2

Prerequisites

LCSs are designed to solve classification as well as more general *reinforcement learning* (RL) problems. LCSs solve these problems by evolving a rule-base of classifiers by the means of a critic based on RL techniques for rule evaluation and a GA for rule evolution. Before jumping directly into the LCS arena, we first look at these prerequisites.

Since optimization and learning is comparable to a *search for expected structure*, we first look at the problem types and problem structures we are interested in. We differentiate between *optimization problems*, *classification problems*, and *Markov decision process problems*. Each problem causes different but related challenges. Thus, successful learning architectures need to be endowed with different but related learning mechanisms and learning biases. LCSs are facing RL problems but might also be applied to classification or even optimization problems.

Since it is provenly impossible that there exists a learning technique that can solve all possible problems (assuming that all expressible problems given a certain representation are equally likely) better than simple enumeration or random search (Wolpert & Macready, 1995; Wolpert, 1996b; Wolpert, 1996a; Wolpert & Macready, 1997), it is important to characterize and distinguish different problem structures. In this problem introduction we focus on such different problem properties. We then use these properties to reveal what GAs and LCSs are actually searching for; in other words, we identify the learning biases of the algorithms. As we will see, GAs and LCSs are very general learning mechanisms that are searching for lower order dependency structures in a problem, often referred to as building blocks (BBs). The advantage of such a search bias is that natural problems typically but arguably obey such a hierarchically composed problem structure (Simon, 1969; Gibson, 1979; Goldberg, 2002).

Apart from the necessary understanding of LCS-relevant problem types and typical problem structures, the second major prerequisite is a general understanding of RL techniques and *genetic algorithms* (GAs). Section 2.2 introduces RL including the most relevant Q-learning algorithm (Watkins,

1989). We show that RL is well-suited to serve as an online learning actor/critic system that is capable of evaluating rules, distributing reward, and making action (or classification) decisions. Section 2.3 introduces GAs, which are well-suited to learn relevant rule structures given a quality or *fitness* measure. Additionally, we highlight the importance of the mentioned *facetwise analysis* approach taken in the GA literature to promote understanding of GA functioning, scale-up behavior, and parameter settings as well as to enable the design of more elaborate, *competent GAs* (Goldberg, 2002).

Summary and conclusions summarize the most important issues addressed in this chapter, pointing towards the integration of the addressed mechanisms into LCSs.

2.1 Problem Types

LCSs may be applied in two major problem domains. One is the world of classification problems. The second is the one of RL problems either modeled by a Markov decision process (MDP) or by a more general, partially observable Markov decision process (POMDP).

In classification problems, feedback is provided instantly. Successive problem instances are usually sampled independently and identically distributed. On the other hand, in RL problems feedback may be delayed in time and successive problem input depends on the underlying problem structure as well as on the chosen actions. Thus, internal reinforcement propagation becomes necessary, which poses an additional learning challenge.

Before introducing classification and RL problems, we give a short introduction to optimization problems emphasizing similarities and differences as well as typically expectable problem structures and properties.

2.1.1 Optimization Problems

An optimization problem is a problem in which a particular structure or solution needs to be optimized. Thus, given a particular solution space, an optimization algorithm searches for the best solution in the solution space. Optimization problems cover a huge range of problems such as (1) the optimization of a particular object, such as an engine, (2) the optimization of a particular method, such as a construction process, (3) the optimization or detection of a state of lowest energy, such as a physical problem, or (4) the optimization of a solution to any search problem, such as the problem of satisfiability or the traveling salesman problem.

More formally, a simple binary optimization problem is defined for a problem space S that is characterized by a bit string of a certain length l: $\mathcal{S} = \{0,1\}^l$. Each bit string represents a particular problem solution. Feedback is provided in terms of a scalar reward (fitness) value that rates the

solution quality. An optimization algorithm should be designed to *effectively* search the solution space for the global optimum.

Given, for example, the problem of optimizing a smoothie drink with a choice of additional ingredients mango, banana, and honey available, the problem may be coded by three bits indicating the absence (0) or presence (1) of each ingredient. The fitness is certainly very subjective in this example, but assuming that we like all three ingredients equally well, prefer any combination of two of the ingredients over just one ingredient alone, and like the combination of all three ingredients best, we constructed a *one-max problem*.

Table 2.1 (first numeric column) gives possible numerical values for a four bit one-max problem. The best solution to the problem is denoted by `1111` and the fitness is determined by the number of ones in the problem. Certainly, the one-max problem is a very easy problem. There is only one (global) optimum and the closer to the global optimum, the higher the fitness. Thus, the problem provides strong *fitness guidance* towards the global optimum. This guidance can be even more clearly observed in the six bit visualization of the one-max problem shown in Figure 2.1a: The more ones that are specified in a solution, the higher the fitness of the solution.

However, problems can be *misleading* in that the fitness measure may give incorrect clues in which direction to search for more promising solutions. Table 2.1 and Figure 2.1 show progressively more misleading types of problems. While the *royal-road* function still provides the steps that lead to the best solution (Mitchell, Forrest, & Holland, 1991), the *needle in the haystack* problem provides no directional clues whatsoever—only the optimum results in high fitness. The *trap problem* provides quality clues in the opposite direction: Fitness leads towards a local optimum away from the global optimum. In a trap problem the combination of a number of factors (e.g. ingredients) makes the solution better, but the usage of only parts of these factors makes the solution actually worse than the base solution that uses none.

The reader should not be misled by these abstract problems thinking that a local optimum (if existent) and the global optimum will always be the exact inverse of a binary string. In other, albeit related problems, this certainly does not need to be the case. Finally, the meaning of zeroes and ones may be (partially) swapped so that the global optimum does not need to be all ones but the problem structure may still be identical to one of the types outlined in the table. Thus, although the examples are very simplified, they characterize important problem structures that pose different challenges to the learning algorithm.

The problem substructures may be combined to form larger, more complex problems. Each substructure can then be regarded as a building block (BB) in the larger problem, that is, a substructure that is evaluated rather independently of the other structures. Certainly, there can be higher-order dependencies or overlapping BB structures. Also, the fitness contribution of each BB may vary.

Table 2.1. There are several ways in which the solution quality measure can lead towards the global optimal solution (here: 1111). In the *one-max* problem, the closer the solution is to the optimal solution, the higher its quality. In the *royal-road* problem, the path leads to the optimal solution step-wise. In the *needle in the haystack* problem, fitness gives no hints about the optimal solution. In the *trap problem*, the quality measure is *misleading* since the more the solution differs from the optimal solution, the higher its fitness.

	One-Max	Royal-Road	Needle i.H.	Trap Prob.
0000	0	0	0	3
0001	1	0	0	2
0010	1	0	0	2
0100	1	0	0	2
1000	1	0	0	2
0101	2	0	0	1
1001	2	0	0	1
0110	2	0	0	1
1010	2	0	0	1
0011	2	2	0	1
1100	2	2	0	1
0111	3	2	0	0
1011	3	2	0	0
1101	3	2	0	0
1110	3	2	0	0
1111	4	4	4	4

Two most basic combinations can be distinguished. In the simplest case the fitness contribution of each substructure is equal and simply added together yielding an *uniformly scaled* problem. Another method is to exponentially scale the utility of each substructure. In this case the fitness of the second block matters only once the optimum of the first block is found and so forth, yielding an *exponentially scaled* problem. It should be noted that in a uniformly scaled problem the blocks can be solved in parallel because fitness provides information about all blocks. On the other hand, in an exponentially scaled problem, the BBs need to be solved sequentially since the exponentially scaled fitness nearly eliminates fitness information from later blocks that are not most relevant yet.

Problem structures that are composed of many smaller dependency structures are often referred to as *decomposable problems* or problems of bounded difficulty (Goldberg, 2002). Boundedly difficult problems are bounded in that the BB size of lower-level interactions is bounded to a certain *order*—the order of problem difficulty. For example, from the problems in Table 2.1, the one-max problem has an order of difficulty of one because it is actually composed of four uniformly scaled BBs of size one. The royal-road function is of order two since two BBs are combined. Both the needle in the haystack and

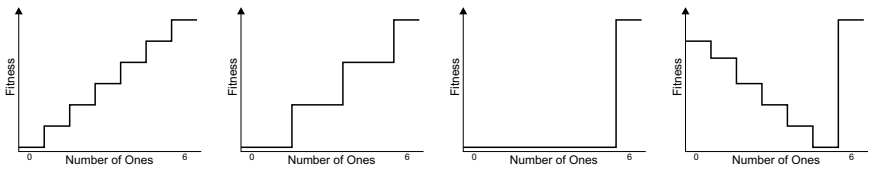


Fig. 2.1. The figures show progressively more challenging optimization problems. In the one-max problem (a), fitness progressively increases on the way to the globally optimal solution (here: all ones). In the royal-road problem (b), larger steps need to be taken towards the optimal solution. In the needle in the haystack problem (c), fitness gives no information about where the optimal solution is situated. Finally, in the trap problem (d), fitness actually misleads to a local optimum (here: all zeroes) and thus away from the globally optimal solution (here: all ones).

the trap problem are of order four since there is no further decomposition possible.

Regardless of the problem structure, the problem may actually have multiple optimal solutions, the quality measure may be noisy, or the provision of several near-optimal solutions may be more desirable than the detection of one (completely) optimal solution. Often, an expert may want to choose from such a set of (near-) optimal solutions. In this case, a learner would be required to find not only one globally optimal solution but rather a set of several different (near-) optimal solutions.

To summarize, optimization problems are problems in which a best solution must be found given a solution space. Feedback is available that rates solutions proposed by the learner. The feedback may or may not provide hints where to direct the further search for the optimal solution. Finally, the number of optimal solutions may vary and, dependent on the problem, one or many optimal (or near optimal) solutions may need to be found.

2.1.2 Classification Problems

A classification problem poses further difficulties to the learning algorithm. Although a classification problem may be reduced to an optimization problem, the reduction is tedious and often destroys much of the available problem structure and information inherent in a classification problem.

We define a classification problem as a problem that consists of problem instances $s \in \mathcal{S}$. Each problem instance belongs to one class (traditionally termed an action in LCSs) $a \in \mathcal{A}$. In machine learning terms, s may be termed a feature vector and a a concept class. The mapping from $\mathcal{S}$ to $\mathcal{A}$ is represented by a *target concept* belonging to a set of concepts (that is, the *concept space*). The goal of a classification system is to learn the target concept. Thus, the classification system learns to which class a_i each problem

instance s_i belongs to. The desirable properties of such a learning system are that the learner learns a maximally *accurate* problem solution, measured usually by the percentage of correct problem instance classifications, and a maximally *general* problem solution, which can be characterized as a solution that generalizes well to other (unseen) problem instances. Given that the learner has a certain *hypothesis space* of expressible solutions, the learner looks for the maximally accurate, maximally general hypothesis with respect to the target concept.

As in optimization problems, a classification problem may be composed of several subproblems characterizable as BBs. However, such BBs cannot be directly related to fitness but can only increase the probability that a problem instance belongs to a certain class. Solution hypotheses may be represented in a more distributed fashion in that different subsolutions may be responsible for different problem subspaces. This is expected to be particularly useful if different problem subspaces are expected to have a quite different class distribution. In this case, different BBs will be relevant dependent on the current problem subspace. Thus, in contrast to optimization problems, different BBs may need to be detected and propagated in different problem subspaces. Nonetheless, the global BB distribution can also be expected to yield important information that can improve the search for accurate problem (sub-)solutions.

Boolean Function Problems

In most of this work, we focus on Boolean function problems. In these problems, the problem instance space is restricted to the binary space, that is, $\mathcal{S} \subseteq \{0,1\}^l$ where l denotes the fixed problem length. Similarly, a Boolean function problem has only two output classes $\mathcal{A} = \{0,1\}$. Consequently, any Boolean function can be represented by a logical formula and consequently also by a logical formula in disjunctive normal form (DNF). Appendix C introduces the Boolean function problems investigated herein showing an exemplar DNF representation and discussing their general structure and problem difficulty.

As an example, let us consider the well-known multiplexer problem, which is widely studied in LCS research (De Jong & Spears, 1991; Wilson, 1995; Wilson, 1998; Butz, Kovacs, Lanzi, & Wilson, 2001). It has been shown that LCSs are superior compared to standard machine learning algorithms, such as the decision tree learner *C4.5*, in the multiplexer task (De Jong & Spears, 1991). The problem is of particular interest due to its dependency structure and its distributed niches, or subsolutions. The problem is defined for binary strings of length $l = k + 2^k$. The output of the multiplexer function is determined by the bit situated at the position referred to by k position bits (usually but not necessarily located at the first k positions). The disjunctive normal form of the 6-multiplexer for example can be written as follows:

$$6MP(x_1, x_2, x_3, x_4, x_5, x_6) = \neg x_1 \neg x_2 x_3 \vee \neg x_1 x_2 x_4 \vee x_1 \neg x_2 x_5 \vee x_1 x_2 x_6; \quad (2.1)$$

for example, $f(100010) = 1$, $f(000111) = 0$, or $f(110101) = 1$. More information on the multiplexer problem can be found in Appendix C. It is interesting to see that the DNF form of the multiplexer problem consists of conjunctions that are *non-overlapping*. Any problem instance belongs, if at all, to only one of the conjunctive terms in the problem. Later, we will see that the amount of overlap in a problem is an important problem property for LCSs' learning success.

Real-valued Problems

Boolean function problems are a rather restricted class of classification problems. In the general case, a problem instance s may be represented by a feature vector. Each feature may be a binary attribute, a nominal attribute, an integer attribute, or a real-valued attribute. Mixed representations are possible.

We refer to such real-world classification problems as datamining problems. The problem is represented by a set of problem instances with their corresponding class. A problem instance may consist of a mixture of features and there may be more than two classification classes. Since the target concept is generally unknown in datamining problems, performance of the learner is often evaluated by the means of stratified ten-fold cross-validation (Mitchell, 1997) that trains the system on a subset of the data set and tests it on the remaining problem instances. The data is partitioned into ten subsets. The learner is trained on nine of the ten subsets and tested on the remaining subset. To avoid sampling biases, this procedure is repeated ten times, each time training and testing on different subsets. Stratification assures that the class distribution is approximately equal in all folds. Ten-fold cross-validation is very useful in evaluating the generalization capabilities of the learner since performance is tested on previously unseen data instances.

2.1.3 Reinforcement Learning Problems

In contrast to optimization and classification problems, feedback in the form of a class or immediate reinforcement is not necessarily available immediately in RL problems. Rather, feedback is provided in terms of a scalar reinforcement value that indicates the quality of a chosen action (or classification). Additionally, successive problem instances may be dependent upon each other in that subsequent input usually depends on previous input and on the executed actions. RL problems are thus more difficult, but also more natural, simulating interaction with an actual outside world. Figure 2.2 shows the agent-environment interaction typical in RL problems.

Despite the environmental interaction metaphor, a classification problem may be redefined as an RL problem providing reward feedback about the accuracy of the chosen class. For example, a reward of 1000 may be provided for the correct class and a reward of 0 for the incorrect class. In this case reward is not delayed. We refer to such redefined classification problems as

Fig. 2.2. In RL problems an adaptive agent interacts with an environment executing actions and receiving state information and reinforcement feedback.

single-step RL problems. On the other hand, *multistep RL problems* refer to RL problems in which reward is delayed and successive states depend on each other and on the chosen action. In the latter case, reward (back-)propagation is necessary.

Later, we see that LCSs are online generalizing RL mechanisms. Classification problems are usually converted into single-step RL problems when learned by LCSs. For convenience reasons we usually refer to these single-step RL problems as classification problems. However, the reader should keep in mind that when referring to classification problems in conjunction with an LCS application, the LCS actually faces a single-step RL problem.

Two types of multistep RL problems need to be distinguished: those that are modeled by a Markov decision process (MDP) and those that are modeled by a partially observable Markov decision process (POMDP).

Markov Decision Processes

We define a multistep problem as a Markov decision process (MDP), reusing notation from the classification problems where appropriate. An MDP problem consists of a set of possible sensory inputs $s \in \mathcal{S}$ (i.e. the states in the MDP); a set of possible actions $a \in \mathcal{A}$; a state transition function $f : \mathcal{S} \times \mathcal{A} \to \Pi(\mathcal{S})$, where $\Pi(\mathcal{S})$ denotes a probability distribution over all possible next states ($\mathcal{S}$); and a reinforcement function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \Re$. The state transition function defines probabilities for reaching all possible next states given the current state and the current action. The reinforcement function defines the resulting reward, which depends on the current state transition. For example, at a certain point in time t, state s_t may be given. The system may then decide on the execution of action a_t. The execution of a_t leads to the reception of reward r_t and the perception of the consequent state s_{t+1}.

An MDP is called a *Markov* decision process because it satisfies the Markov property: Future probabilities and thus the optimal action decision can be determined from the current state information alone since the state transition probabilities depend solely on the current state information.

A simple example of a (multistep) MDP problem is Maze 1—a simple maze environment shown in Figure 2.3. The learning system, or *agent*, may

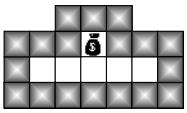


Fig. 2.3. Maze 1 is a simple MDP problem

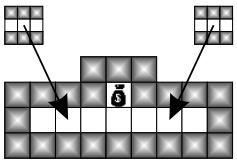


Fig. 2.4. The two identical looking states in Maze 2 turn the problem into a POMDP problem.

reside in one of the five positions in the maze perceiving the eight surrounding positions. An empty position may be coded by 0, a blocked position by 1. The money bag indicates a reward position in which reward is received and the agent is reset to a randomly chosen empty position. Note that each position has a unique perception code so that the problem can be modeled by an MDP process.

Partially Observable Markov Decision Processes

More difficult than MDP problems, are POMDP problems in which current sensory input may not be sufficient to determine the optimal action. Formally, a POMDP can be defined by a state space $\mathcal{X}$, a set of possible sensations $\mathcal{S}$, a set of possible actions $\mathcal{A}$, a state transition function $f : \mathcal{X} \times \mathcal{A} \rightarrow \Pi(\mathcal{X})$. In contrast to an MDP problem, however, states are not perceived directly but are converted into a sensation using an observation function $O : \mathcal{X} \rightarrow \Pi(\mathcal{S})$ that converts a particular state into a sensation. Similarly, the reward function does not rely on the sensations but on the underlying (unobservable) states $R : \mathcal{X} \times \mathcal{A} \times \mathcal{X} \rightarrow \Re$. In contrast to the MDP, a POMDP might violate the Markov property in that optimal action decisions cannot be made solely based on current sensory input.

A simple example of a POMDP problem is shown in Figure 2.4. Although only slightly larger than Maze 1, this maze does not satisfy the Markov property since the second empty position on the left looks identical to the second empty position on the right. Thus, given that the agent is currently in either of the two positions, it is impossible to know if the fastest way to the reward is to the left or to the right. Only an internal state or a short-term memory (that may, for example, indicate that the agent came from the left-most empty position) can disambiguate the two states and allow the agent to act optimally in the positions in question.

In the studies herein, we focus on MDP problems. Applications of the learning classifier system XCS to POMDP problems can be found elsewhere (Lanzi & Wilson, 2000; Lanzi, 2000), in which the system is enhanced with internal memory mechanisms somewhat similar to Holland's original message list idea.

2.2 Reinforcement Learning

Facing an MDP or POMDP problem, an RL system is the most appropriate system to solve the problem. In essence, the investigated rule-based evolutionary systems are RL systems that use GAs to evolve their state-action-value representation. Excellent introductions to RL are available (see Kaelbling, Littman, & Moore, 1996; Dietterich, 1997; Sutton & Barto, 1998) and the following overview can only provide a brief glance at RL. We introduce terminology and basic understanding necessary for the remainder of the book.

The task of an RL system is to learn an optimal behavioral policy interacting with an MDP (or POMDP) problem. A behavioral policy is a policy that decides on action execution. Given current sensory input (and possibly further internal state information) the behavioral policy is responsible for the action decision. A behavioral policy is optimal if it results in the maximum expectable reward in the long run. The most often used expression to formalize the expected reward is the *cumulative discounted reward*:

$$E(\sum_{t=0}^{\infty} \gamma^t r_t), \tag{2.2}$$

where $\gamma \in [0, 1]$ denotes the discount factor that weighs the importance of more distant rewards. Setting γ to zero results in a single-step RL problem in which only current reward is important. Setting γ to one results in a system in which cumulative reward needs to be optimized. Usually, γ is set to values close to one such as 0.9. RL essentially searches for a behavioral policy that maximizes the cumulative discounted reward.

Looking back at our small maze problem in Figure 2.3, we can see how much reward can be expected in each state executing an optimal behavioral policy. Assuming that the environment triggers a reward of 1000 when the rewarding position is reached, the three positions that are one step away from the reward position have an expected reward of 1000 and the two outermost positions have an expected reward of 900.

RL systems learn state value or state-action value representations using temporal difference learning techniques to estimate and maximize the expected discounted reward expressed in Equation 2.2. Two approaches can be distinguished: (1) *Model-free* learners learn an optimal behavioral policy directly without learning the state-transition function; (2) *Model-based* learners learn the state-transition function using it to learn or improve their behavioral policy.

2.2.1 Model-Free Reinforcement Learning

Two major approaches comprise the model-free RL realm: (1) $TD(\lambda)$ and (2) *Q-learning*. While the former needs an interactive mechanism that updates the RL-based critic and the behavioral policy in turn, the latter is doing the

same more naturally. After a short overview of $TD(\lambda)$ we focus on Q-learning since it is able to learn an optimal policy *off-policy* (that is, independent of the current behavioral policy). The RL mechanism implemented in the subsequently investigated learning classifier system XCS closely resembles Q-learning.

$TD(\lambda)$ methods interactively, or in turn, update their current behavioral policy π and the critic V^π that evaluates the policy π. Given the current critic V^π, a k-armed bandit optimization mechanism may be used to optimize π. Given current policy π, V^π may be updated using the $TD(\lambda)$ strategy. Hereby, each state value $V(s)$ is updated using

$$V(s) \leftarrow V(s) + \beta(r + \gamma V(s') - V(s))e(s), \tag{2.3}$$

$$e(s) = \sum_{k=1}^{t} (\lambda\gamma)^{t-k} \delta_{s,s_k}, \text{ where } \delta_{s,s_k} = \begin{cases} 1 & \texttt{if } s = s_k \\ 0 & \texttt{otherwise} \end{cases}$$

where $e(s)$ denotes the eligibility of state s, meaning its involvement in the achievement of the current reward. Parameter β denotes the learning rate somewhat reflecting the belief in the new information vs. the old information. A large β assumes very low prior knowledge resulting in a large change in the value estimate, whereas a small β assumes solid knowledge resulting in a small change in the value estimate. Parameter λ controls the importance of states in the past and δ monitors the occurrences of the states. With $\lambda = 0$, past states are not considered and only the currently encountered state transition is updated. Similarly, setting λ to one, all past states are considered as equally relevant. Note that the update is still discounted by γ so that the influence of parameter λ specifies the belief in the relevancy of the current update for the states encountered in the past (Kaelbling, Littman, & Moore, 1996).

A somewhat more natural approach is Watkins' Q-learning mechanism (Watkins, 1989). Instead of learning state values, Q-learning learns state-action values, effectively combining policy and critic in one. A *Q-value* of a state action pair (s, a) essentially estimates the expected discounted future reward if executing action a in state s and pursuing the optimal policy thereafter. Q-values are updated by

$$Q(s,a) \leftarrow Q(s,a) + \beta(r + \gamma \ \texttt{max}_{a'} Q(s',a') - Q(s,a)). \tag{2.4}$$

Due to the `max` operator, the Q-value predicts the average discounted reward of executing a in state s reaching state s' and following the optimal policy thereafter. Q-learning is guaranteed to converge to the optimal values as long as it is guaranteed that in the long run all state-action pairs are executed infinitely often and learning rate β is decayed appropriately. If the optimal Q-values are determined, the optimal policy is determined by

$$\pi^*(s) = \texttt{arg max}_a Q(s,a), \tag{2.5}$$

which chooses the action that is expected to maximize the consequent Q-value if executed in current state s. To ensure an infinite exploration of the

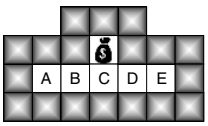
state-action space, an ϵ-greedy exploration strategy may be used:

$$\pi(s) = \begin{cases} \texttt{arg max}_a Q(s,a) & \texttt{with probability } 1-\epsilon \\ \texttt{rand}(a) & \texttt{otherwise} \end{cases}, \tag{2.6}$$

which chooses a random action with probability ϵ.

In our running Maze 1 example, the Q-table learned by a Q-learner is shown in Table 2.2. The environment provides a constant reward of 1000 when the money position is reached. Additionally, after reward reception, the agent is reset to a random position in the maze. The reward is propagated backward through the maze yielding lower reward to more distant positions. In effect, Q-learning is an online distance learning mechanism to a reward source where the reward prediction, or *Q-value*, indicates the worthiness or value of executing an action given the current situation. Note that worthiness in Q-learning is defined as the expected discounted future reward using Equation 2.4. Other discount mechanisms often work just as well depending on the task setup. For example, parameter γ could be set to one but actions may have an associated, potentially fixed, cost value which will be deducted from the expected future reward. In this case, explicit discounting is unnecessary since the environment—actually reflecting the inner architecture of the agent—would take care of discounting. The cost would reflect a potentially body-specific effort measure of the executed action.

Table 2.2. The Q-values in the shown Q-table reflect the value of each available action in each possible state of Maze 1 (shown on the left). Each value is defined as the immediate reward plus the expected discounted future reward received after executing the action indicated by the arrows in the state defined by the letters. The corresponding sensation is derived by specifying the surrounding conditions starting north and coding clockwise, specifying an obstacle by symbol 1 and a free space by symbol 0. The discount level is set to $\gamma = .9$.



state	sensation	↑	↗	→	↘	↓	↙	←	↖
A	11011111	810	810	900	810	810	810	810	810
B	10011101	900	1000	900	900	900	900	810	900
C	01011101	1000	900	900	900	900	900	900	900
D	11011100	900	900	810	900	900	900	900	1000
E	11111101	810	810	810	810	810	810	900	810

To summarize, Q-learning learns a Q-function that determines state-dependent value estimates for each available action in each state of the environment. Due to its well-designed interactive learning mechanism and proven convergence properties, it is commonly used in RL. As we will see, Q-learning forms the fundamental basis for the RL component used in the investigated LCSs.

2.2.2 Model-Based Reinforcement Learning

In addition to the reward prediction values, model-based RL techniques learn (potentially an approximation of) the state-transition function of the underlying MDP problem. Once the state-transition function is learned with sufficient accuracy, an optimal behavioral policy can be learned by simply simulating state-transitions offline using *dynamic programming* techniques (Bellman, 1957).

In dynamic programming, the state transition function is known to the system and used to estimate the payoff for each state-action pair. The literature on dynamic programming is broad, conveying many convergence results concerning different environmental properties such as circles, probabilistic state transitions, and probabilistic reward (see e.g. Bellman, 1957; Gelb, Kasper, Nash, Price, & Sutherland, 1974; Sutton & Barto, 1998).

In model-based RL, the state-transition function needs to be learned as suggested in Sutton's DYNA architecture (Sutton, 1990). DYNA uses state-transition experiences in two ways: (1) to learn a behavioral policy (using $TD(\lambda)$ or Q-learning); (2) to learn a predictive model of the state-transition function. Additionally, DYNA executes offline policy updates (independent of the environmental interactions) using the learned predictive model. Moore and Atkeson (1993) showed that the offline learning mechanism can be sped up significantly if the offline learning steps are executed in a prioritized fashion favoring updates that promise to result in large state(-action) value changes. DYNA and related techniques usually represent their knowledge in tabular form.

Later, we will see that GAs can be applied in LCSs to learn a more generalized representation of Q-values and predictions. LCSs essentially try to cluster the state space so that each cluster accurately predicts a certain value. GAs are used to learn the appropriate clusters.

2.3 Genetic Algorithms

Prior to LCSs, John H. Holland proposed GAs (Holland, 1971; Holland, 1975). Somewhat concurrently, evolution strategies (Rechenberg, 1973; Bäck & Schwefel, 1995) were proposed, which are very similar to GAs but are not discussed any further herein. Goldberg (1989) provides a comprehensive introduction to GAs including LCSs. Goldberg's recent book (Goldberg, 2002) provides a much more detailed analysis of GAs including scale-up behavior as well as problem and parameter/operator dependencies.

This section gives a short overview of the most important results and basic features of GAs. The interested reader is referred to Goldberg (1989) for a comprehensive introduction and to Goldberg (2002) for a detailed analysis of GAs leading to the design of *competent GAs*—GAs that solve boundedly difficult problems quickly, accurately, and reliably.

2.3.1 Basic Genetic Algorithm

GAs are evolutionary-based search or optimization techniques. They evolve a *population* (or set) of individuals, which are represented by a specific *genotype*. The decoded individual, also called *phenotype*, specifies the meaning of the individual. For example, when facing the problem of optimizing the ingredients of a certain dish, an individual may code the presence (1) or absence (0) of the available ingredients. The decoded individual, or phenotype, would then be the actual ingredients put together.

Basic GAs face an optimization problem as specified in Section 2.1.1. Feedback is provided in form of a fitness value that specifies the quality of an individual, usually in the form of a scalar value. If we face a maximization problem, a high fitness value denotes high quality. GAs are designed to progressively generate (that is, evolve) individuals that yield higher fitness.

Given a population of evaluated individuals, *evolutionary pressures* are applied to the population generating offspring and deleting old individuals. A basic GA comprises the following steps executed in each iteration given a population:

1. evaluation of current population
2. selection of high fitness individuals from current population
3. mutation and recombination of selected individuals
4. generation of new population

The evaluation is necessary to derive a fitness measure for each individual. Selection focuses the current population on more fit individuals, which represent more accurate subsolutions. Mutation is a diversification operator that searches in the syntactic, genotypic neighborhood of an individual by slightly changing its structure. Recombination, also called *crossover*, recombines the structure of parental individuals in the hope of generating offspring that combines the positive structural properties of both parents. Finally, the generation of the new population decides which individuals are part of the new population. In the simplest case, the number of reproduced classifiers equals the population size and the offspring individuals replace the old individuals. In a more steady-state version, less individuals are reproduced and only a subset of the old population is replaced. Replacement may also depend on fitness.

The remainder of this chapter analyzes the different methods in more detail. Hereby, we follow Goldberg's facetwise GA theory.

2.3.2 Facetwise GA Theory

To understand a system as complex as a GA, it is helpful to partition the system into its most relevant components and investigate those components separately Once the single components are sufficiently well understood, they may then be combined appropriately, respecting interaction constraints. This

is the essential idea behind Goldberg's facetwise approach to GA theory (Goldberg, 1991; Goldberg, Deb, & Clark, 1992; Goldberg, 2002).

Before we can do the partitioning, though, it is necessary to understand *how* a GA is supposed to work and *what* it is supposed to learn.

Since GAs are targeted to solve optimization problems evolving the solution that yields highest fitness, fitness is the crucial factor along the way to an optimal solution. Learning success in GAs relies on sufficient *fitness guidance* towards the optimal solution. However, fitness may be misleading as illustrated above in the trap problem (Table 2.1). Thus, the structural assumption made in GAs is that of *bounded difficulty*, which means that the overall optimization problem is composed of substructures, or BBs, that are of bounded length. The internal structure of one BB might be misleading in that a BB might, for example, resemble a trap problem. However, the overall problem is assumed to provide some fitness guidance so that good combinations of BBs are expected to lead towards the optimal solution.

With this objective in mind, it is clear that GAs should detect and propagate subproblems effectively. The goal is to design *competent GAs* (Goldberg, 2002)—GAs that solve boundedly difficult problems quickly, accurately, and reliably. Hereby, quickly means to solve the problems in low-order polynomial time (ideally subquadratic), with respect to the problem length. Accurately means to find a solution in small fitness distance from the optimal solution, and reliably means to find this solution with a low error probability.

The actual GA design theory (Goldberg, 1991; Goldberg, Deb, & Clark, 1992; Goldberg, 2002) then stresses the following points for GA success:

1. Know what GAs process: Building blocks
2. Know the GA challenge: BB-wise difficult problems
3. Ensure an adequate supply of raw BBs
4. Ensure increased market share for superior BBs
5. Know BB takeover and convergence times
6. Make decisions well among competing BBs
7. Mix BBs well

While we characterized building blocks and BB-wise difficult problems (that is, boundedly difficult problems) above, we elaborate on the latter points in the subsequent paragraphs. First, we focus on supply and increased market share. Next, we look at diversification, BB decision making, and effective BB mixing.

2.3.3 Selection and Replacement Techniques

As in real-live Darwinian evolution (Darwin, 1859), selection, reproduction, and deletion decide the life and death of individuals and groups of individuals in a population. Guided by fitness, individuals are evolved that are more fit for the problem at hand. Several selection and deletion techniques exist,

each with certain advantages and disadvantages. For our purposes most important are (1) *proportionate selection* (also often referred to as *roulette-wheel selection*) and (2) *tournament selection*. Most other commonly used selection mechanisms are comparable to either of the two. More detailed comparisons of different selection schemes can be found elsewhere (Goldberg, 1989; Goldberg & Deb, 1991; Goldberg & Sastry, 2001).

Proportionate Selection

Proportionate selection is the most basic and most natural selection mechanism: Individuals are selected for reproduction proportional to their fitness. The higher the fitness of an individual, the higher its probability of being selected. In effect, high fitness individuals are evolved. Given a certain individual with fitness f_i and an overall average fitness in the population $\overline{f}$, the proportion of individual p_i is expected to change as

$$p_i \leftarrow \frac{f_i}{\overline{f}} p_i \tag{2.7}$$

A similar equation can be derived for the proportion of a BB representation and its fitness contribution in the population (Goldberg & Sastry, 2001). The equation shows that structural growth by the means of proportionate selection strongly depends on *fitness scaling* and the current *fitness distribution* in the population (Baker, 1985; Goldberg & Deb, 1991; Goldberg & Sastry, 2001).

A simple example clarifies the dependence on fitness scaling. Given any fitness measure that needs to be maximized, the probability of reproducing the best individual is the following:

$$P(\texttt{rep. of best individual}) = \frac{f_{max}}{\overline{f}} p_i \tag{2.8}$$

Assuming now that we add a positive value x to the fitness function, the resulting probability of reproduction decreases to

$$P(\texttt{rep. of best individual scaled}) = \frac{f_{max} + x}{\overline{f} + x} p_i \tag{2.9}$$

The larger x, the smaller the probability that the best individual is selected for reproduction until all selection probabilities are equal to their current proportion, although the best individual should usually receive significantly more reproductive opportunities. The manipulation is certainly inappropriate but shows that a fitness function needs to be well-designed—or appropriately scaled—to ensure sufficient selection pressure when using proportionate selection.

The dependency on the current fitness distribution has a strong impact on the convergence to the global best individual of a population. The more similar the fitness values in a population, the less selection pressure is encountered by

the individuals. Thus, once the population has nearly converged to the global optimum, fitness values tend to be similar so that selection pressure due to proportionate selection is weak. This effect is undesirable if a single global solution is searched.

However, complete convergence may not necessarily be desired or may even be undesired if multiple solutions are being searched for. In this case, proportionate selection mechanisms might actually be appropriate given a reasonable fitness value estimate. For example, as investigated elsewhere (Horn, 1993; Horn, Goldberg, & Deb, 1994), proportionate selection can guarantee that all similarly good solutions (or subsolution niches) can be sustained with a population size that grows linearly in the number of niches and logarithmically in the time they are assured to be sustained. The only requirement is that fitness sharing techniques are applied and that the different niches are sufficiently non-overlapping.

Tournament Selection

In contrast to proportionate selection, tournament selection does not depend on fitness scaling (Goldberg & Deb, 1991; Goldberg, 2002). In tournament selection, tournaments are held among randomly selected individuals. The tournament size s specifies how many individuals take part in a tournament. The individual with the highest fitness wins the tournament and consequently undergoes mutation and crossover and then serves as a candidate for the next generation.

If replacing the whole population by individuals selected by tournament selection, the best individuals can be expected to be selected s times so that the proportion p_i of the best individual i can be expected to grow with s, that is:

$$p_i \leftarrow sp_i \tag{2.10}$$

That means that the best individual is expected to take over the population quickly since the proportion of the best individual grows exponentially in s. Thus, in contrast to proportionate selection, which naturally stalls late in the run, tournament selection pushes the better individuals until the best available individual takes over the whole population.

Supply

Before selection can actually be successful, BBs need to be available in the population. This leads to the important issue of initial BB *supply*. If the initial population is too small to guarantee the presence of all important BBs (expressed in usually different) individuals, a GA relies on mutation to generate the missing structures by chance. However, an accidental successful mutation is very unlikely (exponentially decreasing in the number of missing BB values). Thus, a sufficiently large population with an initially sufficiently large diversity is mandatory for GA success.

Niching

Very important for a successful application of GAs in the LCS realm is the parallel sustenance of equally important subsolutions. Usually, *niching* techniques are applied to accomplish this sustenance. Hereby, two techniques reached significant impact in the literature: (1) crowding, and (2) sharing.

In crowding (De Jong, 1975; Mahfoud, 1992; Harik, 1994) the replacement of classifiers is restricted to classifiers that are (usually syntactically) similar. For example, in the restricted tournament selection technique (Harik, 1994), offspring is compared with a subset of other individuals in the population. The offspring competes with the (syntactically) closest individual, replacing it, if its fitness is larger.

In sharing techniques (Goldberg & Richardson, 1987) fitness is shared among similar individuals where similarity is defined by an appropriate distance measure (e.g. Hamming distance in the simplest case). The impact of sharing has been investigated in detail (Horn, 1993; Horn, Goldberg, & Deb, 1994; Mahfoud, 1995). Horn, Goldberg, and Deb (1994) highlight the importance of fitness sharing in the realm of LCSs showing the important impact of sharing on the distribution of the population and potentially near infinite niche sustenance due to the applied sharing technique.

Although sharing can be beneficial in non-overlapping (sub-)solution representations, the more the solutions overlap, the less beneficial fitness sharing techniques become. Horn, Goldberg, and Deb (1994) propose that sharing works successfully as long as the overlap proportion is smaller than the fitness ratio between the competing individuals. Thus, if the individuals have identical fitness, only a complete overlap eliminates the sharing effect. In general, the higher the degree of overlap, the smaller the sharing effect and thus the higher the probability of losing important BB structures due to genetic drift.

2.3.4 Adding Mutation and Recombination

Selection alone certainly does not do much good. In essence, selection uses a complicated method to find the best individuals in a given set of individuals. Certainly, this can be done much faster by simple search mechanisms. Thus, selection needs to be combined with other search techniques that search in the *neighborhoods* of the current best individuals for even better individuals. The two basic operators that accomplish such a search in GAs are *mutation* and *crossover*.

Simple mutation takes an individual as input and outputs a slight variation of the individual. In the simplest form when coding an individual in binary, mutation randomly flips bits in the individual with a certain probability μ. Effectively, mutation searches in the *syntactic* neighborhood of the individual where the shape and size of the neighborhood depend on the mutation operator and the mutation probability μ.

Crossover is designed to recombine current best individuals. Thus, rather than searching in the syntactic neighborhood of one individual, crossover searches in the neighborhood defined by two individuals, resulting in a certain type of knowledge exchange among the crossed individuals. In the simplest case when coding individuals in binary, *uniform crossover* exchanges each bit with a 50% probability, whereas one-point or two-point crossover choose one or two positions in the bit strings and exchange the right or the inner part of the resulting partition, respectively.

It should be noted that uniform crossover does not assume any relationship among bit positions, whereas one- and two-point crossover implicitly assume that bits that are close to each other depend on each other since substrings are exchanged. One-point crossover additionally assumes that beginning and ending are unrelated to each other, whereas two-point crossover assumes a more circular coding structure. More detailed analyses of simple crossover operators can be found elsewhere (Bridges & Goldberg, 1987; Booker, 1993).

It is important to appreciate the effects of mutation and crossover alone, disregarding selection for a moment. If selecting randomly and simply mutating individuals, mutation causes a general *diversification* in the population. In the long run, each attribute in an individual will be set independently uniformly distributed resulting in a population with maximum entropy in its individuals. In combination with selection, mutation causes a search in the syntactic neighborhood of an individual where the spread of the neighborhood is controlled by the mutation type and rate. Mutation may be biased, incorporating potentially available problem knowledge to improve the neighborhood search as well as to obey problem constraints.

Recombination, on the other hand, exchanges information among individuals syntactically dependent on the structural bias in the applied crossover operator. Selecting randomly, random crossover results in randomized shuffling of the individual genotypes. In the long run, crossover results in a random distribution of attribute values over the classifiers but does not affect the proportion of each value in the population.

In conjunction with selection, crossover is responsible for effective BB processing. Crossover needs to be designed to ensure proper BB recombination by exchanging important BBs among individuals and preventing the disruption of BBs in the meantime. Goldberg (2002) compares this very important exchange of individual substructures with innovation. Since innovation essentially refers to a successful (re-)combination of available knowledge in a novel manner, GAs are essentially designed (or should be designed) to do just that—be innovative in a certain sense.

Unfortunately, standard crossover operators are not guaranteed to propagate BBs effectively. BBs may also be disrupted by destroying important BB structures when recombining individuals. Dependent on the crossover operator and BB structure, a probability may be derived that the BB is not fully exchanged but cut by crossover. If it is cut and only a part is exchanged, the

BB structure may get lost dependent on the structure present in the other individual.

To prevent such disruption and design a more directed form of innovation, estimation of distribution algorithms (EDA) were recently introduced to GAs (Pelikan, Goldberg, & Lobo, 2002; Larrañaga, 2002). These algorithms estimate the current structural distribution of the best individuals in the population and use this distribution estimation to constrain the crossover operator or to generate better offspring directly from the distribution. In Chapter 7, we show how to incorporate mechanisms from the extended compact GA (ECGA) (Harik, 1999), which learns a non-overlapping BB structure, as well as from the Bayesian optimization algorithm (BOA), which learns a Bayes model of the BB structure, into the investigated XCS classifier system.

2.3.5 Component Interaction

We already discussed that a combination of selection with mutation and recombination leads to local search plus potentially innovative search. In addition to the impact of selection on growth and convergence, selection strength is interdependent with mutation and recombination. The two methods interact in that selection propagates better individuals while mutation and crossover search in the neighborhood of these individuals for even better solutions. Consequently, the growth of better individuals (or better BBs), often characterized by a *take over time* (Goldberg, 2002), needs to be balanced with the search in the neighborhood of the current best solutions. Too strong selection pressure may result in a collapse of the population to one only locally optimal individual, preventing effective search and innovation via mutation and crossover. On the other hand, too weak selection pressure may allow *genetic drift* that can cause the loss of important BB structures by chance.

These insights led to the proposition of a *control map* for GAs (Goldberg, Deb, & Thierens, 1993; Thierens & Goldberg, 1993; Goldberg, 1999) that characterizes a region of selection and recombination parameter settings in which a GA can be expected to work. The region is bounded by *drift*, when selection pressure is too low, *cross-competition*, when selection is too high, and *mixing*, when knowledge exchange is too slow with respect to the selection pressure. The mixing bound essentially characterizes the boundary below which knowledge exchange caused by crossover is not strong enough with respect to the selection pressure applied. Essentially the time until the expected generation of a better individual needs to be shorter than the mentioned take over time of the current best individual. For further details on these important factors, the interested reader is referred to Goldberg (2002).

2.4 Summary and Conclusions

This chapter introduced the three major problem types addressed in this book: (1) optimization problems, (2) classification problems, and (3) reinforcement

learning problems. Optimization problems require effective search techniques to find the best solution to the problem at hand. Classification problems require a proper structural partition into different problem classes. RL problems additionally require reward backpropagation.

RL techniques are methods that solve MDP problems online applying dynamic programming techniques in the form of temporal difference learning to estimate discounted future reward. The behavioral policy is optimized according to the estimated reward values. In the simplest case, RL techniques use tables to represent the expected cumulative discounted reward with respect to a state or a state-action tuple.

The most prominent RL technique is Q-learning, which is able to learn an optimal behavioral policy online without learning the underlying state transition function in the problem. Q-learning learns off-policy, meaning that it does not need to pursue the current optimal policy in order to learn the optimal policy.

Genetic algorithms (GAs) are optimization techniques derived from the idea of Darwinian evolution. GAs combine fitness-based selection with mutation and recombination operators to search for better individuals with respect to the current problem. Individuals usually represent complete solutions to a problem.

Effective building block (BB) processing is mandatory in order to solve problems of bounded difficulty. Effective BB processing was recently successfully accomplished using statistical modeling techniques that estimate the dependency structures in the current population and bias recombination towards propagating the identified dependencies.

Niching techniques are very important when the task is to sustain a subset of equally good solutions or different subsolutions for different subspaces (or niches) in the problem space. Fitness sharing and crowding are the most prominent niching methods in GAs.

Goldberg's *facetwise analysis* approach to GA theory significantly improved GA understanding and enabled the design of competent GAs. Although the facetwise approach has the drawback that the found models may need to be calibrated to the problem at hand, the advantages of the approach are invaluable for the analysis and design of highly interactive systems. First, crude models of system behavior are derivable cheaply. Second, analysis is more effective and more general since it is adaptable to the actual problem at hand and focuses only on most relevant problem characteristics. Finally, the approach enables more effective system design and system improvement due to the consequently identifiable rather independent aspects of problem difficulty.

The following chapters investigate how RL and GA techniques are combined in LCSs to solve classification problems and RL problems effectively. Similar to the facetwise decomposition of GA theory and design, we propose a facetwise approach to LCS theory and design in the next chapter. We then

pursue the facetwise approach to analyze the XCS classifier system qualitatively and quantitatively. The analysis also leads to the design of improved XCS learning mechanisms and to the proposition of more advanced LCS-based learning architectures.