Algorithms* (NSGA) and *Strength Pareto Evolutionary Algorithms* (SPEA) are variants of EA (respectively of GA) used in multiobjective optimization (also referred to as multicriteria or Pareto optimization), that solves optimization problems with more than one objective function to be optimized simultaneously. This kind of optimization is a kind of multiple criteria decision-making where the result of the optimization process is a multi-dimensional hyper-surface of non-dominated (Pareto optimal) and dominated solutions.

Two important example methods of EA, which are effective to solve such multiobjective optimization problems are the *Non-dominated Sorting Genetic Algorithms* (NSGA and the improved NSGA-II) (Deb et al. 2002) and the *Strength Pareto Evolutionary Algorithms* (SPEA and the improved SPEA2) (Zitzler et al. 2001). They use the advantage that a GA is a population-based method. Applied to multiobjective optimization problems, the individuals converge to the Pareto front, while the distribution mechanisms within these methods ensure that the single individuals (candidate solutions) are widely spread along the Pareto front.

2.2.2.6 Memetic Algorithms

*Memetic Algorithms* (Chen et al. 2011) (MA) are a set of hybrid algorithms combining EA (or at least any population-based approach) with local search or learning strategies, often deterministic ones. Examples of MAs are also mentioned as Baldwinian or Lamarckian EAs, cultural algorithms or genetic local search. The basic idea is that the evolution process of the EA is influenced, controlled or adapted by selectively modifying the individuals in the population within each
iteration step. A meme is a cultural feature (e.g. a trait or behavior) passed on from person to person. Roughly speaking, MAs are extended EAs, where additional meme features provide additional strategies or knowledge about the problem in order to improve the optimization process.

The standard MAs (sometimes called MAs of the first generation) integrate an additional improved local search during the evolution process of the EAs, because EAs are effective in their population-based global search, but often not suited to converge exactly to the optimum. Consequently, near the optimum a local refinement strategy is more effective. Standard MAs combine these two strategies on the level of evolving the individuals (i.e. solution candidates) individually by selectively applying local search strategies.

The extended MAs, often referred to as multi-meme MAs (or MAs of the second generation) directly modify the individuals by applying possible meme features, i.e. additional knowledge about the problem, to the solution candidates. Within the iteration steps, not only the solution candidates improve, but also the kind and manner of memes which are applied. It is a kind of self-fulfilling prophecy: The individuals carrying the better meme features are superior, which leads to the point that these memes are selected to improve the solution candidates still further. While in this kind of MA the set of possible memes is predefined, self-generating MAs (or MAs of the third generation) develop themselves possible memes within the (meta-)optimization process.

2.2.2.7 Other Evolutionary Computation Techniques

In a Learning Classifier System (LCS) not only states or properties are evolved, but also rules. The rules themselves and the sequence of applying them are optimized to undertake certain actions in a particular environment. These rules are encoded in the chromosomes, which are modified using the GA operators. The principle of reinforcement learning computes the fitness value measuring the quality of these rules and the related actions in the particular environment.

Differential Evolution (DE) is related to EA, working with real-valued populations and operators to be used in ES and GA. DE is mainly used for optimization problems with real-valued multidimensional, often noisy and time-changing problems, where no gradient information is available. Its main extension lies in the generation of the child individuals (candidate solutions) using the information of three parent candidate solutions. The combination of information stored in these three parents allows guiding the optimization process faster in the direction of higher improvement.

Harmony Search (Geem et al. 2001) (HS) iteratively modifies the vector components in a set of solution candidates following the analogy of improvising melodious, harmonic music. Its specialty lies in the combination of all solution candidate vectors and the corresponding fitness to one matrix, the harmony memory. The entries of the matrix are selectively modified by applying methods of composing or arranging music, converging to a global optimum.
Other recent evolutionary computation techniques are *Neuroevolution* (evolving neural networks using principles of GP), *Evolutionary Robotics* (evolving control- lers for autonomous robots), and *Learnable Evolution Model* (the evolution is guided by hypotheses rather than by the randomly applied EA operators for generating new individuals).

### 2.2.3 Swarm Intelligence

The interest in research in *Swarm Intelligence* (SI) started around the early 1990s (Meystell et al. 1990). The methods belonging to SI follow the principle of swarms, which is commonly stated as *optimization via emergence* or *the whole is more than its single components*. Emergence means here the appearance of patterns, larger entities, or regularities. Thereby, SI is copying the collective behavior of self-organized multi-agent systems, found in nature or natural processes: A population of simple agents (individuals, candidate solutions) are interacting locally, mostly following simple rules, and leading to an intelligent global behavior. The single individual has usually only a minor importance.

An intelligent behavior appears only after the interaction with other, mostly similar individuals. Stochastic mechanisms provide a variation within the population in order to explore the living environment (i.e. search space) and solution paths sufficiently without (almost) any additional knowledge about the problem. This swarm behavior is mostly used to solve optimization and search problems. A large applications area is logistics, where often combinatorial aspects lead to very large search spaces.

*Swarm Algorithms* (SA) are often mentioned as a sub-category of both, SI and EC, because they follow the principles of swarms and of evolution. Of late, SA are considered more and more as the representative method of SI. Sometimes, SA is even used as a synonym for SI.

The most important methods belonging to SI respectively SA are Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO), Bees Algorithm (BA, recently more commonly denoted as Artificial Bee Colony (ABC) optimization) and Cuckoo Search (CS). Many other methods can, for instance, be found in the classification by Fister Jr et al. (2013). PSO knows further sub-methods such as binary, discrete or combinatorial PSO. The Firefly Algorithms (FA) and the very similar Glowworm Swarm Optimization (GSO) are considered as sub-methods of SA, too. Constantly, further sub-methods of SI respectively SA are emerging.

The application areas of SI methods mostly surround logistic or networking problems, such as routing, transportation and network configuration problems, but as well the simulation of crowds or heaps of particles and/or of particular items. Other application areas are the development of software controllers and the field of robots, consequently mentioned as swarm robotics.
2.2.3.1 Particle Swarm Optimization

*Particle Swarm Optimization* (PSO) is used to find the global optimum—in its standard form—in a continuous \( n \)-dimensional search space. In PSO, the candidate solutions are named *particles* instead of individuals as used in EC. These particles form a population, the swarm, which is moving through the search space. Iteratively, the particles of the swarm with the relatively better fitness attract the other particles to let the swarm move towards a local optimum while some particles search always the space for a better fitness.

In the basic variant of PSO, every particle \( p_i \) of the swarm of the size \( n \) is moving in the search space depending on its actual position \( x_i \) and its particular velocity vector \( v_i \). The velocity vector of each particle, thus each individual’s movement, is additionally influenced by the positions and velocities of the other particles in the swarm, preferably, the most by the recently best known particle(s) of the swarm. After an iteration, all the particles \( p_i \) move to a new position depending on their former position and their adapted velocity vectors. Another particle might now be the best particle of the swarm.

State of particle \( p_i \) with its initial position \( x_i \) and velocity \( v_i \):

\[
p_i = (x_i, v_i)
\]

\( v_i^{\text{new}} \) is the adapted velocity of particle \( p_i \) depending on the velocity of the other particles \( j \) calculated with the influence factors \( a_{ij} \). These influence factors mainly depend on the distance between the two particles and the relative fitness of particle \( j \) compared to particle \( p_i \).

\[
v_i^{\text{new}} = v_i + \sum_{j=1}^{n} a_{ij} (v_j - v_i)
\]

New position \( x_i^{\text{new}} \) of particle \( p_i \) after the iteration step \( \Delta t \) is:

\[
x_i^{\text{new}} = x_i + v_i^{\text{new}} \cdot \Delta t
\]

An improvement of PSO was introduced by James Kennedy and Russell Eberhart (Kennedy and Eberhart 1995). They defined the new velocity \( v_i^{\text{new}} \) as a function of two summands added to the recent velocity. The first summand is history-related in the sense of the position \( p_i^{\text{best}} \), where the particle \( p_i \) has found its best fitness in the search space. The second summand is defined as the position \( p_i^{\text{global}} \), where the swarm has found the best fitness so far.

\[
v_i^{\text{new}} = w_i v_i + c_i^{\text{best}} r_i^{\text{best}} (p_i^{\text{best}} - x_i) + c_i^{\text{global}} r_i^{\text{global}} (p_i^{\text{global}} - x_i)
\]
Here, the coefficients $c_{\text{best}}$ and $c_{\text{global}}$ are acceleration factors which allow the particle to adapt more or less to the behavior of the entire swarm or to be more individual. The random coefficients $r_{\text{best}}$ and $r_{\text{global}}$ follow a uniform random distribution on the interval $[0, 1]$, while $w$ is an inertia coefficient, normally reduced during the optimization process to increase the convergence ability.

### 2.2.3.2 Discrete Particle Swarm Optimization

The *Discrete Particle Swarm Optimization* (DPSO), also referred to as *Combinatorial PSO*, or in a digital case as *Binary PSO*, is extending PSO to discrete search spaces, as found in combinatorial problems. Here, the position $x_i$ of the particle $p_i$ is a solution vector of discrete numbers in an $n$-dimensional discrete search space. For example, in the combinatorial problem of arranging uniquely the five numbers $\{1, 2, 3, 4, 5\}$ downward with respect to their value, a solution candidate might be $x_i = (3, 2, 4, 5, 1)$, where the optimal solution is $x_{\text{opt}} = (5, 4, 3, 2, 1)$.

In contrast to adding in each iteration a velocity-driven component to the recent position, the velocity vector $v_i$ now defines one or even a set of exchange operations to be performed on the candidate solution $x_i$. E.g. $v_i = \{(1, 4)\}$ defines that the first and the forth entry are exchanged:

$$x_{i}^{\text{new}} = v_i(x_i) = v_i^{(1,4)}(x_i) = v_i^{(1,4)}(3, 2, 4, 5, 1) = (5, 2, 4, 3, 1)$$

### 2.2.3.3 Ant Colony Optimization

*Ant Colony Optimization* (ACO) was proposed by Marco Dorigo in the early 1990s (Colorni et al. 1991). In ACO, the individuals are named *ants*, which try to find the shortest way from their colony, the starting point, to a so-called food source through a graph of possible ways in the search space. Often, ACO is applied in scheduling, routing or assignment problems, where the sequence of discrete steps has to be optimized. Today, many different variants of ACO exist.

Originally, the foundations of the computational optimization algorithm ACO are highly inspired by the behavior of real ants: Ants mark their crawling trails using pheromones, an odorous substance, which is continuously set while the ants move on these trails. The pheromone level is slowly vaporizing over time. But, the more often the same ants or further new ants crawl along the same trail, the more intensive the pheromone level will be. Then, passing ants are influenced by the pheromone intensity and will—with higher probability—change their route to the more attractive trail with the higher pheromone level.

Every ant is representing a solution candidate vector $x_i$ (the trail) as a sequence of discrete edges, e.g. $x_i = (A, D, B, C, E)$ from starting point A to endpoint E. At the beginning, all ants and pheromone levels on the edges are randomly initialized. The fitness of each ant is computed. Then, the ants (preferably only those with the better fitness values) add a new pheromone level to their trails, thus on all the edges
of their trials. Afterwards, every ant is updating its solution candidate vector $x_i$, where some edges can be changed due to a trade-off between the part-fitness of the edge and the pheromone level set on it. From iteration to iteration, the ants tend to follow the sub-trails with stronger pheromone level. Over the time, the best route is found marked with the strongest pheromone level.

2.2.4 Neural Networks

The computational models, which simulate the functionality of real brains or at least of small parts of them on the computer, are called Neural Networks (NN). Literature sometimes uses the name Artificial Neural Networks (ANN) as well in order to distinguish between those neural networks of biological brains of animals and human beings, and the computational model of NNs. In this book, we use only the abbreviation NN for the computational models, omitting the “A” for the adjective “artificial”.

NNs use the principle of natural cognition. Inspired by real biological neural networks of brains, NNs are processing information through a network of interconnected artificial neurons—in an abstract view similar to the information processing in biological brains. Various types of NNs exist, where the most prominent ones are Feedforward Neural Networks (FNN) respectively Perceptron Neural Networks (PNN), and Self-Organizing Maps (SOM).

The FNNs are the simplest NNs. They use a data flow from the neurons of the input layer through neurons of hidden layers (if there are any at all) to the neurons of the output layer. While multiplying the respective input values with the weights of the connections between the neurons, each neuron is summing up its input. An activation function defines a threshold when the neuron transmits an output value larger than zero. This simple algorithm is called perceptron. During a training phase of the NN prior to its application the weights and possibly other network parameters are adjusted. Using sets of training data all the weights in the NN are adjusted so that the input data is matched as good as possible to the wanted output data. The main aspect is back-propagation of some information of the transmitted data in order to train the NNs. In simple NNs the number of neurons per layer and the number of layers is predefined. High-level NNs allow to increase the number of neurons, layers and connections and thus the depth within the network.

SOMs are used for unsupervised mapping of high dimensional to low dimensional data, e.g. used in clustering problems. SOMs are often called Kohonen maps named after Teuvo Kohonen who introduced this kind of NNs in the 1980s (Kohonen 1982). In contrast to the classical NNs, SOMs use certain kinds of neighborhood functions to identify topological properties of the input vector. SOMs work with training data and then with mapping. The training data vectors build a map, which is then used to classify an unknown new data vector.
The main application areas of NNs are classification and optimization. Here, NNs implicitly evaluate the relationships between input and output data, finding patterns. NNs are fault- and noise-tolerant using non-linear threshold activation functions.

2.2.5 Fuzzy Logic

Fuzzy Logic (FL) applies the principle of fuzziness by assuming that logical values are not just true or false, or either 1 or 0, but can adopt any value from the interval [0, 1]. Similar to human cognition and human reasoning, FL allows a fuzzy modelling of the problem. FL uses a many-valued logic represented by a set of logical values allowing approximate values and incomplete or ambiguous data. FL is able to provide conclusions and approximate solutions based on fuzzy information. Its application areas are the modelling of fuzzy expressions, making inference based on fuzzy information, generating fuzzy sets of rules, and clustering of data. FL is often considered as fault- and noise-tolerant using weak rules.

2.2.6 Artificial Immune System

The methods which belong to Artificial Immune System (AIS) follow the principles of combatted pathogens in biological bodies. The human immune system identifies particular pathogens as a potential source of illness, which are then called antigens. Thus, AIS is inspired by the immune system of humans and animals. The AIS methods use a population of agents which learn to identify certain patterns of antigens that intrude a system. Over the generations, these agents improve via positive stimulated selection, clonal reproduction and adaptation to identify possibly changing pathogens as well. Therefore, the characteristics and processes of the immune system for learning and memory are transferred to AIS. Application areas of AIS are the development of adaptive systems and optimal behavior for problem solving as well as pattern recognition.

2.2.7 Further Related Methods

2.2.7.1 Reinforcement Learning

Reinforcement Learning (Kramer 2009) (RIL) follows the principle of “reward and punishment”. Desired behavior is rewarded whereas undesired or not goal-oriented behavior is punished. Different learning strategies (e.g. dynamic programming,
temporal difference learning) allow the system to iteratively learn a better behavior. Application areas are the control of characteristics and the learning of optimal behavior.

### 2.2.7.2 Simulated Annealing

*Simulated Annealing* is a generic probabilistic metaheuristic algorithm. Although this method is not inspired by the (living) nature, but by physics or more precisely by thermodynamics (the annealing process in metallurgy), we count Simulated Annealing to the CI methods. It follows the thermodynamic principle of a controlled cooling down of melted metal in order to increase the size of its arising crystals and reduce their defects. During the incremental steps of successively reducing the temperature, local minima or maxima are skipped in order to find the global optimum. The application areas of Simulated Annealing are in global optimization, where the search space is mostly large and discrete.

### 2.2.7.3 Further Metaheuristics: Local and Tabu Search

In the following, we briefly discuss further metaheuristics, which are not nature-inspired, but share similarities with several of those nature-inspired approaches portrayed above. An additional reason to include them in this book is that these metaheuristics belong to the most successful methods applied to various hard-to-solve problems in logistics.

#### Local Search, Neighborhood Search

One of the most common approaches in metaheuristics is the strategy of performing a *Local Search*—also denoted as *Neighborhood Search*. The underlying concept is that it is usually more promising to search improved solutions in the neighborhood of already determined solutions than in arbitrary regions of the entire search space.

Another reason is that local search is often used for large search spaces, which grow exponentially with the number of variables. While searching the whole set of feasible solutions is mostly not acceptable with respect to computation time, searching a small neighborhood area can be done very quickly. Obviously, local search requires the definition of neighborhoods of solutions, which depend on the particular problem type and can be referred to as distance between solutions or number of operations to transfer one solution into another one. Local search is—in its basic form—an iterative method, which is “jumping” from one solution to another better one in the neighborhood until a termination criterion (e.g. maximum number of iterations or inability to improve) is fulfilled. An
obvious drawback of local search is that it can get stuck in a local, non-global optimum.

Quite often, local search is used as a background concept, on which other more complex metaheuristics are built, or which use local search as an embedded method for the improvement of intermediate solutions. For instance, simulated annealing uses local search in a more complex way and memetic algorithms include them in the framework of evolutionary algorithms. Also, typical mutations in evolutionary algorithms could be interpreted as terms of local search, because they usually prefer the determination of new solutions close to parent solutions.

Local search is further extended by combining it with other metaheuristics: In Multistart Local Search (Hart 1998), the problem of ending up in a local optimum, but not the global one, is tackled by restarting the local search from new (for instance randomly determined) starting solutions.

In Iterated Local Search (ILS) (Den Besten et al. 2001), the problem is treated by a particular perturbation of the found local optimum. The search continues repeatedly. In contrast to a multistart local search, the perturbed solutions should be sufficiently different from the previous solutions (to escape the local optima), but not “totally random”.

Tabu Search

One of the most well-known advancements of local search is Tabu Search (Glover 1989, 1990), where the name “tabu” denotes forbidden. The central idea of this approach is to use a list of previously visited solutions, which are to be avoided in the subsequent search. Instead of storing visited solutions it is also possible to store some properties of previously visited solutions which are to be avoided in the subsequent search. The idea of a so-called tabu list addresses the problem that often, especially in combinatorial optimization problems, identical solutions are visited again and again during the local search. Apart from avoiding unnecessary computation time, the tabu list may also help to escape local optima together with the concept of allowing the search also to progress with worse solutions found during the iterations. (Nevertheless the best found solution is always stored for consideration at the end of the search process.) There are many variants with respect to tabu search, e.g. concerning the definition and length of the tabu list and its usage within the search.

Variable Neighborhood Search

Another successful and widely used variant of local search is the Variable Neighborhood Search (VNS) (Mladenović and Hansen 1997). VNS assumes that not only one, but several neighborhoods of different sizes exist. Usually, it is assumed that
there are \( k \) neighborhoods, where a neighborhood with a higher index includes the neighborhoods with smaller indices.

VNS is based on the idea that local search should first take place in the smaller, closer neighborhood and then continue (when not being successful) with the next larger and wider neighborhood. Once a better solution is found the search continues, firstly again in its smallest neighborhood. This helps avoid getting stuck in a local optimum of a very small neighborhood, but also avoids spending too much computation time for search in large neighborhoods.

There are several variants of VNS. For instance, in Basic VNS a single, randomly determined point from a considered neighborhood is improved by local search before probing the next neighborhood. In Reduced VNS the embedded step of local search is omitted, i.e. if a randomly selected point is not directly better, the search continues with the next neighborhood. Variable Neighborhood Descent (VND) is based on the idea of doing a deterministic local search in a considered neighborhood.

**Greedy Randomized Adaptive Search Procedure (GRASP)**

As a last example of frequently used metaheuristics let us mention the Greedy Randomized Adaptive Search Procedure (GRASP) (Feo and Resende 1995), which is based on iterations for constructing and then improving solutions by local search. The construction phase is problem specific. The general idea is given by adding “solution elements” successively, which increase their “quality” in a greedy way. While adding these solutions elements, a particular randomness is included in order to maintain variability.

**References**


Computational Intelligence in Logistics and Supply Chain Management
Hanne, Th.; Dornberger, R.
2017, XX, 176 p. 20 illus., 14 illus. in color., Hardcover
ISBN: 978-3-319-40720-3