2

SLS Algorithms

Stochastic local search (SLS) has become a widely accepted approach to solving hard combinatorial optimisation problems. An important characteristic of many recently developed SLS methods is the fact that they can be applied to many different problems in a rather straightforward way. In this chapter we present some of the most prominent SLS techniques, 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 more complex SLS algorithms such as Ant Colony Optimisation or 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 the different SLS techniques.

2.1 Iterative Improvement (revisited)

In Section 1.5, we introduced Iterative Improvement as one of the simplest, yet reasonably effective SLS algorithms. 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 making 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. When using the standard k-exchange neighbourhoods introduced in Section 1.5, it is easy to see that for growing k, the size of the local neighbourhoods (*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 local neighbourhoods is in $\mathcal{O}(n^k)$, *i.e.*, the neighbourhood size increases exponentially with k.

Generally, larger neighbourhoods contain more and potentially better candidate solutions, and hence they typically offer better chances of facilitating 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.[1] 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), where depending on problem size, even quadratic or cubic time per search step might already be prohibitively high.

Fortunately, the evaluation function values of individual neighbours of a candidate solution can be done more efficiently by caching and updating the

[1] Efficiently searchable exact neighbourhoods exist in a few cases; for example, the simplex algorithm in linear programming is an iterative improvement algorithm which uses a polynomially searchable, exact neighbourhood and is hence guaranteed to find a globally optimal solution.

respective values after each search step rather than computing them from scratch. As a simple example, consider the k-exchange neighbourhood for the TSP, where the length of a candidate tour after a search step can be computed by subtracting the edge lengths of the removed edges from the current tour length and then adding the sum of the length of the newly introduced edges. In many cases, the use of this and similar techniques leads to very significant speedups which are crucial for the success of the respective SLS algorithms in solving combinatorial problems in practice.

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 which are unlikely to, or which 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 pruning techniques, the same 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, which has been shown to be useful in practice, is the use of candidate lists that 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 a candidate list is based on the intuition that high quality solutions will be likely to include short edges between neighbouring vertices (*cf.* Figure 1.1). In fact, the TSP is also an example, where pruning techniques have shown significant impact on local search performance not only for large neighbourhoods but also already for rather small sized neighbourhoods like 2-exchange. SLS techniques for the TSP that make use of this pruning technique will be further discussed in Chapter 8.

Other neighbourhood pruning techniques identify neighbours which provably cannot lead to improvements in the evaluation function based on insights into the structure of a given problem. An example for such pruning

techniques is described by Nowicki and Smutnicki in their Tabu Search approch to the Job Shop Problem [**?**]. We will introduce these techniques in Chapter 9, where SLS algorithms for a variety of scheduling problems will be discussed.

Best Improvement *vs* First Improvement

Another possible method for speeding up the local search is to select the next search step more efficiently. In the context of iterative improvement algorithms, the mechanism for selecting search steps, which implements the step function from Definition 1.5 is also called *pivoting rule* [149], and most widely used neighbour selection mechanisms are the so-called *Best Improvement* and *First Improvement* strategies.

Iterative Best Improvement randomly selects in each search step one of the neighbouring candidate solutions which result in a maximal improvement in the evaluation function. Formally, the corresponding step function can be defined in the following way: Given a search position s, $g^* = min\{g(s') \mid s' \in N(s)\}$ is 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 $step(s)(s') = 1/|I^*(s)|$ 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, 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 $g(s') < g(s)$, *i.e.*, 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, one can also use random ordering. For fixed evaluation orderings, repeated runs of 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 (we

will give some example results illustrating that issue for the TSP in Chapter 8). In this sense, randomised First Improvement inherently leads to a certain diversification of the search process.

As in the case of large neighbourhoods, there is a tradeoff between the number of search steps required for finding a local optimum and the computation time for each search step. Typically, for First Improvement search steps can be computed more efficiently than when using Best Improvement, since especially as long as there are multiple improving search steps from a current candidate solution, only a small part of the local neighbourhood is evaluated by First Improvement. However, the improvement obtained by each step of First Improvement local search is typically smaller than for Best Improvement and therefore, more search steps have to be applied 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. In Section **??** we will show an example in which, as a result, Best Improvement can even be faster than First Improvement.

> **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 $I(s)$ which 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 local search has been applied in a number of SLS algorithms [**?**], but was only recently systematised by the *Variable Neighbourhood Search* (VNS) heuristic [99, 56]. VNS comprises a number of algorithmic approaches including *Variable Neighborhood Descent* (VND), an iterative improvement algorithm which is derived from the general VNS idea. In VND, k neighbourhood relations $N_1, N_2, \ldots, N_k$ are used, which are typically ordered according to increasing size, are used. 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 local search starts again in N_1. In general, there are variants of this basic VND which 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 [56].

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 [75, 82] for the Graph Partitioning Problem and the TSP. 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.

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 underlying mechanism can be understood best by considering a sequence of Hamiltonian paths, *i.e.*, paths which visit each vertex in the given graph G exactly once. Figure 2.1a shows an example in

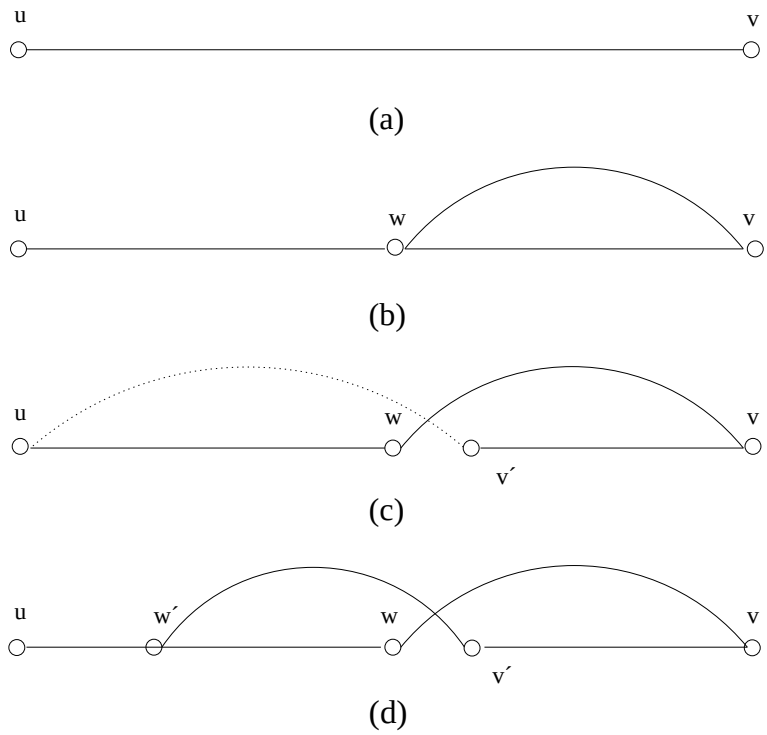

Figure 2.1: Schematic view of a Lin–Kernighan exchange move: (a) gives a Hamiltonian path, (b) a possible δ-path, (c) the next Hamiltonian path (the dotted line should be introduced to close the tour again), and (d) indicates a next possible δ-path.

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 2.1b). The resulting subgraph can also be viewed as a spanning tree of G with one additional edge; it is called a δ-path or a *one tree*. The cycle in this δ-path can be broken by removing a uniquely defined edge (w, v') incident to w; the result is a new Hamiltonian path which 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 dotted edge (v', u) in Fig-

ure 2.1c). Alternatively, a different edge can be added, leading to a new δ-path as indicated in Figure 2.1d.

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 δ-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 δ-path can be obtained with weight smaller than that of the best Hamiltonian cycle found so far. Finally, the minimal weight Hamiltonian cycle s' which has been 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 which alternate between paths and Hamiltonian cycles.

Additional restrictions on the edges that can be added or removed within the construction of complex search steps help to further keep the length of the considered sequence low. For example, in the original LK procedure, any edge that has been added cannot be removed and any edge that has been removed cannot be introduced anymore. This tabu restriction has the effect that a candidate sequence for a complex step is never longer than n.

All VDS algorithms use two types of restrictions, *cost restrictions* and *tabu restrictions*, to limit the time complexity of constructing complex search steps by reducing the constituting simple search steps. Typically, these conditions are very problem specific. Apart from the TSP, a number of other problems have been attacked with considerable success by VDS algorithms. Generally, the implementation of high-performance VDS algorithms requires considerable effort. For example, the best currently available implementations of the LK algorithm for solving large TSP instances represent many person-months of implementation effort (see [98, 72, **?**] for an overview of implementation issues and efficient variants of the basic LK algorithm).

Dynasearch

An interesting variation of the general idea behind VDS forms the basis of Dynasearch [**?**], an iterative improvement algorithm in which complex search steps consist of the best possible combination of mutually *indepen-*

dent simple steps. For example, given a TSP instance and a specific Hamiltonian cycle, two 2-exchange steps, m_1 and m_2 can be considered independent if all vertices adjacent to edges involved in m_1 occur either before or after those adjacent to edges involved in m_2. The neighbourhood explored in Dynasearch is exponentially large w.r.t. instance size, but based on an ingenious dynamic programming algorithm, best neighbours can be identified in polynomial time. The set of independent moves can then be executed in parallel, leading to an overall improvement equal to the sum of the improvements achieved by the simple component steps. Only in the worst case, one complex Dynasearch step consists of a single simple step; but in general, it achieves significant improvements over simple Iterative Improvement. Although Dynasearch is a very recent local search technique, it already has shown excellent performance on some combinatorial optimisation problems, such as the Single Machine Weighted Total Tardiness Problem (a well-known scheduling problem discussed in more detail in Chapter 9).

2.2 'Simple' Stochastic Local Search Methods

The previous section introduced some ways of extending simple exchange neighbourhoods which significantly can enhance the performance of Iterative Improvement. 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 Section 1.5, the simplest technique for achieving this is to use randomised variants of Iterative Improvement or with a restart strategy which re-initialises the search process whenever it gets stuck in a local optimum. In this section we will discuss a number of different techniques 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.

Randomised Iterative Improvement

One of the simplest ways in which iterative improvement algorithms can be extended such that worsening steps can be performed and escaping from lo-

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 $(u \leq wp)$ **then**
    $s' := step_{URW}(\pi, s)$
  else
    $s' := step_{II}(\pi, s)$
  end
  return s'
end *step-RII*

Figure 2.2: Standard implementation of the step function for Randomised Iterative Improvement; $random(0, 1)$ returns a random number between zero and one using a uniform distribution.

cal optima becomes possible, 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 startegy 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 local optima escape. Therefore, it is more appropriate 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 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

$step_{II}(s)(s')$ the step function for the iterative improvement algorithm (*cf.* Section 1.5). As shown in Figure 2.2, the RII step function is typically implemented as a two level choice, where first a probabilistic decision is made 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 end the search after a limit on the CPU time or the number of search steps has been reached; alternatively, the search may be terminated if a given number of search steps has been performed without achieving any improvement.

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 which has a large "basin of attraction" in the sense that many worsening steps may be required to make sure that subsequent improvement steps have a chance of leading into different local optima.

[The following might be moved to Chapter 4, where it would fit well into a general discussion of convergence / probabilistic completeness. This would also allow a slightly more formal treatment.]

In fact, provided that the neighbourhood graph is strongly connected and all solutions have a same neighbourhood size $|N(s)|$, the property of arbitrary long sequences of random walk moves also leads to an easy and intuitive way of proving that RII will eventually find the optimal solution to any combinatorial optimization problem: In this case from an arbitrary solution s exists a search trajectory to a globally optimal solution. For simplicity, let us assume that the shortest possible such search trajectory of length l is taken. Then we can compute the probability of exactly making this trajectorie as $(wp/|N(s)|)^l > 0$. Since the diameter Δ of the neighbourhood graph is an upper bound for l, we can give a worst case estimate for the probability of making the desired trajectory as $(wp \cdot 1/|N(s)|)^{\Delta} > 0$. This consideration holds for an arbitrary solution s and it is easy to see that for the number of visted solutions tending to infinity, the probability of having visited the global optimum tends to one.

Example 2.1: Randomised Iterative Improvement for SAT

RII can easily be applied to SAT by combining the uninformed random walk algorithm presented in Example 1.5 and an iterative improvement algorithm like that of Example 1.5, using the same search space, solution set, neighbourhood relation, and initialisation function as defined there. The only difference is that for this example, we will apply a best improvement local search algorithm instead of the descent method proposed in Example 1.5: At each step the best improvement algorithm flips a variable which leads to a maximal increase in the evaluation function. Note that such a best improvement algorithm need not stop at local optima, because at a local optimum the maximally improving variable flip is a perfectly valid worsening step (actually a least worsening move). The step function is composed of the two step functions for this greedy improvement algorithm and for uninformed random walk as described above: With a probability wp a random neighbouring solution is returned, otherwise with a probability $1 - wp$ a greedy improvement step is applied. We call the resulting algorithm GUWSAT.

Interesting, a slight variation of the GUWSAT algorithm for SAT, called GSAT with Random Walk (GWSAT), has been proven rather successful (see also Chapter 6). 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 occuring in currently unsatisfied clauses. Technically, this is done by first selecting a currently unsatisfied clause c at random. Then, one of the variables appearing in c is flipped, thus effectively forcing c to become satisfied. 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 which 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 of that the probability of accepting a worsening step should depend on the respective deterioration in the 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 the Simulated Annealing algorithms which we will discuss later in this chapter. Let $p(g, s)$ be a function that, based on the evaluation function values in the current neighbourhood of a candidate solution s, determines a probability distribution over neighbouring candidate solutions of s. In each search step, PII simply selects a neighbour of the current candidate solution according to p. 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 Section 1.5 as well as Randomised Iterative Improvement, can be seen as special cases of PII which 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.

Example 2.2: PII / Constant Temperature Annealing for 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, and the same set as our set of feasible candidate solutions. This simply means that we consider each Hamiltonian cycle in G as a valid solution. As a neighbourhood relation, we use a reflexive variant of the 2-exchange neighbourhood defined in 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 which randomly picks a Hamiltonian cycle from S. The step function is implemented as a two-stage process, in which first a neighbour $s' \in N(s)$ is

selected at random, which is then accepted according to the following probability distribution:

$$p_{accept}(T, s, s') = \begin{cases} 1 & \text{if } f(s') < f(s) \\ \exp(\frac{f(s)-f(s')}{T}) & \text{otherwise} \end{cases} \quad (2.1)$$

This distribution is known as the Metropolis distribution. 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.

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 [77] and Cerný [14]. 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 modeled 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 was to solve combinatorial optimisation problems by a process analogous to the physical annealing. In this analogy, the candidate solutions of the given problem instance correspond to the states of the physical system, the evaluation function models the thermal energy of

procedure *step-SA*(π, s, T)
  input *problem instance* π, *candidate solution* s, *temperature* T
  output *candidate solution* s'
  $s' := proposal(\pi, s)$
  $s' := accept(\pi, s, s', T)$
  return s'
end *step-SA*

Figure 2.3: Standard step function for Simulated Annealing; $proposal$ randomly selects a neighbour of s, $accept$ chooses probabilistically between s and s', dependent on temperature T.

the solid, and the globally optimal solutions correspond to the ground state 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 2.2, where in each step first a neighbour s' of s is randomly chosen, 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. One standard choice for this acceptance criterion is a probabilistic choice according to the Metropolis distribution (*cf.* Equation 2.1), which was also used in an early article on the simulation of the physical annealing process [**?**], where the parameter T corresponded to the actual 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 which for each run-time t (typically measured as the number of search steps) determines a temperature value $T(t)$. Annealing schedules are commonly specified by an initial temperature T_0, a temperature update scheme, a number of iterations to be performed at each temperature and a termination condition. Simulated Annealing can use a variety of termination predicates; a specific termination condition, which is often used for SA, is based on the acceptance ratio. In this case, the search process is terminated if the ratio of proposed steps to accepted steps falls below a certain threshold or if no improving candidate

solution could be found for a given number of iterations.

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 [77, 71, 145]. A simple geometric cooling schedule in which temperature is updated as $T := \alpha \cdot T$ has been shown to be quite efficient in many cases [77, 71]. Finally, the number of steps performed at each temperature setting is often chosen as a multiple of the neighborhood size.

Example 2.3: Simulated Annealing for the TSP

The PII algorithm for the TSP specified in Example 2.2 can be easily extended into a Simulated Annealing algorithm (see also [72]). The search space, solution set, and neighbourhood relation are defined as in Example 2.2. 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' with a probability which is given by the Metropolis distribution dependent on T. 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 for five consecutive temperature values no improvement of the evaluation function was 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 procedure, the SA algorithm presented in Example 2.2 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, 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 is computationally expensive. By precomputing a table of function values for a range of argument values, and looking up the values of $\exp(f(s) - f(s')/T)$ from that table, a very significant speedup (in our example about 40%) can be obtained [72].

[The following might be moved to Chapter 4, where it would fit well into a general discussion of convergence / probabilistic completeness. This would also allow a slightly more formal treatment.]

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 [38, 53, 84, 124]. However, the practical usefulness of these results is limited. Theoretically, an infinite number of candidate solutions have to be visited by the SA algorithm to guarantee convergence. Furthermore, practical annealing schedules decrease the temperature much faster than required in the context of the theoretical convergence results, which for such schedules do not apply any longer. Finally, even if SA would be guaranteed to converge towards optimal solutions under realistic conditions, it is not clear whether this notion of convergence would be useful. It should be noted that optimal solutions can always be found in finite time by exhaustive enumeration of the search space. Furthermore, whether or not a search trajectory ends in an optimal solution is typically not relevant for the performance of an SLS algorithm. Instead, since SLS algorithms for optimisation generally keep track of the best candidate solution found during the search process, the important question is whether an optimal solution is encountered at some point during the search. But even for simple SLS algorithms, such as Random Picking or Uninformed Random Walk, optimal solutions are encountered during the search with probability approaching one as arbitrarily high numbers of search steps are performed, yet empirically it is known that these algorithms perform extremely poorly.

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 which systematically utilises memory for guiding the search

process [43, 44, 45, 55]. In the simplest and most widely applied version of TS (which is also called Simple Tabu Search), an iterative improvement algorithm is enhanced with a form of short term memory which enables it to escape from local optima.

Tabu Search uses an aggressive local improvement strategy, like Best Improvement, to select a best possible 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 in 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 a number of previously visited candidate solutions and ruling out any step that would lead back to those. More commonly, reversing recent search steps is prevented by forbidding the reintroduction of solution components (such as edges in the TSP) which have just been removed from the current candidate solution. A parameter tl, 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 neighborhood $N(s)$ 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 neighborhood search technique [61].

Sometimes, this tabu mechanism can forbid search steps leading to attractive, unvisited candidate solutions. Therefore, many tabu search algorithms implement so-called aspiration criteria which specify 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 which lead to an improvement in solution quality compared to the best solution encountered so far.

Figure 2.4 shows the step function which forms the core of Tabu Search. It uses a function $admissibleNeighbours$ to determine the neighbours of the current candidate solution which are not tabu or satisfy the aspiration criterion. In a second stage, a maximally improving step is randomly selected from this set of admissible neighbours.

Example 2.4: Tabu Search for SAT

Using the same definition for the search space, solution set, and neighbour-

procedure *step-TS*(π, s, tl)
input *problem instance* π, *candidate solution* s, *tabu tenure* tl
output *candidate solution* s'
$N' := admissibleNeighbours(\pi, s)$
$s' := selectBest(N')$
return s'
end *step-SA*

Figure 2.4: Standard step function for Tabu Search; $admissibleNeighbours(\pi, s)$ returns the set of admissible neighbours of s, $selectBest(N')$ randomly chooses an element of N' with maximal evaluation function value.

hood relation as in Example 1.5, and the same evaluation function as in Example 1.5, Tabu Search can be applied in a straightforward way to SAT. The search starts with a randomly chosen variable assignment. Each search step corresponds to a single variable flip, which 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 tl 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 which when flipped yields a maximal decrease (or, equivalently, a minimal increase) in the number of unsatisfied clauses is selected 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 with Tabu Search; it has been shown empirically to achieve very good performance on a broad range of SAT problems. When implementing GSAT with Tabu Search, it is crucial to keep the time complexity of the individual search steps minimal, which can be achieved using special data structures and a dynamic caching and updating technique for the evaluation function (this will be discussed in more detail in Chapter 6). 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 time (*i.e.*, the search step number) t_x when it was flipped last and comparing the difference between the current time t (in this case measured as the number of steps performed since the beginning of the search) and t_x to the tabu tenure parameter, tl: if $t - t_x$ is smaller than tl, then the variable x is tabu.

In general, the performance of Tabu Search crucially depends on the setting of the tabu tenure parameter, tl. If tl is chosen too small, cycling 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 tl can only be found empirically and often requires considerable fine tuning. Therefore, several approaches to make the particular settings of tl more robust or to adjust tl dynamically during the run of the algorithm have been introduced. *Robust Tabu Search* [141] achieves an increased robustness of performance w.r.t. the tabu tenure by repeatedly choosing tl randomly from an interval $[tl_{min}, tl_{max}]$. A slight 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 [8] uses the search history to adjust the tabu tenure tl dynamically during the search. In particular, if candidate solutions are repeatedly encountered this is interpreted as evidence that cycling occurs and the tabu tenure is increased. If, on the contrary, no repetitions are found during a sufficiently long period, the tabu tenure is decreased gradually. Additionally, an escape mechanism based on a series of random changes is used to avoid getting trapped in a specific region of the search space.

Generally, the efficiency of Tabu Search can be further increased by using techniques exploiting a *long-term memory* on the search process to achieve an intensification or diversification of the search. Intensification strategies correspond to efforts of revisiting promising regions of the search space, for example by recovering elite candidate solutions, *i.e.* candidate solutions which are amongst the best that have been found in the search process so far. When recovering an elite solution, all tabu restrictions associated to it can be cleared, in which case the search may follow a different search path than when the solution was encountered the first time. Another possibility is to freeze certain solution components and 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 exploring regions of the search space which have not been visited yet. Many long term memory strategies for Tabu Search are based on a frequency memory for the occurrence of solution components. For a detailed discussion of particular diversification and intensification techniques that exploit long-term memory we refer to [45, 46, 47].

Overall, tabu search algorithms are amonst the most successful SLS methods to date. For several problems, TS implementations are among the best known algorithms w.r.t. the tradeoff between solution quality and computation time [108, 144]. Yet, to obtain maximum efficiency, a carefully chosen neighbourhood structure as well as a very significant effort of fine-tuning a potentially large number of parameters and implementation choices is typically required [61]. Despite its empirical success, the theoretical knowledge on Tabu Search is very limited; a convergence proof similar to the one for Simulated Annealing is only available for specific tabu search variants, such as probabilistic tabu search [44, 32], and as discussed for SA, the practical relevance of such results is extremely limited. For another, deterministic TS variant, called *xxx*, recently it could be proved that it implicitly enumerates the search space and stop when all candidate solutions have been visited [**?**]. Unfortunately, in this enumeration process candidate solutions may be revisited several times leading to the fact that a systematic enumeration of the search space by a systematic algorithm gives the same guarantee with less solution evaluations.

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

procedure *step-DLS*(π, s)
  input *problem instance* π*, candidate solution* s
  output *candidate solution* s'
  $g' := g + \sum\{penalty(i) \cdot ind(i) \mid i$ is a solution component$\}$
  $s' := localSearch(\pi, g', s)$
  $updatePenalties(\pi, s')$
  return s'
end *step-DLS*

Figure 2.5: Step function for Dynamic Local Search; $penalty(i)$ is the penalty associated with solution component i, $ind(i)(s)$ is an indicator function for the set of solution components used in a candidate solution s, $localSearch(\pi, g', s)$ is a lower level local search procedure, and $updatePenalties$ is a procedure for updating the solution component penalties. (Details are given in the text.)

of the corresponding solution components are increased, leading 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 have become available. This general approach provides the basis for a number of SLS algorithms which we collectively call Dyanmic Local Search (DLS).

Figure 2.5 shows the step function which forms the core 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 $g(\pi, s)$:

$$g'(\pi, s) = g(\pi, s) + \sum\{penalty(i) \cdot ind(i)(s) \mid i \text{ is a solution component}\}$$

where $ind(i)$ is an indicator function which is used to only include

penalties for solution components actually used in s:

$$ind(i)(s) = \begin{cases} 1 & \text{if } i \text{ is used in } s \\ 0 & \text{otherwise} \end{cases}$$

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 $localSearch(\pi, g', s)$; in many cases, this increase consists of incrementing the penalty by a constant factor, λ. Additionally, some DLS techniques might also decrease the penalties of solution components which are not used in s' (*cf.* [129, 130]).

Penalising all solution components of a locally optimal candidate solution can cause difficulties, if certain solution components which 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 which are least likely to occur in globally optimal solutions. One specific mechanism which implements this idea uses the solution quality contribution of a solution component i at candidate solution s', $f_i(\pi, s')$ to estimate the utility of increasing $penalty(i)$:

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

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.

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 contribution is simply given by their weight. 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 the solution components of s'. This is the case, for example, for the Quadratic Assignment Problem

(*cf.* Section **??**), where DLS algorithms typically use approximations of the actual solution cost contribution [147].

Example 2.5: Dynamic Local Search for the TSP

This example follows the first application of DLS to the TSP, as presented in [147], where the algorithm is called Guided LocalSearch. Given a TSP instance in form of an edge-weighted graph, the same search space, solution set, and 2-exchange neighbourhood is used as in Figure 1.4. The solution components are the edges of $G = (V, E, w)$, 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.

$updatePenalties(\pi, s)$ increments the penalties of all edges of maximal utility contained in candidate solution s by a factor λ, which should be chosen in dependence of the average length of good tours; in particular a setting of

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

where $f(s_{2-opt})$ is the objective function value of a 2-optimal tour, and n is the number of vertices in G, showed good results on a set of standard TSP benchmark instances [147].

The fundamental idea underlying DLS of adaptively modify the evaluation function during a local search process has also been used as the basis of number of other algorithms. GENET [21], an algorithm that adaptively modifies the weight of constraints to be satisfied, has directly inspired the DLS algorithm of [147]. Closely related SLS algorithms, which can be seen as instances of the general DLS algorithm presented here, have been developed for Constraint Satisfaction and SAT, where penalties are typically associated with the clauses of a given CNF formula [132, 100, 15, 36]; this particular approach is also known as clause weighting. Some of the best-peforming SLS algorithms for SAT and more general Boolean Integer Programming problems are based on clause weighting schemes which are inspired by Lagrangean relaxation techniques [148, 130].

2.3 Hybrid Stochastic Local Search Algorithms

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, really is a hybrid SLS algorithm obtained by probabilistically combining standard Iterative Improvement and Uninformed Random Walk (*cf.* Section 2.2). Similarly, many SLS implementations make use of a random restart mechanism, which terminates and restarts the search process from a randomly chosen initial position based on standard termination conditions, which can be seen as a hybrid combination of the underlying SLS algorithm and Random Picking. In this section, we present a number of well-known and very successful SLS algorithms, which can be seen as hybrid combinations of various 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) [**?**], an SLS method which essentially uses the two types of search steps described above alternatingly to perform a walk in the space of local optima w.r.t. a given evaluation function.

Figure 2.6 shows an algorithmic outline for ILS algorithms. As usual, the search process can be initialised in various ways, *e.g.* 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 pro-

```
procedure ILS(π')
    input problem instance π' ∈ Π', objective function f(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))
            ŝ := s''
        end
        s = accept(π', s, s'')
    end
    if ŝ ∈ S' then
        return ŝ
    else
        return ∅
    end
end ILS
```

Figure 2.6: Algorithmic outline of Iterated Local Search (ILS) for optimisation problems. (For details, see text.)

cess is continued. Both functions, $perturb$ and $accept$ can use aspects of the search history; *e.g.,* when the same local optima are repeatedly encountered, stronger perturbation steps might 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 concrete choice of these procedures has a crucial impact on the performance of an ILS algorithm for any application. 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 final 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.* Iterative Improvement on the 3-exchange neighbourhood), 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. (Note however, that results like this generally depend strongly on the desired solution quality and the run-time allowed for the local search process.) While often, iterative improvement methods are used for the subsidiary local search within ILS, it is perfectly possible to use also 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 which will not be immediately undone by the subsequent local search phase, such that the search process can 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 neighborhood than the one used by $localSearch$ may be sufficient for achieving this goal. 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 iterative improvement algorithm takes less steps to identify a local optimum. If the perturbation is too weak, however, the local search will often fall back into the local optimum just visited leading to a stagnation of the search process. 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 [83], 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 nature and effectiveness of the walk in the space of the local optima performed by 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 perform iterative improvement in the space of local optima.

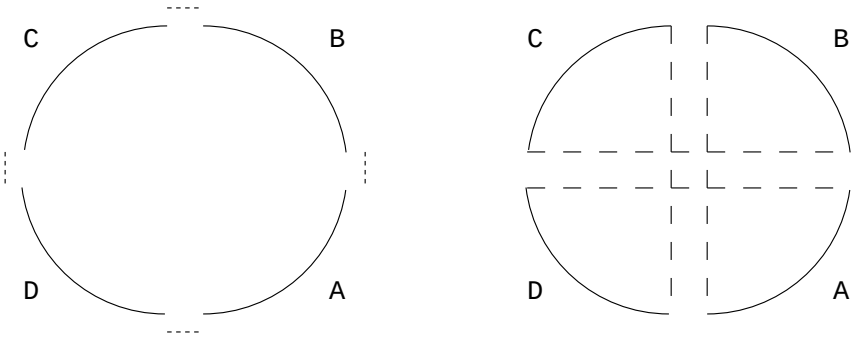

Figure 2.7: Schematic representation of the double-bridge move used in ILK. The four dotted edges are removed and the remaining parts A, B, C, D are reconnected by the dashed edges.

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 over the local optima. Between these extremes, many intermediate choices exist; for example, the Metropolis acceptance criterion known from Simulated Annealing has been used in one of the first classes of ILS algorithms which was called *large step Markov chains* [89]. While all these acceptance criteria are actually Markovian, it has been shown recently that acceptance criteria which take into account aspects of the search history, such as the number of search steps since the last improvement to decide whether to trigger a diversification phase, often help to enhance ILS performance ¸[136].

Example 2.6: Iterated Local Search for the TSP

In this example we describe the Iterated Lin-Kernighan algorithm (ILK), an ILS algorithm which is currently amongst the best performing approximation algorithms for the Travelling Salesman Problem. ILK is based on the same search space and solution set as used as in Example 2.2. The subsidiary local search procedure $localSearch$ is the Lin-Kernighan ariable depth search algorithm (LK) described in Section **??**.

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 2.7; 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 independent of problem size.

Finally, an acceptance criterion is used which always returns the better of the two candidate solutions s and s''. In one of the first detailed studies of the ILK algorithms, this was identified as the best amongst a number of alternative acceptance criteria [70, 72]. Yet, more recent studies revealed (see also Chapter 8) that for very long runs of the algorithm, better solution quality can be obtained by acceptance criteria which take into account aspects of the search history [**?**].

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 lead to frequent re-discoveries and many variants, most of which are known under various names, such as *Large Step Markov Chains*[89], *Chained Local Search* [88], or, when applied to particular algorithms, to specific techniques such as Iterated Lin-Kernighan [72].

ILS algorithms are attractive, not only because of the simplicity of the underlying idea, but also 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 approximation methods for many combinatorial problems, the most prominent application being the Travelling Salesman Problem [72, 88].

Greedy Randomized Adaptive Search Procedures

A standard approach for finding high-quality solutions for a given combinatorial optimisation problem in short time is to apply a greedy construction search method (see also Section 1.4) that, starting from an empty candidate solution, at each construction step adds the best ranked solution component based on a heuristic selection function, and subsequently use a perturbative local search algorithm to improve the candidate solution thus obtained. In

```
procedure GRASP(π')
    input problem instance π' ∈ Π', objective function f(π)
    output solution ŝ ∈ S(π') or ∅
    s := ∅
    ŝ := s
    while not terminate(π', s) do
        s := construct(π')
        s' := localSearch(π', s)
        if (f(s') < f(ŝ))
            ŝ := s'
        end
    end
    if ŝ ∈ S' then
        return ŝ
    else
        return ∅
    end
end GRASP
```

Figure 2.8: Algorithmic outline of GRASP for optimisation problems. (For details, see text.)

practice, this type of hybrid search method often obtains much better solution quality than simple SLS methods initialised at candidate solutions obtained by Uniformed Random Picking (see Section 1.5). Additionally, when starting from a greedily constructed candidate solution, the subsequent perturbative local search process typically takes much less improvement steps to reach a local optimum. To further increase solution quality over a single run, one possible solution is to repeatedley generate greedy solutions and apply local search.

Unfortunately, greedy construction search methods can typically only generate one or a very limited number of different candidate solutions. Greedy Randomised Adaptive Search Procedures (GRASP) [33, 34] 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 2.8 shows an algorithm outline for GRASP. In each iteration of the algorithm, first a candidate solution s is generated using a randomised constructive search method $construct$. Then, a local search method $localSearch$ 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 cardinality restriction, only the k best ranked solution components are included in the RCL, while in the case of a value restriction, the RCL contains only solution components which are within a factor of α from the currently best heuristic value.

The constructive search process performed within GRASP is 'adaptive' in the sense that the heuristic value for each solution component can depend on the components which are already present in the current partial candidate solution. This takes more computation time than using static heuristic values which 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 [**?**]. 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 neighborhood; 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 2.7: GRASP **for SAT**

GRASP can be applied to SAT in a rather straightforward way [122]. 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 which 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 which become satisfied as a consquence 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 randomly selects an atomic variable assignment from the RCL according to a uniform distribution. In [122], a slightly more complex assignment strategy is followed: whenever there are atomic assignments $x_i := v$ for which *(i)* there is at least one currently unsatisfied clause containing only one unassigned variable and which *(ii)* do not leave any clauses unsatisfied whose only unassigned variable is x_i, one of those atomic assignments is selected; only if no such atomic assignment exists, a random element of the RCL is selected instead.

After having generated a full assignment, the candidate solution is improved by a best-improvement variant of the iterative improvement algorithm for SAT from Example 1.5. The search process is terminated after a fixed number of iterations.

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 [122]. While the results were reasonably good at the time the algorithm was first presented, it is now outperformed by more recent SLS variants for SAT.

GRASP has been applied to a large number of combinatorial problems, including MAX-SAT, Quadratic Assignment, and various scheduling problems. 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 α used in value restricted RCLs is

dynamically adapted, or combinations with Tabu Search [**?**]. We refer to [**?**] for a recent overview of research on GRASP.

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 rather obvious and appealing. One way of implementing this idea is to use weights associated with the possible decisions which are made during the construction process. These weights are adapted over multiple iterations of the search process to reflect the search experience mentioned before. This leads to a family of SLS algorithms which we call Adaptive Iterated Construction Search (AICS).

An algorithm outline of AICS is shown in Figure 2.9. At the beginning of the search process, the weights are initialised to some small uniform value τ_0. Each 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 objective function value of s'. As usual, a variety of 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, for h a standard heuristic functions, as used for greedy constructions or in the context of tree searches, can be chosen; alternatively, h can be based on lower bounds on the solution quality of s, similar to the ones 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, irrespectively 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

```
procedure AICS(π')
    input problem instance π' ∈ Π', objective function f(π')
    output solution ŝ ∈ S(π') or ∅
    s := ∅
    ŝ := s
    initWeights(π')
    while (not terminate(π', s)) do
        s := construct(π')
        s' := localSearch(π', s)
        if f(s') < f(ŝ)
            ŝ = s'
        end
        adaptWeights(π', s')
    end
    if ŝ ∈ S' then
        return ŝ
    else
        return ∅
    end
end AICS
```

Figure 2.9: Algorithm outline of Adaptive Iterated Construction Search for optimisation problems. (For details, see text.)

possible and potentially beneficial to use more powerful SLS methods which 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 depending on the given problem domain.

The adjustment of the weights, as implemented in the procedure $adaptWeights$, is typically done by increasing the weights of 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 best candidate solution found in the search process so far 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 2.8: 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 [28, 29], enhanced by an additional perturbative search phase, which in practice improves the performance of the original algorithm. It uses the same search space and solution set as used as in Example 2.2.

Weights $\tau_{ij} \in \mathbb{R}$ 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, τ_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 ϕ by probabilistically selecting from the vertices not contained in ϕ according to the following distribution:

$$p_{ij} = \frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in N'(i)} \tau_{il} \cdot \eta_{ij}} \qquad \text{if } j \in N'(i) \tag{2.2}$$

where $N'(i)$ is the feasible neighborhood of vertex i, *i.e.*, the set of all neighbours of i wich are not contained in the current partial permutation ϕ, and α and β are parameters which together 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 τ_{ij} is done by first decreasing all τ_{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', *i.e.*, for all edges (i, j), the following update is performed:

$$\tau_{ij} := (1 - \rho) \cdot \tau_{ij} + \Delta(i, j, s') \tag{2.3}$$

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 ρ helps to avoid unlimited increased of the weights τ_{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 that are contained in good candidate solutions should be used with higher probability in subsequent constructions than edges with high weight. 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 which 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 by Bresina [13], the Squeeky-Wheel Optimisation algorithm by Joslin and Clements [**?**], and the Adaptive Probing algorithm by Ruml [126]. Furthermore, AICS can be viewed as a particular type of Ant Colony Optimization, a prominent SLS method based on an adaptive iterated construction process involving populations of candidate solutions.

Ant Colony Optimization

Ant colony optimization(ACO) is a population-based search metaphor for solving combinatorial problems which is inspired by the pheromone trail laying and following behaviour of particular ant species [24, 29, 25]. Ants are social insects which live together in colonies. Compared to their rather limited individual capabilities, ants show astonishingly complex cooperative behaviour, including the ability of finding shortest paths from a food source to their home colony [51]. This collective capability becomes possible by means of a communication mechanism based on pheromones (odorous chemical substances) which the ants may lay down in varying quantity to mark, for example, a path. While isolated ants essentially appear to be moving at random, ants which detect previously laid pheromone, follow this pheromone trail with a probability which increases with the intensity of the pheromone. At the same time, when following a pheromone trail, ants may lay down additional pheromone, which can result in a positive feedback loop: the more ants have chosen the pheromone trail in the past, the more ants will follow it in the future.

ACO is a stochastic local search algorithm inspired by aspects of the pheromone trail-following behaviour of real ants; it has been first introduced by Dorigo, Maniezzo and Colorni [28] as a metaphor for solving hard combinatorial problems like the TSP. ACO can be seen as a population-based extension of AICS based on a population of agents (ants) which communicate via a distributed, dynamically changing information (pheromone trail levels) that reflects the collective search experience and is exploited by the ants in their attempts of solving a given problem instance. The pheromone trail levels used in ACO correspond exactly to the weights in AICS.[2]

An algorithmic outline of ACO for static optimisation problems[3] is shown in Figure 2.10. One main difference to the SLS algorithms discussed so far is that now, the search state comprises a population of candidate solutions rather than just a single candidate solution; the size of the population is usually fixed to a number k. Conceptually, the algorithm is usually tought 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, τ_0. In each iteration of ACO, first a population sp of k candidate solutions is generated by a constructive search algorithm $construct$. This construction process is performed by each candidate solution starting with an empty candidate solution and iteratively extending the current partial candidate solution with solution components which are selected probabilistically according to the pheromone trail levels and a heuristic function, h.

Next, in an optional phase, a perturbative local search procedure $localSearch$ can 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.

Finally, the pheromone trail levels are updated based on the candidate solutions in sp' and their respective solution qualities. How this is done ex-

[2] We use the term "trail level" instead of "weight" to be consistent with the literature on Ant Colony Optimization.

[3] Static problems are those for which all the necessary data defining an instance are fixed and completely known before solving the problem and do not change during the solution process.

```
procedure ACO(π')
    input problem instance π' ∈ Π', objective function f(π')
    output solution ŝ ∈ S(π') or ∅
    sp := {∅}
    ŝ := best(π', sp)
    initTrails(π')
    while (not terminate(π', sp)) do
        sp := construct(π', τ, h)
        sp' := localSearch(π', sp)      % optional
        if f(best(π', sp')) < f(ŝ)
            ŝ = best(π', sp')
        end
        updateTrails(π', sp')
    end
    if ŝ ∈ S' then
        return ŝ
    else
        return ∅
    end
end ACO
```

Figure 2.10: Algorithm outline of Ant Colony Optimization for static optimisation problems; $best(\pi', sp)$ denotes the individual from population sp with the best objective function value. (For details, see text.)

actly, differs among the various ACO algorithms. A typical procedure for the pheromone update first uniformly decreases all pheromone trail levels (corresponding to the physical process of pheromone evaporation) by a constant factor, 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 solution 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; here, these can include conditions on the makeup of the current population, sp', such as the variation in solution quality across the elements of sp' or their average

distance.

It should be noted that after the construction phase as well as the perturbative local search phase are executed for each candidate solution independently. Pheromone trail update, however, is based on the solution quality of each element of the population, and hence requires synchronisation. When applying ACO to dynamic optimisation problems, *i.e.*, optimisation problems where aspects 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* presented in [26, 25, **?**], which can be applied to both, static and dynamic combinatorial problems.

Example 2.9: 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 since it was the first application of the ant colony metaphor to solving combinatorial optimisation problems [28, 24, 29].

This algorithm can be seen as a slight extension of the AICS algorithm from Example 2.3. The initialisation of the pheromone trail levels is performed exactly like the weight initialisation in the Adaptive Iterated Construction Search example. The functions $construct$ and $localSearch$ are straightforward extensions of the ones from Example 2.3, which 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 2.3, 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:

$$\Delta(i, j, sp') = \sum \{\Delta(i, j, s') \mid s' \in sp'\} \tag{2.4}$$

$\Delta(i, j, s')$ 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

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 losely as the actions of ants, which walk the edges of the given graph to construct tours (using a memory in order 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 2.3 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 [27, 87, 138, 137].

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 was capable of solving some non-trivial instances of these problems, but did not reach the performance of state-of-the-art algorithms. Subsequently, many other Ant Colony Optimization algorithms have been developped, including Ant Colony System [27], $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System [137, 139], and the the ANTS algorithm [86]. These algorithms differ in important aspects of the search control and introduced advanced features such as pheromone trail level updates or the use of look-ahead during the construction phase, or diversification mechanisms like bounds on the range of possible pheromone trail levels. Some its most prominent applications are to dynamic optimisation problems, such as routing in telecommunications networks, in which traffic patterns are subject to significant changes over time [22]. We refer to the recent overview by Dorigo and Stützle for a detailed account on the different ACO variants and their applications [**?**].

Evolutionary Algorithms

In Ant Colony Optimization we saw a first example for a population-based SLS method; however, the only interaction between the individual elements of a population of candidate solutions was of a very indirect nature. 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 (EA).

In a broad sense, Evolutionary Algorithms (EA) are a large and diverse class of algorithms inspired by models of the natural evolution of species [6, **?**]. They transfer the principle of evolution through mutation, recombination, and selection of the fittest, which leads to the development of species which 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 (totally or partially) replaced by a new set of individuals; 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 individuals 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 which introduces small, often random modifications. Recombination is based on an operation which generates a new individual (called the offspring) by combining information from two or more individuals (called the parents). The most commonly used types of recombination mechanism are called *crossover*; these are originally inspired by a fundamental mechanism in biological evolution of the same name, and essentially assemble 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 which combine parents in a way that the resulting offspring is likely to share desirable properties of the parents while improving over their fitness.

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 which comprises the search state.

For the purpose of this discussion, let us focus on one of the most prominent types of Evolutionary algorithms Genetic Algorithms [62, 48] that – different from other kinds of Evolutionary algorithms which we will briefly address towards the end of this section – are mainly applied to combinatorial problems. 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 Genetic 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 Genetic Algorithms often seem to lack the capability of sufficient search intensification, *i.e.*, the ability to reach high-quality candidate solutions efficiently when a good starting position is given, *e.g.* as the result of a recombination or mutation. Hence, in many cases, the peformance of Genetic Algorithms for combinatorial problems can be significantly improved by adding a local search phase after applying mutation and recombination [12, 140, 104, 142, 92, 93] or by incorporating a local search process into the recombination operator [105]. The class of Evolutionary algorithms thus obtained is usually called *Genetic Local Search* [142, 78, 92] or *Memetic Algorithms* [101, 103, 102].

In Figure 2.11 we show the outline of a generic Genetic Local Search 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, *e.g.*, 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

procedure $GeneticLocalSearch(\pi')$
  input *problem instance* $\pi' \in \Pi'$, *objective function* $f(\pi')$
  output *solution* $\hat{s} \in S(\pi')$ **or** $\emptyset$
  $sp := init(\pi')$
  $\hat{s} := best(\pi', sp)$
  while (**not** $terminate(\pi', sp)$) **do**
    $sp' := recomb(\pi', sp)$
    $sp'' := mutate(\pi', sp, sp')$
    $sp''' := localSearch(\pi', sp', sp'')$
    if $f(best(\pi', sp' \cup sp'' \cup sp''')) < f(\hat{s})$
      $\hat{s} = best(\pi', sp' \cup sp'' \cup sp''')$
    end
    $sp := select(\pi', sp, sp''')$
  end
  if $\hat{s} \in S'$ **then**
    return $\hat{s}$
  else
    return $\emptyset$
  end
end $GeneticLocalSearch$

Figure 2.11: Algorithm outline of Genetic Local Search for optimisation problems; $best(\pi', sp)$ denotes the individual from a population sp with the best objective function value. (For details, see text.)

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 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 "inherits" certain subsets of solution components from its parents. One of the most commonly used recombination mechanisms is the one-point bi-

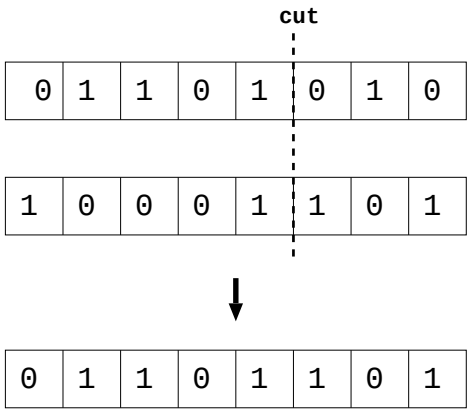

Figure 2.12: Schematic representation of the one-point crossover operator.

nary 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 $\{1, 2, \ldots, n-1\}$. 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 2.12).

In the first GA applications, individual candidate solutions were typically represented as bit strings of fixed length [62]. Using this approach, interesting theoretical properties of certain Genetic Algorithms can be proven, one of the most prominent of these being the so-called Schema Theorem [62]. Yet, this type of representation proved to be disadvantageous in practice for solving certain types of combinatorial problems [94]; in particular, this is the case for permutation problems such as the TSP, which are represented more naturally using different encodings. 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, where the solution candidates are represented by permutations of the vertex set which are written as vectors $(u_1, u_2, \ldots u_n)$. Usingq a simple one-point crossover operation, as defined above, as the basis for recombination obviously leads to vectors which 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.

The role of function $mutate(\pi, sp')$ is to introduce relatively small perturbations in the individuals in sp'. Typically, these perturbations are of stochastic nature, and they are performed independently for each individual in sp', where the amount of perturbation applied is controlled by a parameter called the mutation rate. It should be noted that permutation need not be applied to all individuals of sp'; instead, a subsidiary selection function can determine which candidate solutions are to be mutated. Until rather recently, the role of mutation compared to recombination for the performance of Genetic Algorithms has been widely underestimated [6].

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

Finally, the selection function used for determining the individuals which form the next generation sp of candidate solutions, typically considers both elements of the original population, as well as the newly obtained candidate solutions in sp''' 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 2.10: A Genetic Local Search 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 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 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 crossovers (as defined above) on pairs of randomly selected assignments from sp, resulting in a set sp' of n offspring assignments.

$mutate(\pi', sp')$ simply flips μ randomly chosen bits of each assignment in sp', where $\mu \in \{1, \ldots n\}$ is a parameter of the algorithm; this corresponds to performing μ steps of Uninformed Random Walk independently for all $s \in sp'$ (see also Section 1.5). For the perturbative local search procedure, we use the same best improvement algorithm as in Example 2.2, which is run until a locally minimal assignment is obtained. Function $localSearch(\pi', s', s'')$ returns a set s''' of assignments obtained by applying this procedure to each element in s'' (ignoring the elements in s').

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

So far, we are not aware of any Genetic Local Search or Evolutionary Algorithm for SAT which 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.

Besides Genetic Algorithms, there are two other major approaches based on the same metaphor of Evolutionary Computation: Evolution Strategies [118, 131] and Evolutionary Programming [35]. All three approaches have been developed independently and, although all of them originated in the 1960s and 1970s, only in the beginning of the nineties researchers became fully aware of the common underlying principles [6]. These three types of Evolutionary Algorithms tend to be primarily applied to different types of problems: while Genetic Algorithms are rather used for solving discrete, combinatorial problems, Evolution Strategies and Evolutionary Programming were originally intended to solve numerical parameter optimisation problems. For a detailed discussion of the similarities and differences between these different types of Evolutionary algorithms and their applications, we we refer to the book by Bäck [6].

2.4 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 which will provide additional material and further references.

There are relatively few books which provide a general introduction to and overview of different SLS techniques. One of these is the recent book by Michalewicz and Fogel on "Modern Heuristics" [95], which mainly focuses on Evolutionary Algorithms; another one is the book by Sait and Youssef [128] which includes the discussion of two less known SLS techniques: Simulated Evolution and Stochastic Evolution. For a tutorial-like introduction into some of the SLS techniques covered in this chapter, such as SA, TS, or GA, we refer to the book edited by Reeves [119]. Slightly more advanced material is provided in the book on local search edited by Aarts and Lenstra [2], 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.

There is a large number of books dedicated to individual SLS techniques. This is particularly true for Evolutionary Algorithms, one of the old-

est and most developped SLS methods. Currently, the classics in this field are certainly the early books describing these techniques [62, 48, 131, 35]. A recent, very readable introduction into Genetic Algorithms is given by the book by Mitchell [**?**]. Similarly, there exist a number of books dedicated to Simulated Annealing, including [1] or [145]. For an overview of the literature on SA as of 1988 we refer to [18]. A more recent, tutorial style overview of SA is given in [31] and a summary of theoretical results and statistical annealing schedules in given in [**?**]. For general overviews on Tabu Search and detailed discussions of its features, we refer to the book by Glover and Laguna [47]. This book also covers in detail various more advanced strategies such as Strategic Oscillation or Path Relinking, as well as some less known tabu search techniques.

For virtually all of the SLS techniques 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 on dedicated conferences or workshop series. Again, Evolutionary Algorithms is 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 more recent ANTS (From Ant Colonies to Artificial Ants: A Series of International Workshops on Ant Algorithms) series of workshops provides a specialised forum for research on Ant Colony Optimization algorithms and their applications. 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 techniques described in Sections 2.2 and 2.3 of this chapter. The corresponding post-conference collections of articles [109, 146, 57] are a good reference for recent developments in this general area. Additionally, an extensive, commented bibliography on various SLS algorithms is found in [110].

In the Operations Research community, now frequently papers on SLS algorithms appear in journals like the INFORMS Journal on Computing, Operations Research, European Journal of Operational Research. There even exists one Journal, the Journal of Heuristics, which is dedicated to

research on SLS algorithms.

Since the early 1990s, SLS algorithms have also been very prominent in the Artificial Intelligence community, particularly applications to SAT, Constraint Satisfaction, Planning, and Scheduling Problems. 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", 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, but some of which are closely related to the approaches we discussed. These include, for example, *Swarm Intelligence* techniques [10] the most successful example of which is, in fact, Ant Colony Optimization, Threshold Accepting [**?**], Extremal Optimisation [**?**], Scatter Search [42, **?**], Ejection Chains [47], and many others. Several of these and additional SLS techniques are described in the book *New Ideas in Optimisation*, edited by Corne, Dorigo, and Glover [20].

[hh: add brief comments on Markov-Chain Monte-Carlo methods, and Expectation-Minimisation (EM) – TODO(hh)]

[hh: possibly add brief comments on and mention some references to local search methods for numeric optimisation – DISCUSS]

2.5 Summary

At the beginning of this chapter we revisited the Iterative Improvement algorithm introduced in Chapter 1 and discussed important details and refinements. Large neighbourhoods can be used to improve the performance of Iterative Improvement, 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 speed up the search process. Advanced strategies, such as Variable Neighbourhood Descent, Variable Depth Search, 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 cominatorial 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, *i.e.*, search steps which achieve no improvement in the given evaluation or objective function, such as in Simulated Annealing (SA), Tabu Search, and many Iterated Local Search and Evolutionary Algorithms; dynamically modifying the evaluation function, as exemplified in Guided Local Search; and using adaptive constructive search methods for providing better initial candidate solutions for perturbative search methods, as seen in GRASP and, Adaptive Iterated Construction Search, and Ant Colony Optimization.

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 introduce 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 [143]).

Our presentation made a prominent distinction between 'simple' and hybrid SLS methods, where hybrid methods can be seen as combinations of various 'simple' SLS techniques. In some cases, such as ILS and EA, 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 techniques 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, which 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 EA, 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 EA, can be characterised as population-based search techniques; these maintain a population of candidate solutions which 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 EA, 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 [63, 136].

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, then 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 search is TS.

Many SLS algorithms actually were inspired by some phenomena occuring in nature. Examples of such SLS algorithms are SA, ACO, EA, and several others that we did not introduce explicitely. Because of this inspiration by nature, it is often the case that the terminology used in such algorithms uses a heavy jargon from the corresponding natural phenomenon. Yet, actual implementations of such algorithms and especially those implementations showing state-of-the-art performance are typically very far from their inspiring source. The main contribution of such methods is the introduction of new, abstract concepts for guiding the search.

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 stagnation of the search process. In principle, knowing the right strategy for guiding the search towards (high quality) solutions would achieve the same goal more efficiently; but given the inherent hardness of the problems to which SLS methods are typically applied, it is hardly surprising that in practice, devising such strategies for guiding the search process reliably and efficiently is typically not possible.

2.6 Exercises

Exercise 2.1 (Medium) Which role do 2-exchange steps play in the Lin-Kernighan procedure?

Exercise 2.2 (Easy) Show that Iterative Improvement and Randomised Iterative Improvement can be seen special cases of Probabilistic Iterative Im-

provement.

Exercise 2.3 (Medium) What tabu attributes would you choose when applying 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.

Exercise 2.4 (Medium) Why is it preferable in Dynamic Local Search to associate penalties with solution components rather than with candidate solutions?

Exercise 2.5 (Medium) Design a population-based extension of Iterated Local Search and describe its application to the TSP.

Exercise 2.6 (Easy) Discuss similarities and differences between Ant Colony Optimisation and Genetic Local Search.