3 Ant Colony Optimization Algorithms for the Traveling Salesman Problem

But you're sixty years old. They can't expect you to keep traveling every week.
—Linda in act 1, scene 1 of *Death of a Salesman*, Arthur Miller, 1949

The traveling salesman problem is an extensively studied problem in the literature and for a long time has attracted a considerable amount of research effort. The TSP also plays an important role in ACO research: the first ACO algorithm, called Ant System, as well as many of the ACO algorithms proposed subsequently, was first tested on the TSP.

There are several reasons for the choice of the TSP as the problem to explain the working of ACO algorithms: it is an important $\mathcal{NP}$-hard optimization problem that arises in several applications; it is a problem to which ACO algorithms are easily applied; it is easily understandable, so that the algorithm behavior is not obscured by too many technicalities; and it is a standard test bed for new algorithmic ideas—a good performance on the TSP is often taken as a proof of their usefulness. Additionally, the history of ACO shows that very often the most efficient ACO algorithms for the TSP were also found to be among the most efficient ones for a wide variety of other problems.

This chapter is therefore dedicated to a detailed explanation of the main members of the ACO family through examples of their application to the TSP: algorithms are described in detail, and a guide to their implementation in a C-like programming language is provided.

3.1 The Traveling Salesman Problem

Intuitively, the TSP is the problem of a salesman who, starting from his hometown, wants to find a shortest tour that takes him through a given set of customer cities and then back home, visiting each customer city exactly once. More formally, the TSP can be represented by a complete weighted graph $G = (N, A)$ with N being the set of nodes representing the cities, and A being the set of arcs. (Note that if the graph is not complete, one can always add arcs to obtain a new, complete graph G' with exactly the same optimal solutions as G; this can be achieved by assigning to the additional arcs weights that are large enough to guarantee that they will not be used in any optimal solution.) Each arc $(i, j) \in A$ is assigned a value (length) d_{ij}, which is the distance between cities i and j, with $i, j \in N$. In the general case of the asymmetric TSP, the distance between a pair of nodes i, j is dependent on the direction of traversing the arc, that is, there is at least one arc (i, j) for which $d_{ij} \neq d_{ji}$. In the symmetric TSP, $d_{ij} = d_{ji}$ holds for all the arcs in A. The goal in the TSP is to find a minimum length Hamiltonian circuit of the graph, where a Hamiltonian circuit is a

closed path visiting each of the $n = |N|$ nodes of G exactly once. Thus, an optimal solution to the TSP is a permutation π of the node indices $\{1, 2, \ldots, n\}$ such that the length $f(\pi)$ is minimal, where $f(\pi)$ is given by

$$f(\pi) = \sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)} + d_{\pi(n)\pi(1)}. \tag{3.1}$$

In the remainder of this chapter we try to highlight differences in performance among ACO algorithms by running computational experiments on instances available from the TSPLIB benchmark library (Reinelt, 1991), which is accessible on the Web at www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/. TSPLIB instances have been used in a number of influential studies of the TSP (Grötschel & Holland, 1991; Reinelt, 1994; Johnson & McGeoch, 2002) and, in part, they stem from practical applications of the TSP such as drilling holes for printed circuit boards (Reinelt, 1994) or the positioning of X-ray devices (Bland & Shallcross, 1989). Most of the TSPLIB instances are geometric TSP instances, that is, they are defined by the coordinates of a set of points and the distance between these points is computed according to some metric. Figure 3.1 gives two examples of such instances. We refer the reader to the TSPLIB website for a detailed description of how the distances are generated. In any case, independently of which metric is used, in all TSPLIB instances the distances are rounded to integers. The main reason for this choice is of a historical nature: in early computers integer computations were much quicker to perform than computations using floating numbers.

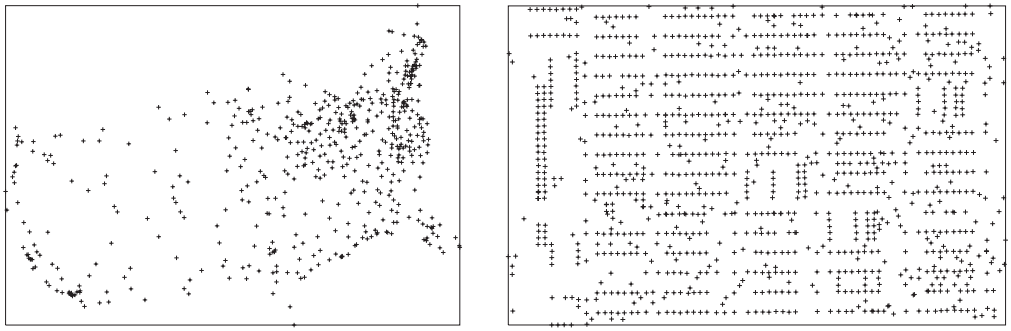


Figure 3.1
Examples of TSP: The figure on the left shows the TSP instance `att532`, which comprises 532 cities in the United States. The figure on the the right shows instance `pcb1173`, which represents the location of 1173 holes to be drilled on a printed circuit board. Each point gives the localization of cities or holes to be drilled, respectively. Both instances are taken from TSPLIB.

3.2 ACO Algorithms for the TSP

ACO can be applied to the TSP in a straightforward way, as described in section 2.3.1; the construction graph $G = (C, L)$, where the set L fully connects the components C, is identical to the problem graph, that is, $C = N$ and $L = A$; the set of states of the problem corresponds to the set of all possible partial tours; and the constraints Ω enforce that the ants construct only feasible tours that correspond to permutations of the city indices. This is always possible, because the construction graph is a complete graph and any closed path that visits all the nodes without repeating any node corresponds to a feasible tour.

In all available ACO algorithms for the TSP, the pheromone trails are associated with arcs and therefore τ_{ij} refers to the desirability of visiting city j directly after city i. The heuristic information is chosen as $\eta_{ij} = 1/d_{ij}$, that is, the heuristic desirability of going from city i directly to city j is inversely proportional to the distance between the two cities. In case $d_{ij} = 0$ for some arc (i, j), the corresponding η_{ij} is set to a very small value. As we discuss later, for implementation purposes pheromone trails are collected into a pheromone matrix whose elements are the τ_{ij}'s. This can be done analogously for the heuristic information.

Tours are constructed by applying the following simple constructive procedure to each ant: (1) choose, according to some criterion, a start city at which the ant is positioned; (2) use pheromone and heuristic values to probabilistically construct a tour by iteratively adding cities that the ant has not visited yet (see figure 3.2), until all cities have been visited; and (3) go back to the initial city. After all ants have completed their tour, they may deposit pheromone on the tours they have followed. We will see that, in some cases, before adding pheromone, the tours constructed by

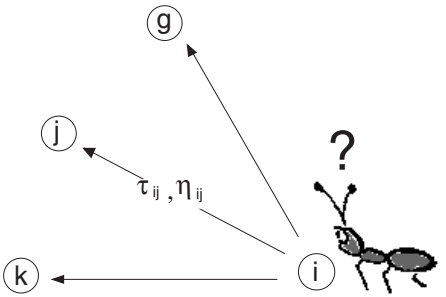


Figure 3.2
An ant arriving in city i chooses the next city to move to as a function of the pheromone values τ_{ij} and of the heuristic values η_{ij} on the arcs connecting city i to the cities j the ant has not visited yet.

```
procedure ACOMetaheuristicStatic
  Set parameters, initialize pheromone trails
  while (termination condition not met) do
    ConstructAntsSolutions
    ApplyLocalSearch        % optional
    UpdatePheromones
  end
end
```

Figure 3.3
Algorithmic skeleton for ACO algorithms applied to "static" combinatorial optimization problems. The application of a local search algorithm is a typical example of a possible daemon action in ACO algorithms.

the ants may be improved by the application of a local search procedure. This high-level description applies to most of the published ACO algorithms for the TSP, one notable exception being Ant Colony System (described in chapter 3, section 3.4.1), in which pheromone evaporation is interleaved with tour construction. In fact, when applied to the TSP and to virtually any other *static* combinatorial optimization problem (see chapter 2, section 2.2), most ACO algorithms employ a more specific algorithmic scheme than the general one of the ACO metaheuristic given in figure 2.1. This algorithm's scheme is shown in figure 3.3; after initializing the parameters and the pheromone trails, these ACO algorithms iterate through a main loop, in which first all of the ants' tours are constructed, then an optional phase takes place in which the ants' tours are improved by the application of some local search algorithm, and finally the pheromone trails are updated. This last step involves pheromone evaporation and the update of the pheromone trails by the ants to reflect their search experience. In figure 3.3 the DaemonActions procedure of figure 2.1 is replaced by the ApplyLocalSearch procedure, and by a routine (not shown in the figure and most often integrated in the UpdatePheromones procedure to facilitate implementation) that helps selecting the ants that should be allowed to deposit pheromone.

As already mentioned, the first ACO algorithm, Ant System (Dorigo, 1992; Dorigo et al., 1991a, 1996), was introduced using the TSP as an example application. AS achieved encouraging initial results, but was found to be inferior to state-of-the-art algorithms for the TSP. The importance of AS therefore mainly lies in the inspiration it provided for a number of extensions that significantly improved performance and are currently among the most successful ACO algorithms. In fact, most of these extensions are direct extensions of AS in the sense that they keep the same solution construction procedure as well as the same pheromone evaporation procedure. These extensions include elitist AS, rank-based AS, and $\mathcal{MAX}$–$\mathcal{MIN}$ AS. The main dif-

Table 3.1
ACO algorithms according to chronological order of appearance

ACO algorithm	TSP	Main references
Ant System (AS)	yes	Dorigo (1992); Dorigo, Maniezzo, & Colorni (1991a,b, 1996)
Elitist AS	yes	Dorigo (1992); Dorigo, Maniezzo, & Colorni (1991a,b, 1996)
Ant-Q	yes	Gambardella & Dorigo (1995); Dorigo & Gambardella (1996)
Ant Colony System	yes	Dorigo & Gambardella (1997a,b)
$\mathcal{MAX}$–$\mathcal{MIN}$ AS	yes	Stützle & Hoos (1996, 2000); Stützle (1999)
Rank-based AS	yes	Bullnheimer, Hartl, & Strauss (1997, 1999c)
ANTS	no	Maniezzo (1999)
Hyper-cube AS	no	Blum, Roli, & Dorigo (2001); Blum & Dorigo (2004)

In the column TSP we indicate whether this ACO algorithm has already been applied to the traveling salesman problem.

ferences between AS and these extensions are the way the pheromone update is performed, as well as some additional details in the management of the pheromone trails. A few other ACO algorithms that more substantially modify the features of AS were also proposed in the literature. These extensions, presented in section 3.4, include Ant-Q and its successor Ant Colony System (ACS), the ANTS algorithm, which exploits ideas taken from lower bounding techniques in mathematical programming, and the hyper-cube framework for ACO. We note that not all available ACO algorithms have been applied to the TSP: exceptions are Maniezzo's ANTS (see section 3.4.2) and ACO implementations based on the hyper-cube framework (see section 3.4.3).

As a final introductory remark, let us note that we do not present the available ACO algorithms in chronological order of their first publication but rather in the order of increasing complexity in the modifications they introduce with respect to AS. The chronological order of the first references and of the main publications on the available ACO algorithms is indicated in table 3.1. Striking is the relatively large gap between 1991–92 and 1995–96. In fact, the seminal publication on AS in *IEEE Transactions on Systems, Man, and Cybernetics*, although submitted in 1991, appeared only in 1996; starting from that publication the interest in ACO has grown very quickly.

3.3 Ant System and Its Direct Successors

In this section we present AS and those ACO algorithms that are largely similar to AS. We do not consider the use of the optional local search phase; the addition of local search to ACO algorithms is the topic of section 3.7.

3.3.1 Ant System

Initially, three different versions of AS were proposed (Dorigo et al., 1991a; Colorni, Dorigo, & Maniezzo, 1992a; Dorigo, 1992). These were called *ant-density*, *ant-quantity*, and *ant-cycle*. Whereas in the ant-density and ant-quantity versions the ants updated the pheromone directly after a move from one city to an adjacent city, in the ant-cycle version the pheromone update was only done after all the ants had constructed the tours and the amount of pheromone deposited by each ant was set to be a function of the tour quality. Nowadays, when referring to AS, one actually refers to ant-cycle since the two other variants were abandoned because of their inferior performance.

The two main phases of the AS algorithm constitute the ants' solution construction and the pheromone update. In AS a good heuristic to initialize the pheromone trails is to set them to a value slightly higher than the expected amount of pheromone deposited by the ants in one iteration; a rough estimate of this value can be obtained by setting, $\forall(i, j)$, $\tau_{ij} = \tau_0 = m/C^{nn}$, where m is the number of ants, and C^{nn} is the length of a tour generated by the nearest-neighbor heuristic (in fact, any other reasonable tour construction procedure would work fine). The reason for this choice is that if the initial pheromone values τ_0's are too low, then the search is quickly biased by the first tours generated by the ants, which in general leads toward the exploration of inferior zones of the search space. On the other side, if the initial pheromone values are too high, then many iterations are lost waiting until pheromone evaporation reduces enough pheromone values, so that pheromone added by ants can start to bias the search.

Tour Construction

In AS, m (artificial) ants concurrently build a tour of the TSP. Initially, ants are put on randomly chosen cities. At each construction step, ant k applies a probabilistic action choice rule, called *random proportional* rule, to decide which city to visit next. In particular, the probability with which ant k, currently at city i, chooses to go to city j is

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in \mathcal{N}_i^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta}, \quad \text{if } j \in \mathcal{N}_i^k, \tag{3.2}$$

where $\eta_{ij} = 1/d_{ij}$ is a heuristic value that is available a priori, α and β are two parameters which determine the relative influence of the pheromone trail and the heuristic information, and $\mathcal{N}_i^k$ is the feasible neighborhood of ant k when being at city i, that is, the set of cities that ant k has not visited yet (the probability of choosing a

city outside $\mathcal{N}_i^k$ is 0). By this probabilistic rule, the probability of choosing a particular arc (i, j) increases with the value of the associated pheromone trail τ_{ij} and of the heuristic information value η_{ij}. The role of the parameters α and β is the following. If $\alpha = 0$, the closest cities are more likely to be selected: this corresponds to a classic stochastic greedy algorithm (with multiple starting points since ants are initially randomly distributed over the cities). If $\beta = 0$, only pheromone amplification is at work, that is, only pheromone is used, without any heuristic bias. This generally leads to rather poor results and, in particular, for values of $\alpha > 1$ it leads to the rapid emergence of a *stagnation* situation, that is, a situation in which all the ants follow the same path and construct the same tour, which, in general, is strongly suboptimal (Dorigo, 1992; Dorigo et al., 1996). Good parameter values for the algorithms presented in this section are summarized in box 3.1.

Box 3.1
Parameter Settings for ACO Algorithms without Local Search

Our experimental study of the various ACO algorithms for the TSP has identified parameter settings that result in good performance. For the parameters that are common to almost all the ACO algorithms, good settings (if no local search is applied) are given in the following table.

ACO algorithm	α	β	ρ	m	τ_0
AS	1	2 to 5	0.5	n	m/C^{nn}
EAS	1	2 to 5	0.5	n	$(e+m)/\rho C^{nn}$
AS_{rank}	1	2 to 5	0.1	n	$0.5r(r-1)/\rho C^{nn}$
$\mathcal{MM}$AS	1	2 to 5	0.02	n	$1/\rho C^{nn}$
ACS	—	2 to 5	0.1	10	$1/nC^{nn}$

Here, n is the number of cities in a TSP instance. All variants of AS also require some additional parameters. Good values for these parameters are:

EAS: The parameter e is set to $e = n$.

AS_{rank}: The number of ants that deposit pheromones is $w = 6$.

$\mathcal{MM}$AS: The pheromone trail limits are $\tau_{max} = 1/\rho C^{bs}$ and $\tau_{min} = \tau_{max}(1 - \sqrt[n]{0.05})/((avg - 1) \cdot \sqrt[n]{0.05})$, *where avg* is the average number of different choices available to an ant at each step while constructing a solution (for a justification of these values see Stützle & Hoos (2000). When applied to small TSP instances with up to 200 cities, good results are obtained by using always the iteration-best pheromone update rule, while on larger instances it becomes increasingly important to alternate between the iteration-best and the best-so-far pheromone update rules.

ACS: In the local pheromone trail update rule: $\xi = 0.1$. In the pseudorandom proportional action choice rule: $q_0 = 0.9$.

It should be clear that in individual instances, different settings may result in much better performance. However, these parameters were found to yield reasonable performance over a significant set of TSP instances.

Each ant k maintains a memory $\mathcal{M}^k$ which contains the cities already visited, in the order they were visited. This memory is used to define the feasible neighborhood $\mathcal{N}_i^k$ in the construction rule given by equation (3.2). In addition, the memory $\mathcal{M}^k$ allows ant k both to compute the length of the tour T^k it generated and to retrace the path to deposit pheromone.

Concerning solution construction, there are two different ways of implementing it: parallel and sequential solution construction. In the parallel implementation, at each construction step all the ants move from their current city to the next one, while in the sequential implementation an ant builds a complete tour before the next one starts to build another one. In the AS case, both choices for the implementation of the tour construction are equivalent in the sense that they do not significantly influence the algorithm's behavior. As we will see, this is not the case for other ACO algorithms such as ACS.

Update of Pheromone Trails

After all the ants have constructed their tours, the pheromone trails are updated. This is done by first lowering the pheromone value on *all* arcs by a constant factor, and then adding pheromone on the arcs the ants have crossed in their tours. Pheromone evaporation is implemented by

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij}, \quad \forall (i,j) \in L, \tag{3.3}$$

where $0 < \rho \leq 1$ is the pheromone evaporation rate. The parameter ρ is used to avoid unlimited accumulation of the pheromone trails and it enables the algorithm to "forget" bad decisions previously taken. In fact, if an arc is not chosen by the ants, its associated pheromone value decreases exponentially in the number of iterations. After evaporation, all ants deposit pheromone on the arcs they have crossed in their tour:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^k, \quad \forall (i,j) \in L, \tag{3.4}$$

where $\Delta\tau_{ij}^k$ is the amount of pheromone ant k deposits on the arcs it has visited. It is defined as follows:

$$\Delta\tau_{ij}^k = \begin{cases} 1/C^k, & \text{if arc } (i,j) \text{ belongs to } T^k; \\ 0, & \text{otherwise;} \end{cases} \tag{3.5}$$

where C^k, the length of the tour T^k built by the k-th ant, is computed as the sum of the lengths of the arcs belonging to T^k. By means of equation (3.5), the better an

ant's tour is, the more pheromone the arcs belonging to this tour receive. In general, arcs that are used by many ants and which are part of short tours, receive more pheromone and are therefore more likely to be chosen by ants in future iterations of the algorithm.

As we said, the relative performance of AS when compared to other metaheuristics tends to decrease dramatically as the size of the test-instance increases. Therefore, a substantial amount of research on ACO has focused on how to improve AS.

3.3.2 Elitist Ant System

A first improvement on the initial AS, called the *elitist strategy* for Ant System (EAS), was introduced in Dorigo (1992) and Dorigo et al., (1991a, 1996). The idea is to provide strong additional reinforcement to the arcs belonging to the best tour found since the start of the algorithm; this tour is denoted as T^{bs} (*best-so-far* tour) in the following. Note that this additional feedback to the best-so-far tour (which can be viewed as additional pheromone deposited by an additional ant called *best-so-far* ant) is another example of a *daemon action* of the ACO metaheuristic.

Update of Pheromone Trails

The additional reinforcement of tour T^{bs} is achieved by adding a quantity e/C^{bs} to its arcs, where e is a parameter that defines the weight given to the best-so-far tour T^{bs}, and C^{bs} is its length. Thus, equation (3.4) for the pheromone deposit becomes

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} + e\Delta\tau_{ij}^{bs}, \tag{3.6}$$

where $\Delta\tau_{ij}^{k}$ is defined as in equation (3.5) and $\Delta\tau_{ij}^{bs}$ is defined as follows:

$$\Delta\tau_{ij}^{bs} = \begin{cases} 1/C^{bs}, & \text{if arc } (i,j) \text{ belongs to } T^{bs}; \\ 0, & \text{otherwise.} \end{cases} \tag{3.7}$$

Note that in EAS, as well as in all other algorithms presented in section 3.3, pheromone evaporation is implemented as in AS.

Computational results presented in Dorigo (1992) and Dorigo et al. (1991a, 1996) suggest that the use of the elitist strategy with an appropriate value for parameter e allows AS to both find better tours and find them in a lower number of iterations.

3.3.3 Rank-Based Ant System

Another improvement over AS is the *rank-based* version of AS (AS_{rank}), proposed by Bullnheimer et al. (1999c). In AS_{rank} each ant deposits an amount of pheromone that

decreases with its rank. Additionally, as in EAS, the best-so-far ant always deposits the largest amount of pheromone in each iteration.

Update of Pheromone Trails
Before updating the pheromone trails, the ants are sorted by increasing tour length and the quantity of pheromone an ant deposits is weighted according to the rank r of the ant. Ties can be solved randomly (in our implementation they are solved by lexicographic ordering on the ant name k). In each iteration only the $(w-1)$ best-ranked ants and the ant that produced the best-so-far tour (this ant does not necessarily belong to the set of ants of the current algorithm iteration) are allowed to deposit pheromone. The best-so-far tour gives the strongest feedback, with weight w (i.e., its contribution $1/C^{bs}$ is multiplied by w); the r-th best ant of the current iteration contributes to pheromone updating with the value $1/C^r$ multiplied by a weight given by $\max\{0, w-r\}$. Thus, the AS_{rank} pheromone update rule is

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{r=1}^{w-1} (w-r)\Delta\tau_{ij}^{r} + w\Delta\tau_{ij}^{bs}, \tag{3.8}$$

where $\Delta\tau_{ij}^{r} = 1/C^r$ and $\Delta\tau_{ij}^{bs} = 1/C^{bs}$. The results of an experimental evaluation by Bullnheimer et al. (1999c) suggest that AS_{rank} performs slightly better than EAS and significantly better than AS.

3.3.4 $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System

$\mathcal{MAX}$–$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) (Stützle & Hoos, 1997, 2000; Stützle, 1999) introduces four main modifications with respect to AS. First, it strongly exploits the best tours found: only either the iteration-best ant, that is, the ant that produced the best tour in the current iteration, or the best-so-far ant is allowed to deposit pheromone. Unfortunately, such a strategy may lead to a stagnation situation in which all the ants follow the same tour, because of the excessive growth of pheromone trails on arcs of a good, although suboptimal, tour. To counteract this effect, a second modification introduced by $\mathcal{MM}$AS is that it limits the possible range of pheromone trail values to the interval $[\tau_{min}, \tau_{max}]$. Third, the pheromone trails are initialized to the upper pheromone trail limit, which, together with a small pheromone evaporation rate, increases the exploration of tours at the start of the search. Finally, in $\mathcal{MM}$AS, pheromone trails are reinitialized each time the system approaches stagnation or when no improved tour has been generated for a certain number of consecutive iterations.

Update of Pheromone Trails

After all ants have constructed a tour, pheromones are updated by applying evaporation as in AS [equation (3.3)], followed by the deposit of new pheromone as follows:

$$\tau_{ij} \leftarrow \tau_{ij} + \Delta\tau_{ij}^{best}, \tag{3.9}$$

where $\Delta\tau_{ij}^{best} = 1/C^{best}$. The ant which is allowed to add pheromone may be either the best-so-far, in which case $\Delta\tau_{ij}^{best} = 1/C^{bs}$, or the iteration-best, in which case $\Delta\tau_{ij}^{best} = 1/C^{ib}$, where C^{ib} is the length of the iteration-best tour. In general, in $\mathcal{MM}$AS implementations both the iteration-best and the best-so-far update rules are used, in an alternate way. Obviously, the choice of the relative frequency with which the two pheromone update rules are applied has an influence on how greedy the search is: When pheromone updates are always performed by the best-so-far ant, the search focuses very quickly around T^{bs}, whereas when it is the iteration-best ant that updates pheromones, then the number of arcs that receive pheromone is larger and the search is less directed.

Experimental results indicate that for small TSP instances it may be best to use only iteration-best pheromone updates, while for large TSPs with several hundreds of cities the best performance is obtained by giving an increasingly stronger emphasis to the best-so-far tour. This can be achieved, for example, by gradually increasing the frequency with which the best-so-far tour T^{bs} is chosen for the trail update (Stützle, 1999).

Pheromone Trail Limits

In $\mathcal{MM}$AS, lower and upper limits τ_{min} and τ_{max} on the possible pheromone values on any arc are imposed in order to avoid search stagnation. In particular, the imposed pheromone trail limits have the effect of limiting the probability p_{ij} of selecting a city j when an ant is in city i to the interval $[p_{min}, p_{max}]$, with $0 < p_{min} \leq p_{ij} \leq p_{max} \leq 1$. Only when an ant k has just one single possible choice for the next city, that is $|\mathcal{N}_i^k| = 1$, we have $p_{min} = p_{max} = 1$.

It is easy to show that, in the long run, the upper pheromone trail limit on any arc is bounded by $1/\rho C^*$, where C^* is the length of the optimal tour (see proposition 4.1 in chapter 4). Based on this result, $\mathcal{MM}$AS uses an estimate of this value, $1/\rho C^{bs}$, to define τ_{max}: each time a new best-so-far tour is found, the value of τ_{max} is updated. The lower pheromone trail limit is set to $\tau_{min} = \tau_{max}/a$, where a is a parameter (Stützle, 1999; Stützle & Hoos, 2000). Experimental results (Stützle, 1999) suggest that, in order to avoid stagnation, the lower pheromone trail limits play a more

important role than τ_{max}. On the other hand, τ_{max} remains useful for setting the pheromone values during the occasional trail reinitializations.

Pheromone Trail Initialization and Reinitialization
At the start of the algorithm, the initial pheromone trails are set to an estimate of the upper pheromone trail limit. This way of initializing the pheromone trails, in combination with a small pheromone evaporation parameter, causes a slow increase in the relative difference in the pheromone trail levels, so that the initial search phase of $\mathcal{MM}$AS is very explorative.

As a further means of increasing the exploration of paths that have only a small probability of being chosen, in $\mathcal{MM}$AS pheromone trails are occasionally reinitialized. Pheromone trail reinitialization is typically triggered when the algorithm approaches the stagnation behavior (as measured by some statistics on the pheromone trails) or if for a given number of algorithm iterations no improved tour is found.

$\mathcal{MM}$AS is one of the most studied ACO algorithms and it has been extended in many ways. In one of these extensions, the pheromone update rule occasionally uses the best tour found since the most recent reinitialization of the pheromone trails instead of the best-so-far tour (Stützle, 1999; Stützle & Hoos, 2000). Another variant (Stützle, 1999; Stützle & Hoos, 1999) exploits the same pseudorandom proportional action choice rule as introduced by ACS [see equation (3.10) below], an ACO algorithm that is presented in section 3.4.1.

3.4 Extensions of Ant System

The ACO algorithms we have introduced so far achieve significantly better performance than AS by introducing minor changes in the overall AS algorithmic structure. In this section we discuss two additional ACO algorithms that, although strongly inspired by AS, achieve performance improvements through the introduction of new mechanisms based on ideas not included in the original AS. Additionally, we present the hyper-cube framework for ACO algorithms.

3.4.1 Ant Colony System

ACS (Dorigo & Gambardella, 1997a,b) differs from AS in three main points. First, it exploits the search experience accumulated by the ants more strongly than AS does through the use of a more aggressive action choice rule. Second, pheromone evaporation and pheromone deposit take place only on the arcs belonging to the best-so-far tour. Third, each time an ant uses an arc (i, j) to move from city i to city

j, it removes some pheromone from the arc to increase the exploration of alternative paths. In the following, we present these innovations in more detail.

Tour Construction

In ACS, when located at city i, ant k moves to a city j chosen according to the so-called *pseudorandom proportional* rule, given by

$$j = \begin{cases} \text{argmax}_{l \in \mathcal{N}_i^k} \{\tau_{il}[\eta_{il}]^\beta\}, & \text{if } q \leq q_0; \\ J, & \text{otherwise;} \end{cases} \tag{3.10}$$

where q is a random variable uniformly distributed in $[0, 1]$, q_0 $(0 \leq q_0 \leq 1)$ is a parameter, and J is a random variable selected according to the probability distribution given by equation (3.2) (with $\alpha = 1$).

In other words, with probability q_0 the ant makes the best possible move as indicated by the learned pheromone trails and the heuristic information (in this case, the ant is exploiting the learned knowledge), while with probability $(1 - q_0)$ it performs a biased exploration of the arcs. Tuning the parameter q_0 allows modulation of the degree of exploration and the choice of whether to concentrate the search of the system around the best-so-far solution or to explore other tours.

Global Pheromone Trail Update

In ACS only one ant (the best-so-far ant) is allowed to add pheromone after each iteration. Thus, the update in ACS is implemented by the following equation:

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} + \rho\Delta\tau_{ij}^{bs}, \quad \forall (i, j) \in T^{bs}, \tag{3.11}$$

where $\Delta\tau_{ij}^{bs} = 1/C^{bs}$. It is important to note that in ACS the pheromone trail update, both evaporation and new pheromone deposit, only applies to the arcs of T^{bs}, not to all the arcs as in AS. This is important, because in this way the computational complexity of the pheromone update at each iteration is reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$, where n is the size of the instance being solved. As usual, the parameter ρ represents pheromone evaporation; unlike AS's equations (3.3) and (3.4), in equation (3.11) the deposited pheromone is discounted by a factor ρ; this results in the new pheromone trail being a weighted average between the old pheromone value and the amount of pheromone deposited.

In initial experiments, the use of the iteration-best tour was also considered for the pheromone updates. Although for small TSP instances the differences in the final tour quality obtained by updating the pheromones using the best-so-far or the iteration-best tour was found to be minimal, for instances with more than 100 cities the use of the best-so-far tour gave far better results.

Local Pheromone Trail Update

In addition to the global pheromone trail updating rule, in ACS the ants use a local pheromone update rule that they apply immediately after having crossed an arc (i, j) during the tour construction:

$$\tau_{ij} \leftarrow (1 - \xi)\tau_{ij} + \xi\tau_0, \tag{3.12}$$

where ξ, $0 < \xi < 1$, and τ_0 are two parameters. The value of τ_0 is set to be the same as the initial value for the pheromone trails. Experimentally, a good value for ξ was found to be 0.1, while a good value for τ_0 was found to be $1/nC^{nn}$, where n is the number of cities in the TSP instance and C^{nn} is the length of a nearest-neighbor tour. The effect of the local updating rule is that each time an ant uses an arc (i, j) its pheromone trail τ_{ij} is reduced, so that the arc becomes less desirable for the following ants. In other words, this allows an increase in the exploration of arcs that have not been visited yet and, in practice, has the effect that the algorithm does not show a stagnation behavior (i.e., ants do not converge to the generation of a common path) (Dorigo & Gambardella, 1997b). It is important to note that, while for the previously discussed AS variants it does not matter whether the ants construct the tours in parallel or sequentially, this makes a difference in ACS because of the local pheromone update rule. In most ACS implementations the choice has been to let all the ants move in parallel, although there is, at the moment, no experimental evidence in favor of one choice or the other.

Some Additional Remarks

ACS is based on Ant-Q, an earlier algorithm proposed by Gambardella & Dorigo (1995) (see also Dorigo & Gambardella, 1996). In practice, the only difference between ACS and Ant-Q is in the definition of the term τ_0, which in Ant-Q is set to $\tau_0 = \gamma \max_{j \in \mathcal{N}_i^k}\{\tau_{ij}\}$, where γ is a parameter and the maximum is taken over the set of pheromone trails on the arcs connecting the city i on which ant k is positioned to all the cities the ant has not visited yet (i.e., those in the neighborhood $\mathcal{N}_i^k$).

This particular choice for τ_0 was motivated by an analogy with a similar formula used in Q-learning (Watkins & Dayan, 1992), a well-known reinforcement learning algorithm (Sutton & Barto, 1998). Because it was found that setting τ_0 to a small constant value resulted in a simpler algorithm with approximately the same performance, Ant-Q was abandoned.

There also exists an interesting relationship between $\mathcal{MM}$AS and ACS: they both use pheromone trail limits, although these are explicit in $\mathcal{MM}$AS and implicit in ACS. In fact, in ACS implementations the pheromone trails can never drop below τ_0 because both pheromone update rules [equations (3.11) and (3.12)] always add an

amount of pheromone greater than or equal to τ_0, and the initial pheromone trail value is set to the value τ_0. On the other hand, as discussed in section 4.3.5.2 of chapter 4, it can easily be verified that the pheromone trails can never have a value higher than $1/C^{bs}$. Therefore, in ACS it is implicitly guaranteed that $\forall(i,j) : \tau_0 \leq \tau_{ij} \leq 1/C^{bs}$.

Finally, it should be mentioned that ACS was the first ACO algorithm to use *candidate lists* to restrict the number of available choices to be considered at each construction step. In general, candidate lists contain a number of the best rated choices according to some heuristic criterion. In the TSP case, a candidate list contains for each city i those cities j that are at a small distance. There are several ways to define which cities enter the candidate lists. ACS first sorts the neighbors of a city i according to nondecreasing distances and then inserts a fixed number *cand* of closest cities into i's candidate list. In this case, the candidate lists can be built before solving a TSP instance and they remain fixed during the whole solution process. When located at i, an ant chooses the next city among those of i's candidate list that are not visited yet. Only if all the cities of the candidate list are already marked as visited, is one of the remaining cities chosen. In the TSP case, experimental results have shown that the use of candidate lists improves the solution quality reached by the ACO algorithms. Additionally, it leads to a significant speedup in the solution process (Gambardella & Dorigo, 1996).

3.4.2 Approximate Nondeterministic Tree Search

Approximate nondeterministic tree search (ANTS) (Maniezzo, 1999) is an ACO algorithm that exploits ideas from mathematical programming. In particular, ANTS computes lower bounds on the completion of a partial solution to define the heuristic information that is used by each ant during the solution construction. The name ANTS derives from the fact that the proposed algorithm can be interpreted as an approximate nondeterministic tree search since it can be extended in a straightforward way to a branch & bound (Bertsekas, 1995a) procedure. In fact, in Maniezzo (1999) the ANTS algorithm is extended to an exact algorithm; we refer the interested reader to the original reference for details; here we only present the ACO part of the algorithm.

Apart from the use of lower bounds, ANTS also introduces two additional modifications with respect to AS: the use of a novel action choice rule and a modified pheromone trail update rule.

Use of Lower Bounds

In ANTS, lower bounds on the completion cost of a partial solution are used to compute heuristic information on the attractiveness of adding an arc (i,j). This is

achieved by tentatively adding the arc to the current partial solution and by estimating the cost of a complete tour containing this arc by means of a lower bound. This estimate is then used to compute the value η_{ij} that influences the probabilistic decisions taken by the ant during the solution construction: the lower the estimate the more attractive the addition of a specific arc.

The use of lower bounds to compute the heuristic information has the advantage in that otherwise feasible moves can be discarded if they lead to partial solutions whose estimated costs are larger than the best-so-far solution. A disadvantage is that the lower bound has to be computed at each single construction step of an ant and therefore a significant computational overhead might be incurred. To avoid this as much as possible, it is important that the lower bound is computed efficiently.

Solution Construction

The rule used by ANTS to compute the probabilities during the ants' solution construction has a different form than that used in most other ACO algorithms. In ANTS, an ant k that is situated at city i chooses the next city j with a probability given by

$$p_{ij}^k = \frac{\zeta\tau_{ij} + (1-\zeta)\eta_{ij}}{\sum_{l\in\mathcal{N}_i^k}\zeta\tau_{il} + (1-\zeta)\eta_{il}}, \quad \text{if } j \in \mathcal{N}_i^k, \tag{3.13}$$

where ζ is a parameter, $0 \leq \zeta \leq 1$, and $\mathcal{N}_i^k$ is, as before, the feasible neighborhood (as usual, the probability of choosing an arc not belonging to $\mathcal{N}_i^k$ is 0).

An advantage of equation (3.13) is that, when compared to equation (3.2), only one parameter is used. Additionally, simpler operations that are faster to compute, like sums instead of multiplications for combining the pheromone trail and the heuristic information, are applied.

Pheromone Trail Update

Another particularity of ANTS is that it has no explicit pheromone evaporation. Pheromone updates are implemented as follows:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m}\Delta\tau_{ij}^k. \tag{3.14}$$

In the above equation (3.14), $\Delta\tau_{ij}^k$ is given by

$$\Delta\tau_{ij}^k = \begin{cases} \vartheta\left(1 - \dfrac{C^k - LB}{L_{\text{avg}} - LB}\right), & \text{if arc } (i,j) \text{ belongs to } T^k; \\ 0, & \text{otherwise;} \end{cases} \tag{3.15}$$

where ϑ is a parameter, LB is the value of a lower bound on the optimal solution value computed at the start of the algorithm and we have $LB \leq C^*$, where C^* is the length of the optimal tour, and L_{avg} is the moving average of the last l tours generated by the ants, that is, it is the average length of the l most recent tours generated by the algorithm (with l being a parameter of the algorithm). If an ant's solution is worse than the current moving average, the pheromone trail of the arcs used by the ant is decreased; if the ant's solution is better, the pheromone trail is increased. The additional effect of using equation (3.15) is a dynamic scaling of the objective function differences which may be advantageous if in later stages of the search the absolute difference between the ant's solution qualities becomes smaller and, consequently, C^k moves closer to L_{avg}. (Note that once a solution with objective function value equal to LB is found, the algorithm can be stopped, because this means that an optimal solution is found.)

As said before, ANTS has not been applied to the TSP so far, although some limited experiments were performed for the asymmetric TSP (Maniezzo, 2000). The first and main publication of ANTS concerns the quadratic assignment problem (Maniezzo, 1999), for which it obtained very good results (see chapter 5, section 5.2.1).

3.4.3 Hyper-Cube Framework for ACO

The hyper-cube framework for ACO was introduced by Blum, Roli, & Dorigo (2001) to automatically rescale the pheromone values in order for them to lie always in the interval $[0, 1]$. This choice was inspired by the mathematical programming formulation of many combinatorial optimization problems, in which solutions can be represented by binary vectors. In such a formulation, the decision variables, which can assume the values $\{0, 1\}$, typically correspond to the solution components as they are used by the ants for solution construction. A solution to a problem then corresponds to one corner of the n-dimensional hyper-cube, where n is the number of decision variables. One particular way of generating lower bounds for the problem under consideration is to relax the problem, allowing each decision variable to take values in the interval $[0, 1]$. In this case, the set of feasible solutions $\mathcal{S}_{rx}$ consists of all vectors $\vec{v} \in \mathbb{R}^n$ that are convex combinations of binary vectors $\vec{x} \in \mathbb{B}^n$:

$$\vec{v} \in \mathcal{S}_{rx} \Leftrightarrow \vec{v} = \sum_{\vec{x}_i \in \mathbb{B}^n} \gamma_i \cdot \vec{x}_i, \qquad \gamma_i \in [0, 1], \sum \gamma_i = 1.$$

The relationship with ACO becomes clear once we normalize the pheromone values to lie in the interval $[0, 1]$. In this case, the pheromone vector $\vec{\tau} = (\tau_1, \ldots, \tau_n)$

corresponds to a point in $\tilde{\mathcal{S}}$; in case $\vec{\tau}$ is a binary vector, it corresponds to a solution of the problem.

When applied to the TSP, a decision variable x_{ij} can be associated with each arc (i, j). This decision variable is set to $x_{ij} = 1$ when the arc (i, j) is used, and to $x_{ij} = 0$ otherwise. In this case, a pheromone value is associated with each decision variable. In fact, the reader may have noticed that this representation corresponds to the standard way of attacking TSPs with ACO algorithms, as presented before.

Pheromone Trail Update Rules

In the hyper-cube framework the pheromone trails are forced to stay in the interval $[0, 1]$. This is achieved by adapting the standard pheromone update rule of ACO algorithms. Let us explain the necessary change considering the pheromone update rule of AS [equations (3.3) and (3.4)]. The modified rule is given by

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} + \rho \sum_{k=1}^{m} \Delta\tau_{ij}^{k}, \tag{3.16}$$

where, to compute the rightmost term, instead of equation (3.7) we use

$$\Delta\tau_{ij}^{k} = \begin{cases} \dfrac{1/C^{k}}{\sum_{h=1}^{m}(1/C^{h})}, & \text{if arc } (i, j) \text{ is used by ant } k; \\ 0, & \text{otherwise.} \end{cases} \tag{3.17}$$

This pheromone trail update rule guarantees that the pheromone trails remain smaller than 1; the update rule is also illustrated in figure 3.4: The new pheromone vector can be interpreted as a shift of the old pheromone vector toward the vector given by the weighted average of the solutions used in the pheromone update.

3.5 Parallel Implementations

The very nature of ACO algorithms lends them to be parallelized in the data or population domains. In particular, many parallel models used in other population-based algorithms can be easily adapted to ACO. Most parallelization strategies can be classified into *fine-grained* and *coarse-grained* strategies. Characteristic of fine-grained parallelization is that very few individuals are assigned to single processors and that frequent information exchange among the processors takes place. In coarse-grained approaches, on the contrary, larger subpopulations or even full populations are assigned to single processors and information exchange is rather rare. See, for example, Cantú-Paz (2000) for an overview.

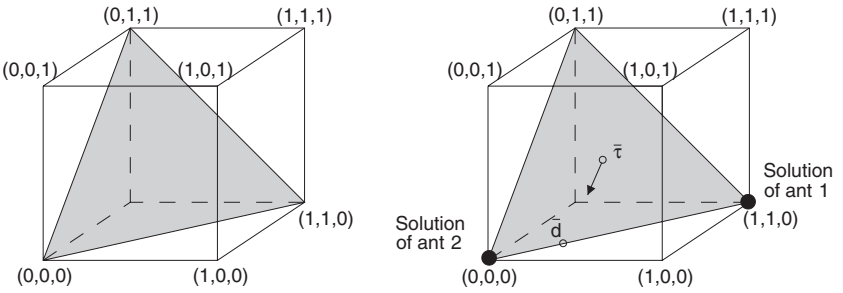


Figure 3.4
Left: Assume that the set of feasible solutions consists of the three vectors $(0,0,0)$, $(1,1,0)$, and $(0,1,1)$. Then, the pheromone vector $\vec{\tau}$ moves over the gray shaded area. Right: The two solutions $(0,0,0)$ and $(1,1,0)$ have been generated by the ants and are used for the pheromone trail update: $\vec{\tau}$ will be shifted toward $\vec{d}$. Note that $\vec{d}$ is the weighted average of the two solutions, so that it belongs to the segment connecting them [in the example $(0,0,0)$ is considered of higher quality than $(1,1,0)$, and therefore $\vec{\tau}$ is closer to $(0,0,0)$ than to $(1,1,0)$].

Fine-grained parallelization schemes have been investigated with parallel versions of AS for the TSP on the Connection Machine CM-2 adopting the approach of attributing a single processing unit to each ant (Bolondi & Bondanza, 1993). Experimental results showed that communication overhead can be a major problem with this approach, since ants end up spending most of their time communicating the modifications they made to pheromone trails. Similar negative results have been reported by Bullnheimer, Kotsis, and Strauss (1998).

As shown by several researches (Bolondi & Bondanza, 1993; Bullnheimer et al., 1998; Krüger, Merkle, & Middendorf, 1998; Middendorf, Reischle, & Schmeck, 2002; Stützle, 1998b), coarse-grained parallelization schemes are much more promising for ACO. In this case, p colonies run in parallel on p processors.

Stützle (1998b) has considered the extreme case in which there is no communication among the colonies. It is equivalent to the parallel independent run of many ACO algorithms, and is the easiest way to parallelize randomized algorithms. The computational results presented in Stützle (1998b) show that this approach is very effective.

A number of other researchers have considered the case in which information among the colonies is exchanged at certain intervals. For example, Bullnheimer et al. (1998) proposed the *partially asynchronous parallel implementation* (PAPI). In PAPI, pheromone information was exchanged among the colonies every fixed number of iterations and a high speedup was experimentally observed. Krüger et al. (1998) investigated the type of information that should be exchanged among the colonies

and how this information should be used to update the colonies' pheromone trail information. Their results showed that it is better to exchange the best solutions found so far and to use them in the pheromone update rather than to exchange complete pheromone matrices. Middendorf et al. (2002), extending the original work of Michel & Middendorf (1998), investigated different ways of exchanging solutions among m ant colonies. They let colonies exchange information every fixed number of iterations. The information exchanged is (1) the best-so-far solution that is shared among all colonies, and (2) either the locally best-so-far solutions or the w iteration-best ants, or a combination of the two, that are sent to neighbor colonies, where the neighborhood was organized as a directed ring. Their main observation was that the best results were obtained by limiting the information exchange to the locally best solutions.

Some preliminary work on the parallel implementation of an ACO algorithm on a shared memory architecture using OpenMP (Chandra, Dagum, Kohr, Maydan, McDonald, & Menon, 2000) is presented in Delisle, Krajecki, Gravel, & Gagné (2001).

3.6 Experimental Evaluation

In order to establish a meaningful comparison of the different versions of ACO discussed in the previous sections, we have reimplemented all of them using the TSP as an application problem, with the exception of ANTS, for which no application to the TSP has been reported in the literature. The resulting software package is available for download at www.aco-metaheuristic.org/aco-code/. We used this software package to study the dependence of the ACO algorithms' behavior on particular configurations or parameters. All the experiments were performed either on a 700 MHz Pentium III double-processor machine with 512 MB of RAM or on a 1.2 GHz Athlon MP double-processor machine with 1 GB of RAM; both machines were running SUSE Linux 7.3. These experiments should be understood as giving an indication of the general behavior of the available ACO algorithms when applied to $\mathcal{NP}$-hard combinatorial optimization problems and as an illustration of what happens when ACO algorithms are combined with local search algorithms. On the contrary, the experiments are not meant to present results competitive with current state-of-the-art algorithms for the TSP. In fact, current state-of-the-art algorithms for the TSP exploit complex data structures and local search routines that have not been implemented for ACO. Nevertheless, the results of our study are interesting because most of our findings remain true when ACO is applied to other $\mathcal{NP}$-hard problems.

3.6.1 The Behavior of ACO Algorithms

Artificial ants iteratively sample tours through a loop that includes a tour construction biased by the artificial pheromone trails and the heuristic information. The main mechanism at work in ACO algorithms that triggers the discovery of good tours is the positive feedback given through the pheromone update by the ants: the shorter the ant's tour, the higher the amount of pheromone the ant deposits on the arcs of its tour. This in turn leads to the fact that these arcs have a higher probability of being selected in the subsequent iterations of the algorithm. The emergence of arcs with high pheromone values is further reinforced by the pheromone trail evaporation that avoids an unlimited accumulation of pheromones and quickly decreases the pheromone level on arcs that only very rarely, or never, receive additional pheromone.

This behavior is illustrated in figure 3.5, where AS is applied to the 14-city TSP instance `burma14` from TSPLIB. The figure gives a visual representation of the pheromone matrix: pheromone trail levels are translated into gray scale, where black

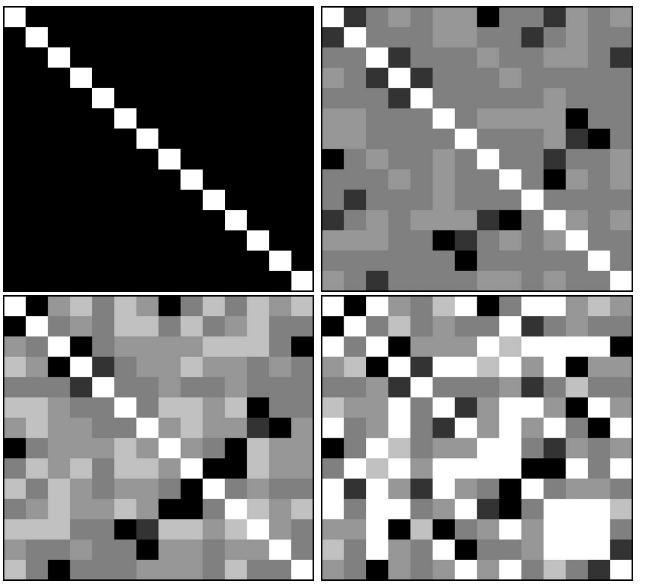


Figure 3.5
A visual representation of the pheromone matrix. The pheromone values on the arcs, stored in the pheromone matrix, are translated into gray-scale values; the darker an entry, the higher the associated pheromone trail value. The plots, from upper left to lower right, show the pheromone value for AS applied to TSPLIB instance `burma14` with 14 cities after 0, 5, 10, and 100 iterations. Note the symmetry with respect to the main diagonal, which is due to the fact that `burma14` is a symmetric TSP instance.

represents the highest pheromone trails and white the lowest ones. The four plots give snapshots of the pheromone matrix after 0, 5, 10, and 100 iterations (from upper left to lower right). At the beginning, all the matrix's cells are black except for those on the diagonal which are always white because they are initialized to zero and never updated. After five iterations, the differences between the pheromone trails are still not very manifest; this is due to the fact that pheromone evaporation and pheromone update could be applied only five times and therefore large differences between the pheromone trails could not be established yet. Also, after five iterations the pheromone trails are still rather high, which is due to the large initial pheromone values. As the algorithm continues to iterate, the differences between the pheromone values become stronger and finally a situation is reached in which only few connections have a large amount of pheromone associated with them (and therefore a large probability of being chosen) and several connections have pheromone values close to zero, making a selection of these connections very unlikely.

With good parameter settings, the long-term effect of the pheromone trails is to progressively reduce the size of the explored search space so that the search concentrates on a small number of promising arcs. Yet, this behavior may become undesirable, if the concentration is so strong that it results in an early stagnation of the search (remember that search stagnation is defined as the situation in which all the ants follow the same path and construct the same solution). In such an undesirable situation the system has ceased to explore new possibilities and no better tour is likely to be found anymore.

Several measures may be used to describe the amount of exploration an ACO algorithm still performs and to detect stagnation situations. One of the simplest possibilities is to compute the standard deviation σ_L of the length of the tours the ants construct after every iteration—if σ_L is zero, this is an indication that all the ants follow the same path (although σ_L can go to zero also in the very unlikely case in which the ants follow different tours of the same length).

Because the standard deviation depends on the absolute values of the tour lengths, a better choice is the use of the variation coefficient, defined as the quotient between the standard deviation of the tour lengths and the average tour length, which is independent of the scale.

The distance between tours gives a better indication of the amount of exploration the ants perform. In the TSP case, a way of measuring the distance $dist(T, T')$ between two tours T and T' is to count the number of arcs contained in one tour but not in the other. A decrease in the average distance between the ants' tours indicates that preferred paths are appearing, and if the average distance becomes zero, then

the system has entered search stagnation. A disadvantage of this measure is that it is computationally expensive: there are $\mathcal{O}(n^2)$ possible pairs to be compared and each single comparison has a complexity of $\mathcal{O}(n)$.

While these measures only use the final tours constructed by the ants, the λ-branching factor, $0 < \lambda < 1$, introduced in Dorigo & Gambardella (1997b), measures the distribution of the pheromone trail values more directly. Its definition is based on the following notion: If for a given city i the concentration of pheromone trail on almost all the incident arcs becomes very small but is large for a few others, the freedom of choice for extending partial tours from that city is very limited. Consequently, if this situation arises simultaneously for all the nodes of the graph, the part of the search space that is effectively searched by the ants becomes relatively small. The λ-branching factor for a city i is defined as follows: If τ^i_{max} is the maximal and τ^i_{min} the minimal pheromone trail value on arcs incident to node i, the λ-branching factor is given by the number of arcs incident to i that have a pheromone trail value $\tau_{ij} \geq \tau^i_{min} + \lambda(\tau^i_{max} - \tau^i_{min})$. The value of λ ranges over the interval $[0, 1]$, while the values of the λ-branching factors range over the interval $[2, n-1]$, where n is the number of nodes in the construction graph (which, in the TSP case, is the same as the number of cities). The average λ-branching factor $\overline{\lambda}$ is the average of the λ-branching factors of all nodes and gives an indication of the size of the search space effectively being explored by the ants. If, for example, $\overline{\lambda}$ is very close to 3, on average only three arcs for each node have a high probability of being chosen. Note that in the TSP the minimal $\overline{\lambda}$ is 2, because for each city there must be at least two arcs used by the ants to reach and to leave the city while building their solutions.

A disadvantage of the λ-branching factor is that its values depend on the setting of the parameter λ. Another possibility for a measure of stagnation would be to use the average $\overline{\mathcal{E}} = \sum_{i=1}^{n} \mathcal{E}_i / n$ of the entropies $\mathcal{E}_i$ of the selection probabilities at each node:

$$\mathcal{E}_i = -\sum_{j=1}^{l} p_{ij} \log p_{ij}, \tag{3.18}$$

where p_{ij} is the probability of choosing arc (i, j) when being in node i, and l, $1 \leq l \leq n-1$, is the number of possible choices. Still another way to measure stagnation is given by the following formula:

$$\frac{\sum_{\tau_{ij} \in T} \min\{\tau_{max} - \tau_{ij}, \tau_{ij} - \tau_{min}\}}{n^2}, \tag{3.19}$$

whose value tends to 0 as the algorithm moves toward stagnation.

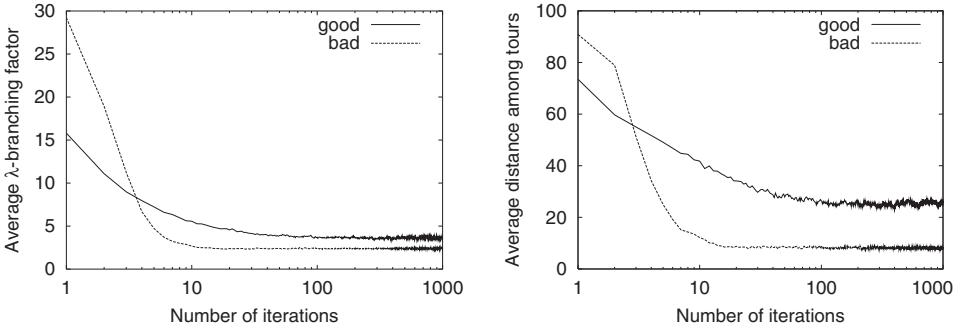


Figure 3.6
Bad behavior because of early stagnation: The plots give (left) the average λ-branching factor, $\lambda = 0.05$, and (right) the average distance among the tours generated by AS on the symmetric TSPLIB instance `kroA100`. "Good" system behavior is observed setting parameters to $\alpha = 1$, $\beta = 2$, $m = n$. "Bad" system behavior is observed setting parameters $\alpha = 5$, $\beta = 0$, $m = n$.

Behavior of AS

In this section we show the typical behavior of the average λ-branching factor and of the average distance among tours when AS has parameter settings that result in either good or bad algorithm performance. The parameter settings are denoted by *good* and *bad* in figure 3.6, and the values used are $\alpha = 1$, $\beta = 2$, $m = n$ and to $\alpha = 5$, $\beta = 0$, $m = n$ respectively. Figure 3.6 shows that for bad parameter settings the λ-branching factor converges to its minimum value much faster than for good parameter settings (λ is set to 0.05). A similar situation occurs when observing the average distance between tours. In fact, the experimental results of Dorigo et al. (1996) suggest that AS enters stagnation behavior if α is set to a large value, and does not find high-quality tours if α is chosen to be much smaller than 1. Dorigo et al. (1996) tested values of $\alpha \in \{0, 0.5, 1, 2, 5\}$. An example of bad system behavior that occurs if the amount of exploration is too large is shown in figure 3.7. Here, *good* refers to the same parameter setting as above and *bad* to the setting $\alpha = 1$, $\beta = 0$, and $m = n$. For both stagnation measures, average λ-branching factor and average distance between tours, the algorithm using the bad parameter setting is not able to focus the search on the most promising parts of the search space.

The overall result suggests that for AS good parameter settings are those that find a reasonable balance between a too narrow focus of the search process, which in the worst case may lead to stagnation behavior, and a too weak guidance of the search, which can cause excessive exploration.

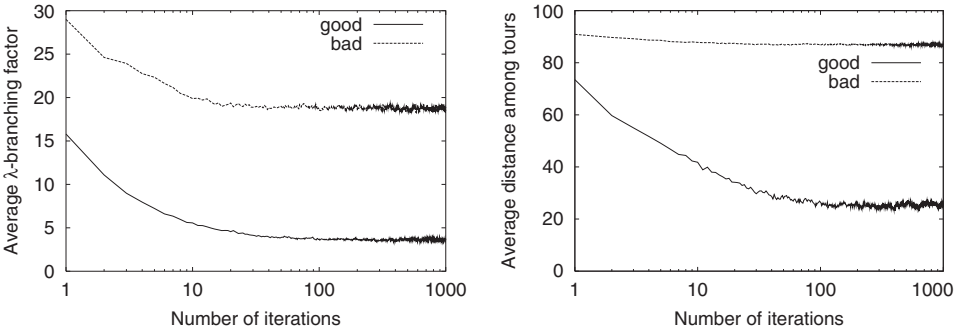


Figure 3.7
Bad behavior because of excessive exploration: The plots give (left) the average λ-branching factor, $\lambda = 0.05$, and (right) the average distance among the tours generated by AS on the symmetric TSPLIB instance `kroA100`. "Good" system behavior is observed setting parameters to $\alpha = 1, \beta = 2, m = n$. "Bad" system behavior is observed setting parameters $\alpha = 1$, $\beta = 0$, $m = n$.

Behavior of Extensions of AS

One particularity of AS extensions is that they direct the ants' search in a more aggressive way. This is mainly achieved by a stronger emphasis given to the best tours found during each iteration (e.g., in $\mathcal{MM}$AS) or the best-so-far tour (e.g., in ACS). We would expect that this stronger focus of the search is reflected by statistical measures of the amount of exploration. Figure 3.8 indicates the development of the λ-branching factor and the average distance between tours as observed in AS, EAS, AS_{rank}, $\mathcal{MM}$AS, and ACS. For this comparison we used the same parameter settings as in box 3.1, except for the value of β which was set to 2 for all algorithms.

The various ACO algorithms show, in part, strongly different behaviors, which gives an indication that there are substantial differences in their ways of directing the search. While ACS shows a low λ-branching factor and small average distances between the tours throughout the algorithm's entire run, for the others a transition from a more explorative search phase, characterized by a rather high average λ-branching factor, to an exploitation phase, characterized by a very low average λ-branching factor, can be observed. While this transition happens very soon in AS and AS_{rank}, it occurs only later in $\mathcal{MM}$AS. On the other hand, AS_{rank} is the only algorithm that enters stagnation when run for a sufficiently high number of iterations. This observation also suggests that AS_{rank} could profit from occasional pheromone trail reinitializations, as was proposed for $\mathcal{MM}$AS (Stützle & Hoos, 2000).

It is interesting to note that, although $\mathcal{MM}$AS also converges to the minimum average λ-branching factor, which suggests stagnation behavior, the average distance

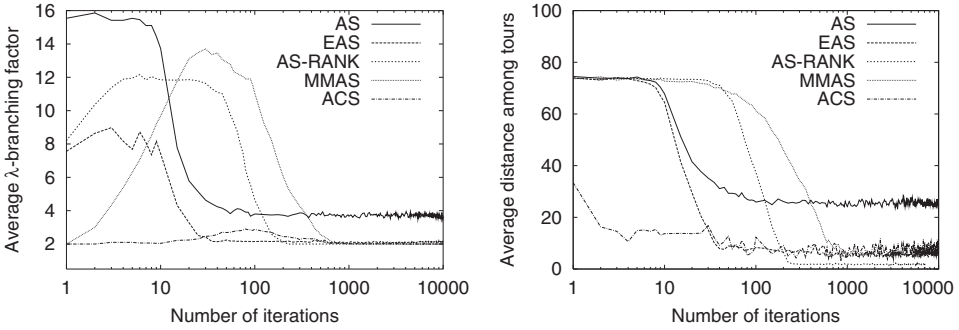


Figure 3.8
Comparing AS extensions: The plots give (left) the average λ-branching factor, $\lambda = 0.05$, and (right) the average distance among the tours for several ACO algorithms on the symmetric TSPLIB instance `kroA100`. Parameters were set as in box 3.1, except for β which was set to $\beta = 2$ for all the algorithms.

between the tours it generates remains significantly higher than zero. The reason for this apparently contradictory result is that $\mathcal{MM}$AS uses pheromone trail limits τ_{max} and τ_{min}. So, even when the pheromone trails on the arcs of a tour reach the value τ_{max} and all others have the value τ_{min}, new tours will still be explored.

A common characteristic of all of the AS extensions is that their search is focused on a specific region of the search space. An indication of this is given by the lower λ-branching factor and the lower average distance between the tours of these extensions when compared to AS. Because of this, AS extensions need to be endowed with features intended to counteract search stagnation.

It should be noted that the behavior of the various ACO algorithms also depends strongly on the parameter settings. For example, it is easy to force $\mathcal{MM}$AS to converge much faster to good tours by making the search more aggressive through the use of only the best-so-far update or by a higher evaporation rate. Nevertheless, the behavior we show in figure 3.8 is typical for reasonable parameter settings (see box 3.1).

In the following, we discuss the behavior of $\mathcal{MM}$AS and ACS in more detail. This choice is dictated by the fact that these two algorithms are the most used and often the best-performing of ACO algorithms.

Behavior of $\mathcal{MM}$AS

Of the ACO algorithms considered in this chapter, $\mathcal{MM}$AS has the longest explorative search phase. This is mainly due to the fact that pheromone trails are initialized to the initial estimate of τ_{max}, and that the evaporation rate is set to a low

value (a value that gives good results for long runs of the algorithm was found to be $\rho = 0.02$). In fact, because of the low evaporation rate, it takes time before significant differences among the pheromone trails start to appear.

When this happens, $\mathcal{MM}$AS behavior changes from explorative search to a phase of exploitation of the experience accumulated in the form of pheromone trails. In this phase, the pheromone on the arcs corresponding to the best-found tour rises up to the maximum value τ_{max}, while on all the other arcs it decreases down to the minimum value τ_{min}. This is reflected by an average λ-branching factor of 2.0. Nevertheless, the exploration of tours is still possible, because the constraint on the minimum value of pheromone trails has the effect of giving to each arc a minimum probability $p_{min} > 0$ of being chosen. In practice, during this exploitation phase $\mathcal{MM}$AS constructs tours that are similar to either the best-so-far or the iteration-best tour, depending on the algorithm implementation.

Behavior of ACS

ACS uses a very aggressive search that focuses from the very beginning around the best-so-far tour T^{bs}. In other words, it generates tours that differ only in a relatively small number of arcs from the best-so-far tour T^{bs}. This is achieved by choosing a large value for q_0 in the pseudorandom proportional action choice rule [see equation (3.10)], which leads to tours that have many arcs in common with the best-so-far tour. An interesting aspect of ACS is that while arcs are traversed by ants, their associated pheromone is diminished, making them less attractive, and therefore favoring the exploration of still unvisited arcs. Local updating has the effect of lowering the pheromone on visited arcs so that they will be chosen with a lower probability by the other ants in their remaining steps for completing a tour. As a consequence, the ants never converge to a common tour, as is also shown in figure 3.8.

3.6.2 Comparison of Ant System with Its Extensions

There remains the final question about the solution quality returned by the various ACO algorithms. In figure 3.9 we compare the development of the average solution quality measured in twenty-five trials for instance `d198` (left side) and in five trials for instance `rat783` (right side) of several ACO algorithms as a function of the computation time, which is indicated in seconds on the x-axis. We found experimentally that all extensions of AS achieve much better final solutions than AS, and in all cases the worst final solution returned by the AS extensions is better than the average final solution quality returned by AS.

In particular, it can be observed that ACS is the most aggressive of the ACO algorithms and returns the best solution quality for very short computation times.

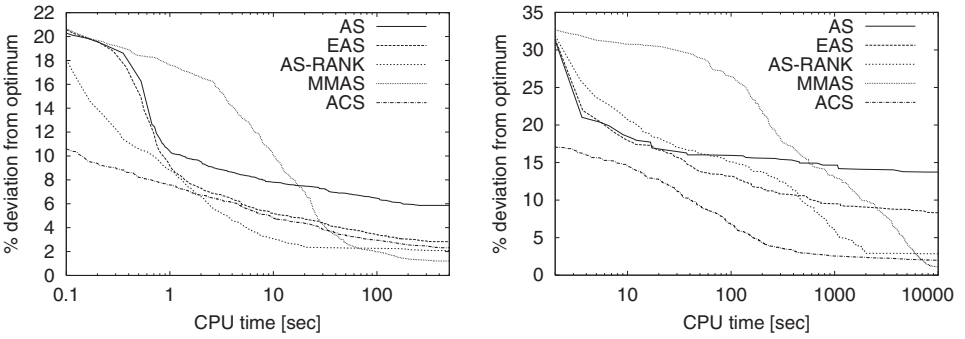


Figure 3.9
Comparing AS extensions: The plots give the development of the average percentage deviation from the optimum as a function of the computation time in seconds for AS, EAS, AS_{rank}, $\mathcal{MM}$AS, and ACS for the symmetric TSPLIB instances `d198` (left), and `rat783` (right). Parameters were set as indicated in box 3.1, except for β, which was set to $\beta = 5$ for all algorithms.

Differently, $\mathcal{MM}$AS initially produces rather poor solutions and in the initial phases it is outperformed even by AS. Nevertheless, its final solution quality, for these two instances, is the best among the compared ACO algorithms.

These results are consistent with the findings of the various published research papers on AS extensions: in all these publications it was found that the respective extensions improved significantly over AS performance. Comparisons among the several AS extensions indicate that the best performing variants are $\mathcal{MM}$AS and ACS, closely followed by AS_{rank}.

3.7 ACO plus Local Search

The vast literature on metaheuristics tells us that a promising approach to obtaining high-quality solutions is to couple a local search algorithm with a mechanism to generate initial solutions. As an example, it is well known that, for the TSP, iterated local search algorithms are currently among the best-performing algorithms. They iteratively apply local search to initial solutions that are generated by introducing modification to some locally optimal solutions (see chapter 2, section 2.4.4, for a detailed description of iterated local search).

ACO's definition includes the possibility of using local search (see figure 3.3); once ants have completed their solution construction, the solutions can be taken to their local optimum by the application of a local search routine. Then pheromones are updated on the arcs of the locally optimized solutions. Such a coupling of solution

construction with local search is a promising approach. In fact, because ACO's solution construction uses a different neighborhood than local search, the probability that local search improves a solution constructed by an ant is quite high. On the other hand, local search alone suffers from the problem of finding good starting solutions; these solutions are provided by the artificial ants.

In the following, we study how the performance of one of the ACO algorithms presented before, $\mathcal{MM}$AS, is improved when coupled with a local search. To do so, we implemented three of the most used types of local search for the TSP: 2-opt, 2.5-opt, and 3-opt. 2-opt was explained in box 2.4 (with the name 2-exchange), while 2.5-opt and 3-opt are explained in box 3.2. All three implementations exploit three standard speedup techniques: the use of nearest-neighbor lists of limited length (here 20), the use of a fixed radius nearest-neighbor search, and the use of *don't look bits*. These techniques together make the computation time increase subquadratically with the instance size. See Bentley (1992) and Johnson & McGeoch (1997) for details on these speedup techniques.

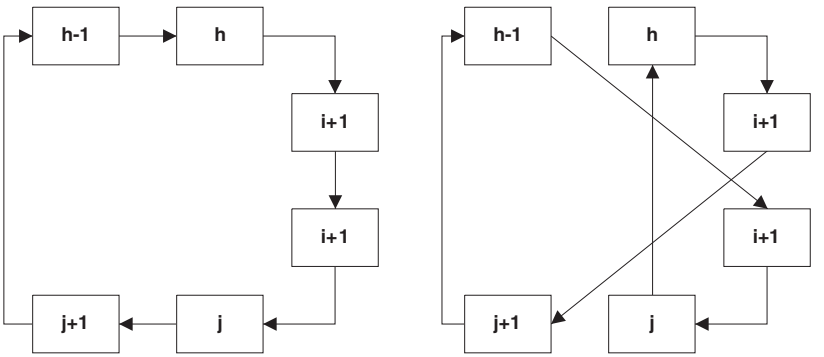
3.7.1 How to Add Local Search to ACO Algorithms?

There exist a large number of possible choices when combining local search with ACO algorithms. Some of these possibilities relate to the fundamental question of how effective and how efficient the local search should be. In fact, in most local search procedures, the better the solution quality returned, the higher the computation time required. This translates into the question whether for a given computation time it is better to frequently apply a quick local search algorithm that only slightly improves the solution quality of the initial solutions, or whether a slow but more effective local search should be used less frequently.

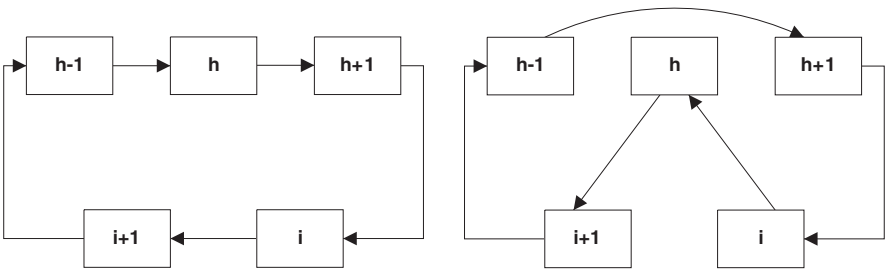
Other issues are related to particular parameter settings and to which solutions the local search should be applied. For example, the number of ants to be used, the necessity to use heuristic information or not, and which ants should be allowed to improve their solutions by a local search, are all questions of particular interest when an ACO algorithm is coupled with a local search routine. An interesting question is whether the implementation choices done and the parameter values chosen in the case of ACO algorithms are still the best once local search is added. In general, there may be significant differences regarding particular parameter settings. For example, for $\mathcal{MM}$AS it was found that when applied without local search, a good strategy is to frequently use the iteration-best ant to update pheromone trails. Yet, when combined with local search a stronger emphasis of the best-so-far update seemed to improve performance (Stützle, 1999).

Box 3.2
2.5-opt and 3-opt

The 3-opt neighborhood consists of those tours that can be obtained from a tour s by replacing at most three of its arcs. The removal of three arcs results in three partial tours that can be recombined into a full tour in eight different ways. However, only four of these eight ways involve the introduction of three new arcs, the other four reduce to 2-opt moves (see box 2.4 for details on the 2-opt neighborhood). (Note that in a 3-opt local search procedure 2-opt moves are also examined.) The figure below gives one particular example of a 3-opt exchange move.



2.5-opt is a local search algorithm that includes a strongly restricted version of a 3-opt move on top of a 2-opt local search. When checking for an improving 2-opt move, it is also checked whether inserting the city between a city i and its successor, as illustrated in the figure below, results in an improved tour.



2.5-opt leads only to a small, constant overhead in computation time over that required by a 2-opt local search but, as experimental results show (Bentley, 1992), it leads to significantly better tours. However, the tour quality returned by 2.5-opt is still significantly worse than that of 3-opt. Implementations of the above-mentioned local search procedures not using any speedup techniques result in the following time complexities for a single neighborhood search: $\mathcal{O}(n^2)$ for 2-opt and 2.5-opt, and $\mathcal{O}(n^3)$ for 3-opt.

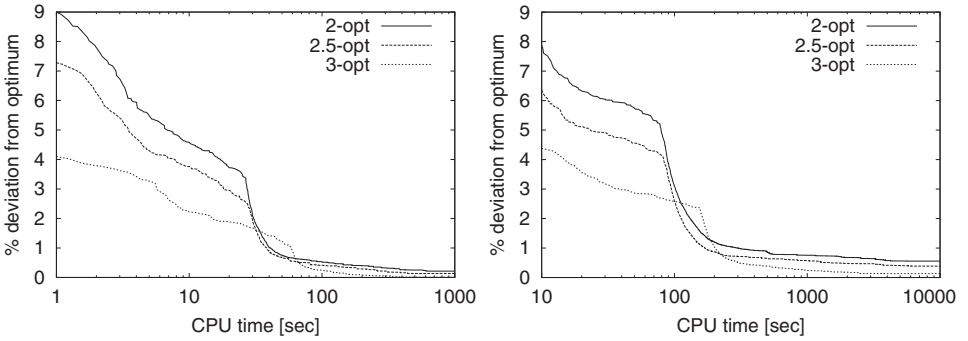


Figure 3.10
Comparing local search procedures: The plots give the average percentage deviation from the optimal tour as a function of the CPU time in seconds for $\mathcal{MM}$AS using a 2-opt, a 2.5-opt, and a 3-opt local search procedure for the symmetric TSPLIB instances `pcb1173` (left) and `pr2392` (right). Parameters are set to the values given in box 3.3.

In the following, we give some exemplary results, focusing our attention on $\mathcal{MM}$AS and ACS. In particular, we examine the influence that the strength of the local search, the number of ants, and the use of heuristic information have on the algorithms' performance.

Strength of the Local Search

We combined $\mathcal{MM}$AS with 2-opt, 2.5-opt, and 3-opt local search procedures. While the solution quality returned by these local search algorithms increases from 2-opt to 3-opt, the same is true for the necessary computation time to identify local optima (Reinelt, 1994; Johnson & McGeoch, 2002).

Figure 3.10 plots the solution quality as a function of the CPU time. For the largest amount of computation time, $\mathcal{MM}$AS combined with 3-opt gives the best average solution quality. The fact that for a short interval of time $\mathcal{MM}$AS combined with 2-opt or 2.5-opt gives slightly better results than $\mathcal{MM}$AS combined with 3-opt can be explained as follows. First, remember (see section 3.6.2 and figure 3.9) that $\mathcal{MM}$AS moves from an initial explorative phase to an exploitation phase by increasing over time the relative frequency with which the best-so-far pheromone update is applied with respect to the iteration-best pheromone update. Second, we have seen (see box 3.2) that 3-opt has a higher time complexity than 2-opt and 2.5-opt. This means that an iteration of the $\mathcal{MM}$AS with 3-opt algorithm requires more CPU time than an iteration of $\mathcal{MM}$AS with 2-opt or 2.5-opt. Therefore, the explanation for the observed temporary better behavior of $\mathcal{MM}$AS with 2-opt or 2.5-opt is that there is a

Box 3.3
Parameter Settings for ACO Algorithms with Local Search

The only ACO algorithms that have been applied with local search to the TSP are ACS and $\mathcal{MM}$AS. Good settings, obtained experimentally (see, e.g., Stützle & Hoos [2000] for $\mathcal{MM}$AS and Dorigo & Gambardella [1997b] for ACS), for the parameters common to both algorithms are indicated below.

ACO algorithm	α	β	ρ	m	τ_0
$\mathcal{MM}$AS	1	2	0.2	25	$1/\rho C^{nn}$
ACS	—	2	0.1	10	$1/nC^{nn}$

The remaining parameters are:

$\mathcal{MM}$AS: τ_{max} is set, as in box 3.1, to $\tau_{max} = 1/(\rho C^{bs})$, while $\tau_{min} = 1/(2n)$. For the pheromone deposit, the schedule for the frequency with which the best-so-far pheromone update is applied is

$$f_{bs} = \begin{cases} \infty & \text{if } i \leq 25 \\ 5 & \text{if } 26 \leq i \leq 75 \\ 3 & \text{if } 76 \leq i \leq 125 \\ 2 & \text{if } 126 \leq i \leq 250 \\ 1 & \text{otherwise} \end{cases} \tag{3.20}$$

where f_{bs} is the number of algorithm iterations between two updates performed by the best-so-far ant (in the other iterations it is the iteration-best ant that makes the update) and i is the iteration counter of the algorithm.

ACS: We have $\xi = 0.1$ and $q_0 = 0.98$.

Common to both algorithms is also that after each iteration all the tours constructed by the ants are improved by the local search. Additionally, in $\mathcal{MM}$AS occasional pheromone trail reinitializations are applied. This is done when the average λ-branching factor becomes smaller than 2.00001 and if for more than 250 iterations no improved tour has been found.

Note that on individual instances different settings may result in much better performance.

period of time in which while $\mathcal{MM}$AS with 3-opt is still in the explorative phase, $\mathcal{MM}$AS with 2-opt and $\mathcal{MM}$AS with 2.5-opt are already in the exploitation phase.

In any case, once the final tour quality obtained by the different variants is taken into account, the computational results clearly suggest that the use of more effective local searches improves the solution quality of $\mathcal{MM}$AS.

Number of Ants

In a second series of experiments we investigated the role of the number of ants m on the final performance of $\mathcal{MM}$AS. We ran $\mathcal{MM}$AS using parameter settings of $m \in \{1, 2, 5, 10, 25, 50, 100\}$ leaving all other choices the same. The result was that on small problem instances with up to 500 cities, the number of ants did not matter very

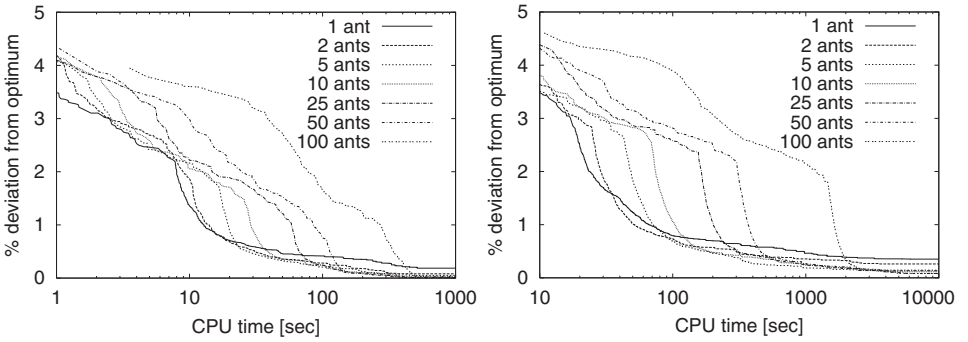


Figure 3.11
Varying the number of ants used: The plots give the average percentage deviation from the optimal tour as a function of the CPU time in seconds for $\mathcal{MM}$AS with 3-opt using a number of ants varying from 1 ant to 100 ants on the symmetric TSPLIB instances `pcb1173` (left) and `pr2392` (right). Parameters are set to the values given in box 3.3.

much with respect to the best final performance. In fact, the best trade-off between solution quality and computation time seems to be obtained when using a small number of ants—between two and ten. Yet, on the larger instances, the usefulness of having a population of ants became more apparent. For instances with more than 500 cities the worst computational results were always obtained when using only one ant and the second worst results when using two ants (see figure 3.11).

Heuristic Information

It is well known that when ACO algorithms are applied to the TSP without local search, the heuristic information is essential for the generation of high-quality tours. In fact, in the initial phases of the search, the pheromones, being set to initial random values, do not guide the artificial ants, which end up constructing (and reinforcing) tours of very bad quality. The main role of the heuristic information is to avoid this, by biasing ants so that they build reasonably good tours from the very beginning. Once local search is added to the ACO implementation, the randomly generated initial tours become good enough. It is therefore reasonable to expect that heuristic information is no longer necessary.

Experiments with $\mathcal{MM}$AS and ACS on the TSP confirmed this conjecture: when used with local search, even without using heuristic information, very high-quality tours were obtained. For example, figure 3.12 plots the average percentage deviation from the optimal tour as a function of CPU time obtained with $\mathcal{MM}$AS and ACS with local search on the symmetric TSPLIB instance `pcb1173`. The figure shows that $\mathcal{MM}$AS without heuristic information converged in most cases somewhat

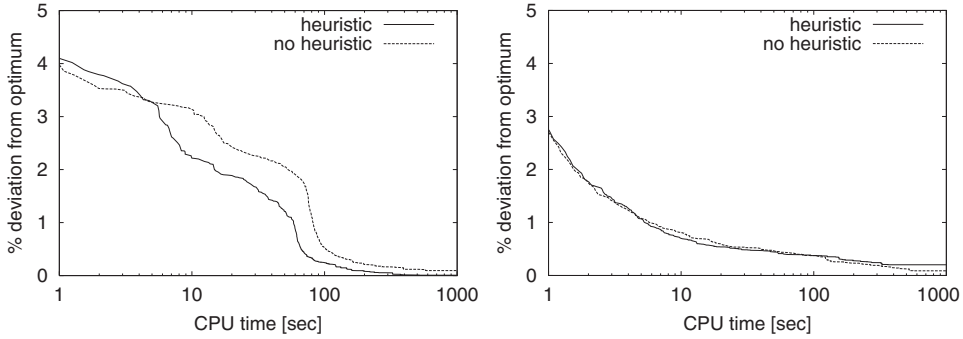


Figure 3.12
The role of heuristic information when using local search: The plots give the average percentage deviation from the optimal tour as a function of the CPU time in seconds for $\mathcal{MM}$AS (left) and ACS (right) with local search on the symmetric TSPLIB instance `pcb1173`, with and without the use of heuristic information during tour construction.

slower to tours that were slightly worse than those obtained using heuristic information, while in most cases ACS's final tour length with heuristic information was slightly worse than without.

One might argue that the question whether heuristic information is used or not is just a matter of parameter settings (not using heuristic information is simply achieved by setting $\beta = 0$). Yet, the importance of our computational results is somewhat more far-reaching. While in the TSP the distance between cities is an obvious and computationally inexpensive heuristic to use, in other problems it may be much more difficult to find, or expensive to compute, meaningful heuristic information which helps to improve performance. Fortunately, if no such obvious heuristic information exists, our computational results suggest that using an ACO algorithm incorporating local search may be enough to achieve good results.

Lamarckian versus Darwinian Pheromone Updates

Let us reconsider the choice of the tour that is used to deposit pheromones after a local search: Each ant produces a tour, say s_1, which is then transformed into another tour, say s_2, by the local search. Then the pheromones are updated. As our goal is to maximize the quality of the final tour s_2, pheromone updates must be proportional to the quality of s_2, not s_1. Once this is accepted, there are still two ways of updating the pheromones:

- We reinforce the pheromones corresponding to the final tour s_2, or
- we reinforce the pheromones corresponding to the intermediate tour s_1.

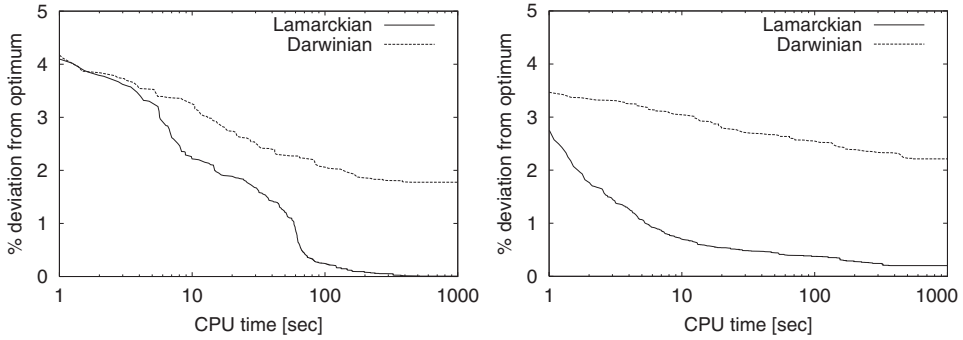


Figure 3.13
Lamarckian versus Darwinian pheromone updates: The plots give the average percentage deviation from the optimal tour as a function of the CPU time in seconds for $\mathcal{MM}$AS (left) and ACS (right) using Lamarckian and Darwinian pheromone updates on the symmetric TSPLIB instance `pcb1173`.

By analogy with similar procedures in the area of genetic algorithms (Whitley, Gordon, & Mathias, 1994), we call the first alternative the Lamarckian approach, and the second the Darwinian approach.

The main argument supporting the Lamarckian approach is that it is reasonable to think that, if the search of the ACO algorithm can be biased by the better tour s_2, then it would be stupid to use the worse tour s_1. In fact, in published ACO implementations, only the Lamarckian alternative has been used. On the other hand, the main argument in favor of the Darwinian approach is the view that what ACO algorithms with local search really do is to learn a way to generate good initial solutions for the local search, where "good" means that the initial solutions allow local search to reach good local optima.

In figure 3.13, we report some results we obtained with $\mathcal{MM}$AS and ACS on one of our test instances. As can be observed, for the TSP case, the Lamarckian approach outperforms by far the Darwinian approach. Analogous tests on other TSP instances and other problems, like the quadratic assignment problem, confirmed this observation and we conjecture that for most combinatorial optimization problems the Lamarckian approach is preferable.

3.8 Implementing ACO Algorithms

This section describes in detail the steps that have to be taken to implement an ACO algorithm for the TSP. Because the basic considerations for the implementation of different ACO algorithm variants are very similar, we mainly focus on AS

and indicate, where appropriate, the necessary changes for implementing other ACO algorithms.

A first implementation of an ACO algorithm can be quite straightforward. In fact, if a greedy construction procedure like a nearest-neighbor heuristic is available, one can use as a construction graph the same graph used by the construction procedure, and then it is only necessary to (1) add pheromone trail variables to the construction graph and (2) define the set of artificial ants to be used for constructing solutions in such a way that they implement, according to equation (3.2), a randomized version of the construction procedure. It must be noted, however, that in order to have an efficient implementation, often additional data structures are required, like arrays to store information which, although redundant, make the processing much faster. In the following, we describe the steps to be taken to obtain an efficient implementation of AS. We will give a *pseudo-code* description of a possible implementation in a C-like notation. This description is general enough to allow a reader with some previous experience in procedural or object-oriented programming (a university-level first-year programming course should suffice) to implement an efficient version of any of the ACO algorithms presented in this chapter. Additionally, a C code of several ACO algorithms is available online at www.aco-metaheuristic.org/aco-code/.

3.8.1 Data Structures

As a first step, the basic data structures have to be defined. These must allow storing the data about the TSP instance and the pheromone trails, and representing artificial ants.

Figure 3.14 gives a general outline of the main data structures that are used for the implementation of an ACO algorithm, which includes the data for the problem representation and the data for the representation of the ants, as explained below.

Problem Representation

Intercity Distances. Often a symmetric TSP instance is given as the coordinates of a number of n points. In this case, one possibility would be to store the x and y coordinates of the cities in two arrays and then compute on the fly the distance between the cities as needed. However, this leads to a significant computational overhead: obviously, it is more reasonable to precompute all intercity distances and to store them in a symmetric *distance matrix* with n^2 entries. In fact, although for symmetric TSPs we only need to store $n(n-1)/2$ distinct distances, it is more efficient to use an n^2 matrix to avoid performing additional operations to check whether, when accessing a generic distance $d(i, j)$, entry (i, j) or entry (j, i) of the matrix should be used.

```
% Representation of problem data

integer dist[n][n]          % distance matrix
integer nn_list[n][nn]      % matrix with nearest neighbor lists of depth nn
real    pheromone[n][n] % pheromone matrix
real    choice_info[n][n] % combined pheromone and heuristic information

% Representation of ants

structure single_ant
begin
  integer tour_length % the ant's tour length
  integer tour[n + 1]  % ant's memory storing (partial) tours
  integer visited[n]    % visited cities
end

single_ant ant[m]       % structure of type single_ant
```

Figure 3.14
Main data structures for the implementation of an ACO algorithm for the TSP.

Note that for very large instances it may be necessary to compute distances on the fly, if it is not possible (or too expensive) to keep the full distance matrix in the main memory. Fortunately, in these cases there exist some intermediate possibilities, such as storing the distances between a city and the cities of its nearest-neighbor list, that greatly reduce the necessary amount of computation. It is also important to know that, for historical reasons, in almost all the TSP literature, the distances are stored as integers. In fact, in old computers integer operations used to be much faster than operations on real numbers, so that by setting distances to be integers, much more efficient code could be obtained.

Nearest-Neighbor Lists. In addition to the distance matrix, it is convenient to store for each city a list of its nearest neighbors. Let d_i be the list of the distances from a city i to all cities j, with $j = 1, \ldots n$ and $i \neq j$ (we assume here that the value d_{ii} is assigned a value larger than d_{max}, where d_{max} is the maximum distance between any two cities). The nearest-neighbor list of a city i is obtained by sorting the list d_i according to nondecreasing distances, obtaining a sorted list d_i'; ties can be broken randomly. The position r of a city j in city i's nearest-neighbor list $nn\_list[i]$ is the index of the distance d_{ij} in the sorted list d_i', that is, $nn\_list[i][r]$ gives the identifier (index) of the r-th nearest city to city i (i.e., $nn\_list[i][r] = j$). Nearest-neighbor lists

for all cities can be constructed in $\mathcal{O}(n^2 \log n)$ (in fact, you have to repeat a sorting algorithm over $n-1$ cities for each city).

An enormous speedup is obtained for the solution construction in ACO algorithms, if the nearest-neighbor list is cut off after a constant number *nn* of nearest neighbors, where typically *nn* is a small value ranging between 15 and 40. In this case, an ant located in city *i* chooses the next city among the *nn* nearest neighbors of *i*; in case the ant has already visited all the nearest neighbors, then it makes its selection among the remaining cities. This reduces the complexity of making the choice of the next city to $\mathcal{O}(1)$, unless the ant has already visited all the cities in *nn_list*[*i*]. However, it should be noted that the use of truncated nearest-neighbor lists can make it impossible to find the optimal solution.

Pheromone Trails. In addition to the instance-related information, we also have to store for each connection (i, j) a number τ_{ij} corresponding to the pheromone trail associated with that connection. In fact, for symmetric TSPs this requires storing $n(n-1)/2$ distinct pheromone values, because we assume that $\tau_{ij} = \tau_{ji}$, $\forall(i, j)$. Again, as was the case for the distance matrix, it is more convenient to use some redundancy and to store the pheromones in a symmetric n^2 matrix.

Combining Pheromone and Heuristic Information. When constructing a tour, an ant located on city *i* chooses the next city *j* with a probability which is proportional to the value of $[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}$. Because these very same values need to be computed by each of the *m* ants, computation times may be significantly reduced by using an additional matrix *choice_info*, where each entry *choice_info*[*i*][*j*] stores the value $[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}$. Again, in the case of a symmetric TSP instance, only $n(n-1)/2$ values have to be computed, but it is convenient to store these values in a redundant way as in the case of the pheromone and the distance matrices. Additionally, one may store the η_{ij}^{β} values in a further matrix *heuristic* (not implemented in the code associated with the book) to avoid recomputing these values after each iteration, because the heuristic information stays the same throughout the whole run of the algorithm (some tests have shown that the speedup obtained when no local search is used is approximately 10%, while no significant differences are observed when local search is used). Finally, if some distances are zero, which is in fact the case for some of the benchmark instances in the TSPLIB, then one may set them to a very small positive value to avoid division by zero.

Speeding Up the Pheromone Update. Further optimization can be introduced by restricting the computation of the numbers in the *choice_info* matrix to the connections between a city and the cities of its nearest-neighbor list. In fact, this technique,

which is exploited in the implementation of the various ACO algorithms in the accompanying code, strongly reduces the computation time when ACO algorithms are applied to large TSP instances with several hundreds or thousands of cities.

Representing Ants

An ant is a simple computational agent which constructs a solution to the problem at hand, and may deposit an amount of pheromone $\Delta\tau$ on the arcs it has traversed. To do so, an ant must be able to (1) store the partial solution it has constructed so far, (2) determine the feasible neighborhood at each city, and (3) compute and store the objective function value of the solutions it generates.

The first requirement can easily be satisfied by storing the partial tour in a sufficiently large array. For the TSP we represent tours by arrays of length $n + 1$, where at position $n + 1$ the first city is repeated. This choice makes easier some of the other procedures like the computation of the tour length.

The knowledge of the partial tour at each step is sufficient to allow the ant to determine whether a city j is in its feasible neighborhood: it is enough to scan the partial tour for the occurrence of city j. If city j has not been visited yet, then it is member of the feasible neighborhood; otherwise it is not. Unfortunately, this simple way of determining the feasible neighborhood involves an operation of worst-case complexity $\mathcal{O}(n)$ for each city i, resulting in a high computational overhead. The simplest way around this problem is to associate with each ant an additional array *visited* whose values are set to $visited[j] = 1$ if city i has already been visited by the ant, and to $visited[j] = 0$ otherwise. This array is updated by the ant while it builds a solution.

Finally, the computation of the tour length, stored by the ant in the *tour_length* variable, can easily be done by summing the length of the n arcs in the ant's tour.

Hence, an ant may be represented by a structure that comprises one variable *tour_length* to store the ant's objective function value, one $(n + 1)$-dimensional array *tour* to store the ant's tour, and one n-dimensional array *visited* to store the visited cities (note that in figure 3.14, the array *visited*, part of the data structure *single_ant*, is declared of type **integer**; however, to save memory, it could be declared of type **Boolean**).

Overall Memory Requirement

For representing all the necessary data for the problem we need four matrices of dimension $n \times n$ for representing the distance matrix, the pheromone matrix, the heuristic information matrix, and the *choice_info* matrix, and a matrix of size $n \times nn$ for the nearest-neighbor lists. Additionally, for each of the ants we need two arrays of size $(n + 1)$ and n to store, respectively, the tour and the visited cities, as well as an

integer for storing the tour's length. Finally, we need a variable for representing each of the m ants. Since the number of ants is typically either a small constant (this is the case for ACS and Ant-Q or for most ACO algorithms with local search) or on the order of n (this is the case for AS variants without use of local search), the overall memory requirement is $\mathcal{O}(n^2)$. In addition to these main data structures, it is also necessary to store intermediate results, such as the best solution found so far, and statistical information about the algorithm performance; nevertheless, these additional data require only a very minor amount of memory when compared to the data for representing the colony of ants and the problem.

To derive a more exact estimate of the memory requirements, we can assume that representing an integer value on a computer takes 4 bytes and representing a "real" number takes 8 bytes. Additionally, we assume the number of ants to be $m = n$, and we do not consider the heuristic information matrix. The estimate is obtained as follows (see figure 3.14): $24n^2$ bytes are necessary for the problem data ($4n^2$ for the distance matrix, $4n^2$ for the matrix *nn_list*, $8n^2$ for the pheromone matrix, and $8n^2$ for the matrix *choice_info*), while $8n^2$ bytes are needed for the representation of the ants (there are $m = n$ ants and each ant requires two integer arrays of length n). The overall memory requirement can therefore be assumed to be roughly $32n^2$ bytes, which is a slight underestimate of the real memory consumption. Memory requirements increase strongly with problem size, because the memory requirement is quadratic in n. However, the memory requirements are reasonable when considering the memory available in current computers (using the above $32n^2$ estimate, instances of up to 4000 cities can be tackled by ACO algorithms with a computer with 512 MB of RAM), and the fact that the problem instances of most combinatorial optimization problems to which ACO has been applied (see chapter 5 for an overview) are typically much smaller than those of the TSP, so that memory consumption is rarely an issue.

3.8.2 The Algorithm

The main tasks to be considered in an ACO algorithm are the solution construction, the management of the pheromone trails, and the additional techniques such as local search. In addition, the data structures and parameters need to be initialized and some statistics about the run need to be maintained. In figure 3.15 we give a high-level view of the algorithm, while in the following we give some details on how to implement the different procedures of AS in an efficient way.

Data Initialization

In the data initialization, (1) the instance has to be read; (2) the distance matrix has to be computed; (3) the nearest-neighbor lists for all cities have to be computed; (4)

```
procedure ACOforTSP
  InitializeData
  while (not terminate) do
    ConstructSolutions
    LocalSearch
    UpdateStatistics
    UpdatePheromoneTrails
  end-while
end-procedure
```

Figure 3.15
High-level view of an ACO algorithm for the TSP.

```
procedure InitializeData
  ReadInstance
  ComputeDistances
  ComputeNearestNeighborLists
  ComputeChoiceInformation
  InitializeAnts
  InitializeParameters
  InitializeStatistics
end-procedure
```

Figure 3.16
Procedure to initialize the algorithm.

the pheromone matrix and the *choice_info* matrix have to be initialized; (5) the ants have to be initialized; (6) the algorithm's parameters must be initialized; and (7) some variables that keep track of statistical information, such as the used CPU time, the number of iterations, or the best solution found so far, have to be initialized. A possible organization of these tasks into several data initialization procedures is indicated in figure 3.16.

Termination Condition

The program stops if at least one termination condition applies. Possible termination conditions are: (1) the algorithm has found a solution within a predefined distance from a lower bound on the optimal solution quality; (2) a maximum number of tour constructions or a maximum number of algorithm iterations has been reached; (3) a maximum CPU time has been spent; or (4) the algorithm shows stagnation behavior.

```
      procedure ConstructSolutions
1       for k = 1 to m do
2         for i = 1 to n do
3           ant[k].visited[i] ← false
4         end-for
5       end-for
6       step ← 1
7       for k = 1 to m do
8         r ← random{1, ..., n}
9         ant[k].tour[step] ← r
10        ant[k].visited[r] ← true
11      end-for
12      while (step < n) do
13        step ← step + 1
14        for k = 1 to m do
15          ASDecisionRule(k, step)
16        end-for
17      end-while
18      for k = 1 to m do
19        ant[k].tour[n + 1] ← ant[k].tour[1]
20        ant[k].tour_length ← ComputeTourLength(k)
21      end-for
      end-procedure
```

Figure 3.17
Pseudo-code of the solution construction procedure for AS and its variants.

Solution Construction

The tour construction is managed by the procedure ConstructSolutions, shown in figure 3.17. The solution construction requires the following phases.

1. First, the ants' memory must be emptied. This is done in lines 1 to 5 of procedure ConstructSolutions by marking all cities as unvisited, that is, by setting all the entries of the array *ants.visited* to *false* for all the ants.

2. Second, each ant has to be assigned an initial city. One possibility is to assign each ant a random initial city. This is accomplished in lines 6 to 11 of the procedure. The function random returns a random number chosen according to a uniform distribution over the set $\{1, \ldots, n\}$.

3. Next, each ant constructs a complete tour. At each construction step (see the procedure in figure 3.17) the ants apply the AS action choice rule [equation (3.2)]. The procedure ASDecisionRule implements the action choice rule and takes as parameters the ant identifier and the current construction step index; this is discussed below in more detail.

4. Finally, in lines 18 to 21, the ants move back to the initial city and the tour length of each ant's tour is computed. Remember that, for the sake of simplicity, in the tour representation we repeat the identifier of the first city at position $n + 1$; this is done in line 19.

As stated above, the solution construction of all of the ants is synchronized in such a way that the ants build solutions in parallel. The same behavior can be obtained, for all AS variants, by ants that construct solutions sequentially, because the ants do not change the pheromone trails at construction time (this is not the case for ACS, in which case the sequential and parallel implementations give different results).

While phases (1), (2), and (4) are very straightforward to code, the implementation of the action choice rule requires some care to avoid large computation times. In the action choice rule an ant located at city i probabilistically chooses to move to an unvisited city j based on the pheromone trails τ_{ij}^{α} and the heuristic information η_{ij}^{β} [see equation (3.2)].

Here we give pseudo-codes for the action choice rule with and without consideration of candidate lists. The pseudo-code for the first variant ASDecisionRule is given in figure 3.18. The procedure works as follows: first, the current city c of ant k is determined (line 1). The probabilistic choice of the next city then works analogously to the *roulette wheel* selection procedure of evolutionary computation (Goldberg, 1989): each value $choice\_info[c][j]$ of a city j that ant k has not visited yet determines a slice on a circular roulette wheel, the size of the slice being proportional to the weight of the associated choice (lines 2–10). Next, the wheel is spun and the city to which the marker points is chosen as the next city for ant k (lines 11–17). This is implemented by

1. summing the weight of the various choices in the variable *sum_probabilities*,
2. drawing a uniformly distributed random number r from the interval $[0, sum\_probabilities]$,
3. going through the feasible choices until the sum is greater or equal to r.

Finally, the ant is moved to the chosen city, which is marked as visited (lines 18 and 19).

```
    procedure ASDecisionRule(k, i)
      input  k   % ant identifier
      input  i   % counter for construction step
1     c ← ant[k].tour[i − 1]
2     sum_probabilities = 0.0
3     for j = 1 to n do
4       if ant[k].visited[ j] then
5         selection_probability[ j] ← 0.0
6       else
7         selection_probability[ j] ← choice_info[c][ j]
8         sum_probabilities ← sum_probabilities + selection_probability[ j]
9       end-if
10    end-for
11    r ← random[0, sum_probabilities]
12    j ← 1
13    p ← selection_probability[ j]
14    while (p < r) do
15      j ← j + 1
16      p ← p + selection_probability[ j]
17    end-while
18    ant[k].tour[i] ← j
19    ant[k].visited[ j] ← true
    end-procedure
```

Figure 3.18
AS without candidate lists: pseudo-code for the action choice rule.

These construction steps are repeated until the ants have completed a tour. Since each ant has to visit exactly n cities, all the ants complete the solution construction after the same number of construction steps.

When exploiting candidate lists, the procedure ASDecisionRule needs to be adapted, resulting in the procedure NeighborListASDecisionRule, given in figure 3.19. A first change is that when choosing the next city, one needs to identify the appropriate city index from the candidate list of the current city c. This results in changes of lines 3 to 10 of figure 3.18: the maximum value of index j is changed from n to nn in line 3 and the test performed in line 4 is applied to the j-th nearest neighbor given by $nn\_list[c][j]$. A second change is necessary to deal with the situation in which all the cities in the candidate list have already been visited by ant k. In this case, the variable *sum_probabilities* keeps its initial value 0.0 and one city out of those not in

```
    procedure NeighborListASDecisionRule(k, i)
      input  k  % ant identifier
      input  i  % counter for construction step
1     c ← ant[k].tour[i − 1]
2     sum_probabilities ← 0.0
3     for j = 1 to nn do
4       if ant[k].visited[nn_list[c][ j]] then
5         selection_probability[ j] ← 0.0
6       else
7         selection_probability[ j] ← choice_info[c][nn_list[c][ j]]
8         sum_probabilities ← sum_probabilities + selection_probability[ j]
9       end-if
10    end-for
11    if (sum_probabilities = 0.0) then
12      ChooseBestNext(k, i)
13    else
14      r ← random[0, sum_probabilities]
15      j ← 1
16      p ← selection_probability[ j]
17      while (p < r) do
18        j ← j + 1
19        p ← p + selection_probability[ j]
20      end-while
21      ant[k].tour[i] ← nn_list[c][ j]
22      ant[k].visited[nn_list[c][ j]] ← true
23    end-if
    end-procedure
```

Figure 3.19
AS with candidate lists: pseudo-code for the action choice rule.

```
procedure ChooseBestNext(k, i)
  input  k   % ant identifier
  input  i   % counter for construction step
  v ← 0.0
  c ← ant[k].tour[i − 1]
  for j = 1 to n do
    if not ant[k].visited[ j] then
      if choice_info[c][ j] > v then
        nc ← j          % city with maximal τ^α η^β
        v ← choice_info[c][ j]
      end-if
    end-if
  end-for
  ant[k].tour[i] ← nc
  ant[k].visited[nc] ← true
end-procedure
```

Figure 3.20
AS: pseudo-code for the procedure ChooseBestNext.

the candidate list is chosen: the procedure ChooseBestNext (see pseudo-code in figure 3.20) is used to identify the city with maximum value of $[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}$ as the next to move to.

It is clear that by using candidate lists the computation time necessary for the ants to construct solutions can be significantly reduced, because the ants choose from among a much smaller set of cities. Yet, the computation time is reduced only if the procedure ChooseBestNext does not need to be applied too often. Fortunately, as also suggested by the computational results presented in Gambardella & Dorigo (1996), this seems not to be the case.

Local Search

Once the solutions are constructed, they may be improved by a local search procedure. While a simple 2-opt local search can be implemented in a few lines, the implementation of an efficient variant is somewhat more involved. This is already true to some extent for the implementation of the 3-opt local search, and even more for the Lin-Kernighan heuristic. Since the details of the local search are not important for understanding how ACO algorithms can be coded efficiently, we refer to the accompanying code (available at www.aco-metaheuristic.org/aco-code/) for more information on the local search implementation.

```
procedure ASPheromoneUpdate
  Evaporate
  for k = 1 to m do
    DepositPheromone(k)
  end-for
  ComputeChoiceInformation
end-procedure
```

Figure 3.21
AS: management of the pheromone updates.

```
procedure Evaporate
  for i = 1 to n do
    for j = i to n do
      pheromone[i][j] ← (1 − ρ) · pheromone[i][j]
      pheromone[j][i] ← pheromone[i][j]   % pheromones are symmetric
    end-for
  end-for
end-procedure
```

Figure 3.22
AS: implementation of the pheromone evaporation procedure.

Pheromone Update

The last step in an iteration of AS is the pheromone update. This is implemented by the procedure ASPheromoneUpdate (figure 3.21), which comprises two pheromone update procedures: pheromone evaporation and pheromone deposit. The first one, Evaporate (figure 3.22), decreases the value of the pheromone trails on all the arcs (i, j) by a constant factor ρ. The second one, DepositPheromone (figure 3.23), adds pheromone to the arcs belonging to the tours constructed by the ants. Additionally, the procedure ComputeChoiceInformation computes the matrix *choice_info* to be used in the next algorithm iteration. Note that in both procedures care is taken to guarantee that the pheromone trail matrix is kept symmetric, because of the symmetric TSP instances.

When attacking large TSP instances, profiling the code showed that the pheromone evaporation and the computation of the *choice_info* matrix can require a considerable amount of computation time. On the other hand, when using candidate lists in the solution construction, only a small part of the entries of the pheromone

```
procedure DepositPheromone(k)
  input  k  % ant identifier
  Δτ ← 1/ant[k].tour_length
  for i = 1 to n do
    j ← ant[k].tour[i]
    l ← ant[k].tour[i + 1]
    pheromone[ j][l] ← pheromone[ j][l] + Δτ
    pheromone[l][ j] ← pheromone[ j][l]
  end-for
end-procedure
```

Figure 3.23
AS: implementation of the pheromone deposit procedure.

matrix are ever required. Therefore, the exploitation of candidate lists speeds up also the pheromone update. In fact, the use of candidate lists with a constant number of nearest neighbors reduces the complexity of these two procedures to $\mathcal{O}(n)$, although with a large constant hidden in the $\mathcal{O}(\cdot)$ notation.

Concerning pheromone depositing, we note that, differently from AS, the best-performing ACO algorithms typically allow only one or, at most, very few ants to deposit pheromone. In this case, the complexity of the pheromone deposit is of order $\mathcal{O}(n)$. Therefore, only for AS and EAS is the complexity of the pheromone trail deposit procedure $\mathcal{O}(n^2)$ if the number of ants m is set to be proportional to n, as suggested in the original papers (Dorigo et al., 1991a,b, 1996; Bauer et al., 2000).

Note that this type of speedup technique for the pheromone trail update is not necessary for ACS, because in ACS only the pheromone trails of arcs that are crossed by some ant have to be changed and the number of ants in each iteration is a low constant.

Statistical Information about ACO Algorithm Behavior
The last step in the implementation of AS is to store statistical data on algorithm behavior (examples are the best-found solution since the start of the algorithm run, or the iteration number at which the best solution was found). Details about these procedures are available at www.aco-metaheuristic.org/aco-code/.

3.8.3 Changes for Implementing Other ACO Algorithms

When implementing AS variants, most of the above-described procedures remain unchanged. Some of the necessary adaptations are described in the following:

procedure ACSLocalPheromoneUpdate(k, i)
  input k % ant identifier
  input i % counter for construction step
  $h \leftarrow ant[k].tour[i-1]$
  $j \leftarrow ant[k].tour[i]$
  $pheromone[h][j] \leftarrow (1-\xi)pheromone[h][j] + \xi\tau_0$
  $pheromone[j][h] \leftarrow pheromone[h][j]$
  $choice\_info[h][j] \leftarrow pheromone[h][j] \cdot \exp(1/dist[h][j], \beta)$
  $choice\_info[j][h] \leftarrow choice\_info[h][j]$
end-procedure

Figure 3.24
Implementation of the local pheromone update in ACS.

- When depositing pheromone, the solution may be given some weight, as is the case in EAS and AS_{rank}. This can be accomplished by simply adding a weight factor as an additional argument of the procedure DepositPheromone.
- $\mathcal{MM}$AS has to keep track of the pheromone trail limits. The best way to do so is to integrate this into the procedure ASPheromoneUpdate.
- Finally, the search control of some of the AS variants may need minor changes. Examples are occasional pheromone trail reinitializations or the schedule for the frequency of the best-so-far update according to equation (3.20) in $\mathcal{MM}$AS.

Unlike AS variants, the implementation of ACS requires more significant changes, as listed in the following:

- The implementation of the pseudorandom proportional action choice rule [see equation (3.10)] requires the generation of a random number q uniformly distributed in the interval $[0, 1]$ and the application of the procedure ChooseBestNext if $q < q_0$, or of the procedure ASDecisionRule otherwise.
- The local pheromone update [equation (3.12)] can be managed by the procedure ACSLocalPheromoneUpdate (see figure 3.24) that is always invoked immediately after an ant moves to a new city.
- The implementation of the global pheromone trail update [equation (3.11)] is similar to the procedure for the local pheromone update except that pheromone trails are modified only on arcs belonging to the best-so-far tour.
- Note that the integration of the computation of new values for the matrix *choice_info* into the local and the global pheromone trail update procedures avoids having to modify this matrix in any other part of the algorithm except for the initialization.

3.9 Bibliographical Remarks

The Traveling Salesman Problem

The TSP is one of the oldest and most studied combinatorial optimization problems. The first references to the TSP and closely related problems date back to the 19th century (see the overview paper on the history of combinatorial optimization by Schrjiver [2002] and the webpage Solving TSPs accessible at www.math.princeton.edu/tsp/ for more details). The TSP has been studied intensively in both operations research and computer science since the '50s. Therefore, it is not surprising that a large number of different algorithmic techniques were either applied to the TSP or developed because of the challenge posed by this problem. Up to the early '80s these approaches comprised mainly construction heuristics (Clarke & Wright, 1964; Christofides, 1976; Golden & Stewart, 1985; Bentley, 1992), iterative improvement algorithms (Flood, 1956; Croes, 1958; Lin, 1965; Lin & Kernighan, 1973), and exact methods like branch & bound or branch & cut (Dantzig, Fulkerson, & Johnson, 1954; Grötschel, 1981; Padberg & Grötschel, 1985; Grötschel & Holland, 1991; Applegate et al., 1995). An in-depth overview of these early approaches is given in Lawler et al. (1985). Extensive experimental evaluations of construction heuristics and iterative improvement algorithms may be found in Bentley (1992), Reinelt (1994), and Johnson & McGeoch (1997, 2002).

Since the beginning of the '80s, more and more metaheuristics have been tested on the TSP. In fact, the TSP was the first problem to which simulated annealing, one of the first metaheuristic approaches, was applied (Cerný, 1985; Kirkpatrick, Gelatt, & Vecchi, 1983). Following SA, virtually any metaheuristic used the TSP as a test problem. These include tabu search (Knox, 1994; Zachariasen & Dam, 1996), guided local search (Voudouris & Tsang, 1999), evolutionary algorithm (Merz & Freisleben, 1997; Walters, 1998), ACO algorithms (see this chapter and Stützle & Dorigo, 1999b), and iterated local search (ILS) (Baum, 1986; Martin et al., 1991; Johnson & McGeoch, 1997; Applegate et al., 2003).

The state of the art (until 1997) for solving symmetric TSPs with heuristics is summarized in the overview article by Johnson & McGeoch (1997). This article contains a discussion of the relative performance of different metaheuristic approaches to the TSP and concludes that ILS algorithms using fine-tuned implementations of the Lin-Kernighan heuristic (Lin & Kernighan, 1973) are the most successful. The most recent effort in collecting the state of the art for TSP solving by heuristic methods was undertaken by the "8th DIMACS Implementation Challenge on the TSP"; details of this benchmark challenge can be found at www.research.att.com/~dsj/chtsp/ and the

results as of February 2002, including construction heuristics, iterative improvement algorithms, metaheuristics, and more TSP-specific approaches, are summarized in a paper by Johnson & McGeoch (2002). The conclusion of this recent undertaking is that, when running time is not much of a concern, the best-performing algorithms appear to be the tour-merging approach (a TSP-specific heuristic) of Applegate et al. (1999) and the iterated version of Helsgaun's Lin-Kernighan variant (Helsgaun, 2000). In this context, it is interesting to note that the iterated version of Helsgaun's implementation of the Lin-Kernighan heuristic uses a constructive approach (as does ant colony optimization) to generate the initial tours for the local searches, where the best-so-far solution strongly biases the tour construction.

Finally, let us note that the results obtained with exact algorithms for the TSP are quite impressive. As of spring 2002, the largest instance provably solved to optimality comprises 15112 cities. Solving such a large instance required a network of 110 processors and took a total time estimated to be equivalent to 22.6 CPU-years on a 500 MHz, EV6 Alpha processor (more details on optimization algorithms for the TSP, the most recent results, and the source code of these algorithms are available at www.math.princeton.edu/tsp/). Although these results show the enormous progress that has been made by exact methods, they also divert attention from the fact that these results on the TSP are not really representative of the performance of exact algorithms on many other combinatorial optimization problems. There are in fact a large number of problems that become intractable for exact algorithms, even for rather small instances.

ACO Algorithms

The currently available results obtained by ACO algorithms applied to the TSP are not competitive with the above-mentioned approaches. By adding more sophisticated local search algorithms like the implementation of the Lin-Kernighan heuristic available at www.math.princeton.edu/tsp/ or Helsgaun's variant of the Lin-Kernighan heuristic, ACO's computational results on the TSP can certainly be strongly improved, but it is an open question whether the results of the best algorithms available can be reached. Nevertheless, as already stated in the introduction, the main importance of the TSP for the ACO research field is that it is a problem on which the behavior of ACO algorithms can be studied without obscuring the algorithm behavior by many technicalities. In fact, the best-performing variants of ACO algorithms on the TSP often reach world-class performance on many other problems (see chapter 5 for several such applications).

In addition to the ACO algorithms discussed in this chapter, recently a new variant called best-worst Ant System (BWAS) was proposed (Cordón et al., 2000;

Cordón, de Viana, & Herrera, 2002). It introduces three main variations with respect to AS. First, while using, in a way similar to $\mathcal{MM}$AS and ACS, an aggressive update rule in which only the best-so-far ant is allowed to deposit pheromone, it also exploits the worst ant of the current iteration to subtract pheromone on the arcs it does not have in common with the best-so-far solution. Second, BWAS relies strongly on search diversification through the frequent reinitialization of the pheromone trails. Third, as an additional means for diversifying the search, it introduces pheromone mutation, a concept borrowed from evolutionary computation. The influence of these three features was systematically analyzed in Cordón et al. (2002). The currently available results, however, are not fully conclusive, so that it is not possible to judge BWAS's performance with respect to the currently best-performing ACO algorithms for the TSP: $\mathcal{MM}$AS and ACS.

ACO algorithms have also been tested on the asymmetric TSP (Dorigo & Gambardella, 1997b; Stützle & Hoos, 2000; Stützle & Dorigo, 1999b), where the distance between a pair of nodes i, j is dependent on the direction of traversing the arc. ACO algorithms for the symmetric TSP can be extended very easily to the asymmetric case, by taking into account that in general $\tau_{ij} \neq \tau_{ji}$, because the direction in which the arcs are traversed has to be taken into account. Experimental results suggest that ACO algorithms can find optimal solutions to ATSP instances with up to a few hundred nodes. In the ATSP case, at the time the research was done, results competitive with those obtained by other metaheuristics were obtained. However, recent results on algorithmic approaches to the ATSP (see Johnson, Gutin, McGeoch, Yeo, Zhang, & Zverovitch, 2002) suggest that current ACO algorithms do not reach state-of-the-art performance for the ATSP.

It is also worth mentioning that there are at least two ant algorithms not fitting into the ACO metaheuristic framework that have been applied to combinatorial optimization problems. These are *Hybrid Ant System* (HAS) by Gambardella, Taillard, & Dorigo (1999b) and *Fast Ant System* (FANT) by Taillard (1998). HAS does not use pheromone trails to construct solutions but to guide a solution modification process similar to perturbation moves as used in ILS. FANT differs from ACO algorithms mainly in the pheromone management process and in the avoidance of explicit evaporation of pheromones, which are decreased by occasional reinitializations. Both HAS and FANT were applied to the quadratic assignment problem and were found to yield good performance. However, adaptations of both to the TSP resulted in significantly worse performance than, for example, $\mathcal{MM}$AS (Stützle & Linke, 2002).

The combination of ACO algorithms with local search was considered for the first time by Maniezzo et al. (1994) for the application of AS to the quadratic assignment

problem. When applied to the TSP, local search was combined for the first time with ACS by Dorigo & Gambardella (1997b) and with $\mathcal{MM}$AS by Stützle & Hoos (1996). For a detailed experimental investigation of the influence of parameter settings on the final performance of ACO algorithms for the TSP, see those papers.

3.10 Things to Remember

- The TSP was the first combinatorial optimization problem to be attacked by ACO. The first ACO algorithm, called Ant System, achieved good performance on small TSP instances but showed poor scaling behavior to large instances. The main role of AS was that of a "proof-of-concept" that stimulated research on better-performing ACO algorithms as well as applications to different types of problems.
- Nowadays, a large number of different ACO algorithms are available. All of these algorithms include a strong exploitation of the best solutions found during the search and the most successful ones add explicit features to avoid premature stagnation of the search. The main differences between the various AS extensions consist of the techniques used to control the search process. Experimental results show that for the TSP, but also for other problems, these variants achieve a much better performance than AS.
- When applying ACO algorithms to the TSP, the best performance is obtained when the ACO algorithm uses a local optimizer to improve the solutions constructed by the ants. As we will see in chapter 5, this is typical for the application of ACO to $\mathcal{NP}$-hard optimization problems.
- When using local search, it is typically sufficient to apply a small constant number of ants to achieve high performance, and experimental results suggest that in this case the role played by the heuristic information becomes much less important.
- The implementation of ACO algorithms is often rather straightforward, as shown in this chapter via the example of the implementation of AS for the TSP. Nevertheless, care should be taken to make the code as efficient as possible.
- The implementation code for most of the ACO algorithms presented in this chapter is available at www.aco-metaheuristic.org/aco-code/.

3.11 Computer Exercises

Exercise 3.1 In all ACO algorithms for the TSP the amount of pheromone deposited by an ant is proportional to the ant's tour length. Modify the code in such a way

that the amount of pheromone deposited is a constant and run tests with the various ACO algorithms.

For which ACO algorithms would you expect that this change does not influence the performance very strongly? Why?

Exercise 3.2 Use a profiler to identify how much computation time is taken by the different procedures (solution construction, pheromone evaporation, local search, etc.) of the ACO algorithms. Identify the computationally most expensive parts.

Exercise 3.3 There exist some ACO algorithms that were proposed in the literature but that have never been applied to the symmetric TSP. These ACO algorithms include the ANTS algorithm (Maniezzo, 1999) and the hyper-cube framework for ACO (Blum et al., 2001; Blum & Dorigo, 2004). Extend the implementation of the ACO algorithms that is available at www.aco-metaheuristic.org/aco-code/ to include these two ACO algorithms.

Hint: For ANTS care has to be taken that the computation of the lower bounds is as efficient as possible, because this is done at each construction step of each ant.

Exercise 3.4 ACO algorithms have mainly been tested on Euclidean TSP instances available from TSPLIB. Many TSP algorithms are experimentally tested on random distance matrix instances, where each entry in the distance matrix is a random number sampled from some interval. Download a set of such instances from the webpage of the 8th DIMACS Implementation Challenge on the TSP (www.research.att.com/~dsj/chtsp/) and test the ACO algorithms on these types of instances.

Exercise 3.5 The implementations described in this chapter were designed for attacking symmetric TSP problems. Adapt the available code to solve ATSP instances.

Exercise 3.6 Compare the results obtained with the ACO algorithms to those obtained with the approaches described in the review paper on heuristics for the asymmetric TSP by Johnson et al. (2002).

Exercise 3.7 The solution construction procedure used in all ACO algorithms is a randomized form of the nearest-neighbor heuristic, in which at each step the closest, still unvisited, city to the current city is chosen and becomes the current city. However, a large number of other solution construction procedures exist (e.g., see Bentley, 1992; Reinelt, 1994; Johnson & McGeoch, 2002). Promising results have been reported among others for the *savings heuristic*, the *greedy heuristic*, and the *insertion heuristic*.

Adapt the ACO algorithms' code so that these construction heuristics can be used in place of the nearest-neighbor heuristic.

Exercise 3.8 Combine the available ACO algorithms with implementations of the Lin-Kernighan heuristic. You may adapt the publicly available Lin-Kernighan codes of the Concorde distribution (available at www.math.princeton.edu/tsp/concorde.html) or Keld Helsgaun's Lin-Kernighan variant (available at www.dat.ruc.dk/~keld/research/LKH/) and use these to improve the solutions generated by the ants (do not forget to ask the authors of the original code for permission to modify/adapt it).

Exercise 3.9 Extend the available code for the TSP to the sequential ordering problem (see chapter 2, section 2.3.2, for a definition of the problem). For a description of an ACO approach to the SOP, see chapter 5, section 5.1.1.