2 The Ant Colony Optimization Metaheuristic

A metaheuristic refers to a master strategy that guides and modifies other heuristics to produce solutions beyond those that are normally generated in a quest for local optimality.
—*Tabu Search*, Fred Glover and Manuel Laguna, 1998

Combinatorial optimization problems are intriguing because they are often easy to state but very difficult to solve. Many of the problems arising in applications are $\mathcal{NP}$-hard, that is, it is strongly believed that they cannot be solved to optimality within polynomially bounded computation time. Hence, to practically solve large instances one often has to use approximate methods which return near-optimal solutions in a relatively short time. Algorithms of this type are loosely called *heuristics*. They often use some problem-specific knowledge to either build or improve solutions.

Recently, many researchers have focused their attention on a new class of algorithms, called metaheuristics. A *metaheuristic* is a set of algorithmic concepts that can be used to define heuristic methods applicable to a wide set of different problems. The use of metaheuristics has significantly increased the ability of finding very high-quality solutions to hard, practically relevant combinatorial optimization problems in a reasonable time.

A particularly successful metaheuristic is inspired by the behavior of real ants. Starting with Ant System, a number of algorithmic approaches based on the very same ideas were developed and applied with considerable success to a variety of combinatorial optimization problems from academic as well as from real-world applications. In this chapter we introduce ant colony optimization, a metaheuristic framework which covers the algorithmic approach mentioned above. The ACO metaheuristic has been proposed as a common framework for the existing applications and algorithmic variants of a variety of ant algorithms. Algorithms that fit into the ACO metaheuristic framework will be called in the following ACO algorithms.

2.1 Combinatorial Optimization

Combinatorial optimization problems involve finding values for discrete variables such that the optimal solution with respect to a given objective function is found. Many optimization problems of practical and theoretical importance are of combinatorial nature. Examples are the shortest-path problems described in the previous chapter, as well as many other important real-world problems like finding a minimum cost plan to deliver goods to customers, an optimal assignment of employees to tasks to be performed, a best routing scheme for data packets in the Internet, an optimal sequence of jobs which are to be processed in a production line, an allocation of flight crews to airplanes, and many more.

A combinatorial optimization problem is either a *maximization* or a *minimization* problem which has associated a set of problem instances. The term *problem* refers to the general question to be answered, usually having several parameters or variables with unspecified values. The term *instance* refers to a problem with specified values for all the parameters. For example, the traveling salesman problem (TSP), defined in section 2.3.1, is the general problem of finding a minimum cost Hamiltonian circuit in a weighted graph, while a particular TSP instance has a specified number of nodes and specified arc weights.

More formally, an instance of a combinatorial optimization problem Π is a triple $(\mathcal{S}, f, \Omega)$, where $\mathcal{S}$ is the *set of candidate solutions*, f is the *objective function* which assigns an objective function value $f(s)$ to each candidate solution $s \in \mathcal{S}$, and Ω is a set of constraints. The solutions belonging to the set $\tilde{\mathcal{S}} \subseteq \mathcal{S}$ of candidate solutions that satisfy the constraints Ω are called *feasible solutions*. The goal is to find a *globally optimal* feasible solution s^*. For minimization problems this consists in finding a solution $s^* \in \tilde{\mathcal{S}}$ with minimum cost, that is, a solution such that $f(s^*) \leq f(s)$ for all $s \in \tilde{\mathcal{S}}$; for maximization problems one searches for a solution with maximum objective value, that is, a solution with $f(s^*) \geq f(s)$ for all $s \in \tilde{\mathcal{S}}$. Note that in the following we focus on minimization problems and that the obvious adaptations have to be made if one considers maximization problems.

It should be noted that an instance of a combinatorial optimization problem is typically not specified explicitly by enumerating all the candidate solutions (i.e., the set $\mathcal{S}$) and the corresponding cost values, but is rather represented in a more concise mathematical form (e.g., shortest-path problems are typically defined by a weighted graph).

2.1.1 Computational Complexity

A straightforward approach to the solution of combinatorial optimization problems would be exhaustive search, that is, the enumeration of all possible solutions and the choice of the best one. Unfortunately, in most cases, such a naive approach becomes rapidly infeasible because the number of possible solutions grows exponentially with the instance size n, where the instance size can be given, for example, by the number of binary digits necessary to encode the instance. For some combinatorial optimization problems, deep insight into the problem structure and the exploitation of problem-specific characteristics allow the definition of algorithms that find an optimal solution much quicker than exhaustive search does. In other cases, even the best algorithms of this kind cannot do much better than exhaustive search.

Box 2.1
Worst-Case Time Complexity and Intractability

The *time complexity function* of an algorithm for a given problem Π indicates, for each possible input size n, the maximum time the algorithm needs to find a solution to an instance of that size. This is often called *worst-case time complexity*.

The worst-case time complexity of an algorithm is often formalized using the $\mathcal{O}(\cdot)$ notation. Let $g(n)$ and $h(n)$ be functions from the positive integers to the positive reals. A function $g(n)$ is said to be $\mathcal{O}(h(n))$ if two positive constants *const* and n_0 exist such that $g(n) \leq const \cdot h(n)$ for all $n \geq n_0$. In other words, the $\mathcal{O}(\cdot)$ notation gives asymptotic upper bounds on the worst-case time complexity of an algorithm.

An algorithm is said to be a *polynomial time algorithm* if its time complexity function is $\mathcal{O}(g(n))$ for some polynomial function $g(\cdot)$. If an algorithm has a time complexity function that cannot be bounded by a polynomial, it is called an *exponential time algorithm*. Note that this includes also functions such as $n^{\log n}$, which are sometimes referred to as subexponential; in any case, subexponential functions grow faster than any polynomial. A problem is said to be *intractable* if there is no polynomial time algorithm capable of solving it.

When attacking a combinatorial optimization problem it is useful to know how difficult it is to find an optimal solution. A way of measuring this difficulty is given by the notion of worst-case complexity. Worst-case complexity can be explained as follows (see also box 2.1): a combinatorial optimization problem Π is said to have worst-case complexity $\mathcal{O}(g(n))$ if the best algorithm known for solving Π finds an optimal solution to any instance of Π having size n in a computation time bounded from above by $const \cdot g(n)$.

In particular, we say that Π is solvable in polynomial time if the maximum amount of computing time necessary to solve any instance of size n of Π is bounded from above by a polynomial in n. If k is the largest exponent of such a polynomial, then the combinatorial optimization problem is said to be solvable in $\mathcal{O}(n^k)$ time.

Although some important combinatorial optimization problems have been shown to be solvable in polynomial time, for the great majority of combinatorial problems no polynomial bound on the worst-case solution time could be found so far. For these problems the run time of the best algorithms known increases exponentially with the instance size and, consequently, so does the time required to find an optimal solution. A notorious example of such a problem is the TSP.

An important theory that characterizes the difficulty of combinatorial problems is that of $\mathcal{NP}$-completeness. This theory classifies combinatorial problems in two main classes: those that are known to be solvable in polynomial time, and those that are not. The first are said to be *tractable*, the latter *intractable*.

Combinatorial optimization problems as defined above correspond to what are usually called *search problems*. Each combinatorial optimization problem Π has an associated *decision problem* defined as follows: given Π, that is, the triple $(\mathcal{S}, f, \Omega)$, and a parameter ϱ, does a feasible solution $s \in \tilde{\mathcal{S}}$ exist such that $f(s) \leq \varrho$, in case Π was a minimization problem? It is clear that solving the search version of a combinatorial problem implies being able to give the solution of the corresponding decision problem, while the opposite is not true in general. This means that Π is at least as hard to solve as the decision version of Π and proving that the decision version is intractable implies intractability of the original search problem.

The theory of $\mathcal{NP}$-completeness distinguishes between two classes of problems of particular interest: the class $\mathcal{P}$ for which an algorithm outputs in polynomial time the correct answer ("yes" or "no"), and the class $\mathcal{NP}$ for which an algorithm exists that verifies for every instance, independently of the way it was generated, in polynomial time whether the answer "yes" is correct. (Note that formally, the complexity classes $\mathcal{P}$ and $\mathcal{NP}$ are defined via idealized models of computation: in the theory of $\mathcal{NP}$-completeness, typically Turing machines are used. For details, see Garey & Johnson (1979).) It is clear that $\mathcal{P} \subseteq \mathcal{NP}$, while nothing can be said on the question whether $\mathcal{P} = \mathcal{NP}$ or not. Still, an answer to this question would be very useful because proving $\mathcal{P} = \mathcal{NP}$ implies proving that all problems in $\mathcal{NP}$ can be solved in polynomial time.

On this subject, a particularly important role is played by *polynomial time reductions*. Intuitively, a polynomial time reduction is a procedure that transforms a problem into another one by a polynomial time algorithm. The interesting point is that if problem Π_A can be solved in polynomial time and problem Π_B can be transformed into Π_A via a polynomial time reduction, then also the solution to Π_B can be found in polynomial time. We say that a problem is $\mathcal{NP}$-hard, if every other problem in $\mathcal{NP}$ can be transformed to it by a polynomial time reduction. Therefore, an $\mathcal{NP}$-hard problem is at least as hard as any of the other problems in $\mathcal{NP}$. However, $\mathcal{NP}$-hard problems do not necessarily belong to $\mathcal{NP}$. An $\mathcal{NP}$-hard problem that is in $\mathcal{NP}$ is said to be $\mathcal{NP}$-complete. Therefore, the $\mathcal{NP}$-complete problems are the hardest problems in $\mathcal{NP}$: if a polynomial time algorithm could be found for an $\mathcal{NP}$-complete problem, then all problems in the $\mathcal{NP}$-complete class (and consequently all the problems in $\mathcal{NP}$) could be solved in polynomial time. Because after many years of research efforts no such algorithm has been found, most scientists tend to accept the conjecture $\mathcal{P} \neq \mathcal{NP}$. Still, the "$\mathcal{P} = \mathcal{NP}$?" question remains one of the most important open questions in theoretical computer science.

Until today, a large number of problems have been proved to be $\mathcal{NP}$-complete, including the above-mentioned TSP; see Garey & Johnson (1979) for a long list of such problems.

2.1.2 Solution Methods for $\mathcal{NP}$-Hard Problems

Two classes of algorithms are available for the solution of combinatorial optimization problems: *exact* and *approximate algorithms*.

Exact algorithms are guaranteed to find the optimal solution and to prove its optimality for every finite size instance of a combinatorial optimization problem within an instance-dependent run time. In the case of $\mathcal{NP}$-hard problems, exact algorithms need, in the worst case, exponential time to find the optimum. Although for some specific problems exact algorithms have been improved significantly in recent years, obtaining at times impressive results (Applegate, Bixby, Chvátal, & Cook, 1995, 1998), for most $\mathcal{NP}$-hard problems the performance of exact algorithms is not satisfactory. So, for example, for the quadratic assignment problem (QAP) (to be discussed in chapter 5), an important problem that arises in real-world applications and whose goal is to find the optimal assignment of n items to n locations, most instances of dimension around 30 are currently the limit of what can be solved with state-of-the-art exact algorithms (Anstreicher, Brixius, Goux, & Linderoth, 2002; Hahn, Hightower, Johnson, Guignard-Spielberg, & Roucairol, 2001; Hahn & Krarup, 2001). For example, at the time of writing, the largest, nontrivial QAP instance from QAPLIB, a benchmark library for the QAP, solved to optimality has 36 locations (Brixius & Anstreicher, 2001; Nyström, 1999). Despite the small size of the instance, the computation time required to solve it is extremely high. For example, the solution of instance `ste36a` from a backboard wiring application (Steinberg, 1961) took approximately 180 hours of CPU time on a 800 MHz Pentium III PC. This is to be compared to the currently best-performing ACO algorithms (see section 5.2.1, for how to apply ACO to the QAP), which typically require an average time of about 10 seconds to find the optimal solution for this instance on a comparable machine. In addition to the exponential worst-case complexity, the application of exact algorithms to $\mathcal{NP}$-hard problems in practice also suffers from a strong rise in computation time when the problem size increases, and often their use quickly becomes infeasible.

If optimal solutions cannot be efficiently obtained in practice, the only possibility is to trade optimality for efficiency. In other words, the guarantee of finding optimal solutions can be sacrificed for the sake of getting very good solutions in polynomial time. Approximate algorithms, often also loosely called *heuristic methods* or simply *heuristics*, seek to obtain good, that is, near-optimal solutions at relatively low computational cost without being able to guarantee the optimality of solutions. Based on the underlying techniques that approximate algorithm use, they can be classified as being either *constructive* or *local search* methods (approximate methods may also be

Box 2.2
Constructive Algorithms

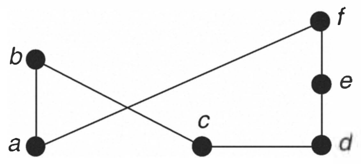
Constructive algorithms build a solution to a combinatorial optimization problem in an incremental way. Step by step and without backtracking, they add solution components until a complete solution is generated. Although the order in which to add components can be random, typically some kind of heuristic rule is employed. Often, greedy construction heuristics are used which at each construction step add a solution component with maximum myopic benefit as estimated by a heuristic function. An algorithmic outline of a greedy construction heuristic is given below.

```
procedure GreedyConstructionHeuristic
  $s_p \leftarrow$ ChooseFirstComponent
  while ($s_p$ is not a complete solution) do
    $c \leftarrow$ GreedyComponent($s_p$)
    $s_p \leftarrow s_p \otimes c$
  end-while
  $s \leftarrow s_p$
  return $s$
end-procedure
```

Here, the function ChooseFirstComponent chooses the first solution component (this is done at random or according to a greedy choice depending on the particular construction heuristic) and GreedyComponent returns a solution component c with best heuristic estimate. The addition of component c to a partial solution s_p is denoted by the operator $\otimes$. The procedure returns a complete solution s.

An example of a constructive algorithm for the TSP is the nearest-neighbor procedure, which treats the cities as components. The procedure works by randomly choosing an initial city and by iteratively adding the closest among the remaining cities to the solution under construction (ties are broken randomly).

In the example tour below the nearest-neighbor procedure starts from city a and sequentially adds cities b, c, d, e, and f.



Box 2.3
Local Search

Local search is a general approach for finding high-quality solutions to hard combinatorial optimization problems in reasonable time. It is based on the iterative exploration of neighborhoods of solutions trying to improve the current solution by local changes. The types of local changes that may be applied to a solution are defined by a neighborhood structure.

Definition 2.1 *A* neighborhood structure *is a function $\mathcal{N} : \mathcal{S} \mapsto 2^{\mathcal{S}}$ that assigns a set of neighbors $\mathcal{N}(s) \subseteq \mathcal{S}$ to every $s \in \mathcal{S}$. $\mathcal{N}(s)$ is also called the neighborhood of s.*

The choice of an appropriate neighborhood structure is crucial for the performance of a local search algorithm and is problem-specific. The neighborhood structure defines the set of solutions that can be reached from s in one single step of a local search algorithm. Typically, a neighborhood structure is defined implicitly by defining the possible local changes that may be applied to a solution, and not by explicitly enumerating the set of all possible neighbors.

The solution found by a local search algorithm may only be guaranteed to be optimal with respect to local changes and, in general, will not be a globally optimal solution.

Definition 2.2 *A* local optimum *for a minimization problem* (a *local minimum*) *is a solution s such that* $\forall s' \in \mathcal{N}(s) : f(s) \leq f(s')$. *Similarly, a* local optimum *for a maximization problem* (a *local maximum*) *is a solution s such that* $\forall s' \in \mathcal{N}(s) : f(s) \geq f(s')$.

A local search algorithm also requires the definition of a neighborhood examination scheme that determines how the neighborhood is searched and which neighbor solutions are accepted. While the neighborhood can be searched in many different ways, in the great majority of cases the acceptance rule is either the *best-improvement* rule, which chooses the neighbor solution giving the largest improvement of the objective function, or the *first-improvement* rule, which accepts the first improved solution found.

obtained by stopping exact methods before completion (Bellman, Esogbue, & Nabeshima, 1982; Jünger, Reinelt, & Thienel, 1994), for example, after some given time bound; yet, here, this type of approximate algorithm will not be discussed further). Usually, if for an approximate algorithm it can be proved that it returns solutions that are worse than the optimal solution by at most some fixed value or factor, such an algorithm is also called an *approximation algorithm* (Hochbaum, 1997; Hromkovic, 2003; Vazirani, 2001).

Constructive algorithms (see box 2.2) generate solutions from scratch by iteratively adding solution components to an initially empty solution until the solution is complete. For example, in the TSP a solution is built by adding city after city in an incremental way. Although constructive algorithms are typically the fastest among the approximate methods, the quality of the solutions they generate is most of the time inferior to the quality of the solutions found by local search algorithms.

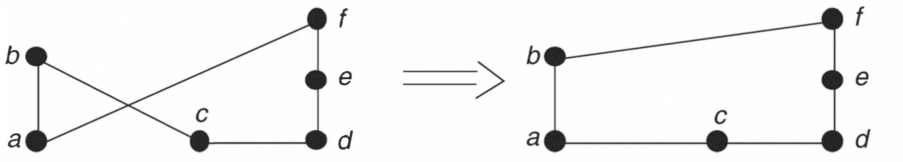
Local search starts from some initial solution and repeatedly tries to improve the current solution by local changes. The first step in applying local search is the definition of a *neighborhood structure* (see box 2.3) over the set of candidate solutions. In

Box 2.4
k–Exchange Neighborhoods

An important class of neighborhood structures for combinatorial optimization problems is that of *k–exchange neighborhoods.*

Definition 2.3 *The* k–exchange neighborhood *of a candidate solution s is the set of candidate solutions s′ that can be obtained from s by exchanging k solution components.*

Example 2.1: The 2–exchange and k–exchange neighborhoods in the TSP Given a candidate solution s, the TSP 2–exchange neighborhood of a candidate solution s consists of the set of all the candidate solutions s' that can be obtained from s by exchanging two pairs of arcs in all the possible ways. The figure below gives an example of one specific 2–exchange: the pair of arcs (b, c) and (a, f) is removed and replaced by the pair (a, c) and (b, f).



The k–exchange neighborhood is the obvious generalization in which a set of k arcs is replaced by a different set of k arcs.

practice, the neighborhood structure defines for each current solution the set of possible solutions to which the local search algorithms can move. One common way of defining neighborhoods is via k-exchange moves that exchange a set of k components of a solution with a different set of k components (see box 2.4).

In its most basic version, often called *iterative improvement*, or sometimes *hill-climbing* or *gradient-descent* for maximization and minimization problems, respectively, the local search algorithm searches for an improved solution within the neighborhood of the current solution. If an improving solution is found, it replaces the current solution and the local search is continued. These steps are repeated until no improving solution is found in the neighborhood and the algorithm terminates in a *local optimum*. A disadvantage of iterative improvement is that the algorithm may stop at very poor-quality local optima.

2.1.3 What Is a Metaheuristic?

A disadvantage of single-run algorithms like constructive methods or iterative improvement is that they either generate only a very limited number of different solutions, which is the case for greedy construction heuristics, or they stop at poor-quality local optima, which is the case for iterative improvement methods. Unfortunately, the obvious extension of local search, that is, to restart the algorithm several times from new starting solutions, does not produce significant improvements in practice (Johnson & McGeoch, 1997; Schreiber & Martin, 1999). Several general approaches, which are nowadays often called metaheuristics, have been proposed which try to bypass these problems.

A *metaheuristic* is a set of algorithmic concepts that can be used to define heuristic methods applicable to a wide set of different problems. In other words, a metaheuristic can be seen as a general-purpose heuristic method designed to guide an underlying problem-specific heuristic (e.g., a local search algorithm or a construction heuristic) toward promising regions of the search space containing high-quality solutions. A metaheuristic is therefore a general algorithmic framework which can be applied to different optimization problems with relatively few modifications to make them adapted to a specific problem.

Examples of metaheuristics include simulated annealing (Cerný, 1985; Kirkpatrick, Gelatt, & Vecchi, 1983), tabu search (Glover, 1989, 1990; Glover & Laguna, 1997), iterated local search (Lourenço, Martin, & Stützle, 2002), evolutionary computation (Fogel, Owens, & Walsh, 1966; Holland, 1975; Rechenberg, 1973; Schwefel, 1981; Goldberg, 1989), and ant colony optimization (Dorigo & Di Caro, 1999b; Dorigo, Di Caro, & Gambardella, 1999; Dorigo, Maniezzo, & Colorni, 1996; Dorigo & Stützle, 2002) (see Glover & Kochenberger [2002] for a comprehensive overview).

The use of metaheuristics has significantly increased the ability of finding very high-quality solutions to hard, practically relevant combinatorial optimization problems in a reasonable time. This is particularly true for large and poorly understood problems. A detailed description of the ant colony optimization metaheuristic is given in the next section; the other metaheuristics mentioned above are briefly described in section 2.4.

2.2 The ACO Metaheuristic

Ant colony optimization is a metaheuristic in which a colony of artificial ants cooperate in finding good solutions to difficult discrete optimization problems. Cooperation is a key design component of ACO algorithms: The choice is to allocate the computational resources to a set of relatively simple agents (artificial ants) that communicate indirectly by stigmergy, that is, by indirect communication mediated by the environment (see chapter 1, section 1.4). Good solutions are an emergent property of the agents' cooperative interaction.

ACO algorithms can be used to solve both static and dynamic combinatorial optimization problems. Static problems are those in which the characteristics of the problem are given once and for all when the problem is defined, and do not change while the problem is being solved. A paradigmatic example of such problems is the TSP (Johnson & McGeoch, 1997; Lawler, Lenstra, Rinnooy Kan, & Shmoys, 1985; Reinelt, 1994), in which city locations and their relative distances are part of the problem definition and do not change at run time. On the contrary, dynamic problems are defined as a function of some quantities whose value is set by the dynamics of an underlying system. The problem instance changes therefore at run time and the optimization algorithm must be capable of adapting online to the changing environment. An example of this situation, which we discuss at length in chapter 6, are network routing problems in which the data traffic and the network topology can vary in time.

In this section we give a formal characterization of the class of problems to which the ACO metaheuristic can be applied, of the behavior governing the artificial ants, and of the general structure of the ACO metaheuristic.

2.2.1 Problem Representation

An artificial ant in ACO is a stochastic constructive procedure that incrementally builds a solution by adding opportunely defined solution components to a partial solution under construction. Therefore, the ACO metaheuristic can be applied to any combinatorial optimization problem for which a constructive heuristic can be defined.

Although this means that the ACO metaheuristic can be applied to any interesting combinatorial optimization problems, the real issue is how to map the considered problem to a representation that can be used by the artificial ants to build solutions. In the following we give a formal characterization of the representation that the artificial ants use and of the policy they implement.

Let us consider the minimization problem $(\mathcal{S}, f, \Omega)$, where $\mathcal{S}$ is the *set of candidate solutions*, f is the *objective function* which assigns an objective function (cost) value $f(s,t)$ to each candidate solution $s \in \mathcal{S}$, and $\Omega(t)$ is a *set of constraints*. The parameter t indicates that the objective function and the constraints can be time-dependent, as is the case, for example, in applications to dynamic problems (e.g., in telecommunication network routing problems the cost of links is proportional to traffic, which is time-dependent; and constraints on the reachable nodes can also change with time: think of a network node that suddenly becomes unreachable).

The goal is to find a *globally optimal* feasible solution s^*, that is, a minimum cost feasible solution to the minimization problem.

The combinatorial optimization problem $(\mathcal{S}, f, \Omega)$ is mapped on a problem that can be characterized by the following list of items:

- A finite set $C = \{c_1, c_2, \ldots, c_{N_C}\}$ of *components* is given, where N_C is the number of components.
- The *states* of the problem are defined in terms of sequences $x = \langle c_i, c_j, \ldots, c_h, \ldots \rangle$ of finite length over the elements of C. The set of all possible states is denoted by $\mathcal{X}$. The length of a sequence x, that is, the number of components in the sequence, is expressed by $|x|$. The maximum length of a sequence is bounded by a positive constant $n < +\infty$.
- The set of (candidate) solutions $\mathcal{S}$ is a subset of $\mathcal{X}$ (i.e., $\mathcal{S} \subseteq \mathcal{X}$).
- A set of feasible states $\tilde{\mathcal{X}}$, with $\tilde{\mathcal{X}} \subseteq \mathcal{X}$, defined via a problem-dependent test that verifies that it is not impossible to complete a sequence $x \in \tilde{\mathcal{X}}$ into a solution satisfying the constraints Ω. Note that by this definition, the feasibility of a state $x \in \tilde{\mathcal{X}}$ should be interpreted in a *weak* sense. In fact, it does not guarantee that a completion s of x exists such that $s \in \tilde{\mathcal{X}}$.
- A non-empty set $\mathcal{S}^*$ of optimal solutions, with $\mathcal{S}^* \subseteq \tilde{\mathcal{X}}$ and $\mathcal{S}^* \subseteq \mathcal{S}$.
- A *cost* $g(s,t)$ is associated with each candidate solution $s \in \mathcal{S}$. In most cases $g(s,t) \equiv f(s,t)$, $\forall s \in \tilde{\mathcal{S}}$, where $\tilde{\mathcal{S}} \subseteq \mathcal{S}$ is the set of feasible candidate solutions, obtained from $\mathcal{S}$ via the constraints $\Omega(t)$.
- In some cases a cost, or the estimate of a cost, $J(x,t)$ can be associated with states other than candidate solutions. If x_j can be obtained by adding solution components to a state x_i, then $J(x_i,t) \le J(x_j,t)$. Note that $J(s,t) \equiv g(s,t)$.

Given this formulation, artificial ants build solutions by performing randomized walks on the completely connected graph $G_C = (C, L)$ whose nodes are the components C, and the set L fully connects the components C. The graph G_C is called *construction graph* and elements of L are called *connections*.

The problem constraints $\Omega(t)$ are implemented in the policy followed by the artificial ants, as explained in the next section. The choice of implementing the constraints in the construction policy of the artificial ants allows a certain degree of flexibility. In fact, depending on the combinatorial optimization problem considered, it may be more reasonable to implement the constraints in a hard way, allowing the ants to build only feasible solutions, or in a soft way, in which case the ants can build infeasible solutions (i.e., candidate solutions in $\mathcal{S} \backslash \tilde{\mathcal{S}}$) that can be penalized as a function of their degree of infeasibility.

2.2.2 Ants' Behavior

As we said, in ACO algorithms artificial ants are stochastic constructive procedures that build solutions by moving on the construction graph $G_C = (C, L)$, where the set

L fully connects the components C. The problem constraints $\Omega(t)$ are built into the ants' constructive heuristic. In most applications, ants construct feasible solutions. However, sometimes it may be necessary or beneficial to also let them construct infeasible solutions. Components $c_i \in C$ and connections $l_{ij} \in L$ can have associated a *pheromone trail* τ (τ_i if associated with components, τ_{ij} if associated with connections), and a *heuristic value* η (η_i and η_{ij}, respectively). The pheromone trail encodes a long-term memory about the entire ant search process, and is updated by the ants themselves. Differently, the heuristic value, often called *heuristic information*, represents a priori information about the problem instance or run-time information provided by a source different from the ants. In many cases η is the cost, or an estimate of the cost, of adding the component or connection to the solution under construction. These values are used by the ants' heuristic rule to make probabilistic decisions on how to move on the graph.

More precisely, each ant k of the colony has the following properties:

- It exploits the construction graph $G_C = (C, L)$ to search for optimal solutions $s^* \in \mathcal{S}^*$.
- It has a memory $\mathcal{M}^k$ that it can use to store information about the path it followed so far. Memory can be used to (1) build feasible solutions (i.e., implement constraints Ω); (2) compute the heuristic values η; (3) evaluate the solution found; and (4) retrace the path backward.
- It has a *start state* x_s^k and one or more *termination conditions* e^k. Usually, the start state is expressed either as an empty sequence or as a unit length sequence, that is, a single component sequence.
- When in state $x_r = \langle x_{r-1}, i \rangle$, if no termination condition is satisfied, it moves to a node j in its *neighborhood* $\mathcal{N}^k(x_r)$, that is, to a state $\langle x_r, j \rangle \in \mathcal{X}$. If at least one of the termination conditions e^k is satisfied, then the ant stops. When an ant builds a candidate solution, moves to infeasible states are forbidden in most applications, either through the use of the ant's memory, or via appropriately defined heuristic values η.
- It selects a move by applying a probabilistic decision rule. The probabilistic decision rule is a function of (1) the locally available pheromone trails and heuristic values (i.e., pheromone trails and heuristic values associated with components and connections in the neighborhood of the ant's current location on graph G_C); (2) the ant's private memory storing its current state; and (3) the problem constraints.
- When adding a component c_j to the current state, it can update the pheromone trail τ associated with it or with the corresponding connection.
- Once it has built a solution, it can retrace the same path backward and update the pheromone trails of the used components.

It is important to note that ants act concurrently and independently and that although each ant is complex enough to find a (probably poor) solution to the problem under consideration, good-quality solutions can only emerge as the result of the collective interaction among the ants. This is obtained via indirect communication mediated by the information ants read or write in the variables storing pheromone trail values. In a way, this is a distributed learning process in which the single agents, the ants, are not adaptive themselves but, on the contrary, adaptively modify the way the problem is represented and perceived by other ants.

2.2.3 The Metaheuristic

Informally, an ACO algorithm can be imagined as the interplay of three procedures: ConstructAntsSolutions, UpdatePheromones, and DaemonActions.

ConstructAntsSolutions manages a colony of ants that concurrently and asynchronously visit adjacent states of the considered problem by moving through neighbor nodes of the problem's construction graph G_C. They move by applying a stochastic local decision policy that makes use of pheromone trails and heuristic information. In this way, ants incrementally build solutions to the optimization problem. Once an ant has built a solution, or while the solution is being built, the ant evaluates the (partial) solution that will be used by the UpdatePheromones procedure to decide how much pheromone to deposit.

UpdatePheromones is the process by which the pheromone trails are modified. The trails value can either increase, as ants deposit pheromone on the components or connections they use, or decrease, due to pheromone evaporation (see also section 1.3 of chapter 1). From a practical point of view, the deposit of new pheromone increases the probability that those components/connections that were either used by many ants or that were used by at least one ant and which produced a very good solution will be used again by future ants. Differently, pheromone evaporation implements a useful form of *forgetting*: it avoids a too rapid convergence of the algorithm toward a suboptimal region, therefore favoring the exploration of new areas of the search space.

Finally, the DaemonActions procedure is used to implement centralized actions which cannot be performed by single ants. Examples of daemon actions are the activation of a local optimization procedure, or the collection of global information that can be used to decide whether it is useful or not to deposit additional pheromone to bias the search process from a nonlocal perspective. As a practical example, the daemon can observe the path found by each ant in the colony and select one or a few ants (e.g., those that built the best solutions in the algorithm iteration) which are then allowed to deposit additional pheromone on the components/connections they used.

```
procedure ACOMetaheuristic
  ScheduleActivities
    ConstructAntsSolutions
    UpdatePheromones
    DaemonActions        % optional
  end-ScheduleActivities
end-procedure
```

Figure 2.1
The ACO metaheuristic in pseudo-code. The procedure DaemonActions is optional and refers to centralized actions executed by a daemon possessing global knowledge.

In figure 2.1, the ACO metaheuristic is described in pseudo-code. The main procedure of the ACO metaheuristic manages the scheduling of the three above-discussed components of ACO algorithms via the `ScheduleActivities` construct: (1) management of the ants' activity, (2) pheromone updating, and (3) daemon actions. The `ScheduleActivities` construct does not specify how these three activities are scheduled and synchronized. In other words, it does not say whether they should be executed in a completely parallel and independent way, or if some kind of synchronization among them is necessary. The designer is therefore free to specify the way these three procedures should interact, taking into account the characteristics of the considered problem.

Nowadays numerous successful implementations of the ACO metaheuristic are available and they have been applied to many different combinatorial optimization problems. These applications are summarized in table 2.1 and they are discussed in the forthcoming chapters of this book.

2.3 How Do I Apply ACO?

Probably, the best way of illustrating how the ACO metaheuristic operates is by describing how it has been applied to combinatorial optimization problems. This is done with a full and detailed description of most of the current applications of ACO in chapter 5. Here we limit ourselves to a brief description of the main points to consider when applying ACO algorithms to a few examples of problems representative of important classes of optimization problems.

First, we illustrate the application to permutation problems in their unconstrained and constrained forms: the TSP and the sequential ordering problem. Then we consider generalized assignment as an example of assignment problems, and multiple

Table 2.1
Current applications of ACO algorithms listed according to problem types and chronologically

Problem type	Problem name	Main references
Routing	Traveling salesman	Dorigo, Maniezzo, & Colorni (1991a,b, 1996) Dorigo (1992) Gambardella & Dorigo (1995) Dorigo & Gambardella (1997a,b) Stützle & Hoos (1997, 2000) Bullnheimer, Hartl, & Strauss (1999c) Cordón, de Viana, Herrera, & Morena (2000)
	Vehicle routing	Bullnheimer, Hartl, & Strauss (1999a,b) Gambardella, Taillard, & Agazzi (1999) Reimann, Stummer, & Doerner (2002)
	Sequential ordering	Gambardella & Dorigo (1997, 2000)
Assignment	Quadratic assignment	Maniezzo, Colorni, & Dorigo (1994) Stützle (1997b) Maniezzo & Colorni (1999) Maniezzo (1999) Stützle & Hoos (2000)
	Graph coloring	Costa & Hertz (1997)
	Generalized assignment	Lourenço & Serra (1998, 2002)
	Frequency assignment	Maniezzo & Carbonaro (2000)
	University course timetabling	Socha, Knowles, & Sampels (2002) Socha, Sampels, & Manfrin (2003)
Scheduling	Job shop	Colorni, Dorigo, Maniezzo, & Trubian (1994)
	Open shop	Pfahringer (1996)
	Flow shop	Stützle (1998a)
	Total tardiness	Bauer, Bullnheimer, Hartl, & Strauss (2000)
	Total weighted tardiness	den Besten, Stützle, & Dorigo (2000) Merkle & Middendorf (2000, 2003a) Gagné, Price, & Gravel (2002)
	Project scheduling	Merkle, Middendorf, & Schmeck (2000a, 2002)
	Group shop	Blum (2002a, 2003a)
Subset	Multiple knapsack	Leguizamón & Michalewicz (1999)
	Max independent set	Leguizamón & Michalewicz (2000)
	Redundancy allocation	Liang & Smith (1999)
	Set covering	Leguizamón & Michalewicz (2000) Hadji, Rahoual, Talbi, & Bachelet (2000)
	Weight constrained graph tree partition	Cordone & Maffioli (2001)
	Arc-weighted *l*-cardinality tree	Blum & Blesa (2003)
	Maximum clique	Fenet & Solnon (2003)

Table 2.1
(continued)

Problem type	Problem name	Main references
Other	Shortest common supersequence	Michel & Middendorf (1998, 1999)
	Constraint satisfaction	Solnon (2000, 2002)
	2D-HP protein folding	Shmygelska, Aguirre-Hernández, & Hoos (2002)
	Bin packing	Levine & Ducatelle (2003)
Machine learning	Classification rules	Parpinelli, Lopes, & Freitas (2002b)
	Bayesian networks	de Campos, Gámez, & Puerta (2002b)
	Fuzzy systems	Casillas, Cordón, & Herrera (2000)
Network routing	Connection-oriented network routing	Schoonderwoerd, Holland, Bruten, & Rothkrantz (1996) Schoonderwoerd, Holland, & Bruten (1997) White, Pagurek, & Oppacher (1998) Di Caro & Dorigo (1998d) Bonabeau, Henavy, Guérin, Snyers, Kuntz, & Theraulaz (1998)
	Connectionless network routing	Di Caro & Dorigo (1997, 1998c,f) Subramanian, Druschel, & Chen (1997) Heusse, Snyers, Guérin, & Kuntz (1998) van der Put (1998)
	Optical network routing	Navarro Varela, & Sinclair (1999)

knapsack as an example of subset problems. Finally, applications to two dynamic problems, network routing and dynamic TSP, are briefly discussed.

2.3.1 The Traveling Salesman Problem

Intuitively, the traveling salesman problem is the problem faced by a salesman who, starting from his home town, wants to find a shortest possible trip through a given set of customer cities, visiting each city once before finally returning home. The TSP can be represented by a complete weighted graph $G = (N, A)$ with N being the set of $n = |N|$ nodes (cities), A being the set of arcs fully connecting the nodes. Each arc $(i, j) \in A$ is assigned a weight d_{ij} which represents the distance between cities i and j. The TSP is the problem of finding a minimum length Hamiltonian circuit of the graph, where a Hamiltonian circuit is a closed walk (a tour) visiting each node of G exactly once. We may distinguish between symmetric TSPs, where the distances between the cities are independent of the direction of traversing the arcs, that is, $d_{ij} = d_{ji}$ for every pair of nodes, and the asymmetric TSP (ATSP), where at least for one pair of nodes i, j we have $d_{ij} \neq d_{ji}$.

A solution to an instance of the TSP can be represented as a permutation of the city indices; this permutation is cyclic, that is, the absolute position of a city in a tour is not important at all but only the relative order is important (in other words, there are n permutations that map to the same solution).

Construction graph. The construction graph is identical to the problem graph: the set of components C corresponds to the set of nodes (i.e., $C = N$), the connections correspond to the set of arcs (i.e., $L = A$), and each connection has a weight which corresponds to the distance d_{ij} between nodes i and j. The states of the problem are the set of all possible partial tours.

Constraints. The only constraint in the TSP is that all cities have to be visited and that each city is visited at most once. This constraint is enforced if an ant at each construction step chooses the next city only among those it has not visited yet (i.e., the feasible neighborhood $\mathcal{N}_i^k$ of an ant k in city i, where k is the ant's identifier, comprises all cities that are still unvisited).

Pheromone trails and heuristic information. The pheromone trails τ_{ij} in the TSP refer to the desirability of visiting city j directly after i. The heuristic information η_{ij} is typically inversely proportional to the distance between cities i and j, a straightforward choice being $\eta_{ij} = 1/d_{ij}$. In fact, this is also the heuristic information used in most ACO algorithms for the TSP.

Solution construction. Each ant is initially put on a randomly chosen start city and at each step iteratively adds one still unvisited city to its partial tour. The solution construction terminates once all cities have been visited.

General comments. The TSP is a paradigmatic $\mathcal{NP}$-hard combinatorial optimization problem which has attracted a very significant amount of research (Johnson & McGeoch, 1997; Lawler et al., 1985; Reinelt, 1994). The TSP has played a central role in ACO, because it was the application problem chosen when proposing the first ACO algorithm called Ant System (Dorigo, 1992; Dorigo, Maniezzo, & Colorni, 1991b, 1996) and it was used as a test problem for almost all ACO algorithms proposed later. Chapter 3 gives a detailed presentation of the ACO algorithms available for the TSP.

2.3.2 The Sequential Ordering Problem

The sequential ordering problem (SOP) consists in finding a minimum weight Hamiltonian path on a directed graph with weights on the arcs and the nodes, subject to precedence constraints between nodes. It is easy to remove weights from nodes and to add them to the arcs, obtaining a kind of asymmetric traveling salesman problem in which, once all the nodes have been visited, the path is not closed (i.e., it does not become a tour as in the ATSP).

Construction graph. Similar to the TSP, the set of components C contains all the nodes. Solutions are permutations of the elements of C, and costs (lengths) are associated with connections between nodes.

Constraints. The only significant difference between the applications of ACO to the SOP and to the TSP is the set of constraints: while building solutions, ants choose components only among those that have not yet been used and, if possible, satisfy all precedence constraints.

Pheromone trails and heuristic information. As in the TSP case, pheromone trails are associated with connections, and the heuristic information can, for example, be chosen as the inverse of the costs (lengths) of the connections.

Solution construction. Ants build solutions iteratively by adding, step by step, new unvisited nodes to the partial solution under construction. They choose the new node to add by using pheromone trails, heuristic, and constraint information.

2.3.3 The Generalized Assignment Problem

In the generalized assignment problem (GAP) a set of tasks $i \in I$, has to be assigned to a set of agents $j \in J$. Each agent j has only a limited capacity a_j and each task i assigned to agent j consumes a quantity r_{ij} of the agent's capacity. Also, the cost d_{ij} of assigning task i to agent j is given. The objective then is to find a feasible task assignment with minimum cost.

Let y_{ij} be 1 if task i is assigned to agent j and 0 otherwise. Then the GAP can formally be defined as

$$\min f(y) = \sum_{j=1}^{m} \sum_{i=1}^{n} d_{ij} y_{ij} \tag{2.1}$$

subject to

$$\sum_{i=1}^{n} r_{ij} y_{ij} \le a_j, \quad j = 1, \ldots, m, \tag{2.2}$$

$$\sum_{j=1}^{m} y_{ij} = 1, \quad i = 1, \ldots, n, \tag{2.3}$$

$$y_{ij} \in \{0, 1\}, \quad i = 1, \ldots, n, \quad j = 1, \ldots, m. \tag{2.4}$$

The constraints in equation (2.2) implement the limited resource capacity of the agents, while the constraints given by equations (2.3) and (2.4) impose that each task is assigned to exactly one agent and that a task cannot be split among several agents.

Construction graph. The GAP can easily be cast into the framework of the ACO metaheuristic. For example, the problem could be represented on the construction graph $G_C = (C, L)$ in which the set of components comprises the set of tasks and agents, that is, $C = I \cup J$. Each assignment, which consists of n couplings (i, j) of tasks and agents, corresponds to at least one ant's walk on this graph and costs d_{ij} are associated with all possible couplings (i, j) of tasks and agents.

Constraints. Walks on the construction graph G_C have to satisfy the constraints given by equations (2.3) and (2.4) to obtain a valid assignment. One particular way of generating such an assignment is by an ant's walk which iteratively switches from task nodes (nodes in the set I) to agent nodes (nodes in the set J) without repeating any task node but possibly using an agent node several times (several tasks may be assigned to an agent). Moreover, the GAP involves resource capacity constraints that can be enforced by an appropriately defined neighborhood. For example, for an ant k, $\mathcal{N}_i^k$ could be defined as consisting of all those agents to which task i can be assigned without violating the agents' resource capacity. If no agent meets the task's resource requirement, then the ant is forced to build an infeasible solution; in this case $\mathcal{N}_i^k$ becomes the set of all agents. Infeasibilities can then be handled, for example, by assigning penalties proportional to the amount of resource violations.

Pheromone trails and heuristic information. During the construction of a solution, ants repeatedly have to take the following two basic decisions: (1) choose the task to assign next and (2) choose the agent the task should be assigned to. Pheromone trail information can be associated with any of the two decisions: it can be used to learn an appropriate order for task assignments or it can be associated with the desirability of assigning a task to a specific agent. In the first case, τ_{ij} represents the desirability of assigning task j directly after task i, while in the second case it represents the desirability of assigning agent j to task i.

Similarly, heuristic information can be associated with any of the two decisions. For example, heuristic information could bias task assignment toward those tasks that use more resources, and bias the choice of agents in such a way that small assignment costs are incurred and the agent only needs a relatively small amount of its available resource to perform the task.

Solution construction. Solution construction can be performed as usual, by choosing the components to add to the partial solution from among those that, as explained above, satisfy the constraints with a probability biased by the pheromone trails and heuristic information.

2.3.4 The Multiple Knapsack Problem

Given a set of items $i \in I$ with associated a vector of resource requirements r_i and a profit b_i, the knapsack problem (KP) is the problem of selecting a subset of items from I in such a way that they fit into a knapsack of limited capacity and maximize the sum of profits of the chosen items. The multiple knapsack problem (MKP), also known as multidimensional KP, extends the single KP by considering multiple resource constraints. Let y_i be a variable associated with item i, which has value 1 if i is added to the knapsack, and 0 otherwise. Also, let r_{ij} be the resource requirement of item i with respect to resource constraint j, a_j the capacity of resource j, and m be the number of resource constraints. Then the MKP can be formulated as

$$\max f(y) = \sum_{i=1}^{n} b_i y_i, \tag{2.5}$$

subject to

$$\sum_{i=1}^{n} r_{ij} y_i \leq a_j, \quad j = 1, \ldots, m, \tag{2.6}$$

$$y_i \in \{0, 1\}, \quad i = 1, \ldots, n. \tag{2.7}$$

In the MKP, it is typically assumed that all profits b_i and all weights r_{ij} take positive values.

Construction graph. In the construction graph $G_C = (C, L)$, the set of components C corresponds to the set of items and, as usual, the set of connections L fully connects the set of items. The profit of adding items can be associated with either the connections or the components.

Constraints. The solution construction has to consider the resource constraints given by equation (2.6). During the solution construction process, this can be easily done by allowing ants to add only those components that, when added to their current partial solution, do not violate any resource constraint.

Pheromone trails and heuristic information. The MKP has the particularity that pheromone trails τ_i are associated only with components and refer to the desirability of adding an item i to the current partial solution. The heuristic information, intuitively, should prefer items which have a high profit and low resource requirements. One possible choice for the heuristic information is to calculate the average resource requirement $\bar{r}_i = \sum_{j=1}^{m} r_{ij}/m$ for each item and then to define $\eta_i = b_i/\bar{r}_i$. Yet this choice has the disadvantage that it does not take into account how tight the single resource constraints are. Therefore, more information can be provided if the heuristic information is also made a function of the a_j. One such possibility is to calculate $\bar{r}_i{}' = 1/m \cdot \sum_{j=1}^{m} a_j/r_{ij}$ and to compute the heuristic information as $\eta_i' = b_i/\bar{r}_i{}'$.

Solution construction. Each ant iteratively adds items in a probabilistic way biased by pheromone trails and heuristic information; each item can be added at most once. An ant's solution construction ends if no item can be added anymore without violating any of the resource constraints. This leads to one particularity of the ACO application to the MKP: the length of the ants' walks is not fixed in advance and different ants may have solutions of different length.

2.3.5 The Network Routing Problem

Let a telecommunications network be defined by a set of nodes N, a set of links between nodes L_{net}, and the costs d_{ij} associated with the links. Then, the network routing problem (NRP) is the problem of finding minimum cost paths among all pairs of nodes in the network. It should be noted that if the costs d_{ij} are fixed, then the NRP is reduced to a set of minimum cost path problems, each of which can be solved efficiently via a polynomial time algorithm like Dijkstra's algorithm (Dijkstra, 1959). The problem becomes interesting for heuristic approaches once, as happens in real-world applications like routing in communications networks, costs (e.g., data traffic in links) or the network topology varies in time.

Construction graph. The construction graph is the graph $G_C = (C, L)$, where C corresponds to the set of nodes N, and L fully connects G_C. Note that $L_{net} \subseteq L$.

Constraints. The only constraint is that ants use only connections $l_{ij} \in L_{net}$.

Pheromone trails and heuristic information. Because the NRP is, in reality, a set of minimum cost path problems, each connection $l_{ij} \in L$ should have many different pheromone trails associated. For example, each connection l_{ij} could have associated one trail value τ_{ijd} for each possible destination node d an ant located in node i can have. Each arc can also be assigned a heuristic value η_{ij} independent of the final destination. The heuristic value η_{ij} can be set, for example, to a value inversely proportional to the amount of traffic on the link connecting nodes i and j.

Solution construction. Solution construction is straightforward. In fact, the S-ACO algorithm presented in chapter 1, section 1.3.1, is an example of how to proceed.

Each ant has a source node s and a destination node d, and moves from s to d hopping from one node to the next, until node d has been reached. When ant k is located at node i, it chooses the next node j to move to using a probabilistic decision rule which is a function of the ant's memory, of local pheromones, and heuristic information.

2.3.6 The Dynamic Traveling Salesman Problem

The dynamic traveling salesman problem (DTSP) is a TSP in which cities can be added or removed at run time. The goal is to find as quickly as possible the new shortest tour after each transition.

Construction graph. The same as for the TSP: $G_C = (C, L)$, where $C = C(t)$ is the set of cities and $L = L(t)$ completely connects G_C. The dependence of C and L on time is due to the dynamic nature of the problem.

Constraints. As in the TSP, the only constraint is that a solution should contain each city once and only once.

Pheromone trails and heuristic information. As in the TSP: pheromone trails are associated with connections, and heuristic values can be given by the inverse of the distances between cities. An important question is how to handle the problem of connections that disappear and appear in case a city is removed or a new city is added. In the first case the values no longer used can simply be removed, while in the second case the new pheromone values could be set, for example, either to values proportional to the length of the associated connections or to the average of the other pheromone values.

Solution construction. Solution construction follows the same rules as in the TSP.

2.4 Other Metaheuristics

The world of metaheuristics is rich and multifaceted and, besides ACO, a number of other successful metaheuristics are available in the literature. Some of the best known and most widely applied metaheuristics are simulated annealing (SA) (Cerný, 1985; Kirkpatrick et al., 1983), tabu search (TS) (Glover, 1989, 1990; Glover & Laguna, 1997), guided local search (GLS) (Voudouris & Tsang, 1995; Voudouris, 1997), greedy randomized adaptive search procedures (GRASP) (Feo & Resende, 1989, 1995), iterated local search (ILS) (Lourenço et al., 2002), evolutionary computation (EC) (Fogel et al., 1966; Goldberg, 1989; Holland, 1975; Rechenberg, 1973; Schwefel, 1981), and scatter search (Glover, 1977).

All metaheuristics have in common that they try to avoid the generation of poor-quality solutions by introducing general mechanisms that extend problem-specific, single-run algorithms like greedy construction heuristics or iterative improvement local search. Differences among the available metaheuristics concern the techniques employed to avoid getting stuck in suboptimal solutions and the type of trajectory followed in the space of either partial or full solutions.

A first important distinction among metaheuristics is whether they are constructive or local search based (see boxes 2.2 and 2.3). ACO and GRASP belong to the first class; all the other metaheuristics belong to the second class. Another important distinction is whether at each iteration they manipulate a single solution or a population of solutions. All the above-mentioned metaheuristics manipulate a single solution, except for ACO and EC. Although constructive and population-based metaheuristics can be used without recurring to local search, very often their performance can be greatly improved if they are extended to include it. This is the case for both ACO and EC, while GRASP is defined from the very beginning to include local search.

One further important dimension for the classification of metaheuristics concerns the use of memory. Metaheuristics that exploit memory to direct future search are TS, GLS, and ACO. TS either explicitly memorizes previously encountered solutions or memorizes components of previously seen solutions; GLS stores penalties associated with solution components to modify the solutions' evaluation function; and ACO uses pheromones to maintain a memory of past experiences.

It is interesting to note that, for all metaheuristics, there is no general termination criterion. In practice, a number of rules of thumb are used: the maximum CPU time elapsed, the maximum number of solutions generated, the percentage deviation from a lower/upper bound from the optimum, and the maximum number of iterations without improvement in solution quality are examples of such rules. In some cases, metaheuristic-dependent rules of thumb can be defined. An example is TS which can be stopped if the set of solutions in the neighborhood is empty; or SA, where the termination condition is often defined by an annealing schedule.

In conclusion, we see that ACO possesses several characteristics which in their particular combination make it a unique approach: it uses a *population* (colony) of ants which *construct* solutions exploiting a form of *indirect memory* called *artificial pheromones*. The following sections describe in more detail the metaheuristics we mentioned above.

2.4.1 Simulated Annealing

Simulated annealing (Cerný, 1985; Kirkpatrick et al., 1983) is inspired by an analogy between the physical annealing of solids (crystals) and combinatorial optimization problems. In the physical annealing process a solid is first melted and then cooled very slowly, spending a long time at low temperatures, to obtain a perfect lattice structure corresponding to a minimum energy state. SA transfers this process to local search algorithms for combinatorial optimization problems. It does so by associating the set of solutions of the problem attacked with the states of the physical system, the objective function with the physical energy of the solid, and the optimal solutions with the minimum energy states.

SA is a local search strategy which tries to avoid local minima by accepting worse solutions with some probability. In particular, SA starts from some initial solution s and then proceeds as follows: At each step, a solution $s' \in \mathcal{N}(s)$ is generated (often this is done randomly according to a uniform distribution). If s' improves on s, it is accepted; if s' is worse than s, then s' is accepted with a probability which depends on the difference in objective function value $f(s) - f(s')$, and on a parameter T, called temperature. T is lowered (as is also done in the physical annealing process) during the run of the algorithm, reducing in this way the probability of accepting solutions worse than the current one. The probability p_{accept} to accept a solution s' is often defined according to the Metropolis distribution (Metropolis, Rosenbluth, Rosenbluth, Teller, & Teller, 1953):

$$p_{accept}(s, s', T) = \begin{cases} 1, & \text{if } f(s') < f(s); \\ \exp\left(\dfrac{f(s) - f(s')}{T}\right), & \text{otherwise.} \end{cases} \tag{2.8}$$

Figure 2.2 gives a general algorithmic outline for SA. To implement an SA algorithm, the following parameters and functions have to be specified:

- The function GenerateInitialSolution, that generates an initial solution
- The function InitializeAnnealingParameters that initializes several parameters used in the *annealing schedule*; the parameters comprise
  - an initial temperature T_0
  - the number of iterations to be performed at each temperature (inner loop criterion in figure 2.2)
  - a termination condition (outer loop criterion in figure 2.2)
- The function UpdateTemp that returns a new value for the temperature
- The function GenerateNeighbor that chooses a new solution s' in the neighborhood of the current solution s
- The function AcceptSolution that implements equation (2.8); it decides whether to accept or not the solution returned by GenerateNeighbor

```
procedure SimulatedAnnealing
  $s \leftarrow$ GenerateInitialSolution
  InitializeAnnealingParameters
  $s_{best} \leftarrow s$
  $n \leftarrow 0$
  while (outer-loop termination condition not met) do
    while (inner-loop termination condition not met) do
      $s' \leftarrow$ GenerateNeighbor($s$)
      $s \leftarrow$ AcceptSolution($T_n$, $s$, $s'$)
      if ($f(s) < f(s_{best})$) then
        $s_{best} \leftarrow s$
      end-if
    end-while
    UpdateTemp($n$); $n \leftarrow n + 1$
  end-while
  return $s_{best}$
end-procedure
```

Figure 2.2
High-level pseudo-code for simulated annealing (SA).

SA has been applied to a wide variety of problems with mixed success (Aarts, Korst, & van Laarhoven, 1997). It is of special appeal to mathematicians due to the fact that under certain conditions the convergence of the algorithm to an optimal solution can be proved (Geman & Geman, 1984; Hajek, 1988; Lundy & Mees, 1986; Romeo & Sangiovanni-Vincentelli, 1991). Yet, to guarantee convergence to the optimal solution, an impractically slow annealing schedule has to be used and theoretically an infinite number of states has to be visited by the algorithm.

2.4.2 Tabu Search

Tabu search (TS) (Glover, 1989, 1990; Glover & Laguna, 1997) relies on the systematic use of *memory* to guide the search process. It is common to distinguish between *short-term memory*, which restricts the neighborhood $\mathcal{N}(s)$ of the current solution s to a subset $\mathcal{N}'(s) \subseteq \mathcal{N}(s)$, and *long-term memory*, which may extend $\mathcal{N}(s)$ through the inclusion of additional solutions (Glover & Laguna, 1997).

TS uses a local search that, at every step, makes the best possible move from s to a neighbor solution s' even if the new solution is worse than the current one; in this latter case, the move that least worsens the objective function is chosen. To prevent local search from immediately returning to a previously visited solution and, more generally, to avoid cycling, TS can explicitly memorize recently visited solutions and forbid moving back to them. More commonly, TS forbids reversing the effect of recently applied moves by declaring *tabu* those solution attributes that change in the local search. The tabu status of solution attributes is then maintained for a number tt of iterations; the parameter tt is called the *tabu tenure* or the *tabu list length*. Unfortunately, this may forbid moves toward attractive, unvisited solutions. To avoid such an undesirable situation, an *aspiration criterion* is used to override the tabu status of certain moves. Most commonly, the aspiration criterion drops the tabu status of moves leading to a better solution than the best solution visited so far.

The use of a short-term memory in the search process is probably the most widely applied feature of TS. TS algorithms that only rely on the use of short-term memory are called *simple tabu search* algorithms in Glover (1989). To increase the efficiency of simple TS, long-term memory strategies can be used to intensify or diversify the search. Intensification strategies are intended to explore more carefully promising regions of the search space either by recovering elite solutions (i.e., the best solutions obtained so far) or attributes of these solutions. Diversification refers to the exploration of new search space regions through the introduction of new attribute combinations. Many long-term memory strategies in the context of TS are based on the memorization of the frequency of solution attributes. For a detailed discussion of techniques exploiting long-term memory, see Glover & Laguna (1997).

An algorithmic outline of a simple TS algorithm is given in figure 2.3. The functions needed to define it are the following:

- The function GenerateInitialSolution, which generates an initial solution
- The function InitializeMemoryStructures, which initializes all the memory structures used during the run of the TS algorithm
- The function GenerateAdmissibleSolutions, which is used to determine the subset of neighbor solutions which are not tabu or are tabu but satisfy the aspiration criterion
- The function SelectBestSolution, which returns the best admissible move
- The function UpdateMemoryStructures, which updates the memory structures

To date, TS appears to be one of the most successful and most widely used metaheuristics, achieving excellent results for a wide variety of problems (Glover & Laguna, 1997). Yet this efficiency is often due to a significant fine-tuning effort of a large collection of parameters and different implementation choices (Hertz, Taillard, & de Werra, 1997). However, there have been several proposals such as reactive TS, which try to make TS more robust with respect to parameter settings (Battiti & Tecchiolli, 1994). Interestingly, some theoretical proofs about the behavior of TS exist.

```
procedure SimpleTabuSearch
    $s \leftarrow$ GenerateInitialSolution
    InitializeMemoryStructures
    $s_{best} \leftarrow s$
    while (termination condition not met) do
        $\mathcal{A} \leftarrow$ GenerateAdmissibleSolutions($s$)
        $s \leftarrow$ SelectBestSolution($\mathcal{A}$)
        UpdateMemoryStructures
        if ($f(s) < f(s_{best})$) then
            $s_{best} \leftarrow s$
        end-if
    end-while
    return $s_{best}$
end-procedure
```

Figure 2.3
High-level pseudo-code for a simple tabu search (TS).

Faigle & Kern (1992) presented a convergence proof for probabilistic TS; Hanafi and Glover proved that several deterministic variants of TS implicitly enumerate the search space and, hence, are also guaranteed to find the optimal solution in finite time (Hanafi, 2000; Glover & Hanafi, 2002).

2.4.3 Guided Local Search

One alternative possibility to escape from local optima is to modify the evaluation function while searching. Guided local search (Voudouris, 1997; Voudouris & Tsang, 1995) is a metaheuristic that makes use of this idea. It uses an augmented cost function $h(s)$, $h : s \mapsto \mathbb{R}$, which consists of the original objective function $f(\cdot)$ plus additional penalty terms pn_i associated with each solution feature i. The augmented cost function is defined as $h(s) = f(s) + \omega \cdot \sum_{i=1}^{n} pn_i \cdot I_i(s)$, where the parameter ω determines the influence of the penalties on the augmented cost function, n is the number of solution features, pn_i is the penalty cost associated with solution feature i, and $I_i(s)$ is an indicator function that takes the value 1 if the solution feature i is present in the solution s and 0 otherwise. A solution feature, for example, in the TSP is an arc and the indicator function tells if a specific arc is used or not.

GLS uses the augmented cost function for choosing local search moves until it gets trapped in a local optimum $\hat{s}$ with respect to $h(\cdot)$. At this point, a utility value $u_i = I_i(\hat{s}) \cdot c_i/(1 + pn_i)$ is computed for each feature, where c_i is the cost of feature i. Features with high costs will have a high utility. The utility values are scaled by pn_i

```
procedure GuidedLocalSearch
    $s \leftarrow$ GenerateInitialSolution
    InitializePenalties
    $s_{best} \leftarrow s$
    while (termination condition not met) do
        $h \leftarrow$ ComputeAugmentedObjectiveFunction
        $\hat{s} \leftarrow$ LocalSearch($\hat{s}, h$)
        UpdatePenalties($\hat{s}$)
    end-while
    return $s_{best}$
end-procedure
```

Figure 2.4
High-level pseudo-code for guided local search (GLS).

to avoid the same high cost features from getting penalized over and over again and the search trajectory from becoming too biased. Then, the penalties of the features with maximum utility are incremented and the augmented cost function is adapted by using the new penalty values. Last, the local search is continued from $\hat{s}$, which, in general, will no longer be locally optimal with respect to the new augmented cost function.

Note that during the local search all solutions encountered must be evaluated with respect to both the original objective function and the augmented cost functions. In fact, the two provide different types of information: the original objective function $f(\cdot)$ determines the quality of a solution, while the augmented cost function is used for guiding the local search.

An algorithmic outline of GLS is given in figure 2.4. The functions to be defined for the implementation of a GLS algorithm are the following:

- The function GenerateInitialSolution, which generates an initial solution
- The function InitializePenalties, which initializes the penalties of the solution features
- The function ComputeAugmentedObjectiveFunction, which computes the new augmented evaluation function after an update of the penalties
- The function LocalSearch, which applies a local search algorithm using the augmented evaluation function
- The function UpdatePenalties, which, once the local search is stuck in a locally optimal solution, updates the penalty vector

GLS has been derived from earlier approaches which dynamically modified the evaluation function during the search like the *breakout method* (Morris, 1993) and GENET (Davenport, Tsang, Wang, & Zhu, 1994). More generally, GLS has tight connections to other weighting schemes like those used in local search algorithms for the satisfiability problem in propositional logic (SAT) (Selman & Kautz, 1993; Frank, 1996) or adaptations of Lagrangian methods to local search (Shang & Wah, 1998). In general, algorithms that modify the evaluation function at computation time are becoming more widely used.

2.4.4 Iterated Local Search

Iterated local search (Lourenço et al., 2002; Martin, Otto, & Felten, 1991) is a simple and powerful metaheuristic, whose working principle is as follows. Starting from an initial solution s, a local search is applied. Once the local search is stuck, the locally optimal solution $\hat{s}$ is perturbed by a move in a neighborhood different from the one used by the local search. This perturbed solution s' is the new starting solution for the local search that takes it to the new local optimum $\hat{s}'$. Finally, an acceptance criterion decides which of the two locally optimal solutions to select as a starting point for the next perturbation step. The main motivation for ILS is to build a randomized walk in a search space of the local optima with respect to some local search algorithm.

An algorithmic outline of ILS is given in figure 2.5. The four functions needed to specify an ILS algorithm are as follows:

- The function GenerateInitialSolution, which generates an initial solution
- The function LocalSearch, which returns a locally optimal solution $\hat{s}$ when applied to s
- The function Perturbation, which perturbs the current solution s generating an intermediate solution s'
- The function AcceptanceCriterion, which decides from which solution the search is continued at the next perturbation step

Additionally, the functions Perturbation and AcceptanceCriterion may also exploit the search history to bias their decisions (Lourenço et al., 2002).

The general idea of ILS was rediscovered by many authors, and has been given many different names, such as *iterated descent* (Baum, 1986), *large-step Markov chains* (Martin et al., 1991), *chained local optimization* (Martin & Otto, 1996), and so on. One of the first detailed descriptions of ILS was given in Martin et al. (1991), although earlier descriptions of the basic ideas underlying the approach exist (Baum,

```
procedure IteratedLocalSearch
  $s \leftarrow$ GenerateInitialSolution
  $\hat{s} \leftarrow$ LocalSearch($s$)
  $s_{best} \leftarrow \hat{s}$
  while (termination condition not met) do
    $s' \leftarrow$ Perturbation($\hat{s}$)
    $\hat{s}' \leftarrow$ LocalSearch($s'$)
    if $(f(\hat{s}') < f(s_{best}))$ then
      $s_{best} \leftarrow \hat{s}'$
    end-if
    $\hat{s} \leftarrow$ AcceptanceCriterion($\hat{s}, \hat{s}'$)
  end-while
  return $s_{best}$
end-procedure
```

Figure 2.5
High-level pseudo-code for iterated local search (ILS).

1986; Baxter, 1981). Some of the first ILS implementations have shown that the approach is very promising and current ILS algorithms are among the best-performing approximation methods for combinatorial optimization problems like the TSP (Applegate, Bixby, Chvátal, & Cook, 1999; Applegate, Cook, & Rohe, 2003; Johnson & McGeoch, 1997; Martin & Otto, 1996) and several scheduling problems (Brucker, Hurink, & Werner, 1996; Balas & Vazacopoulos, 1998; Congram, Potts, & de Velde, 2002).

2.4.5 Greedy Randomized Adaptive Search Procedures

Greedy randomized adaptive search procedures (Feo & Resende, 1989, 1995) randomize greedy construction heuristics to allow the generation of a large number of different starting solutions for applying a local search.

GRASP is an iterative procedure which consists of two phases, a construction phase and a local search phase. In the construction phase a solution is constructed from scratch, adding one solution component at a time. At each step of the construction heuristic, the solution components are ranked according to some greedy function and a number of the best-ranked components are included in a *restricted candidate list*; typical ways of deriving the restricted candidate list are either to take the best $\gamma\%$ of the solution components or to include all solution components that have a greedy value within some $\delta\%$ of the best-rated solution component. Then, one

```
procedure GRASP
  while (termination condition not met) do
    $s \leftarrow$ ConstructGreedyRandomizedSolution
    $\hat{s} \leftarrow$ LocalSearch($s$)
    if $f(\hat{s}) < f(s_{best})$ then
      $s_{best} \leftarrow \hat{s}$
    end-if
  end-while
  return $s_{best}$
end-procedure
```

Figure 2.6
High-level pseudo-code for greedy randomized adaptive search procedures (GRASP).

of the components of the restricted candidate list is chosen randomly, according to a uniform distribution. Once a full candidate solution is constructed, this solution is improved by a local search phase.

A general outline of the GRASP procedure is given in figure 2.6. For the implementation of a GRASP algorithm we need to define two main functions:

- The function ConstructGreedyRandomizedSolution, which generates a solution
- The function LocalSearch, which implements a local search algorithm

The number of available applications of GRASP is large and several extensions of the basic GRASP algorithm we have presented here have been proposed; see Festa & Resende (2002) and Resende & Ribeiro (2002) for an overview. Regarding theoretical results, it should be mentioned that standard implementations of GRASP use restricted candidate lists and therefore may not converge to the optimal solution (Mockus, Eddy, Mockus, Mockus, & Reklaitis, 1997). One way around this problem is to allow choosing the parameter γ randomly according to a uniform distribution so that occasionally all the solution components are eligible (Resende, Pitsoulis, & Pardalos, 2000).

2.4.6 Evolutionary Computation

Evolutionary computation has become a standard term to indicate problem-solving techniques which use design principles inspired from models of the natural evolution of species.

Historically, there are three main algorithmic developments within the field of EC: evolution strategies (Rechenberg, 1973; Schwefel, 1981), evolutionary programming

(Fogel et al., 1966), and genetic algorithms (Holland, 1975; Goldberg, 1989). Common to these approaches is that they are population-based algorithms that use operators inspired by population genetics to explore the search space (the most typical genetic operators are *reproduction*, *mutation*, and *recombination*). Each individual in the algorithm represents directly or indirectly (through a decoding scheme) a solution to the problem under consideration. The reproduction operator refers to the process of selecting the individuals that will survive and be part of the next generation. This operator typically uses a bias toward good-quality individuals: The better the objective function value of an individual, the higher the probability that the individual will be selected and therefore be part of the next generation. The recombination operator (often also called crossover) combines parts of two or more individuals and generates new individuals, also called *offspring*. The mutation operator is a unary operator that introduces random modifications to one individual.

Differences among the different EC algorithms concern the particular representations chosen for the individuals and the way genetic operators are implemented. For example, genetic algorithms typically use binary or discrete valued variables to represent information in individuals and they favor the use of recombination, while evolution strategies and evolutionary programming often use continuous variables and put more emphasis on the mutation operator. Nevertheless, the differences between the different paradigms are becoming more and more blurred.

A general outline of an EC algorithm is given in figure 2.7, where *pop* denotes the population of individuals. To define an EC algorithm the following functions have to be specified:

- The function InitializePopulation, which generates the initial population
- The function EvaluatePopulation, which computes the fitness values of the individuals
- The function BestOfPopulation, which returns the best individual in the current population
- The function Recombination, which repeatedly combines two or more individuals to form one or more new individuals
- The function Mutation, which, when applied to one individual, introduces a (small) random perturbation
- The function Reproduction, which generates a new population from the current one

EC is a vast field where a large number of applications and a wide variety of algorithmic variants exist. Because an overview of the EC literature would fill an

```
**procedure** EvolutionaryComputationAlgorithm
  $pop \leftarrow$ InitializePopulation
  EvaluatePopulation($pop$)
  $s_{best} \leftarrow$ BestOfPopulation($pop$)
  **while** (termination condition not met) **do**
    $pop' \leftarrow$ Recombination($pop$)
    $pop'' \leftarrow$ Mutation($pop'$)
    EvaluatePopulation($pop''$)
    $s \leftarrow$ BestOfPopulation($pop''$)
    **if** $f(s) < f(s_{best})$ **then**
      $s_{best} \leftarrow s$
    **end-if**
    $pop \leftarrow$ Reproduction($pop''$)
  **end-while**
  **return** $s_{best}$
**end-procedure**
```

Figure 2.7
High-level pseudo-code for an evolutionary computation (EC) algorithm.

entire book, we refer to the following for more details on the subject: Fogel et al., 1966; Fogel, 1995; Holland, 1975; Rechenberg, 1973; Schwefel, 1981; Goldberg, 1989; Michalewicz, 1994; Mitchell, 1996.

Still, one particular EC algorithm, called population-based incremental learning (PBIL) (Baluja & Caruana, 1995), is mentioned here because of its similarities to ACO. PBIL maintains a vector of probabilities called the *generating vector*. Starting from this vector, a population of binary strings representing solutions to the problem under consideration is randomly generated: each string in the population has the i-th bit set to 1 with a probability given by the i-th value on the generating vector. Once a population of solutions is created, the generated solutions are evaluated and this evaluation is used to increase (or decrease) the probabilities of each separate component in the generating vector with the hope that good (bad) solutions in future generations will be produced with higher (lower) probability. It is clear that in ACO the pheromone trail values play a role similar to PBIL's generating vector, and pheromone updating has the same goal as updating the probabilities in the generating vector. A main difference between ACO and PBIL consists in the fact that in PBIL all the components of the probability vector are evaluated independently, so that PBIL works well only when the solution is separable in its components.

2.4.7 Scatter Search

The central idea of scatter search (SS), first introduced by Glover (1977), is to keep a small population of *reference solutions*, called a *reference set*, and to combine them to create new solutions.

A basic version of SS proceeds as follows. It starts by creating a reference set. This is done by first generating a large number of solutions using a *diversification generation method*. Then, these solutions are improved by a local search procedure. (Typically, the number of solutions generated in this way is ten times the size of the reference set [Glover, Laguna, & Martí, 2002], while the typical size of a reference set is usually between ten and twenty solutions.) From these improved solutions, the reference set *rs* is built. The solutions to be put in *rs* are selected by taking into account both their solution quality and their diversity. Then, the solutions in *rs* are used to build a set *c_cand* of subsets of solutions. The solutions in each subset, which can be of size 2 in the simplest case, are candidates for combination. Solutions within each subset of *c_cand* are combined; each newly generated solution is improved by local search and possibly replaces one solution in the reference set. The process of subset generation, solution combination, and local search is repeated until the reference set does not change anymore.

A general outline of a basic SS algorithm is given in figure 2.8, where *pop* denotes a population of candidate solutions. To define an SS algorithm, the following functions have to be specified:

- The function GenerateDiverseSolutions, which generates a population of solutions as candidates for building the first reference set. These solutions must be diverse in the sense that they must be spread over the search space
- The function LocalSearch, which implements an improvement algorithm
- The function BestOfPopulation, which returns the best candidate solution in the current population
- The function GenerateReferenceSet, which generates the initial reference set
- The function GenerateSubsets, which generates the set *c_cand*
- The function SelectSubset, which returns one element of *c_cand*
- The function CombineSolutions, which, when applied to one of the subsets in *c_cand*, returns one or more candidate solutions
- The function WorstOfPopulation, which returns the worst candidate solution in the current population

```
procedure ScatterSearch
  $pop$ ← GenerateDiverseSolutions
  $pop$ ← LocalSearch($pop$)
  $s_{best}$ ← BestOfPopulation($pop$)
  $rs$ ← GenerateReferenceSet($pop$)
  new_solution ← true
  while (new_solution = true) do
    new_solution ← false
    $c\_cand$ ← GenerateSubsets($rs$)
    while ($c\_cand \neq \emptyset$) do
      $cc$ ← SelectSubset($c\_cand$)
      $s$ ← CombineSolutions($cc$)
      $\hat{s}$ ← LocalSearch($s$)
      $s_{worst}$ ← WorstOfPopulation($rs$)
      if $\hat{s} \notin rs$ and $f(\hat{s}) < f(s_{worst})$ then
        UpdateReferenceSet($\hat{s}$)
        new_solution ← true
      end-if
      if ($f(\hat{s}) < f(s_{best})$) then
        $s_{best} \leftarrow \hat{s}$
      end-if
      c_cand ← c_cand\$cc$
    end-while
  end-while
  return $s_{best}$
end-procedure
```

Figure 2.8
High-level pseudo-code for scatter search (SS).

▪ The function UpdateReferenceSet, which decides whether a candidate solution should replace one of the solutions in the reference set, and updates the reference set accordingly

SS is a population-based algorithm that shares some similarities with EC algorithms (Glover, 1977; Glover et al., 2002; Laguna & Martí, 2003). Solution combination in SS is analogous to *recombination* in EC algorithms; however, in SS solution combination was conceived as a linear combination of solutions that can lead to both convex and nonconvex combinations of solutions in the reference set (Glover, 1977);

nonconvex combination of solutions allows the generation of solutions that are external to the subspace spanned by the original reference set. See Laguna & Martí (2003) for an overview of implementation principles and of current applications.

2.5 Bibliographical Remarks

Combinatorial Optimization

Combinatorial optimization is a widely studied field for which a large number of textbooks and research articles exist. One of the standard references is the book by Papadimitriou & Steiglitz (1982). There also exist a variety of other textbooks which give rather comprehensive overviews of the field. Examples are books by Lawler (1976), by Nemhauser & Wolsey (1988), and the more recent book by Cook, Cunningham, Pulleyblank & Schrijver (1998). For readers interested in digging into the huge literature on combinatorial optimization, a good starting point is the book of annotated bibliographies edited by Dell'Amico, Maffioli, & Martello (1997).

The standard reference on the theory of $\mathcal{NP}$-completeness is the excellent book by Garey & Johnson (1979). A question of particular interest for researchers in metaheuristics concerns the computational complexity of approximation algorithms. A recent detailed overview of the current knowledge on this subject is given in Hochbaum (1997) and in Ausiello, Crescenzi, Gambosi, Kann, Marchetti-Spaccamela, & Protasi (1999). Of particular interest also is the recently developed complexity theory for local search algorithms, introduced in an article by Johnson, Papadimitriou, & Yannakakis (1988).

ACO Metaheuristic

The first algorithm to fall into the framework of the ACO metaheuristic was Ant System (AS) (Dorigo, 1992; Dorigo, Maniezzo, & Colorni, 1991a, 1996). AS was followed by a number of different algorithmic variants that tried to improve its performance. The ACO metaheuristic, first described in the articles by Dorigo & Di Caro (1999a,b) and Dorigo, Di Caro, & Gambardella (1999), is the result of a research effort directed at building a common framework for these algorithmic variants. Most of the available ACO algorithms are presented in chapter 3 (up-to-date information on ACO is maintained on the Web at www.aco-metaheuristic.org). To be mentioned here is also the international workshop series "ANTS: From Ant Colonies to Artificial Ants" on ant algorithms, where a large part of the contributions focus on different aspects of the ACO metaheuristic (see the Web at iridia.ulb.ac.be/~ants/ for up-to-date information on this workshop series). The proceedings of the most recent workshop of this series in 2002 are published in the *Lecture*

Notes in Computer Science series of Springer-Verlag (Dorigo, Di Caro, & Sampels, 2002a).

Other Metaheuristics

The area of metaheuristics has now become a large field with its own conference series, the Metaheuristics International Conference, which has been held biannually since 1995. After each conference, an edited book covering current research issues in the field is published (Hansen & Ribeiro, 2001; Osman & Kelly, 1996; Voss, Martello, Osman, & Roucairol, 1999).

Single-authored books which give an overview of the whole metaheuristics field are few. An inspiring such book is the recent one by Michalewicz & Fogel (2000). Two other books which cover a number of different metaheuristics are those of Sait & Youssef (1999) and Karaboga & Pham (2000). A recent survey paper is that of Blum & Roli (2003).

As far as single metaheuristics are concerned, we gave basic references to the literature in section 2.4. A large collection of references up to 1996 is provided by Osman & Laporte (1996). A book that gives an extensive overview of local search methods is that edited by Aarts & Lenstra (1997), which contains a number of contributions by leading experts. Another book which gives an overview of a number of metaheuristics (including some variants not covered in this chapter) was edited by Reeves (1995). Recent new metaheuristic ideas are collected in a book edited by Corne, Dorigo, & Glover (1999). Currently, the best overview of the field is the *Handbook of Metaheuristics*, edited by Glover & Kochenberger (2002).

2.6 Things to Remember

▪ Combinatorial optimization problems arise in many practical and theoretical problems. Often, these problems are very hard to solve to optimality. The theory of $\mathcal{NP}$-completeness classifies the problems according to their difficulty. For many combinatorial optimization problems it has been shown that they belong to the class of $\mathcal{NP}$-hard problems, which means that in the worst case the effort needed to find optimal solutions increases exponentially with problem size, unless $\mathcal{P} = \mathcal{NP}$.

▪ Exact algorithms try to find optimal solutions and additionally prove their optimality. Despite recent successes, for many $\mathcal{NP}$-hard problems the performance of exact algorithms is not satisfactory and their applicability is often limited to rather small instances.

▪ Approximate algorithms trade optimality for efficiency. Their main advantage is that in practice they often find reasonably good solutions in a very short time.

- A metaheuristic is a set of algorithmic concepts that can be used to define heuristic methods applicable to a wide set of different problems. In other words, a metaheuristic can be seen as a general algorithmic framework which can be applied to different optimization problems with relatively few modifications to make them adapted to a specific problem.

- The ACO metaheuristic was inspired by the foraging behavior of real ants. It has a very wide applicability: it can be applied to any combinatorial optimization problem for which a solution construction procedure can be conceived. The ACO metaheuristic is characterized as being a distributed, stochastic search method based on the indirect communication of a colony of (artificial) ants, mediated by (artificial) pheromone trails. The pheromone trails in ACO serve as a distributed numerical information used by the ants to probabilistically construct solutions to the problem under consideration. The ants modify the pheromone trails during the algorithm's execution to reflect their search experience.

- The ACO metaheuristic is based on a generic problem representation and the definition of the ants' behavior. Given this formulation, the ants in ACO build solutions to the problem being solved by moving concurrently and asynchronously on an appropriately defined construction graph. The ACO metaheuristic defines the way the solution construction, the pheromone update, and possible *daemon actions*—actions which cannot be performed by a single ant because they require access to nonlocal information—interact in the solution process.

- The application of ACO is particularly interesting for (1) $\mathcal{NP}$-hard problems, which cannot be efficiently solved by more traditional algorithms; (2) dynamic shortest-path problems in which some properties of the problem's graph representation change over time concurrently with the optimization process; and (3) problems in which the computational architecture is spatially distributed. The versatility of the ACO metaheuristic has been shown using several example applications.

- The ACO metaheuristic is one out of a number of metaheuristics which have been proposed in the literature. Other metaheuristics, including simulated annealing, tabu search, guided local search, iterated local search, greedy randomized adaptive search procedures, and evolutionary computation, have been discussed in this chapter. Several characteristics make ACO a unique approach: it is a constructive, population-based metaheuristic which exploits an indirect form of memory of previous performance. This combination of characteristics is not found in any of the other metaheuristics.

2.7 Thought and Computer Exercises

Exercise 2.1 We have exemplified the application of the ACO metaheuristic to a number of different combinatorial optimization problems. For each of these problems, do the following (for this exercise and the next, consider only the static example problems introduced in this chapter):

1. Define the set of candidate solutions and the set of feasible solutions.
2. Define a greedy construction heuristic. (Answering the following questions may be of some help: What are appropriate solution components? How do you measure the objective function contribution of adding a solution component? Is it always possible to construct feasible candidate solutions? How many different solutions can be generated with the constructive heuristic?)
3. Define a local search algorithm. (Answering the following questions may be of some help: How can local changes be defined? How many solution components are involved in each local search step? How do you choose which neighboring solution to move to? Does the local search always maintain feasibility of solutions?)

Exercise 2.2 Implement the construction heuristics and the local search algorithms defined in the first exercise in your favorite programming language.

Evaluate the performance of the resulting algorithms using test instances. Test instances that have already been used by other researchers are available, for example, at ORLIB mscmga.ms.ic.ac.uk/info.html. Another possibility is to look at www.metaheuristics.org.

How strongly do the local search algorithms improve the solution quality if they are applied to the solutions generated by the construction heuristics?

Exercise 2.3 Develop a description of how to apply the ACO metaheuristic to the combinatorial optimization problems you are familiar with. To do so, answer the following questions: What are the solution components? Are there different ways of defining the solution components? If yes, in which aspects do the definitions differ? How is the construction graph defined? How are the pheromone trails and the heuristic information defined? Are there different ways of defining the heuristic information? How are the constraints treated? How do you implement the ants' behavior and, in particular, how do you construct solutions?

Exercise 2.4 We have introduced three criteria to classify metaheuristics. One is the use of solution construction versus the use of local search; another is the use, or not, of a population of solutions; and the last the use, or not, of a memory within the search process. Additional criteria concern whether the evaluation function is modified during the search or not, whether an algorithm uses several neighborhoods or only a single one, and whether the metaheuristics are inspired by some process occurring in nature. Recapitulate the classification of section 2.4, for the metaheuristics discussed in this chapter. Extend this classification to also include the three additional criteria given above.

Exercise 2.5 There are a number of additional metaheuristics available, some of which are described in *New Ideas in Optimization* (Corne et al., 1999). Develop short descriptions of these metaheuristics in a format similar to that used in this chapter. To do so, first consider the general principles underlying the metaheuristics, develop a general algorithmic outline for the metaheuristic, and describe the functions that need to be defined to implement the metaheuristic. Finally, consider the range of available applications of that metaheuristic and find out about the theoretical knowledge on the convergence behavior of these metaheuristics.