3

Simple Learning Classifier Systems

Learning Classifier Systems (LCSs) (Holland, 1976; Booker, Goldberg, & Holland, 1989) are rule-based evolutionary learning systems. A basic LCS consists of (1) a set of rules, that is, a *population of classifiers*, (2) a rule evaluation mechanism, which usually is realized by adapted reinforcement learning (RL) (Kaelbling, Littman, & Moore, 1996; Sutton & Barto, 1998) techniques, and (3) a rule evolution mechanism, which is usually implemented by a genetic algorithm (GA) (Holland, 1975). The classifier population codes the current knowledge of the LCS. The evaluation mechanism estimates and propagates rule utility. Based on the estimated utilities, the evolutionary mechanism generates offspring classifiers and deletes less useful classifiers.

LCSs can be distinguished between *online learning* LCSs and *offline learning* LCSs. Moreover, they can be distinguished between LCSs that evolve a single solution, often referred to as Michigan-style LCSs, and LCSs that evolve a set of solutions, often referred to as Pittsburgh-style learning classifier systems (DeJong, Spears, & Gordon, 1993; Llorà & Garrell, 2001b). These systems are usually applied in offline learning scenarios only (also referred to as batch learning).

We analyze LCSs in a modular, facetwise way. That is, different facets of successful LCS learning are analyzed separately and then possible interactions between the facets are considered. The facetwise analysis focuses on appropriate identification, propagation, sustenance, and search of a complete and accurate problem solution. While most of the work focuses on one particular (online learning, Michigan-style) LCS, that is, the accuracy-based learning classifier system XCS (Wilson, 1995), the basic analysis and comparisons as well as the drawn conclusions should readily carry over to other types of LCSs neither restricted to online-learning LCSs nor to Michigan-style LCSs.

This chapter first gives a general introduction to a simple LCS in tutorial form, assuming knowledge about both the basic functioning of a GA as well as basic RL principles. An illustrative example provides more details on basic LCSs. Section 3.3 introduces our facetwise theory approach. Summary and conclusions wrap up the most important lessons of this chapter.

3.1 Learning Architecture

LCSs have a rather simple, but interactive, learning structure combining the strengths of GAs in search, pattern recognition, pattern propagation, and innovation with the value estimation capabilities of RL. The result is a learning system that generates online a generalized state-action value representation. Depending on the complexity of the problem and the number of different states, the generalization capability is able to save space as well as time to learn an optimal behavioral policy. Similarly, in a classification problem scenario, LCSs may be able to detect distributed dependencies in data focusing on the most relevant ones. Conveniently, the dependencies are usually directly reflected in the emerging rules, allowing not only statistical datamining but also more qualitatively oriented datamining.

The basic interaction of the three major components of a learning classifier system and the environment is illustrated in Figure 3.1. While the RL component controls the interaction with the environment, the evolutionary component evolves the problem representation, that is, classifier condition and action parts. Thus, the learning mechanism interacts not only with the environment but also within itself. The evolutionary component relies on appropriate evaluation measures from the RL component and, vice versa, the RL component relies on appropriate classifier structure generated by the GA component to be able to estimate future reinforcement accurately. The interaction between the two components is the key to LCS success. However, proper interaction alone does not assure success. This will become particularly evident in our later analyses. The following paragraphs provide a more concrete definition of a simple learning classifier system LCS1.

3.1.1 Knowledge Representation

To be more concrete, we define a learning classifier system LCS1. LCS1 consists of a population of maximum size N of classifiers. Each classifier cl consists of a condition part $cl.C$, an action part $cl.A$ and a reward prediction value $cl.R$ (using the dot notation to refer to parts of a classifier). Classifier cl predicts reward $cl.R \in \Re$ given its condition $cl.C$ is satisfied, and given further that action $cl.A \in \mathcal{A}$ is executed.

Depending on the representation of the problem space $\mathcal{S}$ (e.g. binary, nominal, real...), conditions may be defined in various ways from simple exact values, over value ranges, to more complex, kernel-based conditions such as radial basis functions. Each classifier condition defines a problem subspace. The population of classifiers as a whole usually covers the complete problem space. Classifiers with non-overlapping conditions (specifying completely different subspaces) are independent with respect to the representation and may be considered as implicitly connected by an `or` operator. Overlapping classifiers compete for activity and reward.

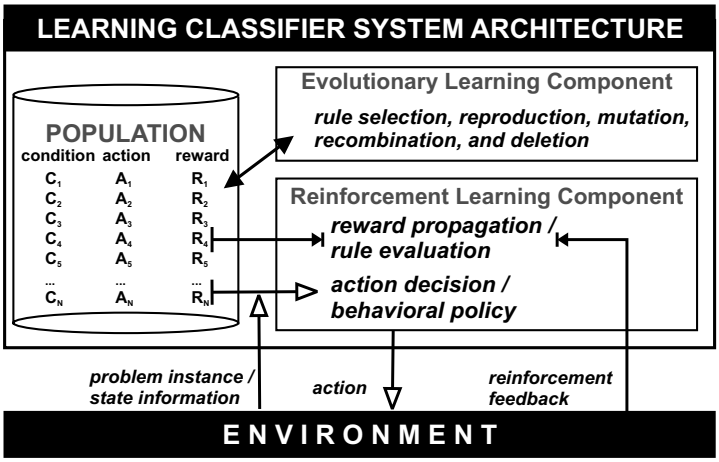

Fig. 3.1. The major components in a learning classifier system are the knowledge base, which is represented by a set of rules (that is, a population of classifiers), the evolutionary component, which evolves classifier structure based on their reward estimation values, and the RL component, which evaluates rules and decides on actions (or classifications).

Let's consider the binary case corresponding to our definition of a Boolean function problem (Chapter 2), as well as our example of Maze 1 (Figure 2.3 on page 17). The binary input string $\mathcal{S} = \{0,1\}^l$ is matched with the conditions that specify the attributes it requires to be correctly set. Traditionally, in its simplest form a condition is represented by the ternary alphabet $C \in \{0,1,\#\}^l$ where the *don't care* symbol # matches both zero and one.[1] If the condition part is satisfied by the current problem instance, the classifier is said to *match*. Table 3.1 shows an example of a potential problem instance and all conditions that match this problem instance.

Introducing a little more notation, a classifier cl may be said to have a certain *specificity* $\sigma(cl)$. In the binary case, we may define specificity as the ratio of the number of specified attributes to the overall number of attributes. For example, given a problem of length l and a classifier with k specialized (not don't care) attributes, the classifier has a specificity of $\frac{k}{l}$. Similar definitions may be used for other problem domains and other condition representations. Essentially, specificity is a measure that characterizes how much of the problem space a classifier covers. A specificity of one means that only one possible problem instance is covered whereas a specificity of zero means that all problem instances (the whole problem space) is covered. Thus, the

[1] Note that the hash symbol might not be expressed explicitly representing a condition part by a set of position-value tuples corresponding to the attributes in the traditional representation that are set to zero or one. This representation has significant computational advantages when the rules only specify a few attributes.

Table 3.1. All classifier conditions whose specified attributes are identical to the corresponding values in the problem instance match the current problem instance. The more general a condition part, the more problem instances it matches.

instance	matching conditions	matching problem instances				condition
1001	`1001`				1001	`1001`
	`100# 10#1 1#01 #001`			1001	1000	`100#`
	`10## 1#0# #00#`		1011	1001	1000	`10##`
	`1##1 #0#1 ##01`	1111	1101	0011	0001	`###1`
	`###1 ##0# #0## 1###`	1111	1110	0001	0000	`####`
	`####`					

larger the specificity of a classifier, the less of the problem space is covered by the classifier. Specificity is an important measure in LCSs, useful for deriving and quantifying evolutionary pressures, problem bounds, and parameter values. Subsequent chapters derive several problem bounds and performance measures based on specificity.

3.1.2 Reinforcement Learning Component

Given a current problem instance, an LCS forms a *match set* $[M]$ of all classifiers in the population $[P]$ whose conditions are satisfied by the current problem instance. The match set reflects the knowledge about the current state (given the current problem instance). An LCS uses the match set $[M]$ to decide on an action.

The action decision is made by the behavioral policy π controlled by the RL component. In the simple case, we can use an adapted ϵ-greedy action selection mechanism. The predicted action value may be decided upon by averaging over the reward predictions of all matching classifiers for each action. The consequent behavioral policy may be written as

$$\pi_{LCS1}(s) = \begin{cases} \texttt{arg max}_a \frac{\sum_{\{cl \in [M] | cl.A = a\}} cl.R}{|\{cl \in [M] | cl.A = a\}|} & \texttt{with prob. } 1-\epsilon \\ \texttt{rand}(a) & \texttt{otherwise} \end{cases}, \tag{3.1}$$

where s denotes the current problem instance, a the chosen action and $\{cl \in [M] | cl.A = a\}$ the set of all classifiers in the match set $[M]$ whose action part specifies action a. As a result of the action decision $a' = \pi_{LCS1}(s)$ a corresponding action set $[A]$ is formed that consists of all classifiers in the current match set $[M]$ that specify action a' ($[A] = \{cl \in [M] | cl.A = a'\}$).

After the reception of the resulting immediate reward r and the next problem instance s_{+1} yielding match set $[M]_{+1}$, all classifier reward predictions in $[A]$ are updated using the adapted Q-learning equation:

$$R \leftarrow R + \beta(r + \gamma \texttt{max}_a \frac{\sum_{\{cl \in [M]_{+1} | cl.A = a\}} cl.R}{|\{cl \in [M]_{+1} | cl.A = a\}|} - R), \tag{3.2}$$

estimating the maximum expected discounted future reward by the average of all participating classifiers. The `max` operation indicates the relation to Q-learning. The difference is that in Q-learning only one value is used to estimate a Q-value, whereas in LCS1 a set of classifiers together estimates the resulting Q-value. If all conditions were completely specific, LCS1 would do Q-learning, predicting each Q-value by the means of a single, fully specific classifier.

3.1.3 Evolutionary Component

At this point, we know how reward prediction values are updated and how they are propagated in an LCS. What remains to be addressed is how the underlying conditional structure evolves. Two components are responsible for classifier structure generation and evolution: (1) a covering mechanism, and (2) a GA.

The covering mechanism is mostly applied early in a run to ensure that all problem instances are covered by at least one rule. Given a problem instance, a rule may be generated that sets the value of each attribute with probability $(1 - P_{\#})$ to the current value and to a don't care symbol otherwise. Note that covering may be mainly avoided by initializing sufficiently general classifiers. Particularly, if adding classifiers for all possible actions with completely general conditions (all don't care symbols) to the population, covering will not be necessary because the completely general classifiers always match. In this case, the GA needs to fully take care of structure evolution starting from completely (over-)general classifiers.

In its simplest form, we use a steady-state GA (similar to an $(N + 2)$ evolution strategy mechanism (Rechenberg, 1973; Bäck & Schwefel, 1995)). That is, in each learning iteration, the evolutionary component selects two offspring classifiers using, for example, proportionate selection based on the reward predictions R. The selected two classifiers are reproduced, mutated, and recombined yielding two offspring classifiers. For example, mutation can change a condition attribute, with a certain probability μ, to one of the other possible values. Additionally, the action part may be mutated with probability μ. Recombination combines the condition parts with a probability χ applying, for example, uniform crossover. The two offspring classifiers replace two other classifiers, which can be selected using proportionate selection on the inverse of their fitness (e.g. $\frac{1}{1+R}$).

In the case of such a simple GA mechanism, the GA searches in the syntactic (genotypic) local neighborhood of the current population. Selection is biased towards selecting higher reward offspring, consequently propagating classifier structures that predict high reward on average and deleting classifiers that expect low reward on average.

In combination with the RL component, the GA should evolve structures that receive high reward on average. Unfortunately, this is not enough to ensure successful learning. Section 3.3 introduces a general theory of learning in LCSs that reveals the drawbacks of this simple LCS system.

3.2 Simple LCS at Work

Let's do a hypothetical run with our simple LCS1 on the Maze 1 problem (see Figure 2.3 on page 17). Perceptions are coded starting north and coding clockwise indicating an obstacle by 1 and a free position by 0. The money position is perceived as a free position. For example, consider the population shown in Figure 3.2 (generated by hand). Classifier 1 is a classifier that identifies a move to the north whenever there is no obstacle to the east, whereas Classifier 2 considers a move to the north whenever there is no obstacle on the west side. The shown reward values reflect the expected reward received if all situations were equally likely and the correct Q-values were propagated (effectively an approximation of the actual values).

In the shown iteration, the provided problem instance `01011101` indicates that there is a free space north, east, and west (as a result of residing in the position just south of the money position). The problem instance triggers the formation of a match set, as indicated in Figure 3.2. In the example, the classifiers shown in the match set predict a reward of 950.75 for action ↑ and 900 for action ↓. When action ↑ is executed, the money position is reached, a reward r of 1000 is received, and the reward predictions of all classifiers in the current action set are updated, applying Equation 3.2 (shown are updates using learning rate $\beta = 0.2$). It can be seen how the reward estimation values of all classifiers increase towards 1000.

Finally, a GA is applied that selects two classifiers from the population, reproduces, mutates, and recombines them, and replaces two existing classifiers by the new classifiers. For example, the GA may select classifiers three and six, reproducing them, mutating them to e.g. $3'$=(`0####1###`,↑) and $6'$=(`0101#101`,↑), recombining them using one-point crossover to e.g. 3^*=(`0####1101`,↑) and 6^*=(`0101####`,↑), and finally reinserting 3^* and 6^* into the population, replacing two other classifiers (e.g. the lower reward classifiers two and five).

We can see that the evolutionary process propagates classifiers that specify how to reach the rewarding position. Due to the bias of reproducing classifiers that predict higher reward, on average, higher-reward classifiers will be reproduced more often and will be deleted less often. Mutation and crossover serve as the randomized search operators that are looking for better solutions in the (syntactically) local neighborhood of the reproduced classifiers.

Our example already exhibits several fundamental challenges for simple learning classifier systems: the problem of *strong overgenerals* (Kovacs, 2000), investigated in detail elsewhere (Kovacs, 2003), the problem of generalization, and the problem of local vs. global competition. These issues are the subject of the following section, in which we develop a facetwise theory approach for LCS analysis and design.

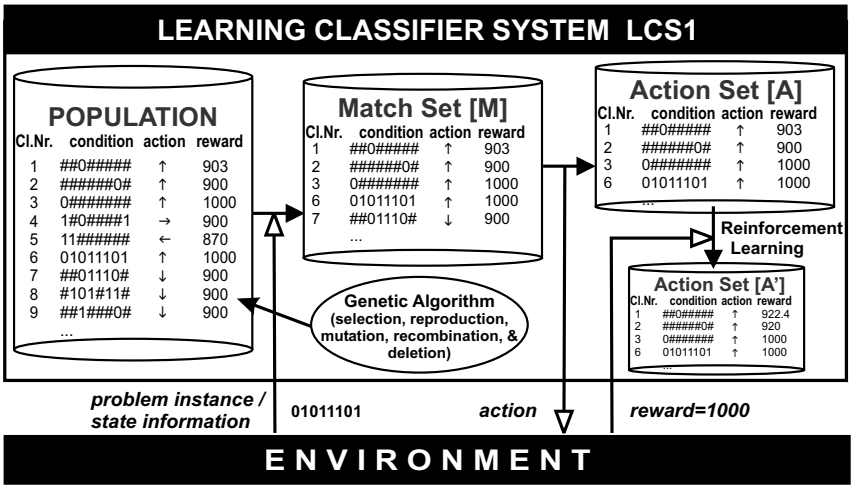

Fig. 3.2. In a typical learning iteration, an LCS receives a current problem instance consequently forming the match set $[M]$. Next, an action is chosen (in this case action $\uparrow$) and executed in the environment. Since the resulting reward equals one thousand, the reward estimates are increased (shown is an increase using learning rate $\beta = 0.2$). Finally, the GA may be applied on the updated population.

3.3 Towards a Facetwise LCS Theory

The introduction of LCS1 should have clarified several important properties of the general LCS learning architecture, such as the online learning property and the interaction of rule evaluation (using reinforcement methods) and rule structure learning (using evolutionary computation methods). However, it remains to be understood if and how the interactions can be assured to learn a complete problem solution.

It is clear that on average classifiers which receive a higher reward will be selected for reproduction more often and will be deleted less often. Thus, the GA mechanism propagates their structure by searching in the local neighborhood of these structures. Can we make specific learning projections? How general will the evolved solution be? How big can the problem be, given a certain population size? How distributed will the final population be?

This section addresses these issues and develops a facetwise theory for LCSs to answer them in a modular way. The section first gives a general outlook of which solution LCSs evolve and *how* this may be accomplished. Next, a theory approach, which addresses *when* this may be accomplished, is proposed.

3.3.1 Problem Solutions and Fitness Guidance

The above sections showed that LCSs are designed to evolve a distributed problem solution represented by a population of classifiers. The condition of each classifier defines the subspace for which the classifier is responsible. The action specifies the proposed solution in the defined subspace. Finally, the reward measure estimates the consequent payoff of the chosen action. It is desired that the action with the highest estimated payoff equals the best action in the current problem state.

To successfully evolve such a distributed problem solution we need to prevent overgeneralization and we rely on sufficient fitness guidance (that is, a fitness gradient) towards better classifiers. The two issues are discussed below starting with the problem of overgenerality and the problem of strong overgenerals in particular.

The problem of *strong overgenerals* (Kovacs, 2001) concerns a particular generality vs. specificity dilemma. The problem is that a general classifier cl_g (one whose conditions are satisfied in many problem instances or states) may have a higher reward prediction value, on average, than a more specialized classifier cl_s, which may match in a subset of cl_g. Consequently, the GA will propagate cl_g. Additionally, given that the actions are different in cl_g and cl_s, action $cl_g.A$ has preference over action $cl_s.A$. However, action $cl_s.A$ may yield higher reward in situations in which cl_s also matches. Thus, although action $cl_s.A$ would be more appropriate in the described scenario, action $cl_g.A$ will be chosen by the behavioral policy and will be propagated by the GA. Such an overgeneral classifier is called *strong overgeneral* classifier because it has a higher reward estimate than other, more specific classifiers and is consequently stronger with respect to evolutionary reproduction and action selection. The following example helps to clarify the problem.

Considering classifiers 1 and 4 in Figure 3.2, it can be seen that both classifiers match in the left-most position of Maze 1 (Figure 2.3 on page 17), which is perceived as `11011111`. The best action in this position is certainly to move to the east, which yields a discounted reward of 900 (as correctly predicted by Classifier 4). However, Classifier 1 predicts a slightly higher reward for action ↑, since its reward reflects the average reward encountered when executing action ↑ in its matching states A, B, C, and D. The values would be exact if the classifier was updated sufficiently often, the learning rate was sufficiently small, and all states were visited equally often, and action north was executed equally often in each of the states. Thus, the incorrect action ↑ may be executed (dependent on the other classifiers in the population) although the action → would be correct.

The basic problem shows that good classifiers cannot be distinguished from bad classifiers as easily as initially thought. In effect, it needs to be questioned if the fitness approach—deriving fitness directly from the absolute reward received—is appropriate, or rather, in which problems a direct reward-based fitness approach is appropriate. Kovacs (Kovacs, 2001; Kovacs, 2003)

analyzes this problem in detail. The most important result is that strong overgenerals can occur in any problem in which more than two reward values may be perceived (and thus essentially in all but the most trivial multistep problems). The severeness of this problem consequently demands that the fitness approach itself should be changed.

Solutions to this problem are the *accuracy-based* fitness approach in XCS (Wilson, 1995), investigated in detail in later chapters, and the application of *fitness sharing* techniques, as applied in ZCS (Wilson, 1994). In the former case, local fitness is modified requiring explicitly that the reward prediction of a classifier is accurate (that is, it has low variance). In the latter case, reward competition in the problem subspaces (defined by the current problem instance) can cause the extinction of strong overgeneral classifiers.

The problem of strong overgenerals illustrates how important it is to exactly define (1) the structure of the problem addressed (to know which challenges are expected) and (2) the objective of the learning system (to know how the system may "misbehave"). Our strength-based system LCS1, for example, works sufficiently well (with respect to strong overgenerals) in all classification problems since only two reward values are received and reward is not propagated. However, other challenges may have to be faced as investigated below.

Once we can assure that classifiers in the optimal solution will have the highest fitness values, we need to ensure that fitness itself *guides* the learning process towards these optimal classifiers. It should be acknowledged that overgeneral classifiers should have lower fitness values by definition of the optimal solution. To what degree fitness guides towards higher fitness values depends on the fitness definition and problem properties. Later chapters address this problem in detail with respect to the XCS classifier system and typical problem properties.

3.3.2 General Problem Solutions

While the problem of strong overgenerals is concerned with a particular phenomenon resulting from the interaction of a classifier structure, reinforcement component, and evolutionary computation component, the problem also points to a much more fundamental problem: the problem of generalization. When we introduced LCSs above, we claimed that they can be characterized as online generalizing RL systems. And in fact, as the classifier structure suggests, rules are often matching in several potential problem instances or states. However, until now it was not addressed at all why and how the evolutionary component may propagate more general classifiers instead of more specific ones.

Considering again our Maze 1 (Figure 2.3 on page 17) and the exemplar population shown in Figure 3.2, classifiers 6 and 3 actually contain the same amount of information: Both classifiers only match in maze position C and both classifiers predict that a move to the north yields a reward of 1000.

Clearly, Classifier 6 is syntactically much more specialized. The general concept of Classifier 3, which only requires a free space to the north and does not care about any other position, appears more appealing and might be the best concept in the addressed environment. In general, the aim is to stay as general as possible, identifying the minimal set of attributes necessary to predict a Q-value correctly.

On the other hand, consider classifiers 7 and 8 in Figure 3.2. Both classifiers predict that moving south results in a reward of 900. Both classifiers are syntactically equally specific, that is, both have an order of five (five specified attributes). However, Classifier 7 is semantically more general than Classifier 8 because it matches in more states than Classifier 8 (all three states below the money vs. only the states south and southeast of the money). Classifier 9 is semantically as general as Classifier 7 but it is syntactically more general. Later, we will see that XCS biases its learning towards syntactic and semantic generality using different mechanisms.

In the general case, the quest for generality leads us to a multi-objective problem in which the objectives are to learn the underlying problem as accurately as possible and to represent the solution with the most general classifiers and the least number of classifiers possible. This problem is addressed explicitly elsewhere (Llorà & Goldberg, 2003; Llorà, Goldberg, Traus, & Bernadó, 2003), in which a Pareto-front of high fitness, high generality classifiers is propagated. Other approaches, including the mechanism in XCS, apply a somewhat constant generalization pressure that is overruled by the fitness pressure if higher fitness is still achievable. Yet another generalization approach, recently proposed by Bull (2003), was applied in the ZCS system. In this case, reproduction causes fitness deduction. The lost fitness can only be regained by reapplication. More general classifiers will be reapplied faster and thus do not suffer as much from the reproduction penalty and eventually take over the population.

In Chapter 5, the effects of different generalization mechanisms are analyzed in more detail.

3.3.3 Growth of Promising Subsolutions

Once we know which solution we intend to evolve with our LCS system, how fitness may guide us to the solution, and how the solution will tend to be general, we need to ensure that our intentions can be put into practice. Thus, we need to ensure the growth of higher fitness classifiers.

To do this, it is necessary to ensure that classifiers with higher fitness are available in the population. Once we can assure that better classifiers are present, we need to assure that the RL component and the genetic component can interact successfully to reproduce higher fitness classifiers. Therefore it is necessary to assure that the RL component has enough *evaluation time* to detect higher fitness classifiers reliably. Moreover, the genetic component

needs to reproduce and thus propagate those better classifiers before they tend to be deleted.

The first aspect is related to the *BB supply* issue in GAs. However, due to the distributed problem representation, a more diverse supply may need to be ensured, and the definition of supply differs. In essence, the initial population needs to be general enough to cover the whole problem space, but it also needs to be diverse enough to have better solutions available for identification and propagation. The diversification and specialization effects of mutation may support the supply issue. These ideas become much more concrete when investigating the XCS classifier system in Chapter 6.

Note that supply is not only relevant in the beginning of a run, but it is actually relevant at all stages of the learning progress, continuously requiring the supply or generation of better offspring classifiers. However, the issue is most relevant in the beginning due to the fact that later in the run, the currently found distributed problem solution usually significantly restricts the search space to the immediate surrounding of these solutions. In the beginning of a run, the whole search space is the surrounding and any randomized search operator, such as mutation, can be expected—dependent on the problem—to have a hard time to find better classifiers by chance.

Once better classifiers are available, we need to ensure that they are identified. Since the RL component requires some time to identify better classifiers (iteratively updating the reward estimates), better classifiers need to have a sufficiently long survival time. Thus, offspring classifiers need to undergo several evaluations before they are deleted.

Finally, if better classifiers are available and the RL component has enough time to identify them, it is necessary to ensure that the genetic component propagates them. Thus, the survival time also needs to be long enough to ensure the reproduction of better classifiers. Additionally, genetic search operators may need additional time to effectively detect important problem substructures and subspaces. Due to the potentially unequally distributed problem complexity in problem space (see, for example, the problem in Figure 3.6 on page 46), different time may need to be available for different problem subspaces. Chapter 6 investigates these issues in detail with respect to the XCS classifier system.

3.3.4 Neighborhood Search

Once we can assure that higher fitness classifiers undergo reproduction and thus grow in the population, we need to implement effective neighborhood search in order to detect even better problem solutions. These problem solutions can be expected to lie in the neighborhood of the currently best subsolution or in further partitions of the current subsolution subspace, defined by the classifier conditions. The neighborhood search especially is very problem dependent and thus it is impossible to define generally optimal search operators. We now first look at simple mutation and crossover and their impact

on genetic search. Next, we discuss the issue of local vs. global search bias in somewhat more detail.

Mutation

Mutation generally searches in the syntactic neighborhood of a selected classifier. A simple mutation operator changes some attributes in the condition part of a classifier as well as the class of the classifier. If the class is changed, the new classifier basically considers the possibility that the relevant attributes for one class might also be appropriate for a reward prediction in another class. This might be helpful especially in multistep problems, where classifiers often develop conditions that identify sets of states that are equally distant from reward.

Mutation of the condition part can have three types of effects that may apply in combination if several attributes of one condition part are mutated: (1) generalization, (2) specialization, (3) knowledge transfer. Considering the ternary alphabet $C \in \{0, 1, \#\}^l$ and given an attribute with value 0, mutation may change the attribute to #. In this case, the classifier is generalized (its specificity decreases) since its condition covers a larger problem subspace (in the binary case, double the space). On the other hand, if the attribute actually was a don't care symbol before and it is mutated to 0 or 1, the classifier is specialized (its specificity is increased) covering a smaller portion of the problem space (in the binary case, half of the space). Finally, a specified attribute (e.g. 0) may be changed to another specific value (e.g. 1), effectively transferring the subspace structure of the rest of the classifier condition to another part of the search space.

Figure 3.3 illustrates the three mutation cases. Given the parental classifier condition `11#11#`, cases (a) and (b) show the potential cases for specialization, that is `11011#`, `11111#` and `11#110`, `11#111`, respectively. Case (c) shows knowledge transfer when a specialized attribute is changed to the other value. In our example, the classifier condition may change to `11#10#`, `11#01#`, `10#11#`, or `01#11#`, effectively moving the hyperrectangle to other subspaces in the problem space. Cases (d), (e), and (f) show how generalization by mutation may change the condition structure. Note that in each case, the original hyperrectangle is maintained and another hyperrectangle with a similar structure is added. The shown cases cover all possible mutation cases of one attribute in the parental classifier. Depending on the mutation probability μ, additional mutations are exponentially less probable, but may result in an extended neighborhood search.

In effect, mutation searches in the general/specific neighborhood of the current solution and it transfers structure from one subspace to another (near-by) subspace. The effectiveness of mutation consequently depends on the complexity distribution over the search space. If syntactic neighborhoods are structured similarly, mutation can be very effective. If there are strong differences between syntactic neighborhoods, mutation can be quite ineffective.

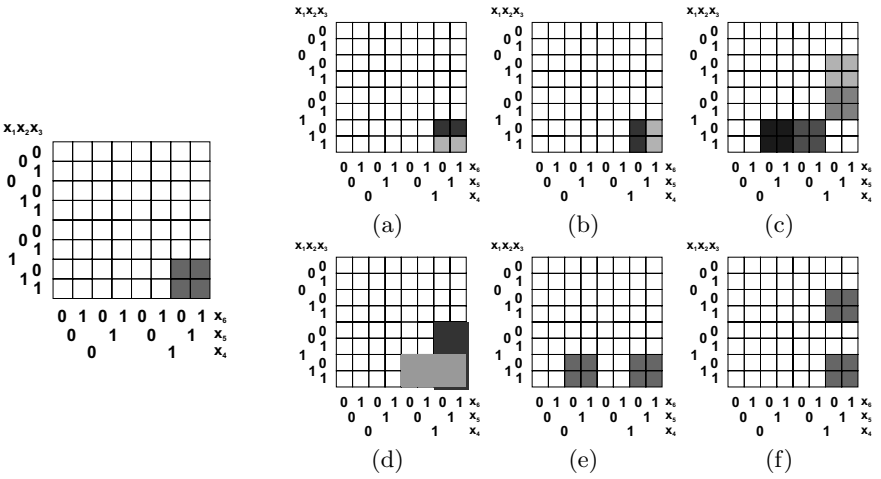

Fig. 3.3. Mutation in action: The parental classifier condition on the left-hand side can be either specialized by one mutation (a,b), projected into a neighboring subspace (c), or generalized including a neighboring subspace (d,e,f). Different grays represent different classifier conditions.

Regardless of the problem search effect, mutation is a general diversification operator that causes the evolutionary search procedure to search in the local (syntactic) neighborhood of currently promising solutions. Doing this, mutation tends to generate an equal number of each available value for an attribute. In the ternary alphabet, mutation consequently pushes the population towards an equal amount of zeros, ones, and #-symbols in classifier conditions. With respect to specificity, mutation pushes towards a 2:1 specific:general distribution. Thus, in the usual case when starting with a fairly general classifier population, mutation has a specialization effect. How mutation influences specificity in XCS and how the specificity influence interacts with other search operators in the system is analyzed in Chapter 5.

Crossover

The nature of crossover strongly differs from mutation in that crossover does not search in the local neighborhood of one classifier, but combines classifier structures. The results are offspring classifiers that specify substructures of the parental classifiers. In contrast to mutation, simple crossover does not affect overall specificity, since the combined specificity of the two offspring classifiers equals the combined specificity of the parents. Thus, although the specificity of individual offspring classifiers might differ from the parental specificity, average specificity is not affected.

For example, consider the overlapping classifier conditions `11####` and `1##00#` shown on the left-hand side of Figure 3.4. The maximal space crossover

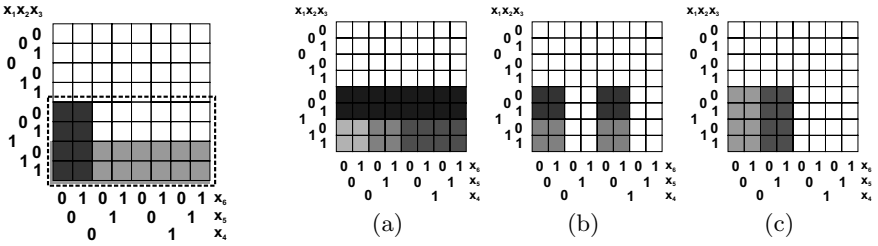

Fig. 3.4. Crossover of two overlapping parental conditions searches in the subspace indicated by the outer dashed box (left-hand side). More specialized offspring conditions (shown in brighter gray) are included in the more general offspring conditions (shown in increasingly darker gray).

searches in is restricted to the maximal general offspring that can be generated from the two parental classifiers, which is `1#####` in our example. Other offspring structures are possible, which are progressively closer to the parental structures as indicated in Figure 3.4 (a), showing progressively more specialized classifier conditions as well as in (b) and (c) showing the four other possible offspring cases. It can be seen that crossover consequently searches in the maximal problem subspace defined by the two classifier conditions. Structure of the two classifiers is exchanged and projected onto other subspaces inside the maximal subspace.

If the parental classifiers are non-overlapping, crossover searches in the maximum subspace, which is defined by the non-overlapping parts of the two subspaces. In the example shown in Figure 3.5, the parental classifiers `1##00#` and `00####` are non-overlapping and the maximum subspaces are characterized by either the upper half of the search space (`0#####`) or the lower half of the search space (`1#####`). Note that in the case of non-overlapping classifiers, the structural exchange may or may not be fruitful and strongly depends on the underlying problem structure. If structure is similar throughout the whole search space, then crossover may be beneficial. However, if structure differs in different search subspaces, crossover can be expected to be mainly disruptive, when non-overlapping conditions are recombined.

In general, crossover recombines previously successful substructures, transferring those substructures to other, nearby problem subspaces. Depending on the complexity and uniformity of problem spaces, crossover may be more or less effective. Also, it can be expected that the recombination of classifiers that cover structurally related problem spaces will be more effective than the recombination of unrelated classifiers. Thus, a good restriction of classifier recombination is expected to result in a more effective genetic search.

As in GAs, the issue of building blocks (BBs) comes into mind. In simple LCSs, BBs are complexity units, which define a subspace that yields high reward on average. The identification and effective propagation of BBs, consequently, should result in another type of more effective search within LCSs.

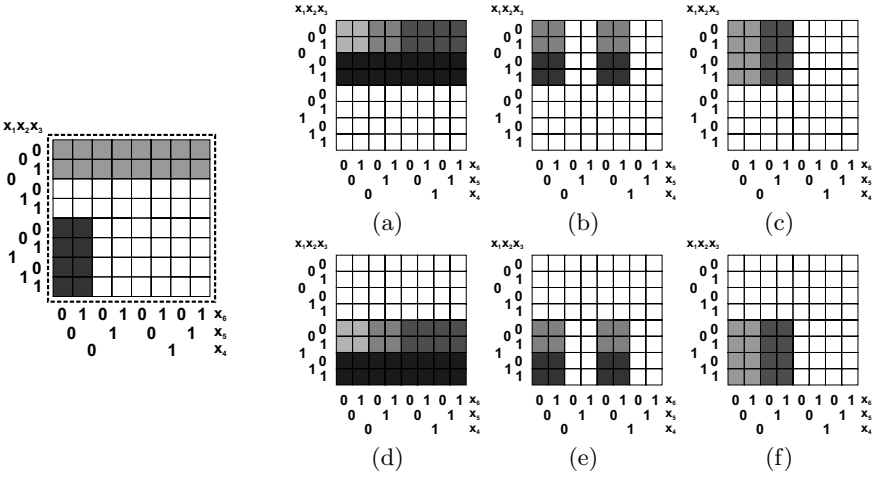

Fig. 3.5. Given two non-overlapping parental conditions (left-hand side), crossover explores structure in either of the resulting maximal general, non-overlapping subspaces. Figures a,b,c,d,e,f show condition subspaces that can be generated by crossover. Brighter gray subspaces are included in darker subspaces but also form a condition subspace on their own.

Considering such search biases, however, it needs to be kept in mind that we are searching for a distributed solution representation in which different subsolutions consist of different BB structures.

Local vs. Global Search Bias

Due to the distributed problem solution representation, LCSs face another challenge in comparison to standard GAs. Although the incoming problem instances must be assumed to be structurally and semantically related, the currently evolved problem subsolutions may be unequally advanced, and good subsolutions may be highly unequally complex, depending on the problem subspace they apply in. For example, in some regions of the problem space, a very low specificity might be sufficient to predict reward correctly whereas in other regions further specificity might be necessary.

Figure 3.6 illustrates a problem space in which some subspaces are highly complex (subspace `00******`), in that the identification of the problem class (black or white) requires several further feature specifications. On the other hand, the rest of the problem space is fairly simple (subspaces `01******`, `10******`, and `11******`) in that classes can be distinguished easily by the specification of only one or two additional features (`01**1***`→`1`, `10**0***`→`1`, and `110**1**`→`1` and `111**0**`→`1`).

Similar differences in complexity can be found in RL problems. For example, in the simple Maze problem in Figure 2.3 on page 17 and the exemplar

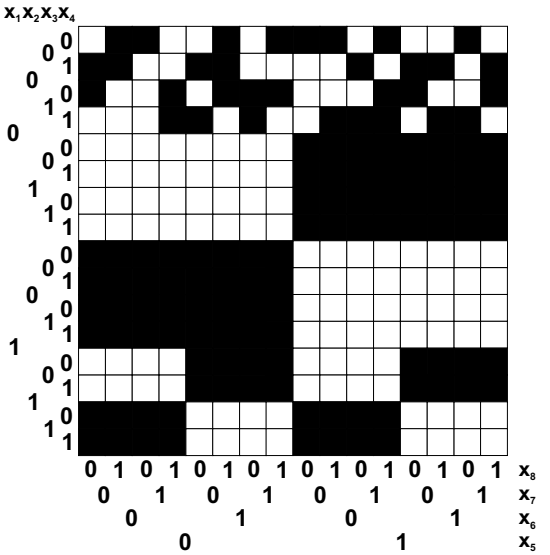

Fig. 3.6. Problem subspaces may vary in complexity dependent on the problem structure. Complexity-dependent niching as well as suitable subspace-dependent search mechanisms are mandatory to enable the effective evolution and sustenance of a complete problem representation.

population shown in Figure 3.2 on page 37, we can see that a classifier of specificity 1/8 (order one—specifying the empty position to the north) suffices to predict a reward of 1000 correctly when going north (Classifier 3). On the other hand, in order to predict a 900 reward correctly when heading south, at least two positions need to be specified (see Classifier 9). Thus, necessary specificities as well as necessary specified positions might differ depending on the problem instance. In effect, global selection and recombination might not be appropriate since different classifiers might represent solutions to completely different subspaces. Thus, a balanced search combining local and global knowledge can be expected to be most effective.

In particular, it is desired to only recombine those classifiers that are compatible in the sense that they address related problems as specified by their condition parts. In the simple LCS, selection, recombination, and crossover are usually applied globally in that two potentially unrelated classifiers are selected from the overall population. The consequent recombination is then likely to be ineffective if solution structure varies over the problem space. We will see that XCS circumvents this problem by reproducing classifiers in action sets. However, if the problem has a much more global structure, further bias towards global selection may result in additional learning advantages. We address this problem in further detail in Chapter 7.

3.3.5 Solution Sustenance

As seen above, a simple GA selects individuals from the whole population, mutates, and potentially recombines them. Since a GA is usually designed to optimize a problem, it usually searches for one globally optimal solution. The sustenance of different solutions is important only early in the run in this case. Sustenance is usually assured through initial population diversity and supply as well as by balancing the focusing effects of selection with the diversification effects of recombination and/or mutation operators.

A different problem arises if the goal is to evolve not only the best solution but a distributed set of solutions. Since LCSs are designed to generate the best solution for every potential problem instance, the population in an LCS needs to evolve optimal solutions for all potential problems in the problem space. Each problem instance represents a new (sub-)problem (defined by the problem instance) that might be related to the other (sub-)problems but represents a new problem in a common problem space.

Due to the necessity of a distributed representation, niching methods (Goldberg & Richardson, 1987; Horn, 1993; Horn, Goldberg, & Deb, 1994; Mahfoud, 1992) are even more important in LCSs than they are in standard GAs. Since LCSs need to evolve and maintain a representation that is able to generate a solution for every possible problem instance, it needs to be assured that the whole problem space is covered by the distributed solution representation. Several niching methods are applicable and different LCSs use different techniques to assure the sustenance of a complete problem solution. Later chapters address the niching issue in further detail.

3.3.6 Additional Multistep Challenges

The above issues mainly targeted problems in which immediate reward indicates the appropriateness of an action. Also, problem instances were thought to be independent of each other. In multistep problems, such as the ones modeled by general MDP or POMDP processes, reward propagation as well as self-controlled problem sampling comes into play.

Evaluation and reproduction time issues need to be reconsidered in this case. Since successive problem instances depend on the executed action, which is chosen by the LCS agent itself, problem instance sampling and problem instance frequencies may become highly skewed. Thus, the time issues with respect to classifier selection and propagation in problem subspaces may need to be reevaluated.

Additionally, the RL and GA component may be influenced by the current action policy and vice versa, the action policy may depend on the RL and GA constraints. For example, exploration may be enhanced in problem subspaces in which no appropriate classifier structure evolved so far. This relates to prioritized sweeping and other biased search algorithms in RL (Moore & Atkeson, 1993; Sutton, 1991).

Besides sampling and learning bias issues, the challenge of reward propagation needs to be faced. Since reinforcement may be strongly delayed depending on the problem and current problem subspace, accurate non-disruptive reward backpropagation needs to be ensured. Thus, competent RL techniques need to be applied. Additionally, due to the rule generalization component, update techniques that weigh the confidence of the predictive contributions of each classifier, similar to error estimation techniques used in neural networks, may be advantageous. These concerns are analyzed in detail in Chapter 10.

3.3.7 Facetwise LCS Theory

The above issues lead to the proposition of the following LCS problem decomposition. To assure that a classifier system evolves an accurate problem solution, the following aspects need to be respected:

1. ***Design evolutionary pressures most effectively***:
   Appropriate fitness definition, parameter estimation, and rule generalization.
   a) **Fitness guidance**: Fitness needs to be defined appropriately to guide towards the optimal solution disabling strong overgenerals.
   b) **Parameter estimation**: Classifier parameters need to be initialized and estimated most effectively to avoid fitness disruption in young and unexperienced classifiers as well as to identify better classifiers as fast as possible.
   c) **Adequate generalization**: Classifier generalization needs to push towards a maximally accurate, maximally general problem solution preventing overgeneralization.
2. ***Ensure solution growth and sustenance***:
   Effective population initialization, classifier supply, classifier growth and niche sustenance.
   a) **Population initialization**: Effective classifier initialization needs to ensure sufficient classifier evaluation and GA application time.
   b) **Schema supply**: Minimal order schema representatives need to be available.
   c) **Schema growth**: Schema representatives need to be identified and reproduced before deletion is expected.
   d) **Solution sustenance**: Niching techniques need to ensure the sustenance of a complete problem solution.
3. ***Enable effective solution search***:
   Effective mutation, recombination, and structural biases.
   a) **Effective mutation**: Mutation needs to search the neighborhoods of current subsolutions effectively ensuring diversity and supply.
   b) **Effective recombination**: Recombination needs to combine building block structures efficiently.

c) **Local vs. global structure**: Search operators need to detect and exploit global structural similarities but also differences in different local problem solution subspaces.

4. ***Consider additional challenges in multistep problems***:
   Behavioral policies, sampling biases, and reward propagation.
   a) **Effective behavior**: A suitable behavioral policy needs to be installed to ensure appropriate environmental exploration and knowledge exploitation.
   b) **Problem sampling reconsiderations**: Subproblem occurrence frequencies might be skewed due to environmental properties and the chosen behavioral policy. Thus, evaluation and reproduction times need to be reevaluated and might be synchronized with currently chosen behavior and encountered environmental properties.
   c) **Reward propagation**: Accurate reward propagation needs to be ensured to allow accurate classifier evaluation.

The next chapters focus on this LCS problem decomposition, investigating how the accuracy-based XCS classifier system faces and solves these problem aspects. Along the way, we also show important improvements in the XCS system in the light of several of the theory facets.

3.4 Summary and Conclusions

In this chapter we introduced the general structure of an LCS. We saw that LCSs are suitable to learn classification problems as well as RL problems. Additionally, we saw that LCSs are online learning systems that learn from one problem instance at a time, potentially interacting with a real environment or problem.

LCSs learn a distributed problem solution represented by a population of classifiers (that is, a set of rules). Each classifier specifies a condition, which identifies the problem subspace in which it is applicable, an action or classification, and a reward prediction, which characterizes the suitability of that action given the current situation. Although we only considered conditions in the binary problem space, applications in other problem spaces including nominal and real-valued inputs are possible. Chapter 9 investigates the performance of the accuracy-based classifier system XCS in such problem spaces.

Classifiers in an LCS population are *evaluated* by RL mechanisms. Classifier structure is *evolved* by a steady-state GA. Thus, while the RL component is responsible for the identification of better classifiers, the GA component is responsible for the propagation of these better classifiers. The consequent strong interdependence of the two learning components needs to be considered when creating an LCS.

Dependent on the classifier condition structure, mutation and crossover have slightly different effects in comparison to search in a simple GA. Mutation changes the condition structure searching for other subsolutions in the

neighborhood of the parental condition. However, mutation does not only cause a diversification pressure, it can also have a direct effect on the *specificity* of the condition of a classifier. Crossover, on the other hand, does not affect combined classifier specificity but recombines problem substructures. As in GAs, crossover may be disruptive and BB identification and propagation mechanisms may improve genetic search. In contrast to GAs, though, LCSs search for a distributed problem solution so that crossover operators need to distinguish and balance search influenced by local and global problem structure.

The proposed facetwise LCS theory approach for analysis and design is expected to result in the following advantages: (1) Simple computational models of the investigated LCS system can be found, (2) The found models are generally applicable, (3) The models are easily modifiable to the actual problem and representation at hand, (4) The models provide a deeper and more fundamental problem and system understanding, (5) The investigated system can be improved effectively focusing on the currently most restricting model facets, and (6) More advanced LCS systems can be designed in a more straightforward manner, targeted to effectively solve the problem at hand. The remainder of this book pursues the facetwise analysis approach, which confirms the expected advantages.