*Perfection has been attained*
*not when nothing remains to be added*
*but when nothing remains to be taken away.*
—Antoine de Saint-Exupéry, Pilot & Writer

# 2 SLS Methods

Stochastic Local Search (SLS) is a widely used approach to solving hard combinatorial optimisation problems. Underlying most, if not all, specific SLS algorithms are general SLS methods that can be applied to many different problems. In this chapter we present some of the most prominent SLS methods and illustrate their application to hard combinatorial problems, using SAT and TSP as example domains.

The techniques covered here range from simple iterative improvement algorithms to complex SLS methods, such as Ant Colony Optimisation and Evolutionary Algorithms. For each of these SLS methods, we motivate and describe the basic technique and discuss important variants. Furthermore, we identify and discuss important characteristics and features of the individual methods and highlight relationships between them.

## 2.1 Iterative Improvement (Revisited)

In Chapter 1, Section 1.5, we introduced Iterative Improvement as one of the simplest, yet reasonably effective SLS methods. We have pointed out that one of the main limitations of Iterative Improvement is the fact that it can, and often does, get stuck in local minima of the underlying evaluation function. Here, we discuss how using larger neighbourhoods can help to alleviate this problem without rendering the exploration of local neighbourhoods prohibitively expensive.

## Large Neighbourhoods

As pointed out before, the performance of any stochastic local search algorithm depends significantly on the underlying neighbourhood relation and, in particular, on the size of the neighbourhood. Consider the standard $k$-exchange neighbourhoods introduced in Chapter 1, Section 1.5. It is easy to see that for growing $k$, the size of the neighbourhood (i.e., the number of direct neighbours for each given candidate solution), also increases. More precisely, for a $k$-exchange neighbourhood, the size of the neighbourhood is in $O(n^k)$, that is, the neighbourhood size increases exponentially with $k$.

Generally, larger neighbourhoods contain more and potentially better candidate solutions, and hence they typically offer better chances for finding locally improving search steps. They also lead to neighbourhood graphs with smaller diameters, which means that an SLS trajectory can potentially more easily explore different regions of the underlying search space. In a sense, the ideal case would be a neighbourhood relation for which any locally optimal candidate solution is guaranteed to be globally optimal. Neighbourhoods which satisfy this property are called *exact*; unfortunately, in most cases exact neighbourhoods are exponentially large with respect to the size of the given problem instance, and searching an improving neighbouring candidate solution may take exponential time in the worst case. (Efficiently searchable exact neighbourhoods exist in a few cases; for example, the Simplex Algorithm in linear programming is an iterative improvement algorithm that uses a polynomially searchable, exact neighbourhood, and is hence guaranteed to find a globally optimal solution.)

This situation illustrates a general tradeoff: using larger neighbourhoods might increase the chance of finding (high-quality) solutions of a given problem in fewer local search steps when using SLS algorithms in general and Iterative Improvement in particular; but at the same time, the time complexity for determining improving search steps is much higher in larger neighbourhoods. Typically, the time complexity of an individual local search step needs to be polynomial w.r.t. the size of the given problem instance. However, depending on problem size, even quadratic or cubic time per search step might already be prohibitively high if the instance is very large.

## Neighbourhood Pruning

Given the tradeoff between the benefits of using large neighbourhoods and the associated time complexity of performing search steps, one attractive idea for improving the performance of Iterative Improvement and other SLS algorithms is to use large neighbourhoods but to reduce their size by never examining neighbours

that are unlikely to (or that provably cannot) yield any improvements in evaluation function value. While in many cases, the use of large neighbourhoods is only practically feasible in combination with such pruning, the same pruning techniques can be applied to relatively small neighbourhoods, where they can lead to substantial improvements in SLS performance.

For the TSP, one such pruning technique that has been shown to be useful in practice is the use of *candidate lists*, which for each vertex in the given graph contain a limited number of their closest direct neighbours, ordered according to increasing edge weight. The search steps performed by an SLS algorithm are then limited to consider only edges connecting a vertex $i$ to one of the vertices in $i$'s candidate list. The use of such candidate lists is based on the intuition that high-quality solutions will be likely to include short edges between neighbouring vertices (cf. Figure 1.1, page 23). In the case of the TSP, pruning techniques have shown significant impact on local search performance not only for large neighbourhoods, but also for rather small neighbourhoods, such as the standard 2-exchange neighbourhood.

Other neighbourhood pruning techniques identify neighbours that provably cannot lead to improvements in the evaluation function based on insights into the properties of a given problem. An example for such a pruning technique is described by Nowicki and Smutnicki [1996a] in their tabu search approach to the Job Shop Problem, which will be described in Chapter 9.

## Best Improvement vs First Improvement

Another method for speeding up the local search is to select the next search step more efficiently. In the context of iterative improvement algorithms, the search step selection mechanism that implements the step function from Definition 1.10 (page 38*f.*) is also called *pivoting rule* [Yannakakis, 1990]; the most widely used pivoting rules are the so-called best improvement and first improvement strategies described in the following.

*Iterative Best Improvement* is based on the idea of randomly selecting in each search step one of the neighbouring candidate solutions that achieve a maximal improvement in the evaluation function. Formally, the corresponding step function can be defined as follows: given a search position $s$, let $g^* := \min\{g(s') \mid s' \in N(s)\}$ be the best evaluation function value in the neighbourhood of $s$. Then $I^*(s) := \{s' \in N(s) \mid g(s') = g^*\}$ is the set of maximally improving neighbours of $s$, and we define $step(s)(s') := 1/\#I^*(s)$ if $s' \in I^*(s)$, 0 otherwise. Best Improvement is also called *greedy hill-climbing* or *discrete gradient descent*. Note that Best Improvement requires a complete evaluation of all neighbours in each search step.

The *First Improvement* neighbour selection strategy tries to avoid the time complexity of evaluating all neighbours by performing the first improving step encountered during the inspection of the neighbourhood. Formally, Iterative First Improvement is best defined by means of a step procedure rather than a step function. At each search position $s$, the First Improvement step procedure evaluates the neighbouring candidate solutions $s' \in N(s)$ in a particular fixed order, and the first $s'$ for which $g(s') < g(s)$, that is, the first improving neighbour encountered, is selected. Obviously, the order in which the neighbours are evaluated can have a significant influence on the efficiency of this strategy. Instead of using a fixed ordering for evaluating the neighbours of a given search position, random orderings can also be used. For fixed evaluation orderings, repeated runs of Iterative First Improvement starting from the same initial solution will end in the same local optimum, while by using random orderings, many different local optima can be reached. In this sense, random-order First Improvement inherently leads to a certain diversification of the search process. The following example illustrates the variability of the candidate solutions reached by random-order First Improvement.

### EXAMPLE 2.1 Random-Order First Improvement for the TSP

In this example, we empirically study a random-order first improvement algorithm for the TSP that is based on the 2-exchange neighbourhood. This algorithm always starts from the same initial tour, which visits the vertices of the given graph in their canonical order (i.e., in the order $v_1, v_2, \ldots, v_n, v_1$). Furthermore, when initialising the search, a random permutation of the integers from 1 to $n$ is generated, which determines the order in which the neighbhourhood is scanned in each search step. (This permutation remains unchanged throughout the search process.) As usual for simple iterative improvement methods, the search is terminated when a local minimum of the given evaluation function (here: weight of the candidate tour) is encountered.

This algorithm was run 1 000 times on `pcb3038`, a TSP instance with 3 038 vertices available from the TSPLIB benchmark library. For each of these runs, the length of the final, locally optimal tour (i.e., the weight of the corresponding path in the graph) was recorded. Figure 2.1 shows the cumulative distribution of the percentage deviations of these solution quality values from the known optimal solution. (The cumulative distribution function specifies for each relative solution quality value $q$ on the $x$-axis the relative frequency with which a solution quality smaller or equal to $q$ is obtained.) Clearly, there is a large degree of variation in the qualities of the 1 000 tours produced by our random-order iterative first improvement algorithm. The average tour length is 8.6% above the known optimum, while the 0.05-

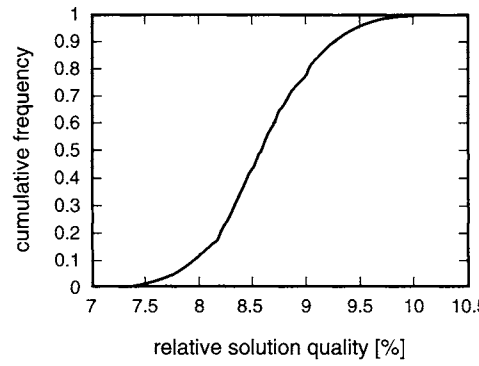


*Figure* **2.1** Cumulative distribution of the solution quality returned by a random-order first improvement 2-exchange algorithm for the TSP on TSPLIB instance pcb3038, based on 1 000 runs of the algorithm.

and 0.95-quantiles of this solution quality distribution can be determined as 7.75% and 9.45% above the optimum.

Based on the shape of this empirical distribution, it can be conjectured that the solution quality data follow a normal distribution. This hypothesis can be tested using the Shapiro-Wilk test [Shapiro and Wilk, 1965], a statistical goodness-of-fit test that specifically checks whether given sample data are normally distributed. In this example, the test accepts the hypothesis that the solution quality data follow a normal distribution with mean 8.6 and standard deviation 0.51 at a p-value of 0.2836. (We refer to Chapter 4 for more details on statistical tests). Normally distributed solution qualities occur rather frequently in the context of SLS algorithms for hard combinatorial problems, such as the TSP.

As for large neighbourhoods, in the context of pivoting rules there is a trade-off between the number of search steps required for finding a local optimum and the computation time for each search step. Search steps in first improvement algorithms can often be computed more efficiently than in best improvement algorithms, since in the former case, typically only a small part of the local neighbourhood is evaluated, especially as long as there are multiple improving search steps from the current candidate solution. However, the improvement obtained by each step of First Improvement is typically smaller than for Best Improvement and therefore, more search steps have to be performed in order to reach a local optimum. Additionally, Best Improvement benefits more than First Improvement from the use of caching and updating mechanisms for evaluating neighbours efficiently.

*Remark:* Besides First Improvement and Best Improvement, iterative improvement algorithms can use a variety of other pivoting rules. One example is *Random Improvement*, which randomly selects a candidate solution from the set $I(s) := \{s' \in N(s) \mid g(s') < g(s)\}$; this selection strategy can be implemented as First Improvement where a new random evaluation ordering is used in each search step. Another example is the *least improvement* rule, which selects an element from $I(s)$ that minimally improves the current candidate solution.

## Variable Neighbourhood Descent

Another way to benefit from the advantages of large neighbourhoods without incurring a high time complexity of the search steps is based on the idea of using standard, small neighbourhoods until a local optimum is encountered, at which point the search process switches to a different (typically larger) neighbourhood, which might allow further search progress. This approach is based on the fact that the notion of a local optimum is defined relative to a neighbourhood relation, such that if a candidate solution $s$ is locally optimal w.r.t. a neighbourhood relation $N_1$ it need not be a local optimum for a different neighbourhood relation $N_2$. The general idea of changing the neighbourhood during the search has been systematised by the *Variable Neighbourhood Search (VNS)* framework [Mladenović and Hansen, 1997; Hansen and Mladenović, 1999].

VNS comprises a number of algorithmic approaches including *Variable Neighborhood Descent (VND)*, an iterative improvement algorithm that realises the general idea behind VNS in a very straightforward way. In VND, $k$ neighbourhood relations $N_1, N_2, \ldots, N_k$ are used, which are typically ordered according to increasing size. The algorithm starts with neighbourhood $N_1$ and performs iterative improvement steps until a local optimum is reached. Whenever no further improving step is found for a neighbourhood $N_i$ and $i + 1 \leq k$, VND continues the search in neighbourhood $N_{i+1}$; if an improvement is obtained in $N_i$, the search process switches back to $N_1$, from where the search is continued as previously described. An algorithm outline for VND is shown in Figure 2.2. In general, there are variants of this basic VND method that switch between neighbourhoods in different ways. It has been shown that Variable Neighbourhood Descent can considerably improve the performance of iterative improvement algorithms both w.r.t. to the solution quality of the local optima reached, as well as w.r.t. the time required for finding (high-quality) solutions compared to using standard Iterative Improvement in large neighbourhoods [Hansen and Mladenović, 1999].

It may be noted that apart from VND, there are several other variants of the general idea underlying Variable Neighbourhood Search. Some of these — in

```
procedure VND(π', N_1, N_2, ..., N_k)
    input: problem instance π' ∈ Π', neighbourhood relations N_1, N_2, ..., N_k
    output: solution ŝ ∈ S'(π') or ∅
    s := init(π');
    ŝ := s;
    i := 1;
    repeat
        find best candidate solution s' in neighbourhood N_i(s);
        if g(s') < g(s) then
            s := s';
            if f(s) < f(ŝ) then
                ŝ := s;
            end
            i := 1;
        else
            i := i + 1;
        end
    until i > k
    if ŝ ∈ S' then
        return ŝ
    else
        return ∅
    end
end VND
```

*Figure* 2.2 Algorithm outline for Variable Neighbourhood Descent for optimisation problems; note that the evaluation function $g$ is used for checking whether the search has reached a local minimum, while the objective function of the given problem instance, $f$, is used for detecting improvements in the incumbent candidate solution. For further details, see text.

particular, Basic VNS and Skewed VNS [Hansen and Mladenović, 2002] — are conceptually closely related to Iterated Local Search, a hybrid SLS method that will be discussed later in this chapter (cf. Section 2.3).

## Variable Depth Search

A different approach to selecting search steps from large neighbourhoods reasonably efficiently is to compose more complex steps from a number of steps in small, simple neighbourhoods. This idea is the basis of *Variable Depth Search*

*(VDS)*, an SLS method introduced first by Kernighan and Lin for the Graph Partitioning Problem [1970] and the TSP [1973]. Generally, VDS can be seen as an iterative improvement method in which the local search steps are variable length sequences of simpler search steps in a small neighbourhood. Constraints on the feasible sequences of simple steps help to keep the time complexity of selecting complex steps reasonably low. (For an algorithm outline of VDS, see Figure 2.3.)

As an example for a VDS algorithm, consider the *Lin-Kernighan (LK) Algorithm for the TSP*. The LK algorithm performs iterative improvement using complex search steps each of which corresponds to a sequence of 2-exchange steps. The mechanism underlying the construction of a complex step can be

```
procedure VDS(π')
    input: problem instance π' ∈ Π'
    output: solution ŝ ∈ S'(π') or ∅

    s := init(π');
    ŝ := s;
    while not terminate (π', s) do
        t := s;
        t̂ := t;
        repeat
            t := selectBestFeasibleNeighbour(π', t);
            if f(t) < f(t̂) then
                t̂ := t;
            end
        until terminateConstruction (π', t, t̂);
        s := t̂;
        if f(s) < f(ŝ) then
            ŝ := s;
        end
    end
    if ŝ ∈ S' then
        return ŝ
    else
        return ∅
    end
end VDS
```

***Figure 2.3*** Algorithm outline for Variable Depth Search for optimisation problems; for details, see text.

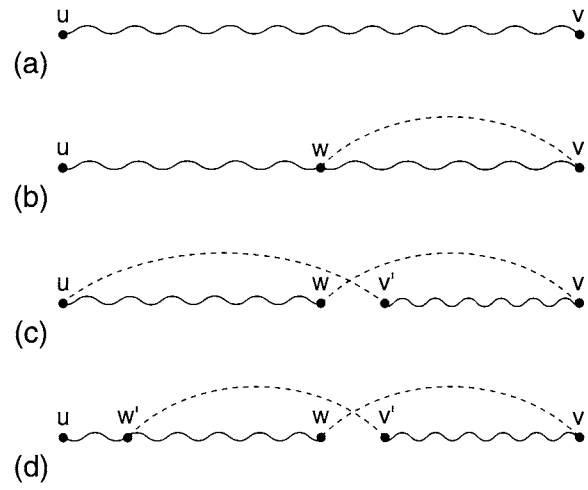


*Figure* **2.4** Schematic view of a Lin–Kernighan exchange step: (a) shows a Hamiltonian path, (b) a possible $\delta$-path, (c) the next Hamiltonian path (which is closed by introducing the left dashed edge) and (d) indicates a next possible $\delta$-path.

understood best by considering a sequence of Hamiltonian paths, that is, paths that contain each vertex in the given graph $G$ exactly once. Figure 2.4a shows an example in which a Hamiltonian path between nodes $u$ and $v$ is obtained from a valid round trip by removing the edge $(u, v)$. Let us fix one of the endpoints in this path, say $u$; the other endpoint is kept variable. We can now introduce a cycle into this Hamiltonian path by adding an edge $(v, w)$ (see Figure 2.4b). The resulting subgraph can also be viewed as a spanning tree of $G$ with one additional edge; it is called a *$\delta$-path*. The cycle in this $\delta$-path can be broken by removing a uniquely defined edge $(w, v')$ incident to $w$, such that the result is a new Hamiltonian path that can be extended to a Hamiltonian cycle (and hence a candidate solution for the TSP) by adding an edge between $v'$ and the fixed endpoint $u$ (this is the dashed edge $(v', u)$ in Figure 2.4c). Alternatively, a different edge can be added, leading to a new $\delta$-path as indicated in Figure 2.4d.

Based on this fundamental mechanism, the LK algorithm computes complex search steps as follows: Starting with the current candidate solution (a Hamiltonian cycle) $s$, a $\delta$-path $p$ of minimal path weight is determined by replacing one edge as described above. If the Hamiltonian cycle $t$ obtained from $p$ by adding a (uniquely defined) edge has weight smaller than $s$, then $t$ (and its weight) is memorised. The same operation is now performed with $p$ as a starting point, and iterated until no $\delta$-path can be obtained with weight smaller than that of the best Hamiltonian cycle found so far. Finally, the minimal weight Hamiltonian cycle $s'$ found in this iterative process provides the end point of a complex search step.

Note that this can be interpreted as a sequence of 1-exchange steps that alternate between $\delta$-paths and Hamiltonian cycles.

In order to limit the time complexity of constructing complex search steps, VDS algorithms use two types of restrictions, *cost restrictions* and *tabu restrictions*, on the selection of the constituting simple search steps. In the case of the LK algorithm, any edge that has been added cannot be removed and any edge that has been removed cannot be introduced any longer. This tabu restriction has the effect that a candidate sequence for a complex step is never longer than $n$, the number of vertices in the given graph. The original LK algorithm also uses a number of additional mechanisms, including a limited form of backtracking, for controlling the generation of complex search steps; as a consequence, the final tour returned by the algorithm is guaranteed to be optimal w.r.t. the standard 3-exchange neighbourhood. Along with other details of the LK algorithm, these mechanisms are described in Chapter 8, Section 8.2.

VDS algorithms have been used with considerable success for solving a number of problems other than the TSP, including the Graph Partitioning Problem [Kernighan and Lin, 1970], the Unconstrained Binary Quadratic Programming Problem [Merz and Freisleben, 2002] and the Generalised Assignment Problem [Yagiura et al., 1999].

## Dynasearch

Like VDS, *Dynasearch* is an iterative improvement algorithm that tries to build a complex search step based on a combination of simple search steps [Potts and van de Velde, 1995; Congram et al., 2002; Congram, 2000]. However, differently from VDS, Dynasearch requires that the individual search steps that compose a complex step are mutually independent. Here, independence means that the individual search steps do not interfere with each other with respect to their effect on the evaluation function value and the feasibility of candidate solutions. In particular, any dynasearch step from a feasible candidate solution $s$ is guaranteed to result in another feasible candidate solution, and the overall improvement in evaluation function value achieved by the dynasearch step can be obtained by summing up the effects of applying each individual search step to $s$.

As an example for this independence condition, consider a TSP instance and a specific Hamiltonian cycle $t = (u_1, \ldots, u_n, u_{n+1})$, where $u_{n+1} = u_1$. A 2-exchange step involves removing two edges $(u_i, u_{i+1})$ and $(u_j, u_{j+1})$ from $t$ (without loss of generality, we assume that $1 \leq i$ and $i + 1 < j \leq n$). Two 2-exchange steps that remove edges $(u_i, u_{i+1})$, $(u_j, u_{j+1})$ and $(u_k, u_{k+1})$, $(u_l, u_{l+1})$, respectively, are independent if, and only if, either $j < k$ or $l < i$. An example of a pair of independent 2-exchange steps is given in Figure 2.5. Any set of independent steps can be executed in parallel, leading to an overall

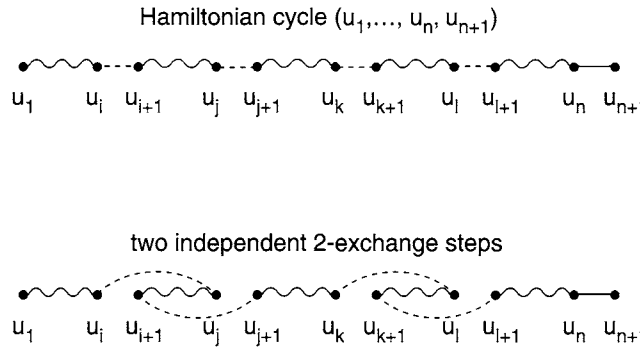


*Figure* **2.5** Example of a pair of independent 2-exchange steps that can potentially form a dynasearch step.

improvement equal to the sum of the improvements achieved by the simple component steps and to a feasible candidate solution, here: another Hamiltonian cycle in the given graph.

The neighbourhood explored by Dynasearch consists of the set of all possible complex search steps; it can be shown that in general this neighourhood can be of exponential size w.r.t. to the size of the underlying simple neighbourhoods. However, through the use of a dynamic programming algorithm it is possible to find the best possible complex search step in polynomial time. (Roughly speaking, the key principle of Dynamic Programming is to iteratively solve a sequence of increasingly large subproblems that leads to a solution of the given problem, exploiting independence assumptions such as the one described previously [Bertsekas, 1995].) Only in the worst case, one complex dynasearch step consists of a single simple step. Although Dynasearch is a very recent local search technique, it has already shown very promising performance on several combinatorial optimisation problems, such as the Single Machine Total Weighted Tardiness Problem [Congram et al., 2002] (we discuss a dynasearch algorithm for this well-known $\mathcal{NP}$-hard scheduling problem in Chapter 9, Section 9.2), the TSP and the Linear Ordering Problem [Congram, 2000].

## 2.2 'Simple' SLS Methods

In the previous section, we introduced several ways of extending simple exchange neighbourhoods that can significantly enhance the performance of Iterative Improvement and prevent this algorithm from getting stuck in very low-quality local optima. Another way of addressing the same problem is to modify the step function, such that for a fixed and fairly simple neighbourhood, the search process can

perform worsening steps which help it to escape from local optima. As mentioned in Chapter 1, Section 1.5, the simplest technique for achieving this is to use randomised variants of Iterative Improvement or a restart strategy that re-initialises the search process whenever it gets stuck in a local optimum. In this section, we will discuss a number of different methods that achieve the same effect often in a more efficient and robust way. These are simple in the sense that they essentially perform only one type of search step, while later in this chapter, we will discuss hybrid SLS algorithms, which combine various different types of search steps, as well as population-based SLS methods.

## Randomised Iterative Improvement

One of the simplest ways of extending iterative improvement algorithms such that worsening steps can be performed is to sometimes select a neighbour at random, rather than an improving neighbour, within the individual search steps. Such uninformed random walk steps may be performed with a fixed frequency, such that the alternation between improvement steps and random walk steps follows a deterministic pattern. Yet, depending on the improvement strategy used, this may easily lead to a situation in which the effect of the random walk steps are immediately undone in subsequent improvement steps, leading to cycling behaviour and preventing an escape from given local optima. Therefore, it is preferable to probabilistically determine in each search step whether to apply an improvement step or a random walk step. Typically, this is done by introducing a parameter $wp \in [0, 1]$, called *walk probability* or *noise parameter*, that corresponds to the probability of performing a random walk step instead of an improvement step.

The resulting algorithm is called *Randomised Iterative Improvement (RII)*. Like Iterative Improvement, it typically uses a random initialisation of the search, as described in Chapter 1, Section 1.5. Its step function can be written as $step_{RII}(s)(s') := wp \cdot step_{URW}(s)(s') + (1 - wp) \cdot step_{II}(s)(s')$, where $step_{URW}(s)(s')$ is the step function for uninformed random walk and $step_{II}(s)(s')$ is a variant of the step function for the Iterative Improvement Algorithm (see Section 1.5) that differs only in that a minimally worsening neighbour is selected if the set $I(s)$ of strictly improving neighbours is empty. As shown in Figure 2.6, the RII step function is typically implemented as a two level choice, where first a probabilistic decision is made on which of the two types of search steps is to be applied, and then the corresponding search step is performed. Obviously, there is no need to terminate this SLS algorithm as soon as a local optimum is encountered. Instead, the termination predicate can be realised in various ways. One possibility is to stop the search after a limit on the CPU time or the number of search steps has been reached; alternatively, the search may be terminated when a given number of search steps has been performed without achieving any improvement.

```
procedure step-RII(π, s, wp)
    input: problem instance π, candidate solution s, walk probability wp
    output: candidate solution s'

    u := random([0, 1]);
    if (u ≤ wp) then
        s' := step_URW(π, s);
    else
        s' := step_II(π, s);
    end
    return s'
end step-RII
```

*Figure 2.6* Standard implementation of the step function for Randomised Iterative Improvement; *random*([0, 1]) returns a random number between zero and one using a uniform probability distribution.

A beneficial consequence of using a probabilistic decision on the type of local search performed in each step is the fact that arbitrarily long sequences of random walk steps (or improvement steps, respectively) can occur, where the probability of performing $r$ consecutive random walk steps is $wp^r$. Hence, there is always a chance to escape even from a local optimum that has a large 'basin of attraction' in the sense that many worsening steps may be required to ensure that subsequent improvement steps have a chance of leading into different local optima. In fact, for RII it can be proven that, when the search process is run long enough, eventually a (optimal) solution to any given problem instance is found with arbitrarily high probability. (More details on this proof can be found in the in-depth section on page 155*ff.*)

EXAMPLE 2.2 **Randomised Iterative Improvement for SAT**

RII can be very easily applied to SAT by combining the uninformed random walk algorithm presented in Example 1.3 (page 41*f.*) and an iterative improvement algorithm like that of Example 1.4 (page 47*f.*), using the same search space, solution set, neighbourhood relation and initialisation function as defined there. The only difference is that here, we will apply a best improvement local search algorithm instead of the simple descent method from Example 1.4: In each step, the best improvement algorithm flips a variable that leads to a maximal increase in the evaluation function. Note that such a best improvement algorithm need not terminate at a local optimum, because in this situation the maximally improving variable flip is a perfectly valid

worsening step (more precisely: a least worsening step). The step function for RII is composed of the two step functions for this greedy improvement algorithm and for uninformed random walk as described previously: With probability $wp$, a random neighbouring solution is returned, otherwise with probability $1 - wp$, a best improvement step is applied. We call the resulting algorithm GUWSAT.

Interestingly, a slight variation of the GUWSAT algorithm for SAT from Example 2.2, called GSAT with Random Walk (GWSAT), has been proven rather successful (see also Chapter 6, page 269*f.*). The only difference between GUWSAT and GWSAT is in the random walk step. Instead of uninformed random walk steps, GWSAT uses 'informed' random walk steps by restricting the random neighbour selection to variables occurring in currently unsatisfied clauses; among these variables, one is chosen according to a uniform distribution. When GWSAT was first proposed, it was among the best performing SLS algorithms for SAT. Yet, apart from this success, Randomised Iterative Improvement is rather rarely applied. This might be partly due to the fact that it is such a simple extension of Iterative Improvement, and more complex SLS algorithms often achieve better performance. Nevertheless, RII certainly deserves attention as a simple and generic extension of Iterative Improvement that can be generalised easily to more complex SLS methods.

### Probabilistic Iterative Improvement

An interesting alternative to the mechanism for allowing worsening search steps underlying Randomised Iterative Improvement is based on the idea that the probability of accepting a worsening step should depend on the respective deterioration in evaluation function value such that the worse a step is, the less likely it would be performed. This idea leads to a family of SLS algorithms called *Probabilistic Iterative Improvement (PII)*, which is closely related to Simulated Annealing, a widely used SLS method we discuss directly after PII. In each search step, PII selects a neighbour of the current candidate solution according to a given function $p(g, s)$, which determines a probability distribution over neighbouring candidate solutions of $s$ based on their respective evaluation function values. Formally, the corresponding step function can be written as $step(s)(s') := p(g, s)$.

Obviously, the choice of the function $p(g, s)$ is of crucial importance to the behaviour and performance of PII. Note that both Iterative Improvement, as defined in Chapter 1, Section 1.5, and Randomised Iterative Improvement can

be seen as special cases of PII that are obtained for particular choices of $p(g, s)$. Generally, PII algorithms for which $p(g, s)$ assigns positive probability to all neighbours of $s$ have properties similar to RII, in that arbitrarily long sequences of worsening moves can be performed and (optimal) solutions can be found with arbitrarily high probability as run-time approaches infinity.

### EXAMPLE 2.3 PII / Constant Temperature Annealing for the TSP

The following, simple application of PII to the TSP illustrates the underlying approach and will also serve as a convenient basis for introducing the more general SLS method of Simulated Annealing. Given a TSP instance represented by a complete, edge-weighted graph $G$, we use the set of all vertex permutations as search space, $S$, and the same set as our set of feasible candidate solutions, $S'$. (This simply means that we consider each Hamiltonian cycle in $G$ as a valid solution.) As the neighbourhood relation, $N$, we use a reflexive variant of the 2-exchange neighbourhood defined in Chapter 1, Section 1.5, which for each candidate solution $s$ contains $s$ itself as well as all Hamiltonian cycles that can be obtained by replacing two edges in $s$.

The search process uses a simple randomised initialisation function that picks a Hamiltonian cycle uniformly at random from $S$. The step function is implemented as a two-stage process, in which first a neighbour $s' \in N(s)$ is selected uniformly at random, which is then accepted according to the following probability function:

$$p_{accept}(T, s, s') := \begin{cases} 1 & \text{if } f(s') \leq f(s) \\ \exp\left(\frac{f(s)-f(s')}{T}\right) & \text{otherwise} \end{cases} \tag{2.1}$$

This acceptance criterion is known as the *Metropolis condition*. The parameter $T$, which is also called *temperature*, determines how likely it is to perform worsening search steps: at low temperature values, the probability of accepting a worsening search step is low, while at high temperature values, the algorithm accepts even drastically worsening steps with a relatively high probability. As for RII, various termination predicates can be used for determining when to end the search process.

This algorithm corresponds to a simulated annealing algorithm in which the temperature is being kept constant at $T$. In fact, there exists some evidence suggesting that compared to more general simulated annealing approaches, this algorithm performs quite well, but in general, the determination of a good value for $T$ may be difficult [Fielding, 2000].

## Simulated Annealing

Considering the example PII algorithm for the TSP, in which a temperature parameter $T$ controls the probability of accepting worsening search steps, one rather obvious generalisation is to allow $T$ to vary over the course of the search process. Conceptually, this leads to a family of SLS algorithms known as *Simulated Annealing (SA)*, which was proposed independently by Kirkpatrick, Gelatt and Vecchi [1983], and Cerný [1985]. SA was originally motivated by the annealing of solids, a physical process in which a solid is melted and then cooled down slowly in order to obtain perfect crystal structures, which can be modelled as a state of minimum energy (also called ground state). To avoid defects (i.e., irregularities) in the crystal, which correspond to meta-stable states in the model, the cooling needs to be done very slowly.

The idea underlying SA is to solve combinatorial optimisation problems by a process analogous to the physical annealing process. In this analogy, the candidate solutions of the given problem instance correspond to the states of the physical system, the evaluation function models the thermodynamic energy of the solid, and the globally optimal solutions correspond to the ground states of the physical system.

Like PII, Simulated Annealing typically starts from a random initial solution. It then performs the same general type of PII steps as defined in Example 2.3, where in each step first a neighbour $s'$ of $s$ is randomly chosen (*proposal mechanism*), and then an acceptance criterion parameterised by the temperature parameter $T$ is used to decide whether the search accepts $s'$ or whether it stays at $s$ (see Figure 2.7). One standard choice for this acceptance criterion is a probabilistic choice according to the Metropolis condition (see Equation 2.1, page 75), which was also used in an early article on the simulation of the physical annealing process [Metropolis et al., 1953], where the parameter $T$ corresponded to the actual

```
procedure step-SA(π, s, T)
    input: problem instance π, candidate solution s, temperature T
    output: candidate solution s''

    s' := proposal(π, s);
    s'' := accept(π, s, s', T);
    return s''
end step-SA
```

**Figure 2.7** Standard step function for Simulated Annealing; *proposal* randomly selects a neighbour of $s$, *accept* chooses probabilistically between $s$ and $s'$, dependent on temperature $T$.

physical temperature. Throughout the search process, the temperature is adjusted according to a given *annealing schedule* (often also called *cooling schedule*).

Formally, an annealing schedule is a function that for each run-time $t$ (typically measured in terms of the number of search steps since initialisation) determines a temperature value $T(t)$. Annealing schedules are commonly specified by an initial temperature $T_0$, a temperature update scheme, a number of search steps to be performed at each temperature and a termination condition.

In many cases, the initial temperature $T_0$ is determined based on properties of the given problem instances such as the estimated cost difference between neighbouring candidate solutions [Johnson et al., 1989; van Laarhoven and Arts, 1987]. Simple geometric cooling schedules in which temperature is updated as $T := \alpha \cdot T$ have been shown to be quite efficient in many cases [Kirkpatrick et al., 1983; Johnson et al., 1989]. The number of steps performed at each temperature setting is often chosen as a multiple of the neighbourhood size.

Simulated Annealing can use a variety of termination predicates; a specific termination condition often used for SA is based on the *acceptance ratio*, that is, the ratio of proposed steps to accepted steps. In this case, the search process is terminated when the acceptance ratio falls below a certain threshold or when no improving candidate solution has been found for a given number of search steps.

EXAMPLE 2.4 **Simulated Annealing for the TSP**

The PII algorithm for the TSP specified in Example 2.3 (page 75) can be easily extended into a Simulated Annealing algorithm (see also Johnson and McGeoch [1997]). The search space, solution set and neighbourhood relation are defined as in Example 2.3. We also use the same initialisation and step functions, where $propose(\pi, s)$ randomly selects a neighbour of $s$, and $accept(\pi, s, s', T)$ probabilistically accepts $s'$ depending on $T$, using the Metropolis condition. The temperature $T$ is initialised such that only 3% of the proposed steps are not accepted, and updated according to a geometric cooling schedule with $\alpha = 0.95$; for each temperature value, $n \cdot (n-1)$ search steps are performed, where $n$ is the size (i.e., number of vertices) of the given problem instance. The search is terminated when for five consecutive temperature values no improvement of the evaluation function has been obtained, and the acceptance rate of new solutions has fallen below 2%.

Compared to standard iterative improvement algorithms, including 3-opt local search (an iterative improvement method based on the 3-exchange neighbourhood on edges) and the Lin-Kernighan Algorithm, the SA algorithm presented in Example 2.4 performs rather poorly. By using additional techniques,

including neighbourhood pruning (cf. Section 2.1), greedy initialisation, low temperature starts and look-up tables for the acceptance probabilities, significantly improved results, which are competitive with those obtained by the Lin-Kernighan Algorithm, can be obtained. Greedy initialisation methods, such as starting with a nearest neighbour tour, help SA to find high-quality candidate solutions more rapidly. To avoid that the beneficial effect of a good initial candidate solution is destroyed by accepting too many worsening moves, the initial temperature is set to a low value. The use of look-up tables deserves particular attention. Obviously, calculating the exponential function in Equation 2.1 (page 75) is computationally expensive compared to the evaluation of one neighbouring solution obtained by one 2-exchange step. By using a precomputed table of values of the function $\exp(\Delta/T)$ for a range of argument values $\Delta/T$ and by looking up the accetance probabilities $\exp((f(s) - f(s'))/T)$ from that table, a very significant speedup (in our example about 40%) can be achieved [Johnson and McGeoch, 1997].

A feature of Simulated Annealing that is often noted as particularly appealing is the fact that under certain conditions the convergence of the algorithm, in the sense that any arbitrarily long trajectory is guaranteed to end in an optimal solution, can be proven [Geman and Geman, 1984; Hajek, 1988; Lundy and Mees, 1986; Romeo and Sangiovanni-Vincentelli, 1991]. However, the practical usefulness of these results is very limited, since they require an extremely slow cooling that is typically not feasible in practice.

## Tabu Search

A fundamentally different approach for escaping from local minima is to use aspects of the search history rather than random or probabilistic techniques for accepting worsening search steps. *Tabu Search (TS)* is a general SLS method that systematically utilises memory for guiding the search process [Glover, 1986; 1989; 1990; Hansen and Jaumard, 1990]. The simplest and most widely applied version of TS, which is also called *Simple Tabu Search*, consists of an iterative improvement algorithm enhanced with a form of short-term memory that enables it to escape from local optima.

Tabu Search typically uses a best improvement strategy to select the best neighbour of the current candidate solution in each search step, which in a local optimum can lead to a worsening or plateau step (*plateau steps* are local search steps which do not lead to a change of the evaluation function value). To prevent the local search to immediately return to a previously visited candidate solution and to avoid cycling, TS forbids steps to recently visited search positions. This can be implemented by explicitly memorising previously visited candidate solutions and ruling out any step that would lead back to those. More commonly,

```
procedure step-TS(π, s, tt)
    input: problem instance π, candidate solution s, tabu tenure tt
    output: candidate solution s'

    N' := admissibleNeighbours(π, s, tt);
    s' := selectBest(N');
    return s'
end step-TS
```

*Figure* **2.8** Standard step function for Tabu Search; $admissibleNeighbours(\pi, s, tt)$ returns the set of admissible neighbours of $s$ given the tabu tenure $tt$, $selectBest(N')$ randomly chooses an element of $N'$ with maximal evaluation function value.

reversing recent search steps is prevented by forbidding the re-introduction of solution components (such as edges in case of the TSP) which have just been removed from the current candidate solution. A parameter $tt$, called *tabu tenure*, determines the duration (in search steps) for which these restrictions apply. Forbidding possible moves using a tabu mechanism has the same effect as dynamically restricting the neighbourhood $N(s)$ of the current candidate solution $s$ to a subset $N' \subset N(s)$ of *admissible neighbours*. Thus, Tabu Search can also be viewed as a dynamic neighbourhood search technique [Hertz et al., 1997].

This tabu mechanism can also forbid search steps leading to attractive, unvisited candidate solutions. Therefore, many tabu search algorithms make use of a so-called *aspiration criterion*, which specifies conditions under which the tabu status of candidate solutions or solution components is overridden. One of the most commonly used aspiration criteria overrides the tabu status of steps that lead to an improvement in the incumbent candidate solution.

Figure 2.8 shows the step function that forms the core of Tabu Search. It uses a function *admissibleNeighbours* to determine the neighbours of the current candidate solution that are not tabu or are tabu but satisfy the aspiration criterion. In a second stage, a maximally improving step is randomly selected from this set of admissible neighbours.

EXAMPLE 2.5 **Tabu Search for SAT**

Using the same definition for the search space, solution set and neighbourhood relation as in Example 1.3 (page 41*f.*), and the same evaluation function as in Example 1.4 (page 47*f.*), Tabu Search can be applied to SAT in a straightforward way. The search starts with a randomly chosen variable assignment. Each search step corresponds to a single variable flip that is selected according to the associated change in the number of unsatisfied clauses and its tabu status. More precisely, in each search step, all variables are considered

admissible that either have not been flipped during the least $tt$ steps, or that, when flipped, lead to a lower number of unsatisfied clauses than the best assignment found so far (this latter condition defines the aspiration criterion). From the set of admissible variables, a variable that, when flipped, yields a maximal decrease (or, equivalently, a minimal increase) in the number of unsatisfied clauses is selected uniformly at random. The algorithm terminates unsuccessfully if after a specified number of flips no model of the given formula has been found.

This algorithm is known as *GSAT/Tabu*; it has been shown empirically to achieve very good performance on a broad range of SAT problems (see also Chapter 6). When implementing GSAT/Tabu, it is crucial to keep the time complexity of the individual search steps minimal, which can be achieved by using special data structures and a dynamic caching and incremental updating technique for the evaluation function (this will be discussed in more detail in Chapter 6, in the in-depth section on page 271*ff.*; Chapter 6 also provides a detailed overview of state-of-the-art SLS algorithms for SAT). It is also very important to determine the tabu status of the propositional variables efficiently. This is done by storing with each variable $x$ the search step number $it_x$ when it was flipped last and comparing the difference between the current iteration number $it$ and $it_x$ to the tabu tenure parameter, $tt$: variable $x$ is tabu if, and only if, $it - it_x$ is smaller than $tt$.

In general, the performance of Tabu Search crucially depends on the setting of the tabu tenure parameter, $tt$. If $tt$ is chosen too small, search stagnation may occur; if it is too large, the search path is too restricted and high-quality solutions may be missed. A good parameter setting for $tt$ can typically only be found empirically and often requires considerable fine-tuning. Therefore, several approaches to make the particular settings of $tt$ more robust or to adjust $tt$ dynamically during the run of the algorithm have been introduced.

*Robust Tabu Search* [Taillard, 1991] achieves an increased robustness of performance w.r.t. the tabu tenure by repeatedly choosing $tt$ randomly from an interval $[tt_{min}, tt_{max}]$. Additionally, Robust Tabu Search forces specific local search moves if these have not been applied for a large number of iterations. For example, in the case of SAT, this corresponds to forcing a specific variable to be flipped if it has not been flipped in the last $k \cdot n$ search steps, where $k > 1$ is a parameter and $n$ is the number of variables in a given formula. (Note that it does not make sense to set $k$ to a value smaller or equal to one in this case.) A variant of Robust Tabu Search is currently amongst the best known algorithms for MAX-SAT, the optimisation variant of SAT (see also Chapter 7).

*Reactive Tabu Search* [Battiti and Tecchiolli, 1994], uses the search history to adjust the tabu tenure $tt$ dynamically during the search. In particular, if candidate solutions are repeatedly encountered, this is interpreted as evidence that search stagnation has occurred, and the tabu tenure is increased. If, on the contrary, no repetitions are found during a sufficiently long period of time, the tabu tenure is gradually decreased. Additionally, an escape mechanism based on a series of random changes is used to prevent the search process from getting trapped in a specific region of the search space. In Section 10.2 (page 482*ff.*) we will present in detail a reactive tabu search algorithm for the Quadratic Assignment Problem.

Generally, the efficiency of Tabu Search can be further increased by using techniques exploiting a form of *intermediate-term* or *long-term memory* to achieve additional intensification or diversification of the search process. Intensification strategies correspond to efforts of revisiting promising regions of the search space, for example, by recovering *elite candidate solutions*, that is, candidate solutions that are amongst the best that have been found in the search process so far. When recovering an elite candidate solution, all tabu restrictions associated with it can be cleared, in which case the search may follow a different search path. (For an example, we refer to the tabu search algorithm presented in Section 9.3, page 446*ff.*) Another possibility is to freeze certain solution components and to keep them fixed during the search. In the TSP case, this amounts to forcing certain edges to be kept in the candidate solutions seen over a number of iterations.

Diversification can be achieved by generating new combinations of solution components, which can help to explore regions of the search space that have not been visited yet. One way of achieving this is by introducing a few rarely used solution components into the candidate solutions. An example for such a mechanism is the forced execution of search steps, as in Robust Tabu Search. Another possibility is to bias the local search by adding a component to the evaluation function contribution of specific search steps based on the frequency with which these were applied. For a detailed discussion of diversification and intensification techniques that exploit intermediate and long-term memory we refer to Glover and Laguna [1997].

Overall, tabu search algorithms have been successfully applied to a wide range of combinatorial problems, and for many problems they are among best known algorithms w.r.t. the tradeoff between solution quality and computation time [Battiti and Protasi, 2001; Galinier and Hao, 1997; Nowicki and Smutnicki, 1996b; Vaessens et al., 1996]. We will discuss several tabu search algorithms in the second part of this book. Crucial for these successful applications of Tabu Search is often a carefully chosen neighbourhood relation, as well as the use of efficient caching and incremental updating schemes for the evaluation of candidate solutions.

## Dynamic Local Search

So far, the various techniques for escaping from local optima discussed in this chapter were all based on allowing worsening steps during the search process. A different approach for preventing iterative improvement methods from getting stuck in local optima is to modify the evaluation function whenever a local optimum is encountered in such a way that further improvement steps become possible. This can be achieved by associating *penalty weights* with individual solution components, which determine the impact of these components on the evaluation function value. Whenever the iterative improvement process gets trapped in a local optimum, the penalties of some solution components are increased. This leads to a degradation in the current candidate solution's evaluation function value until it is higher than the evaluation function values of some of its neighbours (which are not affected in the same way by the penalty modifications), at which point improving moves become available. This general approach provides the basis for a number of SLS algorithms which we collectively refer to as *dynamic local search (DLS)* methods.

Figure 2.9 shows an algorithm outline of DLS. As motivated above, the underlying idea is to find local optima of a dynamically changing evaluation function $g'$ using a simple local search algorithm *localSearch*, which typically performs iterative improvement until a local minimum in $g'$ is found. The modified evaluation function $g'$ is obtained by adding penalties $penalty(i)$ to solution components used in a candidate solution $s$ to the original evaluation function value $g(\pi', s)$:

$$g'(\pi', s) := g(\pi, s) + \sum_{i \in SC(\pi', s)} penalty(i),$$

where $SC(\pi', s)$ is the set of solution components of $\pi'$ used in a candidate solution $s$.

The penalties $penalty(i)$ are initially set to zero and subsequently updated after each subsidiary local search. Typically, *updatePenalties* increases the penalties of some or all the solution components used by the locally optimal candidate solution $s'$ obtained from $localSearch(\pi, g', s)$. Particular DLS algorithms differ in how this update is performed. One main difference is whether the penalty modifications are done in an additive way or in a multiplicative way. In both cases, the penalty modification is typically parameterised by some constant $\lambda$, which also takes into account the range of evaluation function values for the particular instance being solved. Additionally, some DLS techniques occasionally decrease the penalties of solution components not used in $s'$ [Schuurmans and Southey, 2000; Schuurmans et al., 2001].

```
procedure DLS (π')
    input: problem instance π' ∈ Π'
    output: solution ŝ ∈ S(π') or ∅
    s := init(π');
    s := localSearch(π', s);
    ŝ := s;
    while not terminate (π', s) do
        g' := g + Σ_{i∈SC(π',s)} penalty(i);
        s' := localSearch(π, g', s);
        if (f(s') < f(ŝ);
            ŝ := s';
        end
        updatePenalties(π, s');
    end
    if ŝ ∈ S' then
        return ŝ
    else
        return ∅
    end
end DLS
```

***Figure* 2.9** Algorithm outline of Dynamic Local Search for optimisation problems; $penalty(i)$ is the penalty associated with solution component $i$, $SC(\pi', s)$ is the set of solution components used in candidate solution $s$, $localSearch(\pi', g', s)$ is a subsidiary local search procedure using evaluation function $g'$, and *updatePenalties* is a procedure for updating the solution component penalties. (Further details are given in the text.)

Penalising all solution components of a locally optimal candidate solution can cause difficulties if certain solution components that are required for any optimal solution are also present in many other local optima. In this case, it can be useful to only increase the penalties of solution components that are least likely to occur in globally optimal solutions. One specific mechanism that implements this idea uses the solution quality contribution of a solution component $i$ in candidate solution $s'$, $f_i(\pi, s')$, to estimate the utility of increasing $penalty(i)$:

$$util(s', i) := \frac{f_i(\pi, s')}{1 + penalty(i)} \tag{2.2}$$

Using this estimate of utility, *updatePenalties* then only increases the penalties of solution components with maximal utility values. Note that dividing the solution quality distribution by $1 + penalty(i)$ avoids overly frequent penalisation

of specific solution components by reducing their utility. (This mechanism is used in a particular DLS algorithm called Guided Local Search [Voudouris and Tsang, 1995].)

It is worth noting that in many cases, the solution quality contribution of a solution component does not depend on the current candidate solution. In the case of the TSP, for example, the solution components are typically the edges of the given graph, and their solution quality contributions are given by their respective weights. There are cases, however, where the solution quality contributions of individual solution components are dependent on the current candidate solution $s'$, or, more precisely, on all solution components of $s'$. This is the case, for example, for the Quadratic Assignment Problem (see Section 10.2, page 477*ff.*), where DLS algorithms typically use approximations of the actual solution cost contribution [Voudouris and Tsang, 1995; Mills et al., 2003].

EXAMPLE 2.6 **Dynamic Local Search for the TSP**

This example follows the first application of DLS to the TSP, as presented by Voudouris and Tsang [1995; 1999], and describes a particular DLS algorithm called *Guided Local Search (GLS)*. Given a TSP instance in the form of an edge-weighted graph $G$, the same search space, solution set and 2-exchange neighbourhood is used as in Example 2.1 (page 64*f.*). The solution components are the edges of $G$, and the cost contribution of each edge $e$ is given by its weight, $w(e)$. The subsidiary local search procedure *localSearch* performs first improvement steps in the underlying 2-exchange neighbourhood and can be enhanced by using standard speed-up techniques, which are described in detail in Chapter 8, Section 8.2.

In GLS, the procedure *updatePenalties*$(\pi, s)$ increments the penalties of all edges of maximal utility contained in candidate solution $s$ by a factor $\lambda$, which is chosen in dependence of the average length of good tours; in particular a setting of

$$\lambda := 0.3 \cdot \frac{f(s_{2\text{-}opt})}{n}$$

where $f(s_{2\text{-}opt})$ is the objective function value of a 2-optimal tour, and $n$ is the number of vertices in $G$, has been shown to yield very good results on a set of standard TSP benchmark instances [Voudouris and Tsang, 1999].

The fundamental idea underlying DLS of adaptively modifying the evaluation function during a local search process has been used as the basis for a number of SLS algorithms for various combinatorial problems. Among the earliest DLS algorithms is the Breakout Method, in which penalities are added to solution

components of locally optimal solutions [Morris, 1993]. GENET [Davenport et al., 1994], an algorithm that adaptively modifies the weight of constraints to be satisfied, has directly inspired Guided Local Search, one of the most widely applied DLS methods [Voudouris and Tsang, 1995; 2002]. Closely related SLS algorithms, which can be seen as instances of the general DLS method presented here, have been developed for constraint satisfaction and SAT, where penalties are typically associated with the clauses of a given CNF formula [Selman and Kautz, 1993; Cha and Iwama, 1996; Frank, 1997]; this particular approach is also known as *clause weighting*. Some of the best-performing SLS algorithms for SAT and MAX-SAT (an optimisation variant of SAT) are based on clause weighting schemes inspired by Lagrangean relaxation techniques [Hutter et al., 2002; Schuurmans et al., 2001; Wu and Wah, 2000].

## 2.3 Hybrid SLS Methods

As we have seen earlier in this chapter, the behaviour and performance of 'simple' SLS techniques can often be improved significantly by combining them with other SLS strategies. We have already presented some very simple examples of such hybrid SLS methods. Randomised Iterative Improvement, for example, can be seen as a hybrid SLS algorithm, obtained by probabilistically combining standard Iterative Improvement and Uninformed Random Walk (cf. Section 2.2, page 72*ff.*). Similarly, many SLS implementations make use of a random restart mechanism that terminates and restarts the search process from a randomly chosen initial position based on standard termination conditions; this can be seen as a hybrid combination of the underlying SLS algorithm and Uninformed Random Picking. In this section, we present a number of well-known and very successful SLS methods that can be seen as hybrid combinations of simpler SLS techniques.

### Iterated Local Search

In the previous sections, we have discussed various mechanisms for preventing iterative improvement techniques from getting stuck in local optima of the evaluation function. Arguably one of the simplest and most intuitive ideas for addressing this fundamental issue is to use two types of SLS steps: one for reaching local optima as efficiently as possible, and the other for effectively escaping from local optima. This is the key idea underlying *Iterated Local Search (ILS)* [Lourenço et al., 2002], a SLS method that essentially uses these two types of search steps alternatingly to perform a walk in the space of local optima w.r.t. the given evaluation function.

```
procedure ILS(π')
    input: problem instance π' ∈ Π'
    output: solution ŝ ∈ S(π') or ∅

    s := init(π');
    s := localSearch(π', s);
    ŝ := s;
    while not terminate(π', s) do
        s' := perturb(π', s');
        s'' := localSearch(π', s');
        if (f(s'') < f(ŝ)) then
            ŝ := s'';
        end
        s := accept(π', s, s'');
    end
    if ŝ ∈ S' then
        return ŝ
    else
        return ∅
    end
end ILS
```

***Figure 2.10*** Algorithm outline of Iterated Local Search (ILS) for optimisation problems. (For details, see text.)

Figure 2.10 shows an algorithm outline for ILS. As usual, the search process can be initialised in various ways, for example, by starting from a randomly selected element of the search space. From the initial candidate solution, a locally optimal solution is obtained by applying a subsidiary local search procedure *localSearch*. Then, each iteration of the algorithm consists of three major stages: first, a perturbation is applied to the current candidate solution $s$; this yields a modified candidate solution $s'$ from which in the next stage a subsidiary local search is performed until a local optimum $s''$ is obtained. In the last stage, an acceptance criterion *accept* is used to decide from which of the two local optima, $s$ or $s''$, the search process is continued. Both functions, *perturb* and *accept*, can use aspects of the search history, for example, when the same local optima are repeatedly encountered, stronger perturbation steps may be applied. As in the case of most other SLS algorithms, a variety of termination predicates *terminate* can be used for deciding when the search process ends.

The three procedures *localSearch*, *perturb* and *accept* form the core of any ILS algorithm. The specific choice of these procedures has a crucial impact

on the performance of the resulting algorithm. As we will discuss in the following, these components need to complement each other for achieving a good tradeoff between intensification and diversification of the search process, which is critical for obtaining good performance when solving hard combinatorial problems.

It is rather obvious that the subsidiary local search procedure, *localSearch*, has a considerable influence on the performance of any ILS algorithm. In general, more effective local search methods lead to better performing ILS algorithms. For example, when applying ILS to the Travelling Salesman Problem, using 3-opt local search (i.e., an iterative improvement algorithm based on the 3-exchange neighbourhood relation) typically leads to better performance than using 2-opt local search, while even better results than with 3-opt local search are obtained when using the Lin-Kernighan Algorithm as a subsidiary local search procedure. While often, iterative improvement methods are used for the subsidiary local search within ILS, it is perfectly possible to use more sophisticated SLS algorithms, such as SA, TS or DLS, instead.

The role of *perturb* is to modify the current candidate solution in a way that will not be immediately undone by the subsequent local search phase. This helps the search process to effectively escape from local optima, and the subsequent local search phase has a chance to discover different local optima. In the simplest case, a random walk step in a larger neighbourhood than the one used by *localSearch* may be sufficient for achieving this goal. There are also ILS algorithms that use perturbations consisting of a number of simple steps (e.g., sequences of random walk steps in a 1-exchange neighbourhood).

Typically, the strength of the perturbation has a strong influence on the length of the subsequent local search phase; weak perturbations usually lead to shorter local search phases than strong perturbations, because the local search procedure requires fewer steps to reach a local optimum. If the perturbation is too weak, however, the local search will often fall back into the local optimum just visited, which leads to search stagnation. At the same time, if the perturbation is too strong, its effect can be similar to a random restart of the search process, which usually results in a low probability of finding better solutions in the subsequent local search phase. To address these issues, both the strength and the nature of the perturbation steps may be changed adaptively during the search. Furthermore, there are rather complex perturbation techniques, such as the one used in Lourenço [1995], which is based on finding optimal solutions for parts of the given problem instance.

The acceptance criterion, *accept*, also has a strong influence on the behaviour and performance of ILS. A strong intensification of the search is obtained if the better of the two solutions $s$ and $s''$ is always accepted. ILS algorithms using this acceptance criterion effectively perform iterative improvement in the space of local optima reached by the subsidiary local search procecudure. Conversely, if

the new local optimum, $s''$, is always accepted regardless of its solution quality, the behaviour of the resulting ILS algorithm corresponds to a random walk in the space of the local optima of the given evaluation function. Between these extremes, many intermediate choices exist; for example, the Metropolis acceptance criterion known from Simulated Annealing has been used in an early class of ILS algorithms called *Large Step Markov Chains* [Martin et al., 1991]. While all these acceptance criteria are Markovian, that is, they only depend on $s$ and $s''$, it has been shown that acceptance criteria that take into account aspects of the search history, such as the number of search steps since the last improvement of the incumbent candidate solution, often help to enhance ILS performance [Stützle, 1998c].

EXAMPLE 2.7 **Iterated Local Search for the TSP**

In this example we describe the *Iterated Lin-Kernighan (ILK) Algorithm*, an ILS algorithm that is currently amongst the best performing incomplete algorithms for the Travelling Salesman Problem. ILK is based on the same search space and solution set as used in Example 2.3 (page 75). The subsidiary local search procedure *localSearch* is the Lin-Kernighan variable depth search algorithm (LK) described in Section 2.1 (page 68*ff.*).

Like almost all ILS algorithms for the Travelling Salesman Problem, ILK uses a particular 4-exchange step, called a *double-bridge move*, as a perturbation step. This double-bridge move is illustrated in Figure 2.11; it has the desirable property that it cannot be directly reversed by a sequence of 2-exchange moves as performed by the LK algorithm. Furthermore, it was found in empirical studies that this perturbation is effective independently of problem size. Finally, an acceptance criterion is used that always returns the better of the two candidate solutions $s$ and $s''$. An efficient implementation of

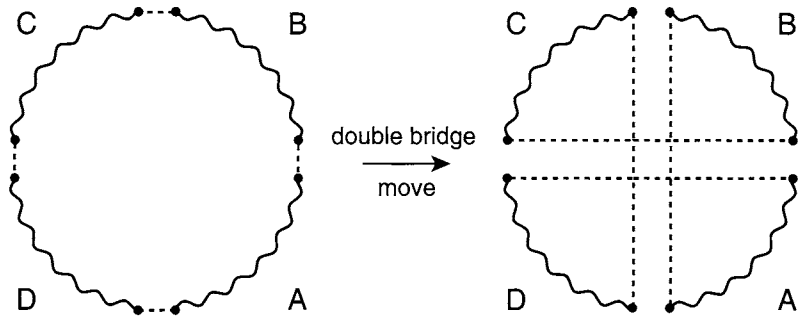


*Figure* 2.11 Schematic representation of the double-bridge move used in ILK. The four dashed edges on the left are removed and the remaining paths A, B, C, D are reconnected as shown on the right side.

this structurally rather simple algorithm has been shown to achieve excellent performance [Johnson and McGeoch, 1997]. (Details on this and other ILS algorithms for the TSP are presented in Section 8.3, page 384*ff.*)

Generally, ILS can be seen as a straight-forward, yet powerful technique for extending 'simple' SLS algorithms such as Iterative Improvement. The conceptual simplicity of the underlying idea led to frequent re-discoveries and many variants, most of which are known under various names, such as *Large Step Markov Chains* [Martin et al., 1991], *Chained Local Search* [Martin and Otto, 1996], as well as, when applied to particular algorithms, to specific techniques, such as Iterated Lin-Kernighan algorithms [Johnson and McGeoch, 1997]. Despite the fact that the underlying ideas are quite different, there is also a close conceptual relationship between ILS and certain variants of Variable Neighbourhood Search (VNS), such as Basic VNS and Skewed VNS [Hansen and Mladenović, 2002].

ILS algorithms are also attractive because they are typically easy to implement: in many cases, existing SLS implementations can be extended into ILS algorithms by adding just a few lines of code. At the same time, ILS algorithms are currently among the best-performing incomplete search methods for many combinatorial problems, the most prominent application being the Travelling Salesman Problem [Johnson and McGeoch, 1997; Martin and Otto, 1996]. For an overview of various issues arising in the design and implementation of ILS algorithms we refer to Lourenço et al. [2002].

## Greedy Randomised Adaptive Search Procedures

A standard approach for quickly finding high-quality solutions for a given combinatorial optimisation problem is to apply a greedy construction search method (see also Chapter 1, Section 1.4) that, starting from an empty candidate solution, at each construction step adds the solution component ranked best according to a heuristic selection function, and to subsequently use a perturbative local search algorithm to improve the candidate solution thus obtained. In practice, this type of hybrid search method often yields much better solution quality than simple SLS methods initialised at candidate solutions obtained by Uninformed Random Picking (see Chapter 1, Section 1.5). Additionally, when starting from a greedily constructed candidate solution, the subsequent perturbative local search process typically takes much fewer improvement steps to reach a local optimum. By iterating this process of greedy construction and perturbative local search, even higher-quality solutions can be obtained.

Unfortunately, greedy construction search methods can typically only generate a very limited number of different candidate solutions. *Greedy Randomised*

```
procedure GRASP(π')
    input: problem instance π' ∈ Π'
    output: solution ŝ ∈ S(π') or ∅

    s := ∅;
    ŝ := s;
    f(ŝ) := ∞;
    while not terminate(π', s) do
        s := construct(π');
        s' := localSearch(π', s);
        if (f(s') < f(ŝ)) then
            ŝ := s';
        end
    end
    if ŝ ∈ S' then
        return ŝ
    else
        return ∅
    end
end GRASP
```

**Figure 2.12** Algorithm outline of GRASP for optimisation problems. (For details, see text.)

*Adaptive Search Procedures (GRASP)* [Feo and Resende, 1989; 1995] try to avoid this disadvantage by randomising the construction method, such that it can generate a large number of different good starting points for a perturbative local search method.

Figure 2.12 shows an algorithm outline for GRASP. In each iteration of the algorithm, first a candidate solution $s$ is generated using a randomised constructive search procedure, *construct*. Then, a local search procedure, *localSearch*, is applied to $s$, yielding an improved (typically, locally optimal) candidate solution $s'$. This two-phase process is iterated until a termination condition is satisfied.

In contrast to standard greedy constructive search methods, the constructive search algorithm used in GRASP does not necessarily add a solution component with maximal heuristic value in each construction step, but rather selects randomly from a set of highly ranked solution components. This is done by defining in each construction step a *restricted candidate list (RCL)* and then selecting one of the solution components in the RCL randomly according to a uniform distribution. In GRASP, there are two different mechanisms for defining the RCL: by cardinality restriction or by value restriction. In the case of a cardinality restriction, only the $k$ best-ranked solution components are included in the RCL.

Value restriction allows the number of elements in the RCL to vary. More specifically, let $g(l)$ be the greedy heuristic value of a solution component $l$, $L$ be the set of feasible solution components, and let $g_{min} := \min\{g(l) \mid l \in L\}$ and $g_{max} := \max\{g(l) \mid l \in L\}$ be the best and worst heuristic values among the feasible components, respectively. Then a component $l$ is inserted into the RCL if, and only if, $g(l) \leq g_{min} + \alpha(g_{max} - g_{min})$. Clearly, the smaller $k$ or $\alpha$, the greedier is the selection of the next solution component.

The constructive search process performed within GRASP is 'adaptive' in the sense that the heuristic value for each solution component typically depends on the components that are already present in the current partial candidate solution. This takes more computation time than using static heuristic values that do not change during the construction process, but this overhead is typically amortised by the higher quality solutions obtained when using the 'adaptive' search method.

Note that it is entirely feasible to perform GRASP without a perturbative local search phase; the respective restricted variants of GRASP are also known as *semi-greedy* heuristics [Hart and Shogan, 1987]. But in general, the candidate solutions obtained from the randomised constructive search process are not guaranteed to be locally optimal with respect to some simple neighbourhood; hence, even the additional use of a simple iterative improvement algorithm typically yields higher quality solutions with rather small computational overhead. Indeed, for a large number of combinatorial problems, empirical results indicate that the additional local search phase improves the performance of the algorithm considerably.

EXAMPLE 2.8 **GRASP for SAT**

GRASP can be applied to SAT in a rather straightforward way [Resende and Feo, 1996]. The constructive search procedure starts from an empty variable assignment and adds an atomic assignment (i.e., an assignment of a truth value to an individual propositional variable of the given CNF formula) in each construction step. The heuristic function used for guiding this construction process is defined by the number of clauses that become satisfied as a consequence of adding a particular atomic assignment to the current partial assignment.

Let $h(i, v)$ be the number of (previously unsatisfied) clauses that become satisfied as a consequence of the atomic assignment $x_i := v$, where $v \in \{\top, \bot\}$. In each construction step, an RCL is built by cardinality restriction; this RCL contains the $k$ variable assignments with the largest heuristic value $h(i, v)$.

In the simplest case, the current partial assignment is extended by an atomic variable assignment that is selected from the RCL uniformly at random. In Resende and Feo [1996], a slightly more complex assignment

strategy is followed. If an unsatisfied clause $c$ exists, in which only one variable is unassigned under the current partial assignment, this variable is assigned the value that renders $c$ satisfied. (This mimics unit propagation, a well-known simplification strategy for SAT that is widely used in complete SAT algorithms.) Only if no such clause exists, a random element of the RCL is selected instead.

After a complete assignment has been generated, the respective candidate solution is improved using a best improvement variant of the iterative improvement algorithm for SAT from Example 1.4 (page 47*f.*). The search process is terminated when a solution has been found or after a given number of iterations has been exceeded.

This GRASP algorithm together with other variants of *construct* was implemented and tested on a large number of satisfiable SAT instances from the DIMACS benchmark suite [Resende and Feo, 1996]. While the results were reasonably good at the time the algorithm was first presented, it is now outperformed by more recent SLS algorithms for SAT (see Chapter 6).

GRASP has been applied to a large number of combinatorial problems, including MAX-SAT, Quadratic Assignment and various scheduling problems; we refer to Festa and Resende [2001] for an overview of GRASP applications. There are also a number of recent improvements and extensions of the basic GRASP algorithm; some of these include reactive GRASP variants in which, for example, the parameter $\alpha$ used in value-restricted RCLs is dynamically adapted [Prais and Ribeiro, 2000], and combinations with tabu search or path relinking algorithms [Laguna and Martí, 1999; Lourenço and Serra, 2002]. For a detailed introduction to GRASP and a discussion of various extensions of the basic GRASP algorithm as presented here, we refer to Resende and Ribeiro [2002].

## Adaptive Iterated Construction Search

Considering algorithms based on repeated constructive search processes, such as GRASP, the idea of exploiting experience gained from past iterations for guiding further solution constructions is appealing. One way of implementing this idea is to use weights associated with the possible decisions that are made during the construction process. These weights are adapted over multiple iterations of the search process to reflect the experience from previous iterations. This leads to a family of SLS algorithms we call *Adaptive Iterated Construction Search (AICS)*.

An algorithm outline of AICS is shown in Figure 2.13. At the beginning of the search process, all weights are initialised to some small value $\tau_0$. Each

```
procedure AICS(π')
    input: problem instance π' ∈ Π'
    output: solution ŝ ∈ S(π') or ∅

    s := ∅;
    ŝ := s;
    f(ŝ) := ∞;
    w := initWeights(π');
    while not terminate(π', s) do
        s := construct(π', w, h);
        s' := localSearch(π', s);
        if f(s') < f(ŝ) then
            ŝ = s';
        end
        w := adaptWeights(π', s', w);
    end
    if ŝ ∈ S' then
        return ŝ
    else
        return ∅
    end
end AICS
```

***Figure* 2.13** Algorithm outline of Adaptive Iterated Construction Search for optimisation problems. (For details, see text.)

iteration of AICS consists of three phases. First, a constructive search process is used to generate a candidate solution $s$. Next, an additional perturbative local search phase is performed on $s$, yielding a locally optimal solution $s'$. Finally, the weights are adapted based on the solution components used in $s'$ and the solution quality of $s'$. As usual, various termination conditions can be used to determine when the search process is ended.

The constructive search process uses the weights as well as a heuristic function $h$ on the solution components to probabilistically select components for extending the current partial candidate solution. Generally, $h$ can be chosen to be a standard heuristic function, as used for greedy methods or in the context of tree search algorithms; alternatively, $h$ can be based on lower bounds on the solution quality of $s$, such as the bounds used in branch & bound algorithms. For AICS, it can be advantageous to implement the solution component selection in such a way that at all points of the construction process, with a small probability,

any component solution can be added to the current partial candidate solution, irrespective of its weight and heuristic value.

As in GRASP, the perturbative local search phase typically improves the quality of the candidate solution generated by the construction process, leading to an overall increase in performance. In the simplest case, iterative improvement algorithms can be used in this context; however, it is perfectly possible and potentially beneficial to use more powerful SLS methods that can escape from local optima of the evaluation function. Typically, there is a tradeoff between the computation time used by the local search phase *vs* the construction phase, which can only be optimised empirically and depends on the given problem domain.

The adjustment of the weights, as implemented in the procedure *adaptWeights*, is typically done by increasing the weights that correspond to the solution components contained in $s'$. In this context, it is also possible to use aspects of the search history; for example, by using the incumbent candidate solution as the basis for the weight update, the sampling performed by the construction and perturbative search phases can be focused more directly on promising regions of the search space.

### EXAMPLE 2.9 A Simple AICS Algorithm for the TSP

The AICS algorithm presented in this example is a simplified version of Ant System for the TSP by Dorigo, Maniezzo and Colorni [1991; 1996], enhanced by an additional perturbative search phase, which in practice improves the performance of the original algorithm. (Ant System is a particular instance of Ant Colony Optimisation, an SLS method discussed in the following section.) It uses the same search space and solution set as used in Example 2.3 (page 75).

Weights $\tau_{ij} \in \mathbb{R}_0^+$ are associated with each edge $(i, j)$ of the given graph $G$, and heuristic values $\eta_{ij} := 1/w((i, j))$ are used, where $w((i, j))$ is the weight of edge $(i, j)$. At the beginning of the search process, all edge weights are initialised to a small value, $\tau_0$. The function *construct* iteratively constructs vertex permutations (corresponding to Hamiltonian cycles in $G$). The construction process starts with a randomly chosen vertex and then extends the partial permutation $\phi$ by probabilistically selecting a vertex not contained in $\phi$ according to the following distribution:

$$p_{ij} := \frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in N'(i)} [\tau_{il}]^\alpha \cdot [\eta_{il}]^\beta} \qquad \text{if } j \in N'(i) \tag{2.3}$$

and 0 otherwise, where $N'(i)$ is the feasible neighbourhood of vertex $i$, that is, the set of all neighbours of $i$ that are not contained in the current partial permutation $\phi$, and $\alpha$ and $\beta$ are parameters that control the relative impact of the weights *vs* the heuristic values.

Upon the completion of each construction process, an iterative improvement search using the 2-exchange neighbourhood is performed until a vertex permutation corresponding to a Hamiltonian cycle with minimal path weight is reached.

The adaption of the weights $\tau_{ij}$ is done by first decreasing all $\tau_{ij}$ by a constant factor and then increasing the weights of the edges used in $s'$ proportionally to the path weight $f(s')$ of the Hamiltonian cycle represented by $s'$, that is, for all edges $(i, j)$, the following update is performed:

$$\tau_{ij} := (1 - \rho) \cdot \tau_{ij} + \Delta(i, j, s') \tag{2.4}$$

where $0 < \rho \leq 1$ is a parameter of the algorithm, and $\Delta(i, j, s')$ is defined as $1/f(s')$ if edge $(i, j)$ is contained in the cycle represented by $s'$ and as zero otherwise.

The decay mechanism controlled by the parameter $\rho$ helps to avoid unlimited increased of the weights $\tau_{ij}$ and lets the algorithm 'forget' the past experience reflected in the weights. The specific definition of $\Delta(i, j, s')$ reflects the idea that edges contained in good candidate solutions should be used with higher probability in subsequent constructions. The search process is terminated after a fixed number of iterations.

Different from most of the other SLS methods presented in this chapter, AICS has not (yet) been widely used as a general SLS technique. It is very useful, however, as a general framework that helps to understand a number of recent variants of constructive search algorithms. In particular, various incomplete tree search algorithms can be seen as instances of AICS, including the stochastic tree search algorithm by Bresina [1996], the Squeeky-Wheel Optimisation algorithm by Joslin and Clements [1999], and the Adaptive Probing algorithm by Ruml [2001]. Furthermore, AICS can be viewed as a special case of Ant Colony Optimisation, a prominent SLS method based on an adaptive iterated construction process involving populations of candidate solutions.

## 2.4 Population-Based SLS Methods

All SLS methods we have discussed so far manipulate only one single candidate solution of the given problem instance in each search step. A straightforward extension is to consider algorithms where several individual candidate solutions are simultaneously maintained; this idea leads to the population-based SLS methods discussed in this section. Although in principle, one could consider population-based search methods in which the population size may vary throughout the

search process, the population-based SLS methods considered here typically use constant size populations.

Note that population-based SLS algorithms fit into the formal definition of an SLS algorithm (Definition 1.10, page 38*f.*) by considering search positions that are sets of individual candidate solutions. Though interesting in some ways, this view is somewhat unintuitive, and for most practical purposes, it is preferable to think of the search process as operating on sets of candidate solutions for the given problem instance. For example, a population-based SLS algorithm for SAT intuitively operates on a set of variable assignments. In the following, unless explicitly stated otherwise, we will use the term 'candidate solution' in this intuitive sense, rather than to refer to entire populations.

The use of populations offers several conceptual advantages in the context of SLS methods. For instance, a population of candidate solutions provides a straightforward means for achieving search diversification and hence for increasing the exploration capabilities of the search process. Furthermore, it facilitates the use of search mechanisms that are based on the combination of promising features from a number of individual candidate solutions.

## Ant Colony Optimisation

Ant Colony Optimisation (ACO) is a population-based SLS method inspired by aspects of the pheromone-based trail-following behaviour of real ants; it was first introduced by Dorigo, Maniezzo and Colorni [1991] as a metaphor for solving hard combinatorial problems, such as the TSP. ACO can be seen as a population-based extension of AICS, based on a population of agents (*ants*) that indirectly communicate via distributed, dynamically changing information, the so-called *(artificial) pheromone trails*. These pheromone trails reflect the collective search experience and are exploited by the ants in their attempts to solve a given problem instance. The pheromone trail levels used in ACO correspond exactly to the weights in AICS. Here, we use the term 'pheromone trail level' instead of 'weight' to be consistent with the literature on Ant Colony Optimisation.

An algorithm outline of ACO for optimisation problems is shown in Figure 2.14. Conceptually, the algorithm is usually thought of as being executed by $k$ ants, each of which creates and manipulates one candidate solution. The search process is started by initialising the pheromone trail levels; typically, this is done by setting all pheromone trail levels to the same value, $\tau_0$. In each iteration of ACO, first a population $sp$ of $k$ candidate solutions is generated by a constructive search procedure *construct*. As in AICS, in this construction process each ant starts with an empty candidate solution and iteratively extends the current partial candidate solution with solution components that are selected probabilistically according to the pheromone trail levels and a heuristic function, $h$.

```
procedure ACO(π')
    input: problem instance π' ∈ Π'
    output: solution ŝ ∈ S'(π') or ∅

    sp := {∅};
    ŝ := ∅;
    f(ŝ) := ∞;
    τ := initTrails(π');
    while not terminate(π', sp) do
        sp := construct(π', τ, h);
        sp' := localSearch(π', sp);
        if f(best(π', sp')) < f(ŝ) then
            ŝ = best(π', sp');
        end
        τ := updateTrails(π', sp', τ);
    end
    if ŝ ∈ S' then
        return ŝ
    else
        return ∅
    end
end ACO
```

*Figure* **2.14** Algorithm outline of Ant Colony Optimisation for optimisation problems; $best(\pi', sp')$ denotes the individual from population $sp$ with the best objective function value. The use of the procedure $localSearch(\pi', sp')$ is optional. (For details, see text.)

Next, a perturbative local search procedure *localSearch* may be applied to each candidate solution in $sp$; typically, an iterative improvement method is used in this context, resulting in a population $sp'$ of locally optimal candidate solutions. If the best of the candidate solutions in $sp'$, $best(\pi', sp')$, improves on the overall best solution obtained so far, this candidate solution becomes the new incumbent candidate solution. As in GRASP and AICS, this perturbative local search phase is optional, but typically leads to significantly improved performance of the algorithm.

Finally, the pheromone trail levels are updated based on the candidate solutions in $sp'$ and their respective solution qualities. The precise pheromone update mechanism differs between various ACO algorithms. A typical mechanism first uniformly decreases all pheromone trail levels by a constant factor (intuitively, this corresponds to the physical process of pheromone evaporation), after which a subset of the pheromone trail levels is increased; this subset and the

amount of the increase is determined from the quality of the candidate solutions in $sp'$ and $\hat{s}$, and from the solution components contained in these. As usual, a number of different termination predicates can be used to determine when to end the search process. As an alternative to standard termination criteria based on CPU time or the number of iterations, these can include conditions on the make-up of the current population, $sp'$, such as the variation in solution quality across the elements of $sp'$ or their average distance from each other.

### EXAMPLE 2.10 A Simple ACO Algorithm for the TSP

In this example, we present a variant of Ant System for the TSP, a simple ACO algorithm which played an important role as the first application of the ant colony metaphor to solving combinatorial optimisation problems [Dorigo et al., 1991; Dorigo, 1992; Dorigo et al., 1996].

This algorithm can be seen as a slight extension of the AICS algorithm from Example 2.9 (page 94*f.*). The initialisation of the pheromone trail levels is performed exactly like the weight initialisation in the AICS example. The functions *construct* and *localSearch* are straightforward extensions of the ones from Example 2.9 that perform the respective construction and perturbative local search processes for each individual candidate solution independently.

The pheromone trail update procedure, *updateTrails*, is also quite similar to the *adaptWeights* procedure from the AICS example; in fact, it is based on the same update as specified in Equation 2.4 (page 95), but instead of $\Delta(i, j, s')$, now a value $\Delta(i, j, sp')$ is used, which is based on contributions from all candidate solutions in the current population $sp'$ according to the following definition:

$$\Delta(i, j, sp') := \sum_{s' \in sp'} \Delta(i, j, s'), \tag{2.5}$$

where $\Delta(i, j, s')$ is defined as $1/f(s')$ if edge $(i, j)$ is contained in the Hamiltonian cycle represented by candidate solution $s'$ and as zero otherwise. According to this definition, the pheromone trail levels associated with edges which belong to the highest-quality candidate solutions (i.e., low-weight Hamiltonian cycles) and which have been used by the most ants are increased the most. This reflects the idea that heuristically, these edges are most likely to be contained in even better (and potentially optimal) candidate solutions and should therefore be selected with higher probability during future construction phases. The search process is terminated after a fixed number of iterations.

Note how, in terms of the biological metaphor, the phases of this algorithm can be interpreted loosely as the actions of ants that walk the edges of the given graph to construct tours (using memory to ensure that

only Hamiltonian cycles are generated as candidate solution) and deposit pheromones to reinforce the edges of their tours.

The algorithm from Example 2.10 differs from the original Ant System (AS) only in that AS did not include a perturbative local search phase. For many (static) combinatorial problems and a variety of ACO algorithms, it has been shown, however, that the use of a perturbative local search phase leads to significant performance improvements [Dorigo and Gambardella, 1997; Maniezzo et al., 1994; Stützle and Hoos, 1996; 1997].

ACO, as introduced here, is typically applied to static problems, that is, to problems whose instances (i) are completely specified before the search process is started and (ii) do not change while solving the problem. (All combinatorial problems covered in this book are static in this sense.) In this case, the construction of candidate solutions, the perturbative local search phase, and the pheromone updates are typically performed in a parallel and fully synchronised manner by all ants. There are, however, different approaches, such as *Ant Colony System*, an ACO method in which ants modify the pheromone trails during the construction phase [Dorigo and Gambardella, 1997]. When applying ACO to dynamic optimisation problems, that is, optimisation problems where parts of the problem instances (such as the objective function) change over time, the distinction between synchronous and asynchronous, decentralised phases of the algorithm becomes very important. This is reflected in the *ACO metaheuristic* [Dorigo et al., 1999; Dorigo and Di Caro, 1999; Dorigo and Stützle, 2004], which provides a general framework for ACO applications to both, static and dynamic combinatorial problems.

ACO algorithms have been applied to a wide range of combinatorial problems. The first ACO algorithm, Ant System, was applied to the TSP and several other combinatorial problems. It has been shown to be capable of solving some non-trivial instances of these problems, but its performance falls substantially short of that of state-of-the-art algorithms. Nevertheless, Ant System can be seen as a proof-of-concept that the ideas underlying ACO can be used to solve combinatorial optimisation problems. Following Ant System, many other Ant Colony Optimisation algorithms have been developed, including Ant Colony System [Dorigo and Gambardella, 1997], $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System [Stützle and Hoos, 1997; Stützle and Hoos, 2000] and the ANTS Algorithm [Maniezzo, 1999]. These algorithms differ in important aspects of the search control and introduced advanced features, such as the use of look-ahead or pheromone trail level updates during the construction phase or diversification mechanisms, such as bounds on the range of possible pheromone trail levels. Some of the most prominent ACO applications are to dynamic optimisation problems, such as routing in telecommunications networks, in which traffic patterns are subject to significant changes over time [Di Caro and Dorigo, 1998]. We refer to the book by

Dorigo and Stützle [2004] for a detailed account of the ACO metaheuristic, different ACO algorithms, theoretical results and ACO applications.

## Evolutionary Algorithms

With Ant Colony Optimisation, we saw an example of a population-based SLS method in which the only interaction between the individual elements of the population of candidate solutions is of a rather indirect nature, through the modification of a common memory (namely, the pheromone trails). Perhaps the most prominent example for a type of population-based SLS algorithms based on a much more direct interaction within a population of candidate solutions is the class of *Evolutionary Algorithms (EAs)*.

In a broad sense, Evolutionary Algorithms are a large and diverse class of algorithms inspired by models of the natural evolution of biological species [Bäck, 1996; Mitchell, 1996]. They transfer the principle of evolution through mutation, recombination and selection of the fittest, which leads to the development of species that are better adapted for survival in a given environment, to solving computationally hard problems. Evolutionary algorithms are generally iterative, population-based approaches: Starting with a set of candidate solutions (the initial population), they repeatedly apply a series of three genetic operators, *selection*, *mutation* and *recombination*. Using these operators, in each iteration of an evolutionary algorithm, the current population is (completely or partially) replaced by a new set of candidate solutions; in analogy with the biological inspiration, the populations encountered in the individual iterations of the algorithm are often called *generations*.

The *selection* operator implements a (generally probabilistic) choice of individual candidate solutions either for the next generation or for the subsequent application of the mutation and recombination operators; it typically has the property that fitter individuals have a higher probability of being selected. *Mutation* is based on a unary operation on individuals that introduces small, often random modifications. *Recombination* is based on an operation that generates one or more new individuals (the *offspring*) by combining information from two or more individuals (the *parents*). The most commonly used type of recombination mechanism is called *crossover*; it is originally inspired by a fundamental mechanism in biological evolution of the same name, and essentially assembles pieces from a linear representation of the parents into a new individual. One major challenge in designing evolutionary algorithms is the design of recombination operators that combine parents in such a way that the resulting offspring is likely to inherit desirable properties from their parents, while improving on their parents' solution quality.

Note how Evolutionary Algorithms fit into our general definition of SLS algorithms, when the notion of a candidate solution as used in an SLS algorithm is applied to populations of candidate solutions of the given problem instance, as used in an EA. The concepts of search space, solution set and neighbourhood, as well as the generic functions *init*, *step* and *terminate*, can be easily applied to this population-based concept of a candidate solution. Nevertheless, to keep this description conceptually simple, in this section we continue to present evolutionary algorithms in the traditional way, where the notion of candidate solution refers to an individual of the population comprising the search state.

Intuitively, by using a population of candidate solutions instead of a single candidate solution, a higher search diversification can be achieved, particularly if the initial population is randomly selected. The primary goal of Evolutionary Algorithms for combinatorial problems is to evolve the population such that good coverage of promising regions of the search space is achieved, resulting in high-quality solutions of a given optimisation problem instance. However, pure evolutionary algorithms often seem to lack the capability of sufficient search intensification, that is, the ability to reach high-quality candidate solutions efficiently when a good starting position is given, for example, as the result of recombination or mutation. Hence, in many cases, the performance of evolutionary algorithms for combinatorial problems can be significantly improved by adding a local search phase after applying mutation and recombination [Brady, 1985; Suh and Gucht, 1987; Mühlenbein et al., 1988; Ulder et al., 1991; Merz and Freisleben, 1997; 2000b] or by incorporating a local search process into the recombination operator [Nagata and Kobayashi, 1997]. The class of Evolutionary Algorithms thus obtained is usually called *Memetic Algorithms (MAs)* [Moscato, 1989; Moscato and Norman, 1992; Moscato, 1999; Merz, 2000] or *Genetic Local Search* [Ulder et al., 1991; Kolen and Pesch, 1994; Merz and Freisleben, 1997].

In Figure 2.15 we show the outline of a generic memetic algorithm. At the beginning of the search process, an initial population is generated using function *init*. In the simplest (and rather common) case, this is done by randomly and independently picking a number of elements of the underlying search space; however, it is equally possible to use, for example, a randomised construction search method instead of random picking. In each iteration of the algorithm, recombination, mutation, perturbative local search and selection are applied to obtain the next generation of candidate solutions. As usual, a number of termination criteria can be used for determining when to end the search process.

The recombination function, $recomb(\pi', sp)$, typically generates a number of offspring solutions by repeatedly selecting a set of parents and applying a recombination operator to obtain one or more offspring from these. As mentioned

```
procedure MA(π')
    input: problem instance π' ∈ Π'
    output: solution ŝ ∈ S'(π') or ∅
    sp := init(π');
    sp := localSearch_1(π', sp);
    ŝ := best(π', sp);
    while not terminate(π', sp) do
        sp' := recomb(π', sp);
        sp' := localSearch_2(π', sp');
        sp'' := mutate(π, sp ∪ sp');
        sp'' := localSearch_3(π', sp'');
        if f(best(π', sp' ∪ sp'')) < f(ŝ) then
            ŝ = best(π', sp' ∪ sp'');
        end
        sp := select(π', sp, sp', sp'');
    end
    if ŝ ∈ S' then
        return ŝ
    else
        return ∅
    end
end MA
```

***Figure* 2.15** Algorithm outline of a memetic algorithm for optimisation problems; $best(\pi', sp)$ denotes the individual from a population $sp$ with the best objective function value. (For details, see text.)

before, this operation is generally based on a linear representation of the candidate solutions, and pieces together the offspring from fragments of the parents; this type of mechanism creates offspring that inherit certain subsets of solution components from their parents. One of the most commonly used recombination mechanisms is the *one-point binary crossover operator*, which works as follows. Given two parent candidate solutions represented by strings $x_1x_2\ldots x_n$ and $y_1y_2\ldots y_n$, first, a cut point $i$ is randomly chosen according to a uniform distribution over the index set $\{2,\ldots,n\}$. Two offspring candidate solutions are then defined as $x_1x_2\ldots x_{i-1}y_iy_{i+1}\ldots y_n$ and $y_1y_2\ldots y_{i-1}x_ix_{i+1}\ldots x_n$ (see also Figure 2.16).

One challenge when designing recombination mechanisms stems from the fact that often, simple crossover operators do not produce valid solution candidates. Consider, for example, a formulation of the TSP in which the solution candidates are represented by permutations of the vertex set, written as vectors

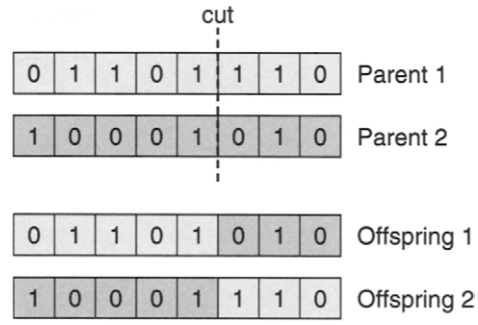


***Figure 2.16*** Schematic representation of the one-point binary crossover operator.

$(u_1, u_2, \ldots, u_n)$. Using a simple one-point binary crossover operation as the basis for recombination obviously leads to vectors that do not correspond to Hamiltonian cycles of the given graph. In cases like this, either a repair mechanism has to be applied to transform the results of a standard crossover into a valid candidate solution, or special crossover operators have to be used, which are guaranteed to produce valid candidate solutions only. In Chapter 8 we give two examples of high-performance memetic algorithms for the TSP that illustrate both possibilities. An overview of different specialised crossover operators for the TSP can be found in Merz and Freisleben [2001].

The role of function *mutate*$(\pi, sp \cup sp')$ is to introduce relatively small perturbations in the individuals in $sp \cup sp'$. Typically, these perturbations are of stochastic nature, and they are performed independently for each individual in $sp \cup sp'$, where the amount of perturbation applied is controlled by a parameter called the *mutation rate*. It should be noted that mutation need not be applied to all individuals of $sp \cup sp'$; instead, a subsidiary selection function can be used to determine which candidate solutions are to be mutated. (Until rather recently, the role of mutation compared to recombination for the performance of one of the most prominent types of evolutionary algorithms, *Genetic Algorithms*, has been widely underestimated [Bäck, 1996].)

As in ACO and AICS, perturbative local search is often useful and necessary for obtaining high-quality candidate solutions. It typically consists of selecting some or all individuals in the current population, and then applying an iterative improvement procedure to each element of this set independently.

Finally, the selection function used for determining the individuals that form the next generation $sp$ of candidate solutions typically considers elements of the original population, as well as the newly obtained candidate solutions, and selects from these based on their respective evaluation function values (which, in this context, are usually referred to as *fitness values*). Generally, the selection is done in such a way that candidate solutions with better evaluation function values

have a higher chance of 'surviving' the selection process. Many selection schemes involve probabilistic choices; however, it is often beneficial to use elitist strategies, which ensure that the best candidate solutions are always selected. Generally, the goal of selection is to obtain a population with good evaluation function values, but at the same time, to ensure a certain diversity of the population.

### EXAMPLE 2.11 A Memetic Algorithm for SAT

As in the case of previous examples of SLS algorithms for SAT, given a propositional CNF formula $F$ with $n$ variables, we define the search space as the set of all variable assignments of $F$, the solution set as the set of all models of $F$, and a basic neighbourhood relation under which two variable assignments are neighbours if, and only if, they differ exactly in the truth value assigned to one variable (1-flip neighbourhood). As an evaluation function, we use the number of clauses in $F$ unsatisfied under a given assignment.

Note that the variable assignments for a formula with $n$ variables can be easily represented as binary strings of length $n$ by using an arbitrary ordering of the variables and representing the truth values $\top$ and $\bot$ by 1 and 0, respectively. We keep the population size fixed at $k$ assignments.

To obtain an initial population, we use $k$ (independent) iterations of Uninformed Random Picking from the search space, resulting in an initial population of $k$ randomly selected variable assignments. The recombination procedure performs $n/2$ one-point binary crossovers (as defined above) on pairs of randomly selected assignments from $sp$, resulting in a set $sp'$ of $n$ offspring assignments.

The function $mutate(F, sp \cup sp')$ simply flips $\mu$ randomly chosen bits of each assignment in $sp \cup sp'$, where $\mu \in \{1, \ldots, n\}$ is a parameter of the algorithm; this corresponds to performing $\mu$ steps of Uninformed Random Walk independently for all $s \in sp \cup sp'$ (see also Section 1.5). For perturbative local search, we use the same iterative best improvement algorithm as in Example 1.4 (page 47*f.*), which is run until a locally minimal assignment is obtained. The function $localSearch_3(F, sp'')$ returns the set of assignments obtained by applying this procedure to each element in $s''$; the same function is used for $localSearch_1$ and $localSearch_2$.

Finally, $select(F, sp, sp', sp'')$ applies a simple elitist selection scheme, in which the $k$ best assignments in $sp \cup sp' \cup sp''$ are selected to form the next generation (using random tie-breaking, if necessary). Note that this selection scheme ensures that the best assignment found so far is always included in the new population. The search process is terminated when a model of $F$ is found or a fixed number of iterations has been performed without finding a model.

So far, we are not aware of any Memetic Algorithm or Evolutionary Algorithm for SAT that achieves a performance comparable to state-of-the-art SAT algorithms. However, even when just following the general approach illustrated in this example, there are many alternate choices for the recombination, mutation, perturbative local search and selection procedures, few of which appear to have been implemented and studied so far.

The most prominent type of Evolutionary Algorithms for combinatorial problem solving has been the class of *Genetic Algorithms (GAs)* [Holland, 1975; Goldberg, 1989]. In early GA applications, individual candidate solutions were typically represented as bit strings of fixed length. Using this approach, interesting theoretical properties of certain Genetic Algorithms can be proven, such as the well-known Schema Theorem [Holland, 1975]. Yet, this type of representation has been shown to be disadvantageous in practice for solving certain types of combinatorial problems [Michalewicz, 1994]; in particular, this is the case for permutation problems such as the TSP, which are represented more naturally using different encodings.

Besides Genetic Algorithms, there are two other major approaches based on the same metaphor of Evolutionary Computation: *Evolution Strategies* [Rechenberg, 1973; Schwefel, 1981] and *Evolutionary Programming* [Fogel et al., 1966]. All three approaches have been developed independently and, although all of them originated in the 1960s and 1970s, only in the beginning of the 1990s did researchers become fully aware of the common underlying principles [Bäck, 1996]. These three types of Evolutionary Algorithms tend to be primarily applied to different types of problems: While Genetic Algorithms are typically used for solving discrete combinatorial problems, Evolution Strategies and Evolutionary Programming were originally developed for solving (continuous) numerical optimisation problems. For a detailed discussion of the similarities and differences between these different types of Evolutionary Algorithms and their applications, we refer to Bäck [1996].

## 2.5 Further Readings and Related Work

There exists a huge amount of literature on the various SLS methods discussed in this chapter. Since it would be impossible to give a reasonably complete list of references, we refer the interested reader to some of the most relevant and accessible literature, and point out books as well as conference and workshop proceedings that will provide additional material and further references.

There are relatively few books that provide a general introduction to and overview of different SLS techniques. One of these is the book on 'modern heuristics' by Michalewicz and Fogel [2000], which is rather focused on Evolutionary Algorithms but also discusses other SLS methods; another one is the book by Sait and Youssef [1999], which includes the discussion of two lesser-known SLS techniques: Simulated Evolution and Stochastic Evolution. For a tutorial-like introduction to some of the SLS techniques covered in this chapter, such as SA, TS or GAs, we refer to the book edited by Reeves [1993b]. More advanced material is provided in the book on local search edited by Arts and Lenstra [1997], which contains expert introductions to individual SLS techniques as well as overviews on the state-of-the-art of applying SLS methods to various combinatorial problems. The Handbook of Metaheuristics [Glover and Kochenberger, 2002] includes reviews of different SLS methods and additional related topics by leading experts on the respective subjects.

There is a large number of books dedicated to individual SLS techniques. This is particularly true for Evolutionary Algorithms, one of the oldest and most developed SLS methods. Currently, the classics in this field are certainly the early books describing these techniques [Holland, 1975; Goldberg, 1989; Schwefel, 1981; Fogel et al., 1966]; the book by Mitchell [1996] offers a good introduction to Genetic Algorithms. Similarly, there exist a number of books dedicated to Simulated Annealing, including Aarts and Korst [1989] or van Laarhoven and Aarts [1987]. For an overview of the literature on SA as of 1988 we refer to Collins et al. [1988]. A tutorial-style overview of SA is given in Dowsland [1993], and a summary of theoretical results and statistical annealing schedules can be found in Aarts et al. [1997]. For a general overview of Tabu Search and detailed discussions of its features, we refer to the book by Glover and Laguna [1997]. This book also covers in detail various more advanced strategies, such as Strategic Oscillation and Path Relinking, as well as some lesser-known tabu search methods. Ant Colony Optimisation is covered in detail in the book by Dorigo and Stützle [2004].

For virtually all of the SLS methods covered in this chapter, large numbers of research articles have been published in a broad range of journals and conference proceedings. Research on some of the most prominent SLS methods is presented at dedicated conferences or workshop series. Again, Evolutionary Algorithms are particularly well represented, with conference series like GECCO (Genetic and Evolutionary Computation Conference), CEC (Congress on Evolutionary Computation) or PPSN (Parallel Problem Solving from Nature) as well as some smaller conferences and workshops dedicated to specific subjects and issues in the general context of Evolutionary Algorithms. Similarly, The ANTS series of workshops (From Ant Colonies to Artificial Ants: A Series of International Workshops on Ant Algorithms) provides a specialised forum for research on Ant Colony Optimisation algorithms and closely related topics. Many of the

most recent developments and results in these areas can be found in the respective proceedings.

The Metaheuristics International Conference (MIC) series, initiated in 1995, has a broader scope, including many of the SLS methods described in Sections 2.2 and 2.3 of this chapter. The corresponding post-conference collections of articles [Osman and Kelly, 1996; Voß et al., 1999; Hansen and Ribeiro, 2001; Resende and de Sousa, 2003] are a good reference for recent developments in this general area. An extensive, commented bibliography on various SLS methods can be found in Osman and Laporte [1996].

In the operations research community, papers on SLS algorithms now appear frequently in journals such as the INFORMS Journal on Computing, Operations Research, European Journal of Operational Research and Computers & Operations Research. There even exists one journal, the Journal of Heuristics that is dedicated to research related to SLS methods.

Since the early 1990s, SLS algorithms have also been very prominent in the artificial intelligence community, particularly in the context of applications to SAT, constraint satisfaction, planning and scheduling. The proceedings of major AI conferences, such as IJCAI (International Joint Conference on Artificial Intelligence), AAAI (AAAI National Conference on Artificial Intelligence), ECAI (European Conference on Artificial Intelligence), as well as the proceedings of the CP (Principles and Practice of Constraint Programming) conferences and leading journals in AI, including Artificial Intelligence and the Journal on AI Research (JAIR), contain a large number of articles on SLS algorithms and their application to AI problems (we will provide many of these references in Part II of this book).

There are a number of SLS methods that we did not present in this chapter, some of which are closely related to the approaches we discussed. First, let us mention that there exists a number of other algorithms that make use of large neighbourhoods. Prominent examples are Ejection Chains [Glover and Laguna, 1997] and Cyclic Exchange Neighbourhoods [Thompson and Orlin, 1989; Thompson and Psaraftis, 1993]. For an overview of SLS methods based on very large scale neighbourhoods we refer to Ahuja et al. [2002]. Other SLS methods include: Threshold Accepting, a variant of Simulated Annealing that uses a deterministic acceptance criterion [Dueck and Scheuer, 1990]; Extremal Optimisation, which in each step tries to eliminate 'defects' of the current candidate solution and accepts every new candidate solution independent of its solution quality [Boettcher and Percus, 2000; Boettcher, 2000]; and Variable Neighbourhood Search (VNS), which is based on the fundamental idea of changing the neighbourhood relation during the search process. The VNS framework comprises Variable Neighbourhood Descent (see Section 2.1) as well as various methods that can be seen as special cases of Iterated Local Search (see Section 2.3); other VNS algorithms, however, such as Variable Neighbourhood Decomposition

Search (VNDS) [Hansen and Mladenović, 2001b], differ significantly from the SLS methods discussed in this chapter.

Ant Colony Optimisation is only the most successful example of a class of algorithms that are often referred to as swarm intelligence methods [Bonabeau et al., 1999; Kennedy et al., 2001]. A technique that is inspired by Evolutionary Algorithms is Estimation of Distribution Algorithms (EDA) [Baluja and Caruana, 1995; Larrañaga and Lozano, 2001; Mühlenbein and Paaß, 1996]; these algorithms build and iteratively update a probabilistic model of good candidate solutions that is used to generate populations of candidate solutions. Another population-based SLS method is Scatter Search [Glover, 1977; Glover et al., 2002; Laguna and Martí, 2003], which is similar to Memetic Algorithms, but typically uses a more general notion of recombination and differs in some other details of how the population is handled.

Several of these and other SLS methods are described in the Handbook of Metaheuristics [Glover and Kochenberger, 2002] and in the book New Ideas in Optimisation [Corne et al., 1999].

## 2.6 Summary

At the beginning of this chapter we discussed important details and refinements of *Iterative Improvement*, one of the most fundamental SLS methods. *Large neighbourhoods* can be used to improve the performance of iterative improvement algorithms, but they are typically very costly to search; in this situation, as well as in general, *neighbourhood pruning techniques* and *pivoting rules,* such as *first-improvement neighbour selection,* can help to increase the efficiency of the search process. More advanced SLS methods, such as *Variable Neighbourhood Descent (VND)*, *Variable Depth Search (VDS)* and *Dynasearch* use dynamically changing or complex neighbourhoods to achieve improved performance over simple iterative improvement algorithms. Although these strategies yield significantly better performance for a variety of combinatorial problems, they are also typically more difficult to implement than simple iterative improvement algorithms and often require advanced data structures to realise their full benefit.

Generally, the main problem with simple iterative improvement algorithms is the fact that they get easily stuck in local optima of the underlying evaluation function. By using large or complex neighbourhoods, some poor-quality local optima can be eliminated; but at the same time, these extended neighbourhoods are typically more costly or more difficult to search. Therefore, in this

chapter we introduced and discussed various other approaches for dealing with the problem of local optima as encountered by simple iterative improvement algorithms: allowing *worsening search steps*, that is, search steps which achieve no improvement in the given evaluation or objective function, such as in *Simulated Annealing (SA)*, *Tabu Search (TS)* and many *Iterated Local Search (ILS) algorithms* and *Evolutionary Algorithms (EAs)*; dynamically modifying the evaluation function, as exemplified in *Dynamic Local Search (DLS)*; and using adaptive constructive search methods for providing better initial candidate solutions for perturbative search methods, as seen in *GRASP*, *Adaptive Iterated Construction Search (AICS)* and *Ant Colony Optimisation (ACO)*.

Each of these approaches has certain drawbacks. Allowing worsening search steps introduces the need to balance the ability to quickly reach good candidate solutions (as realised by a greedy search strategy) *vs* the ability to effectively escape from local optima and plateaus. Dynamic modifications of the evaluation function can eliminate local optima, but at the same time typically introduces new local optima; in addition, as we will see in Chapter 6, it can be difficult to amortise the overhead cost introduced by the dynamically changing evaluation function by a reduction in the number of search steps required for finding (high-quality) solutions. The use of adaptive constructive search methods for obtaining good initial solutions for subsequent perturbative SLS methods raises a very similar issue; here, the added cost of the construction method needs to be amortised.

Beyond the underlying approach for avoiding the problem of search stagnation due to local optima, the SLS algorithms presented in this chapter share or differ in a number of other fundamental features, such as the combination of simple search strategies into *hybrid methods*, the use of *populations* of candidate solutions and the use of *memory* for guiding the search process. These features form a good basis not only for a classification of SLS methods, but also for understanding their characteristics as well as the role of the underlying approaches (see also Vaessens et al. [1995]).

Our presentation made a prominent distinction between *'simple'* and *hybrid SLS methods*, where hybrid methods can be seen as combinations of various 'simple' SLS methods. In some cases, such as ILS and EAs, the components of the hybrid method are various perturbative SLS processes. In other cases, such as GRASP, AICS and ACO, constructive and perturbative search mechanisms are combined. All these hybrid methods can use different types of 'simple' SLS algorithms as their components, including simple iterative improvement methods as well as more complex methods, such as SA, TS or DLS, and a variety of constructive search methods. In this sense, the hybrid SLS methods presented here are higher-order algorithms that require complex procedural or functional

parameters, such as a subsidiary SLS procedure, to be specified in order to be applied to a given problem.

It is interesting to note that some of the hybrid algorithms discussed here, including ACO and EAs, originally did not include the use of perturbative local search for improving individual candidate solutions. However, adding such perturbative local search mechanisms has been found to significantly improve the performance of the algorithm in many applications to combinatorial problems.

Two of the SLS methods discussed here, ACO and EAs, can be characterised as *population-based search methods*; these maintain a population of candidate solutions that is manipulated and evaluated during the search process. Most state-of-the-art population-based SLS approaches integrate features from the individual elements of the population in order to guide the search process. In ACO, this integration is realised by the *pheromone trails* which provide the basis for the probabilistic construction process, while in EAs, it is mainly achieved through *recombination*. In contrast, all the 'simple' SLS algorithms discussed in Section 2.2 as well as ILS, GRASP and AICS manipulate only a single candidate solution in each search step. In many of these cases, such as ILS, various population-based extensions are easily conceivable [Hong et al., 1997; Stützle, 1998c].

Integrating features of populations of candidate solutions can be seen as one (rather indirect) mechanism that uses *memory* for guiding the search process towards promising regions of the search space. The weights used in AICS serve exactly the same purpose. A similarly indirect form of memory is represented by the penalties used by DLS; only here, the purpose of the memory is at least as much to guide the search away from the current, locally optimal search position, as to guide it towards better candidate solutions. The prototypical example of an SLS method that strongly exploits an explicit form of memory for directing the search process is Tabu Search.

Many SLS methods were originally inspired by natural phenomena; examples of such methods are SA, ACO and EAs, along with several other SLS methods that we did not cover in detail. Because of the original inspiration, the terminology used for describing these SLS methods is often heavily based on jargon from the corresponding natural phenomenon. Yet, closer study of these computational methods, and in particular of the high-performance algorithms derived from them, often reveals that their performance has little or nothing to do with the aspects and features that are important in the context of the corresponding natural processes. In fact, it can be argued that the most successful nature-inspired SLS methods for combinatorial problem solving are those that have been liberated to a large extent from the context of the phenomenon that originally motivated them, and use the new mechanisms and concepts derived from that original context to effectively guide the search process.

Finally, it should be pointed out that in virtually all of the local search methods discussed in this chapter, the use of random or probabilistic decisions results in significantly improved performance and robustness of these algorithms when solving combinatorial problems in practice. One of the reasons for this lies in the diversification achieved by stochastic methods, which is often crucial for effectively avoiding or overcoming stagnation of the search process. In principle, it would certainly be preferable to altogether obliviate the need for diversification by using strategies that guide the search towards (high-quality) solutions in an efficient and reliable way. But given the inherent hardness of the problems to which SLS methods are typically applied, it is hardly surprising that in practice, such strategies are typically not available, leaving stochastic local search as one of the most attractive solution approaches.

## Exercises

2.1 [***Easy***] What is the role of 2-exchange steps in the Lin-Kernighan Algorithm?

2.2 [***Medium***] Design and describe a variable depth search algorithm for SAT.

2.3 [***Easy***] Show that the condition for the independence of a pair of 2-exchange steps to be considered for a complex dynasearch move is necessary to guarantee feasibility of the tour obtained after executing a pair of 2-exchange moves. To do so, consider what happens if we have $i < k < j < l$ for the indices of the 2-exchange moves that delete edges $(u_i, u_{i+1})$, $(u_j, u_{j+1})$ and $(u_k, u_{k+1})$, $(u_l, u_{l+1})$, respectively.

2.4 [***Easy***] Show that Iterative Improvement and Randomised Iterative Improvement can be seen as special cases of Probabilistic Iterative Improvement.

2.5 [***Easy***] Explain the impact on the value of the tabu tenure parameter in Simple Tabu Search on the diversification *vs* intensification of the search process.

2.6 [***Medium***] Which tabu attributes would you choose when applying Simple Tabu Search to the TSP? Are there different possibilities for deciding when a move is tabu? Characterise the memory requirements for efficiently checking the tabu status of solution components.

2.7 [***Medium***] Why is it preferable in Dynamic Local Search to associate penalties with solution components rather than with candidate solutions?

2.8 [***Medium; Implementation***] Implement the Guided Local Search (GLS) algorithm for the TSP described in Example 2.6 (page 84). (You can make use of the 2-opt implementation available at `www.sls-book.net`; if you do so, think carefully about how to best integrate the edge penalties into the local search procedure.) For the search initialisation, use a tour returned by Uninformed Random Picking.

Run this implementation of GLS on TSPLIB instance `pcb3038` (available from TSPLIB [2003]). Perform 100 independent runs of the algorithm with $n$ search steps each, where $n = 3\,038$ is the number of vertices in the given TSP instance. Record the best solution quality reached in each run and report the distribution of these solution quality values (cf. Example 2.1, page 64*f.*).

For comparison, modify your implementation such that instead of Guided Local Search, it realises a variant of the randomised first improvement algorithm for TSP described in Example 2.1 (page 64*f.*) that initialises the search by Uninformed Random Picking.

Measure the distribution of solution qualities obtained from 100 independent runs of this algorithm on TSPLIB instance `pcb3038`, where each run is terminated when a local minimum is encountered. Compare the solution quality distribution thus obtained with that for GLS – what do you observe?

What can you say about the run-time required by both algorithms, using the same termination criteria as in the previous experiments?

2.9 [***Medium***] Show precisely how Memetic Algorithms fit the definition of an SLS algorithm from Chapter 1, Section 1.5.

2.10 [***Medium***] The various SLS methods described in this chapter can be classified according to different criteria, including: (1) the use of a population of solutions, (2) the explicit use of memory (other than just for storing control parameters), (3) the number of different neighbourhood relations used in the search, (4) the modification of the evaluation function during the search and (5) the inspiring source of an algorithm (e.g., by natural phenomena). Classify the SLS methods discussed in this chapter according to these criteria.