3

Generalised Local Search Machines

In this chapter, we introduce Generalised Local Search Machines (GLSMs), a formal framework for stochastic local search algorithms. The underlying idea is that most efficient SLS algorithms are obtained by combining simple (pure) search strategies using a control mechanism; in the GLSM model, the control mechanism is essentially realised by a non-deterministic finite state machine (FSM). GLSMs provide a uniform framework capable of representing most modern SLS algorithms in an adequate way; they facilitate representations which clearly separate between search and search control.

After defining the basic GLSM model we establish the relation between our definition of stochastic local search algorithms and the GLSM model. Next, we discuss several aspects of the model, such as structural GLSM types, transitions types, and state types; we also show how various well-known SLS algorithms are represented in the GLSM framework. Finally, we address extensions of the basic GLSM model, such as cooperative, learning, and evolutionary GLSMs.

3.1 The Basic GLSM Model

Finite State Machines (FSMs) are one of the most prominent formal models in the computing sciences [**?**]. They can be seen as abstractions of systems that can be characterised by a finite number of states. Starting in a specific state, the current state of an FSM can change as a response to certain events, *e.g.* a signal received from its environment; these changes in system state are called state transitions. FSMs are used to model algorithms in many domains, *e.g.*, in hardware design or state-of-the-art computer games.

Intuitively, a Generalised Local Search Machine (GLSM) for a given problem class Π is an FSM each state of which corresponds to a simple local search strategy for instances of Π. The machine starts with an initial state z_0 and executes one step of the local search method associated with the current state. Then, according to a transition relation Δ, a new state is selected in a nondeterministic manner. This is iterated until a termination condition is satisfied; this termination condition typically depends on the search state (*e.g.*, evaluation or objective function value), search history (*e.g.*, number of local search steps or state transitions performed), or resource bounds (*e.g.*, total CPU time consumed). In the case of SLS algorithms, the termination predicate used typically depends more on the specific application context than on the underlying search strategy (*cf.* Section 1.5, page 31). Therefore, for simplicity's sake, a termination condition is not explicitly included in our GLSM model; instead, we consider it as part of the execution time behaviour. (Note, however, that analogously to standard FSM models, termination conditions could easily be included in our GLSM model in the form of absorbing final states, *i.e.*, states upon reaching which the machine halts and the search process is terminated, and appropriate state transitions.)

Definition 3.1 (Generalised Local Search Machine)

A *Generalised Local Search Machine (GLSM)* is formally defined as a tuple $M = (Z, z_0, \Delta, \sigma_Z, \sigma_\Delta, \tau_Z, \tau_\Delta)$ where Z is a set of *states* and $z_0 \in Z$ the *initial state*. $\Delta \subseteq Z \times Z$ is the *transition relation for M*; σ_Z and σ_Δ are sets of *state types* and *transition types*, respectively, while $\tau_Z : Z \mapsto \sigma_Z$ and $\tau_\Delta : \Delta \mapsto \sigma_\Delta$ associate the corresponding types to states and transitions. We call $\tau_Z(z)$ the *type of state z* and $\tau_\Delta((z_1, z_2))$ the *type of transition* (z_1, z_2), respectively. $\square$

It is useful to assume that σ_Z, σ_Δ do not contain any types that are not associated with at least one state or transition of the given machine (*i.e.*, τ_Z, τ_Δ are surjective). In this case, we define the *type of machine* M by $\tau_M := (\sigma_Z, \sigma_\Delta)$. However, we allow for several states of M having the same type (*i.e.*, τ_Z need not be injective). Note that we do not require that each of the states in Z can be actually reached when beginning in state z_0; as we will shortly see, it is generally not trivial to decide this form of reachability. Thus, by adding unreachable states, the type of a given machine can be extended in an arbitrary way such that for any two GLSMs M_1, M_2, one can always find functionally equivalent models M_1', M_2' of the same type (*i.e.*, $\tau_{M_1'} = \tau_{M_2'}$).

Example 3.1: Simple 3-State GLSM

The following GLSM models a hybrid SLS algorithm which after initialising the search (state z_0), probabilistically alternates between two search strategies (states z_1 and z_2):

$$M = (\{z_0, z_1, z_2\}, z_0, \Delta, \sigma_Z, \sigma_\Delta, \tau_Z, \tau_\Delta)$$

where

$$\begin{aligned}
\Delta &= \{(z_0, z_1), (z_1, z_2), (z_2, z_1), (z_1, z_1), (z_2, z_2)\} \\
\sigma_Z &= \{z_0, z_1, z_2\} \\
\sigma_\Delta &= \{\text{PROB}(p) \mid p \in [0,1]\} \\
\tau_Z(z_i) &= z_i, \quad i \in \{1,2,3\} \\
\tau_\Delta((z_0, z_1)) &= \text{PROB}(1.0) \\
\tau_\Delta((z_1, z_2)) &= \text{PROB}(p_1) \\
\tau_\Delta((z_2, z_1)) &= \text{PROB}(p_2) \\
\tau_\Delta((z_1, z_1)) &= \text{PROB}(1-p_1) \\
\tau_\Delta((z_2, z_2)) &= \text{PROB}(1-p_2)
\end{aligned}$$

Intuitively, transitions of type PROB(p) from the current state will be executed with a probability p. The generic transition type PROB(p) formally corresponds to unconditional, probabilistic transitions with an associated

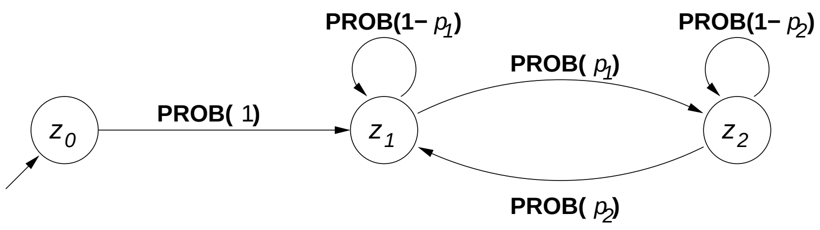

Figure 3.1: Simple 3-state GLSM, representing a hybrid SLS algorithm that, after initialising the search process, probabilistically alternates between two search strategies.

transition probability p; it will be presented in more detail later in this section. This type of GLSM can, for instance, be used to represented a variant of Randomised Iterative Improvement (see also Section 3.3).

> **Remark:** As long as the meaning is clear from the context, we use the same symbols for denoting states and their respective types.

We will usually specify GLSMs more intuitively using a standard graph representation for the finite state machine part. In this graphic notation, GLSM states are represented by circles labelled with the respective state types, and state transitions are represented by arrows, labelled with the respective transition types; the initial state is marked by an incoming arrow that does not originate from another state. The graphic representation for the GLSM from Example **??** is shown in Figure 3.1.

GLSM Semantics

Having formally defined the structure and compontents of a Generalised Local Search Machine, we now need to specify the semantics of this formal model, *i.e.*, the way it works when applied to a specific instance π of a combinatorial problem Π. The following example illustrates the intuitive function of a simple GLSM.

Example 3.2: Semantics of Simple 3-State GLSM

The semantics of the simple 3-state GLSM from Example **??** can be described intuitively as follows: For a given problem instance π, the local search process is initialised by setting the machine to its initial state z_0 and executing one local search step corresponding to state type z_0 (this step is designed to initialise the subsequent local search process, *e.g.*, by randomly generating or selecting a candidate solution). With a probability of one, the machine then switches to state z_1, executing one step of the local search strategy corresponding to state type z_1. Now, with a probability of p_1, the machine switches to state z_2, performing one local search step of type z_2; otherwise it remains in z_1 and does a z_1-step. When in z_2, an analogous behaviour is observed. This results in a local search process which repeatedly and nondeterministically switches between z_1 and z_2-steps. However, only once in each run of the machine, a z_0-step is performed, and that is at the very beginning of the local search process. As discussed above, a number of different termination criteria can be used, but these are not part of the GLSM model.

Remark: For the sake of simplicity, we will generally assume that different state types always represent different local search strategies.

When running a GLSM M for a given instance π of problem Π, the operation of M is uniquely characterised by its state and the search space position (candidate solution) of the local search process realised by M for any given point in time (where time is measured in search steps). This information can be captured in the form of two functions, the *actual state trajectory* and *actual search trajectory of* M, respectively, which specify the machine state and search position of M over time.

However, due to the inherent stochasticity of GLSMs and the SLS algorithms they model, generally each actual search trajectory will only be observed with a certain probability. Therefore, we have to use probability distributions over machines states and search positions when formalising the semantics of a GLSM. This leads to the notions of *state trajectory* and *search trajectory*, which are functions defining the probability distribution over machine states and search positions, respectively, over time.

Obviously, the semantics of any GLSM crucially depend on the search strategies represented by its states and on the nature of the transitions between these states. We will further discuss transition types in the next section and practically relevant examples of state types will be given in Section 3.3.

In Depth: Formal Definition of GLSM Semantics

To formally define the semantics of a GLSM as motivated above, we assume that the semantics of each state type τ are defined and given in form of a search transition function $\gamma_\tau : S \mapsto \mathcal{D}(S)$, where S denotes the set of positions in the search space induced by the given problem instance, and $\mathcal{D}(S)$ represents the set of distributions over S. Intuitively, γ_τ determines for each position in the search space the resulting position after one τ-step has been performed; it corresponds to the step function of the local search strategy associated with τ and can be defined functionally or procedurally (see also Chapter **??**).

We further need the functions $\gamma_Z : S \times Z \mapsto \mathcal{D}(Z)$ which model the direct transitions between states of the GLSM. These are defined on the basis of the transitions from a given state s and their respective types.

The $\gamma_Z(s, z)$ are given by the specific transition types of $\tau((z, z'))$;
for $\tau((z_i, z_k)) =$ PROB$(p_{i,k})$, $\gamma_Z(s, z_i) = D''_z$ with $D''_z(z_k) = p_{i,k}$.

Remark: To facilitate a concise formulation of the definitions, we often use the functional form of discrete probability distributions; thus for $D = \{\ldots, (e, p), \ldots\}$, $D(e)$ is equal to p and denotes the probability of event e.

The direct state transition functions γ_Z can be generalised into functions $\gamma'_Z : \mathcal{D}(S) \times \mathcal{D}(Z) \mapsto \mathcal{D}(Z)$, which map distributions of search positions and GLSM states onto GLSM state distributions.

$\gamma'_Z(D_s, D_z) = D'_z$
with $D'_z(z_k) = \sum_{s \in S, z \in Z} D''_z(z_k) \cdot D_s(s) \cdot D_z(z)$
where $D''_z = \gamma_Z(s, z)$

Based on the functions γ_τ and γ_Z, we next define the direct search transition function $\gamma : S \times Z \mapsto \mathcal{D}(S)$ which determines for a given search position and a state distribution the search position distribution after one step of the GLSM.

$\gamma(s_k, z) = D''_s$
with $D''_s(s_j) = P(\textit{go from state } z \textit{ to } z') \cdot P(\textit{in state } z' \textit{ go from } s_k \textit{ to } s_j)$.

> $P(\textit{go from state } z \textit{ to } z') = D'_z(z)$
> where $D'_z = \gamma_Z(s_k, z)$
>
> $P(\textit{in state } z' \textit{ go from } s_k \textit{ to } s_j) = D'_s(s_j)$
> where $D'_s = \gamma_{\tau(z')}(s_k)$

Again, this is generalised into the corresponding function $\gamma' : \mathcal{D}(S) \times \mathcal{D}(Z) \mapsto \mathcal{D}(S)$.

> $\gamma'(D_s, D_z) = D'_s$
> with $D'_s(s_k) = \sum_{s \in S, z \in Z} D''_s(s_k) \cdot D_s(s) \cdot D_z(z)$
> where $D''_s = \gamma(s, z)$.

Finally, we inductively define the state and search trajectories $\gamma^*_Z : \mathbb{N} \mapsto \mathcal{D}(Z)$ and $\gamma^* : \mathbb{N} \mapsto \mathcal{D}(S)$. The interlocked inductive definitions reflect the operation of a GLSM, where in each step, the next search position and the next state are determined based on the current state and position. The initial search position $s_0 \in S$ can be arbitrarily chosen, since usually the state distribution determined in the first step does not depend on s_0.

> γ^* and γ^*_Z are defined inductively by:
> $\gamma^*(0) = D_0$ with $D_0(s_0) = 1, \forall s_k \in S - \{s_0\} : D_0(s_k) = 0$
> $\gamma^*(t+1) = \gamma'(\gamma^*(t), \gamma^*_Z(t))$
>
> $\gamma^*_Z(0) = z_0$
> $\gamma^*_Z(t+1) = \gamma'_Z(\gamma^*(t), \gamma^*_Z(t))$

It is now rather easy to formally define the notion of an *actual search trajectory* as a function $\delta^* : \mathbb{N} \mapsto S$. To this end, we use a function $draw(D)$ that for any probability distribution D randomly selects an element e' from its domain such that $P(draw(D) = e') = D(e')$. Based on this, we define functions $\delta : S \times Z \mapsto S$ and $\delta_Z : S \times Z \mapsto Z$ that for each given position and state randomly determine the position and state after one step of the GLSM:

> $\delta(s, z) = draw(\gamma'(s, z))$
> $\delta_Z(s, z) = draw(\gamma'_Z(s, z))$

Assuming that the given GLSM is started in state z_0 and at search position s_0, we now define the actual position and state trajectory by another double induction:

> $\delta^* : \mathbb{N} \mapsto S$ and $\delta^*_Z : \mathbb{N} \mapsto Z$ are defined inductively by:
> $\delta^*(0) = s_0$
> $\delta^*(t+1) = \delta(\delta^*(t), \delta^*_Z(t))$
>
> $\delta^*_Z(0) = z_0$
> $\delta^*_Z(t+1) = \delta_Z(\delta^*(t), \delta^*_Z(t))$

Note the similarity between these definitions and the ones for γ^* and γ^*_Z; the only difference is in the use of the $draw$ function to randomly select elements from the underlying probability distributions.

GLSMs as Models for Local Search Algorithms

The GLSM model has been introduced as a means for representating SLS strategies, particularly hybrid search techniques, more adequately. In particular, in the context of the formal definition of SLS Algorithms from Section 1.5 (page 31), GLSMs allow a more detailed and explicit representation of the step function.

Each GLSM, however, still realises a local search scheme and can therefore be described using the components of such a scheme. The notions of search space and solution set are not part of the GLSM model. This is mainly because both are not only problem- but actually instance-specific, they thus form the environment in which a given GLSM operates. Consequently, to characterise the behaviour of a GLSM when applied to a given instance, both the machine definition and this environment are required. Likewise, as explained before, we consider the termination predicate of an SLS algorithm as a part of the respective GLSM's operating environment. The initial distribution is also not an explicit part of the GLSM model. The reason for this is the fact that the initial state, which is part of the model, can easily be used to generate arbitrary initial distributions of search positions. The general local search scheme's step function is what is actually realised by the states of the GLSM and the finite control mechanism as given by the state transition relation.

The remaining component of the general local search scheme, the neighbourhood relation, generally does not have a direct representation in the GLSM model. This is because for a GLSM, each state type can be based on a different neighbourhood relation. However, for each GLSM as a whole, a neighbourhood relation can be defined by constructing a generalised relation which contains the neighbourhood relation for each state type as a subset.

At the same time, each GLSM state represents a local search algorithm of its own. While all these share the same search space and solution set, they

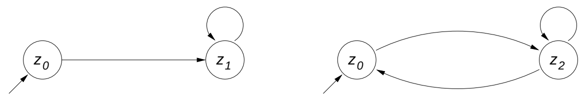

Figure 3.2: Sequential *(left)* and alternating *(right)* 1-state+init GLSM.

generally differ in their neighbourhood relations and step functions. Initial distributions for the individual local search algorithms are not needed since they are defined by the context in which a GLSM state is activated. Furthermore, the termination predicates for this subsidiary local search algorithms are given by the transitions to other states and their respective types.

3.2 Machine, Transition, and State Types

One of the major advantages of using the GLSM model for characterising hybrid local search schemes is the clear distinction between search control and the actual search strategies thus facilitated. By abstracting from state and transition types, and thus concentrating on the structure of the search control mechanism alone, GLSMs can be categorised into the following structural classes:

1-state machines: This is the minimal form of a GLSM. Since initialisation of the local search process has to be realised using a GLSM state, 1-state machines realise a very basic form of local search which basically only picks search space positions without doing actual local search. Consequently, the practical relevance of this machine type is extremely limited. It can, however, be used for analytical purposes, *e.g.* as a reference model when evaluating other types of GLSMs.

1-state+init machines: These machines have one state for search initialisation and one working state, realising the search strategy. There are two sub-types of these machines: *1-state+init sequential machines* visit the initialisation state only once, while *alternating machines* might visit it again in the course of the search process, causing re-initialisations. The structure of these machine models is shown in Figure 3.2.

2-state+init sequential machines: This machine type has three states, one of which is an initialisation state that is only visited once while the other

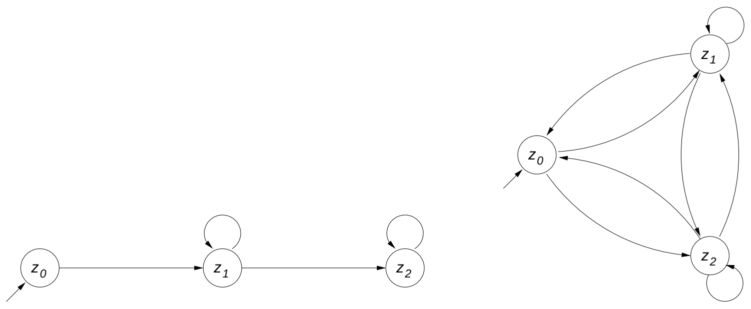

Figure 3.3: Sequential *(left)* and alternating *(right)* 2-state+init GLSM.

two are working states. However, once the machine has switched from the first state to the second, it will never switch back (see Figure 3.3, left side); analogously, the GLSM switches from the second to the third state only once. Thus, each trajectory of such a machine can be partitioned into three phases: one initialisation step, a number of steps in the first working state and a number of steps in the second working state.

2-state+init alternating machines: Like the 2-state+init sequential machine, this machine type has one initialisation state and two working states. Here, however, arbitrary transitions between all states are possible (see Figure 3.3, right side). An interesting sub-type of these machines is given by the special case in which the initial state is only visited once, while the machine might arbitrarily switch between the two working states. Another sub-type that might be distinguished is a uni-directional cyclic machine model which allows the three states to be visited only in one fixed order.

Of course, the categorisation can easily be continued in this manner by successively increasing the number of working states. However, as we will see later, to describe state-of-the-art stochastic local search algorithms, three-state-machines are often sufficient. We therefore conclude this categorisation with a brief look at two potentially interesting cases of the k-state+init machine types:

k-state+init sequential machines: As a straightforward generalisation of the sequential 2-state+init machines, in this machine type we have $k + 1$ states which are visited in a linear order. Consequently, after a machine state has been left, it will never be visited again (see Figure 3.4, left side).

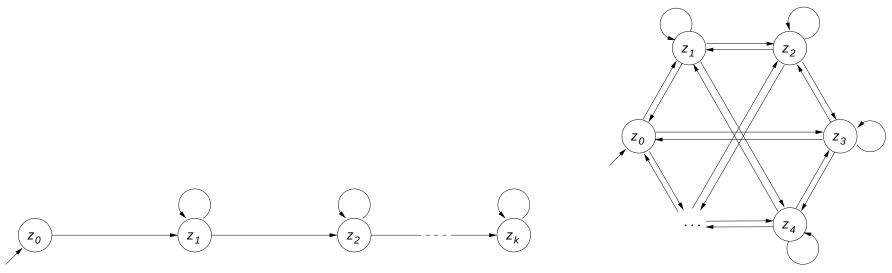

Figure 3.4: Sequential *(left)* and alternating *(right)* k-state+init GLSM.

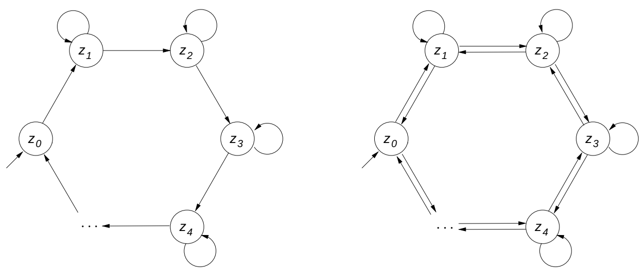

Figure 3.5: Uni-directional *(left)* and bi-directional *(right)* cyclic k-state+init GLSM.

k-state+init alternating machines: These machines allow arbitrary transitions between the $k + 1$ states and may therefore re-initialise the search process and switch between strategies as often as desired (see Figure 3.4, right side). Some special cases worth noting are the uni- and bi-directional cyclic machine models which allow to switch between states in a cyclic manner. In the former case, the cyclic structure can be traversed only in one direction, in the latter case the machine can switch from any state to both its neighbouring states (see Figure 3.5).

This categorisation of machines according to their structure is useful for characterising the structural aspects of the search control as realised by the GLSM model. Obviously, this is a very high-level view of GLSMs which can be refined in many ways, but nevertheless it will prove to be useful for capturing some fundamental differences between various stochastic local search schemes.

Transition Types

In refining the structural view of GLSMs given above, we next focus on transition types. As mentioned before, properties of the transition types are used as a basis for defining GLSM semantics. Here, we will introduce transition types in terms of a hierarchy of increasingly complex or expressive types and describe the semantics of each transition type.

Unconditional deterministic transitions, DET

This is the most basic transition type; DET transitions from state z_i to state z_k cause, when the GLSM is in state z_i, always a deterministic transition into state z_k. Note that this implies that each GLSM state can have only one outgoing transition of type DET. The use of this transition type is fairly limited, because it causes a state with such a transition as its source to be left immediately after being entered. This obviously implies that for each state there can be only one transition leaving it. Consequently, using exclusively DET transitions, one can only realise a very limited class of GLSM structures. However, at least for the transition leaving the initial state, DET transitions are frequently used in practically occurring GLSMs.

Unconditional probabilistic transitions, PROB(p)

A transition of type PROB(p) from a GLSM state z_i to another state z_k takes a GLSM that is in state z_i directly into state z_k with probability p. Clearly, DET transitions are equivalent to a special case of this transition type, namely to PROB(1). For now, we can therefore assume without loss of generality that all transition types in a given GLSM are of type PROB. Note that if the set of transitions leaving z_i is given as $\{t_1, \ldots, t_n\}$, and the type of any t_j is PROB(p_j), the semantics of this state type require $\sum_{j=1}^{n} p_j = 1$ in order to ensure that the selection of the transition from z_i is based on a proper probability distribution.

Note that without loss of generality, by using PROB(0) transitions we can restrict our attention to fully connected GLSMs, where for each pair of states (z_i, z_k), a transition of type PROB(p_{ik}) is defined. This allows a more uniform representation of this class of GLSMs which in turn will facilitate both theoretical investigations and practical implementations of this GLSM type. Furthermore, the behaviour of these GLSMs can be easily modelled using Markov processes [Çinlar, 1975], which facilitates their analysis, as well-known techniques for studying Markov processes can be directly applied.

Conditional transitions, CPROB(C, p) and COND(C)

While until now we have focused on transitions whose execution only depends on the actual state, the following generalisation from PROB(p) introduces context dependent transitions. A CPROB(C, p) transition from state z_i to state z_k is executed with a probability proportional to p only when a condition predicate C is satisfied. If C is not satisfied, all transitions CPROB(C, p) from the current state are blocked, *i.e.*, they cannot be executed.

Obviously, PROB(p) transitions are equivalent to CPROB$(\top, p)$ conditional probabilistic transitions, where $\top$ is the predicate which is always true. Without loss of generality, we can therefore assume that for a given GLSM all transitions are of type CPROB(C, p). An important special case of conditional transitions are *conditional deterministic* transitions. These include the case COND(C) = CPROB$(C, 1)$; they can also arise when for a given GLSM state z_i, all but one of its outgoing transitions are blocked at any given time. Hence, one way to obtain deterministic GLSMs using conditional transitions is to make sure that by their logical structure, the condition predicates for the transitions leaving each state are mutually exclusive. Generally, depending on the nature of the condition predicates used, the decision whether a conditional transition is deterministic or not can be very difficult. For the same reasons it can be difficult to decide for a given GLSM with conditional probabilistic transitions, whether a particular state is reachable from the initial state.

For practically using GLSMs with conditional transitions it is important to make sure that all condition predicates can be evaluated in a sufficiently efficient way (when compared to the cost for executing local search steps). There are two kinds of condition predicates, the first of which is based on the search space position and/or its local neighbourhood; the other is based on search control aspects alone, like the time that has been spent in the

$\top$	always true
count(k)	total number of local search steps $\geq k$
countm(k)	total number of local search steps modulo $k = 0$
scount(k)	number of local search steps in current state $\geq k$
scountm(k)	number of local search steps in current state modulo $k = 0$
lmin	current local search position is local minimum w.r.t. its direct neighbours
obf(x)	current objective function value $\leq x$
noimpr(k)	objective function value could not be improved within last k steps

Table 3.1: Commonly used simple condition predicates.

current GLSM state, or the overall run-time. Naturally, these two kinds of conditions can also be mixed. Some concrete examples for condition predicates can be seen in Table 3.1. Note that all these predicates are based on local information only and can thus be efficiently evaluated during the search.

Usually, for each condition predicate, a positive as well as a negative (negated) form will be defined. Using propositional connectives like “$\wedge$” or “$\vee$”, these simple predicates can be combined into complex predicates. However, it is not difficult to see that every GLSM using complex condition predicates can be reduced to an equivalent GLSM using only simple predicates by introducing additional states and/or transitions. Thus, using complex condition predicates can generally be avoided without restricting the expressive power of the model.

Transition actions

After discussing a hierarchy of increasingly general transition types, we now introduce another conceptual element into the context of transitions: transition actions. Transition actions are associated with the individual transitions and are executed whenever the GLSM executes the corresponding transition.

Generally, transition actions can be added to each of the transition types defined above, while the semantics of the transition (in terms of its effect on the immediate successor state of the GLSM) are not affected. If T is a transition type, we let $T : A$ denote the same transition type with associated action A. The nature of action A has to be such that it neither directly affects the state of the GLSM, nor its search space position. Instead, transition actions generally can be used for

- modifying global parameters of one or more state types,
- realisation of input / output functionality in actual GLSM realisations,
- communication between GLSMs in cooperative GLSM models (this extension of the basic GLSM model will be introduced in Section 3.4.)

By introducing an action NOP ("no operation") without any effects we obtain uniform GLSMs in which all transitions have associated actions. Note, however, that we do not need multiple actions (*i.e.*, sequences or sets of actions which are associated with the same transition), because by introducing 'dummy' GLSM states that do not modify the search position, the (intuitive) semantics of multiple actions can be emulated.

From a minimalist point of view, even simple transition actions are not strictly required because they, too, can be emulated by embedding the corresponding actions into the search strategies associated with the GLSM states. This, however, would confound two different concepts, namely the local search strategies and the actions which are rather part of the search control mechanism represented by the modified finite state machines underlying the GLSM model. But the primary motivation of the GLSM model is to facilitate the *adequate* representation of complex SLS algorithms, and as we will see in Section 3.3, the notion of transition action occurs naturally in several widely known SLS techniques. Therefore, we prefer to model transition actions explicitly in the way described above.

State Types

At this point, state types are the only component that remains to be covered in order to be able to fully specify concrete GLSMs. For practical purposes, state types will usually be defined in a procedural way, usually by using some form of pseudo-code. In some cases, more adequate descriptions of complex state types can be obtained by using other formalisms, such as decision trees. Concrete examples for various state types will be give in the next section and in subsequent chapters. For formally specifying the semantics of a GLSM, the semantics of the individual state types are required to be specified in the form of a trajectory $\pi_\tau : S \mapsto \mathcal{D}(S)$ (see also the "In Depth" section on page 110).

Here, we want to focus on some fundamental distinctions between certain state types. One of these concerns the role of the state within the general local search scheme presented in Section 1.5. Since we are modelling search initialisation and local search steps using the same mechanism, namely GLSM states, there is a distinction between initialisation states and search step states. An initialisation state is usually different from a search step state in that it is left after one corresponding step has been performed. Also, while search step states correspond to moves in a restricted local neigbourhood (like flipping one variable in SAT), one initialisation step can lead to arbitrary search space positions (like a randomly chosen assignment of all variables of a SAT instance). Formally, we define an *initialising state type* as a state type τ for which the local search position after one τ-step is independent of the local search position before the step; the states of an initialising type τ are called *initialising states*. Generally, each GLSM will have at least one initialising state, which is also its initial state. A GLSM can, however, have more than one initialising state and use these states for dynamically restarting the local search process.

If for a given problem there is a natural common neighbourhood relation for local search, we can also distinguish *single-step states* from *multi-step states*. For the SAT problem, most local search algorithms use a 1-exchange neighbourhood relation, in which two variable assignments are direct neighbours if they differ in exactly one variable's value. In this context, a single-step state would flip one variable's value in each step, whereas a multi-step state could flip several variables per local search step. Consequently, initialising states are an extreme case of multi-step states, since they can affect all variable's values at the same time.

3.3 Modelling SLS Algorithms Using GLSMs

Up to this point, we have introduced the GLSM model and discussed various types of GLSMs structures, transitions, and states. We now demonstrate applications of the model by specifying and discussing GLSM representations for many of the well-known SLS techniques described in Chapter 2. This way, important similarities and differences between SLS techniques are

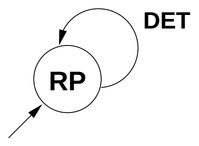

Figure 3.6: GLSM for Random Picking

```
**procedure** *step-RP*($\pi, s$)
    **input** *problem instance* $\pi \in \Pi$, *candidate solution* $s \in S(\pi)$
    **output** *candidate solution* $s \in S(\pi)$
    $s' := selectRandom(S)$
    **return** $s'$
**end** *step-RP*
```

Figure 3.7: Procedural specification of GLSM state RP; the function $selectRandom(S)$ returns an element of S selected randomly according to a uniform distribution over S.

highlighted. GLSM representations of other SLS algorithms are covered in later chapters or exercises.

Random Picking and Random Walk

The simplest possible SLS algorithm is Random Picking, as introduced in Section 1.5. When cast into our SLS definition, the $init$ and $step$ functions of Random Picking are identical and perform a random uniform selection of a candidate solution from the underlying search space. The corresponding GLSM is shown in Figure 3.6. It has only one state of type RP, formally defined by $\tau_{RP}(s)(s') = 1/|S|$ for all s'. As this kind of functional state type definition can get rather complex and difficult to understand for more advanced SLS strategies, we will often use procedural definitions of state types in terms of step functions instead. Such a definition for the random picking state type is shown in Figure 3.7.

The Uninformed Random Walk algorithm (*cf.* Section 1.5) requires an

```
**procedure** *step-RW*($\pi, s$)
    **input** *problem instance* $\pi \in \Pi$, *candidate solution* $s \in S(\pi)$
    **output** *candidate solution* $s \in S(\pi)$
    $s' := selectRandom(\{s' \mid N(s,s')\})$
    **return** $s'$
**end** *step-RW*
```

Figure 3.8: Procedural specification of GLSM state RW; the function $selectRandom(\cdot)$ returns an element of the set selected randomly according to a uniform distribution.

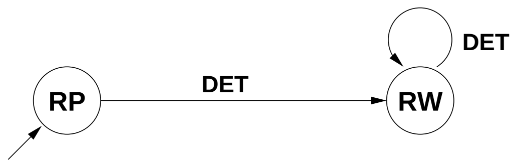

Figure 3.9: GLSM for Uninformed Random Walk

additional state type, RW, whose semantics are defined in Figure 3.8. In its simplest form, the search is initialised by random picking, followed by a series of uninformed random walk steps. The corresponding GLSM is shown in Figure 3.9. In practice, many SLS algorithms are extended by a restart mechanism, by which, in the simplest case, after every m search steps (where m is a parameter of the algorithm), the search process is reinitialised. Generally, other restart conditions can be used for determining when a reinitialisation should occur. Figure 3.10 shows the GLSM for Uninformed Random Walk with Restart; it is obtained from the GLSM for the basic Uninformed Random Walk algorithm without restart by a simple modification of the state transitions. Note how the GLSM representation for both algorithms indicates the fact that Uninformed Random Walk can already be seen as a (albeit very simple) hybrid SLS algorithm, using two types of search steps, Random Picking, and Uninformed Random Walk.

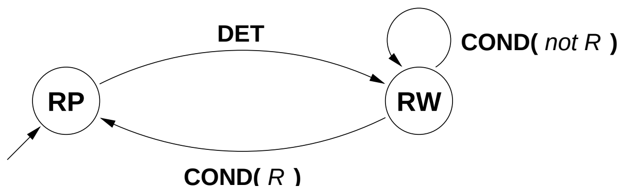

Figure 3.10: GLSM for Uninformed Random Walk with Restart; R is the restart predicate, *e.g.*, countm(m).

While using a restart mechanism for pure Random Walk does not appear to be useful other than for illustrative purposes, the analogous extension of Iterative Improvement covered in the next section leads to significant performance improvements.

Iterative Improvement

The GLSM model for Iterative Improvement (*cf.* Section 1.5) is similar to that for Uninformed Random Walk. Again, we use an RP state to model the search initialisation by random picking, but the second state now captures the semantics of iterative improvement search steps. A procedural specification of a GLSM state BI that models best improvement search steps is given in Figure 3.11; Figure 3.12 shows the GLSM for Iterative Best Improvement Search with Random Restart. Note that the random restart mechanism will enable the algorithm to escape from local minima of the evaluation function and can hence be expected to improve its performance. Notice that the only difference between this GLSM and the one for Uninformed Random Walk with Random Restart shown in Figure 3.9 lies in the type of one state (BI *vs* RW). This reflects the common structure of the search control mechanism underlying both of these simple SLS algorithms. Similarly, the GLSM models for other variants of iterative improvement search, such as First Improvement, or Random Improvement, are obtained by replacing the BI state by a state of an appropriately defined type that reflects the semantics of these different types of iterative improvement steps.

Using the RP, RW, and BI state types, it is easy to construct a GLSM model for Randomised Iterative Improvement, one of the simplest hybrid

```
procedure step-BI($\pi, s$)
    input problem instance $\pi \in \Pi$, candidate solution $s \in S(\pi)$
    output candidate solution $s \in S(\pi)$
    $g^* := \min\{g(s') | g' \in N(s)\}$
    $s' := selectRandom(\{s' \mid N(s, s')$ and $g(s') = g^*\})$
    return $s'$
end step-BI
```

Figure 3.11: Procedural specification of GLSM state BI.

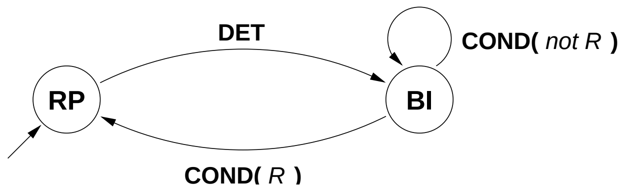

Figure 3.12: GLSM for Iterative Best Improvement with Random Restart; R is the restart predicate, *e.g.*, lmin.

SLS algorithms (*cf.* Section 2.2). The 2-state+init GLSM shown in Figure 3.13 represents the hybrid search mechanism in an explicit and intuitive way. Since the addition of the Random Walk state enables the algorithm in principle to escape from local minima, the restart mechanism included in this GLSM is not as important as in the previous case of pure Iterative Best Improvement. It may be noted that the same SLS algorithm could be modelled by a 1-state+init GLSM, using a single state for Randomised Iterative Improvement steps. This representation, however, would be substantially inferior to the 2-state+init GLSM introduced above, since an important part of the search control strategy is not captured by the structure of the GLSM model.

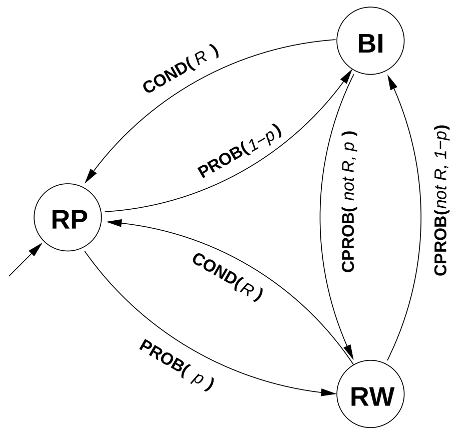

Figure 3.13: GLSM for Randomised Iterative Best Improvement with Random Restart; R is the restart predicate, *e.g.*, countm(m)

Simulated Annealing

To model Simulated Annealing by a GLSM, two approaches can be taken. Both use a state type SA to represent the probabilistic iterative improvement search steps that form the core of Simulated Annealing. This state type can be specified procedurally by the step function introduced in Chapter 2 (*cf.* Figure 2.3 on page 65). The difference between the two approaches lies in the way they conceptually implement the annealing schedule. This can be done by integrating temperature modifications into the search steps; in this case, the probabilistic iterative improvement state type is extended such that it includes temperature updates along with the basic search steps.

An alternative representation uses transition actions to capture the temperature modifications comprising the annealing schedule. This leads to the GLSM model shown in Figure 3.14. Conceptually, this representation is cleaner, since it separates the basic search process (which corresponds to a modification of the current position in the search space) from modifications of search control parameters, such as the temperature T.

Transition actions can be used in a similar way for modelling Reactive Tabu Search (*cf.* Section 2.2). In this case, a state type representing basic tabu search steps is used along with transition actions that modify the

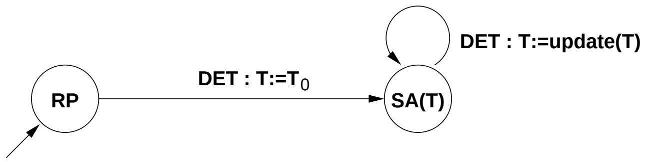

Figure 3.14: GLSM for Simulated Annealing; the initial temperature t_0 and temperature update function $update$ implement the annealing schedule.

tabu tenure parameter during the search. Additionally, an RW state can be integrated to additionally diversify the search when required.

Iterated Local Search

Iterated Local Search is an excellent example of a hybrid SLS technique. There are two basic types of search steps: The steps performed in the subsidiary local search and perturbation steps. Obviously, these as well as the search initialisation are modelled by separate GLSM states. A slight complication is introduced by the acceptance criterion, which is used in ILS to determine whether or not the search continues from the new candidate solution obtained from the last perturbation and subsequent local search phase. There are various ways of modelling this feature in a GLSM. Figure 3.16 shows a GLSM representation of ILS, where the application of the acceptance criterion is modelled as a separate state, AC (for a procedural definition of AC, see Figure 3.15). Note the use of transition actions for storing the current candidate solution before each perturbation phase, which is needed when applying the acceptance criterion in state AC. Furthermore, it might be noted that our GLSM model allows for several perturbation steps to be performed; this is motivated by the fact that in some ILS applications, the perturbation phase consists of several search steps of the same type (for an example, see Chapter 7).

Ant Colony Optimisation

There are different approaches to representing population-based SLS algorithms as GLSMs, corresponding to different views of the underlying

```
**procedure** *step-AC*$(\pi, s, t)$
    **input** *problem instance* $\pi \in \Pi$, *candidate solution* $s \in S(\pi)$
    **output** *candidate solution* $s \in S(\pi)$
    **if** $C(\pi, s, t)$ **then**
        **return** $s$
    **else**
        **return** $t$
    **end**
**end** *step-AC*
```

Figure 3.15: Procedural specification of GLSM state AC; this state type uses a candidate solution t stored earlier in the search process, and selection predicate $C(\pi, s, t)$ which returns $\top$ if s is to be selected, and $\perp$ otherwise. A selection predicate which is often used in the context of ILS is $better(\pi, s, t) := (g(s) < g(t))$, where g is the evaluation function for the given problem instance, π.

stochastic local search process itself. As briefly discussed in Section **??**, one can view populations of candidate solutions for the given problem instance π as search states; under this view, the search space of a population-based SLS algorithm consists of sets of candidate solutions of π. Ant Colony Optimisation, for instance, can then be represented as a GLSM as shown in Figure 3.17. State CI initialises the construction search and state CS performs a single construction step for all ants (*cf.* Example **??**, page **??**). The LS state performs a single step of local search for the whole population of ants. For a typical iterative improvement local search, this means that iterative improvement steps are performed for all ants that have not obtained a locally minimal candidate solution of the given problem instance; in this case, usually a condition predicate CL is used that is satisfied when all ants have obtained a locally minimal candidate solution. Initialisation and update of the pheromone trails are modelled using transition actions.

An alternate GLSM model for ACO is based on a view under which the

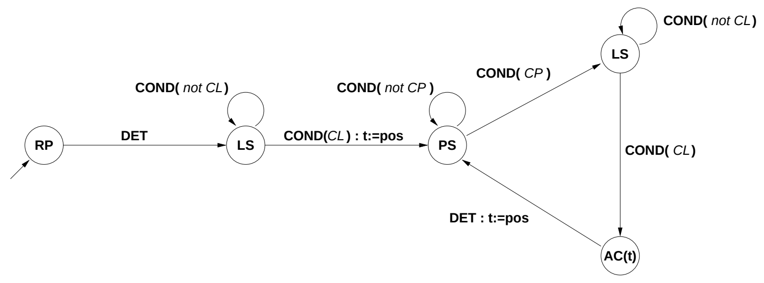

Figure 3.16: GLSM representation of Iterated Local Search; CP and CL are condition predicates which determine the end of the perturbation phase and the local search phase, respectively (see text for further details).

search space consists of probability distributions of candidate solutions for the given problem instance. Note that the probabilistic construction process carried out by each ant induces a probability distribution in which, ideally, higher quality candidate solutions have higher probability of being constructed. The subsequent local search phase then biases this probability distribution further towards better candidate solution. Finally, updating the peromone values modifies the probability distribution underlying the next construction phase. This representation has two disadvantages: Firstly, it does not reflect the fact that ACO effectively samples the probability distribution central to this model in each cycle in order to obtain a set of candidate solutions for the given problem instance; secondly, it does not capture the compact implicit representation of the probability distributions given by the pheromone values. However, this view on ACO is interesting for theoretical reason, *e.g.*, in the context of analysing important theoretical properties of ACO, such as the probability of obtaining a specific solution quality within a given time bound. Also note that the GLSM model corresponding to this view does not require transition actions for manipulating the pheromone trails since these are now essential components of the search state.

Another general approach for modelling population-based SLS algorithms as GLSMs is to represent each member of the population by a separate GLSM. The resulting cooperative GLSM models will be discussed in more detail in the next section.

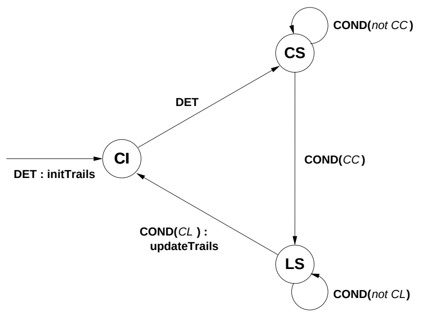

Figure 3.17: GLSM representation of Ant Colony Optimisation; the condition predicates CC and CL determine the end of the the construction and local search phases, respectively.

3.4 Extensions of the Basic GLSM Model

In this section we discuss various extensions of the basic GLSM model. One of the strengths of the GLSM model lies in the fact that these extensions arise quite naturally and can easily be realised within the basic framework. Some extended GLSM models, in particular cooperative GLSMs, have immediate applications in the form of existing SLS algorithms, while others have not yet been studied in detail.

Cooperative GLSM models

A natural extension of the basic GLSM, which is particularly suited for representing population-based SLS algorithms, is to apply several GLSMs simultaneously to the same problem instance. In the simplest case, such an ensemble consists of a number of identical GLSMs without any communication between the individual machines. The semantics of this *homogenous cooperative GLSM model without communication* are conceptually equivalent to executing multiple independent runs of an individual GLSM. It is particularly attractive for parallelisation, because it is very easy to implement, involves virtually no communication overhead, and can be almost arbitrarily scaled in principle.

The restrictions of this model can be relaxed in two directions. One is to allow ensembles of different GLSMs. This *heterogeneous cooperative GLSM model without communication* is particularly useful for modelling robust combinations of various SLS algorithms, each of which shows superior performance on certain types of instances, when the features of the given problem instances are not known *a priori*. This approach has been recently studied in the context of complete algorithms for hard combinatorial problems [Gomes and Selman, 1997b]: In this context the heterogenous ensembles were called *algorithm portfolios*. Generally, this cooperative model has almost the same advantages as its homogeneous variant; it is easy to implement and almost free of communication overhead.

Another generalisation is to allow communication between the individual GLSMs of a cooperative model. This is required for modelling explicitly population-based SLS algorithms in which the individual search trajectories are not independent. As an example, consider variants of Ant Colony Optimisation which allow only the ants that obtained the best solution quality in a given cycle to update the pheromone trails (iteration-best elitist pheromone update) [Stützle and Hoos, 2000]; communication between the ants is required in order to determine the best candidate solutions.

In principle, coooperative GLSM models can be extended with various communication schemes, including blackboard mechanisms, synchronous broadcasting, and one-to-one message passing in a fixed network topology. There are various ways of formally realising these techniques within the GLSM framework. One approach is to use special transition actions for communication (*e.g.*, *send* and *receive*); another option is to allow transition conditions and/or transition actions to access information that is shared between the individual GLSMs.

Many population-based SLS algorithms can be naturally represented as homogeneous cooperative GLSMs. Most ACO algorithms, for example Ant System [Dorigo *et al.*, 1991; 1996], can be easily modelled in the following way: The basic GLSMs corresponding to the individual ants have the same structure as the GLSM model in Figure 3.17 above; only now, the GLSM states represent the construction and local search steps performed by an individual ant and the transition action *updateTrails* performs a joint pheromone trail update for all ants. Note that in this case, the pheromone values are shared information between the individual ants' GLSMs.

Cooperative GLSMs with communication are more difficult to design and to implement than those without communication, since issues like preventing and detecting deadlocks and starvation situations generally have to be considered. Furthermore, the communication between individual GLSMs usually involves a certain amount of overhead. This overhead has to be amortised by the performance gains which may be realised in terms of speedup when applied to a specific problem class, but also in terms of increased robustness w.r.t. different problem classes.

Generally, one way of using communication to improve the performance of cooperative GLSMs is to propagate search space positions with low objective function values (or other attractive properties) within the ensemble such that individual GLSMs which detect that their search is not progressing well can pick up these “hints” and continue their local search from there. This can be easily realised as a homogeneous cooperative GLSM with communication. In such a model, the search effort will be more focussed on exploring promising parts of the search space than in a cooperative model without communication. Another general scheme uses two types of GLSMs, analysts and solvers. Analysts do not attempt to find solutions but rather try to analyse features of the search space. The solvers try to use this information to improve their search strategy. This architecture is an instance of the heterogeneous cooperative GLSM model with communication. It can be extended in a straightforward way to allow for different types of analysts and solvers, or several independent sub-ensembles of analysts and solvers.

Learning via dynamic transition probabilities

One of the features of the basic GLSM model with probabilistic transitions is the fact that the transition probabilities are static, *i.e.*, they are fixed when designing the GLSM. An obvious generalisation, along the lines of learning automata theory [Narendra and Thathachar, 1989], is to let the transition probabilities evolve over time as the GLSM is running. The search control in this model corresponds to a variable structure learning automaton. The environment in which such a dynamic GLSM is operating is given by the evaluation function induced by an individual problem instance or by a class of evaluation functions induced by a class of instances. In the first case (*single-instance learning*), the idea is to optimise the control strategy on one instance during the local search process. The second case (*multi-instance learning*), is based on the assumption that for a given problem domain (or sub-domain), all instances share certain features to which the search control strategy can be adapted.

The modification of the transition probabilities can either be realised by an external mechanism (external adaption control), or within the GLSM framework by means of specialised transition actions (internal adaption control). In both cases, suitable criteria for transition probability updates have to be defined. Two classes of such criteria are those based on trajectory information, and those based on GLSM statistics. The latter category includes state occupancies and transition frequencies, while the former comprises primarily basic descriptive statistics of the objective or evaluation function value along the search trajectory, possibly in conjunction with discounting of past observations. The approach as outlined here captures only a specific form of parameter learning for a given parameterised class of GLSMs. Conceptually, this can be further extended to allow for dynamic changes of transition types (which is equivalent to parameter learning for a more general transition model, such as conditional probabilistic transitions).

In principle, concepts and methods from learning automata theory can be used for analysing and characterising dynamic GLSMs; basic properties, such as expedience or optimality can easily be defined. We conjecture, however, that theoretically proving such properties will be extremely difficult, as the theoretical analysis of standard SLS behaviour is already very complex and limited in its results. Nevertheless, we believe that empirical methodology can provide a sufficient basis for developing and analysing interesting and useful dynamic GLSM models.

Evolutionary GLSM models

For cooperative GLSMs, another form of learning can be realised by letting the number or type of the individual GLSMs vary over time. The population dynamics of these *evolutionary GLSM models* can be interpreted as a learning mechanism. As for the learning GLSMs described above, we can distinguish between *single-instance* and *multi-instance learning* and base the process for dynamically adapting the population on similar criteria.

In the conceptually simplest case, the evolutionary process only affects the composition of the cooperative ensemble: machines that are performing well will spawn off numerous offspring replacing individuals showing inferior performance. This mechanism can be applied to both, homogeneous and heterogeneous models for single-instance learning. In the former case, the selection is based on trajctory information of the individual machines and achieves a similar effect as described above for certain types of homogeneous cooperative GLSMs with communication: The search is concentrated on exploring promising parts of the search space. When applied to heterogeneous models, this scheme allows the realisation of self-optimising algorithm portfolios, which can be useful for single-instance as well as multi-instance learning.

This scheme can be further extended by introducing mutation and possibly recombination operators as known from Evolutationary Algorithms. It is also easily conceivable to combine evolutionary and indivual learning, *e.g.*, by evolving ensembles of dynamic GLSMs. And finally, one could consider models that additionally allow communication within the ensemble. By combining different extensions we can arive at very complex and potentially powerful GLSM models; while these are very expressive, in general they will also be extremely difficult to analyse. Nevertheless, their implementation is quite simple and straightforward and an empirical approach for analysing and optimising their behaviour appears to be viable. We believe that such complex models, which allow for a very flexible and fine-grained search control, will likely be most effective when applied to problem classes with numerous and salient structural features (see also Chapter 5). There is little doubt that, to some extent, this is the case for most real-world problem domains.

Continuous GLSM models

The basic GLSM model and all extensions thereof discussed up to this point model local search algorithms for solving discrete decision or optimisation problems. Yet, by using continuous instead of discrete local search strategies for the GLSM state types, the model can be easily extended to continuous optimisation approaches. Although SLS algorithms for continuous optimisation problems are beyond the scope of this book, we believe that the GLSM model's main feature, the clear distinction between simple search strategies and search-control, is also a useful architectural and conceptual principle for continuous optimisation algorithms.

3.5 Further Reading and Related Work

The main idea underlying the GLSM model, namely to adequately represent complex algorithms as a combination of several simple strategies, is one of the fundemantal concepts in computing science. Here, we applied this general metaphor to local search algorithms for combinatorial decision and optimisation problems using suitably extended finite state machines for search control. The GLSM model is partly inspired by Amir Pnueli's work on hybrid systems [Maler *et al.*, 1992] and Thomas Henzinger's work on hybrid automata; the latter uses finite state machines to model systems with continuous and discrete components and dynamics [Alur *et al.*, 1993; Henzinger, 1996] and is therefore conceptually related to the GLSM model.

The GLSM definition and semantics are heavily based on well-known concepts from automata theory (for a general references, *cf.* [Harrison, 1978; Rozenberg and Salomaa, 1997]). However, when using conditional transitions or transition actions, the GLSM model extends the conventional model of a finite state machine. In its most general form, the GLSM model bears close resemblance to a restricted form of Petri nets [Krishnamurthy, 1989], where only one token is used. Of course, the same type of search control mechanism could be represented using formal systems other than a FSM-based formalism. First of all, other types of automata, such as pushdown automata or Turing machines could be considered. However, we feel that the FSM model, one of the simplest types of automata, is powerful enough for representing most interesting search-control strategies commonly found in the local search literature. Furthermore, it leaves sufficient room for extension, while being analytically more tractable than more complex automata models. Finally, FSMs offer the potential advantage of being implementable in hardware in a rather straightforward way, which might be interesting in the context of applying local search algorithms to time-critical problems (*cf.* [Hamada and Merceron, 1997]). Summarising these arguments, the use of FSMs for formalising the search-control mechanism seems to be sufficient and adequate. Of course, formalisms equivalent to the FSM model, such as rule-based descriptions, could be chosen instead. While this might be advantageous in certain contexts (such as reasoning about properties of a given GLSM), we find that the automaton model provides a slightly more intuitive and easier-to-handle framework for designing and implementing local search algorithms whose nature is primarily procedural.

The GLSM model allows to adequately represent existing algorithmic frameworks for local search, such as GenSAT [Gent and Walsh, 1993a] or Iterated Local Search [Martin *et al.*, 1991; Johnson, 1990]. These frameworks are generally more specific and more detailed than the GLSM model; however, they can be easily realised as generic GLSMs without losing any detail of description. This is done by using structured generic state types to capture the more specific aspects of the framework to be modelled. While the GLSM model can be used to represent *any* local search algorithm, many of these do not really make use of the search control mechanism it provides. Note, however, that some of the most successful local search algorithms for various problem classes (such as R-Novelty for SAT [McAllester *et al.*, 1997], *HRTS* for MaxSAT [Battiti and Protasi, 1997a], and iterated local search schemes for TSP [Martin *et al.*, 1991; Johnson, 1990; Johnson and McGeoch, 1997]) rely on search control mechanisms of the type provided by the GLSM model.

The various extensions of the basic model discussed in this chapter are closely related to established work on learning automata [Narendra and Thathachar, 1989], parallel algorithm architectures [Jájá, 1992], and Evolutionary Algorithms [Bäck, 1996]. While most of the proposed extensions have not been implemented and empirically evaluated so far, they appear to be promising, especially when considering recent work on multiple independent tries parallelisation [Shonkwiler, 1993; Gomes *et al.*, 1998], algorithm portfolios [Gomes and Selman, 1997b], and learning local search approaches for solving hard combinatorial problems [Boyan and Moore, 1998; Minton, 1996].

3.6 Summary

Based on the intuition that adequate local search algorithms are usually obtained by combining several simple search strategies, in this chapter we introduced and discussed the GLSM model. This framework formalises the search control using a finite state machine (FSM) model, which associates simple component search strategies with the FSM states. FSMs belong to the most basic and yet fruitful concepts in computer science; using them to model local search control offers a number of advantages. First, FSM-based models are conceptually simple; consequently, they can be implemented easily and efficiently. At the same time, the formalism is expressive enough to allow for the adequate representation of a broad range of modern local search algorithms. Secondly, in our experience, the GLSM model facilitates the development and design of new, hybrid local search algorithms. In this context, both conceptual and implementational aspects play a role: due to the conceptual simplicity of the GLSM model and its clear representational distinction between search strategies and search control, hybrid combinations of existing local search algorithms can be easily formalised and explored. Using a generic GLSM simulator, which is not difficult to implement, new hybrid GLSM algorithms can be realised and evaluated in a very efficient way. Finally, there is a huge body of work on FSMs; many results and techniques are in principle directly applicable to GLSMs which may be of interest in the context of analysing and optimising SLS algorithms.

As we have shown, based on a clean and simple definition of a GLSM, the semantics of the model can be formalised in a rather straightforward way. We then discussed the tight relation between the GLSM model and a standard generic local search scheme. By categorising GLSM types according to their structure and transition types, we demonstrated how the general model facilitates the systematic study of search control mechanisms. Most modern SLS techniques can be represented by rather simple GLSMs; the fact that the most efficient of them are structurally slightly more complex than others suggests that further improvements can be achieved by developing more complex combinations of simple search strategies. Finally, we pointed out several extensions of the basic GLSM model. Most of these are very natural generalisations, such as cooperative or continous GLSMs; they not only demonstrate the scope of the general idea but also suggest numerous routes for further research. In subsequent chapters of this book, GLSMs will be used for conceptualising and analysing SLS algorithms for various combinatorial problems.

3.7 Exercises

Exercise 3.1 (Easy) Explain how any local search algorithm can be formalised as a 1-state+init GLSM. Why is this formalisation not desirable?

Exercise 3.2 (Easy) Find a good GLSM representation for the following hybrid SLS algorithm for TSP. **[hh: describe in detail: SA for 1000 iterations repeat (3-opt until lmin; 4 x 4-opt – TODO(hh)]**

Exercise 3.3 (Medium) Show how a genetic local search algorithm (see Chapter 2, Section 2.3) can be modelled as a GLSM.

Exercise 3.4 (Medium) Consider the following skeleton of a GLSM for Tabu Search. **[ts: add skeleton – TODO(hh)]** Assuming you don't like GLSMs with transition actions, specify an equivalent GLSM for Tabu Search which does not use transition actions.

Exercise 3.5 (Hard) How could GLSMs be useful in analysing the behaviour of SLS algorithms? (Hint: Think of conditional transitions and observations you can make from monitoring the actual state of the GLSM during search.)