Gradient Descent Methods in Learning Classifier Systems: Improving XCS Performance in Multistep Problems

Martin V. Butz, David E. Goldberg, and Pier Luca Lanzi

***Abstract*—The accuracy-based XCS classifier system has been shown to solve typical data mining problems in a machine-learning competitive way. However, successful applications in multistep problems, modeled by a Markov decision process, were restricted to very small problems. Until now, the temporal difference learning technique in XCS was based on deterministic updates. However, since a prediction is actually generated by a set of rules in XCS and Learning Classifier Systems in general, gradient-based update methods are applicable. The extension of XCS to gradient-based update methods results in a classifier system that is more robust and more parameter independent, solving large and difficult maze problems reliably. Additionally, the extension to gradient methods highlights the relation of XCS to other function approximation methods in reinforcement learning.**

***Index Terms*—Function approximation, gradient descent, learning classifier systems (LCSs), multistop problems, Q-learning, reinforcement learning, XCS.**

I. Introduction

LEARNING CLASSIFIER SYSTEMS (LCSs) are rule-based learning systems that combine temporal difference learning methods with evolutionary computation. Knowledge evolves in a *population* of "condition-action-payoff" prediction rules (i.e., the *classifiers*). LCSs are designed to learn an optimal behavioral policy represented by a compact set of maximally general rules. Our investigations focus on the accuracy-based XCS classifier system [26], which is designed to evolve a complete, accurate, and maximally general representation of the reward function in a given problem.

Recent analysis in data-mining applications have shown that XCS can perform better than some well-known traditional machine learning techniques [4], [11], [18], [30]. However, successful applications in multistep problems were restricted to small problems [26], [27]. Without further additions, XCS is not able to robustly solve environments that allow only few generalizations or require a larger number of steps until reinforcement is encountered [3], [17].

Although LCSs were developed within the evolutionary computation community rather independently from reinforcement learning research, LCSs are actually tightly linked to reinforcement learning methods. For example, the well-known original *bucket-brigade* algorithm is very similar to SARSA [23]. More recent temporal difference learning techniques in LCSs can be compared with Q-learning [24] such as the zero-current switching (ZCS) system [25] and the accuracy-based XCS system [19], [26]. Thus, from this perspective, an LCS is a population-based temporal difference learning method.

Besides the similarity to the most common temporal difference learning approaches, LCSs may be viewed as evolutionary-based function approximation methods that evolve a rule-based representation of a target function. The XCS system, for example, has been shown to be applicable as a pure function approximator [29].

In the realm of multistep environments, which are usually modeled as a Markov decision process (MDP), the task is to learn the state-value function or the state-action value function that predicts the current reward in conjunction with the expected discounted future reward. To stay policy-independent, the system often learns the Q-value function that maps state-action pairs to the value of the immediate reward plus the discounted expected future reward assuming the optimal behavioral policy is executed from that instant on. Due to possibly large state-action spaces, however, it has become clear that tabular-based reinforcement learning scales-up poorly. Thus, function approximation methods have been applied [23]. These mostly neural-based function approximators have been shown to be highly unstable if direct (gradient) methods are used to implement Q-learning [1]. Accordingly, residual gradient methods have been developed to improve robustness [2].

The aim of this paper is to explore the possibility of applying gradient-based update methods in LCSs and in particular in the

Manuscript received December 13, 2003; revised February 17, 2004. The work of M. V. Butz was supported in part by the Computational Science and Engineering Graduate Option Program, University of Illinois at Urbana–Champaign, Urbana, in part by the German Research Foundation under Grant DFG HO1301/4-3, and in part by the Euopean Commission under Contract FP6-511931. This work was supported in part by the Air Force Office of Scientific Research, Air Force Material Command, USAF under Grant F49620-03-l-0129, and in part by the Technology Research, Education, and Commercialization Center (TRECC), University of Illinois at Urbana–Champaign, administered by the National Center for Supercomputing Applications (NCSA) and funded by the Office of Naval Research under Grant N00014-01-l-0175. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research, the National Science Foundation, or the U.S. Government.

M. V. Butz and D. E. Goldberg are with the Illinois Genetic Algorithms Laboratory, Department of General Engineering, University of Illinois at Urbana–Champaign, Urbana, IL 61801 USA (e-mail: butz@illigal.ge.uiuc.edu; deg@illigal.ge.uiuc.edu).

P. Lanzi is with Illinois Genetic Algorithms Laboratory, Department of General Engineering, University of Illinois at Urbana–Champaign, Urbana, IL 61801 USA. He is also with the Artificial Intelligence and Robotics Laboratory, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milan 20133, Italy (e-mail: lanzi@illigal.ge.uiuc.edu; pierluca.lanzi@polimi.it).

Digital Object Identifier 10.1109/TEVC.2005.850265

XCS system. We show how LCSs are related to neural function approximation methods and use similar gradient-based methods to improve stability and robustness. In particular, we apply XCS to large multistep problems using gradient-based temporal difference update methods. We show that XCS with gradient update methods reaches higher learning robustness and stability. Moreover, the system becomes more independent from learning parameter settings.

Paper Organization: The paper is structured as follows. In Section II, we provide the necessary background knowledge on reinforcement learning including Q-learning, the direct (gradient) approach and residual gradient approaches. Next, in Section III, we provide an overview of the XCS classifier system. In Section IV, we highlight the similarities between Q-learning, Q-learning with gradient methods, and XCS; we show how XCS can be modified to include gradient-based updates. In Section V, we describe the design of experiments used in this paper. In Section VI, we compare the performance of Wilson's XCS [26] with that of XCS extended with direct gradient (namely, XCSG). The reported results show that the addition of gradient descent to XCS improves performance and robustness. In Section VII, we take a step further and show how to extend XCS with residual gradient methods; we test the new version of XCS with residual gradient (XCSRG) in Section VIII. In Section IX, we take a different perspective and analyze the performance of XCS and XCS with the gradient approach in terms of approximation of the Q-value function. We show that the addition of gradient to XCS results in a more accurate approximation of the target Q-value function. We conclude this paper with a discussion regarding various implications of our results and consequent future research directions.

II. Reinforcement Learning

Reinforcement learning is defined as the problem of an *agent* that learns to perform a task through *trial and error interactions* with an unknown *environment* which provides feedback in terms of numerical *reward* [23]. The agent and the environment interact continually. At time t, the agent senses the environment to be in state s_t; based on its current sensory input s_t the agent selects an action a_t in the set A of the possible actions; then action a_t is performed in the environment. Depending on the state s_t, on the action a_t performed, and on the effects of a_t in the environment, the agent receives a *scalar reward* r_{t+1} and a new state s_{t+1}. The agent's goal is to *maximize* the amount of reward it receives from the environment *in the long run*, usually expressed as the *discounted expected payoff*, which at time t is defined as follows:

$$E\left[\sum_{k=0}^{\infty}\gamma^k r_{t+1+k}\right] \tag{1}$$

where γ is the *discount factor* $(0 \leq \gamma \leq 1)$ that specifies the importance of future reward. The larger γ, the more important are more distant future rewards.

In reinforcement learning, the agent learns how to maximize the incoming reward by developing an action-value function $Q(\cdot,\cdot)$ [or a state value function $V(\cdot)$] that maps state-action pairs (or states) onto the corresponding expected payoff value (1).

A. Q-Learning

Given a reinforcement learning problem, under adequate hypotheses, the Q-learning algorithm [24] converges with probability one to the optimal action-value function Q^* which maps state-action pairs to the associated expected payoff. More precisely, Q-learning iteratively approximates the table of all values $Q(s,a)$, named the Q-table; $Q(s,a)$ is defined as the payoff predicted under the hypothesis that the agent performs action a in state s, and then it carries on always selecting the actions which predict the highest payoff. The algorithm works as follows. In the beginning, for each state-action pair, $Q(s,a)$ is initialized with random values, at time step t $(t > 0)$, when the agent senses the environment to be in state s_t, and receives reward r_t for performing a_{t-1} in state s_{t-1}, the entry $Q(s_{t-1},a_{t-1})$ is updated according to the formula

$$Q(s_{t-1},a_{t-1}) \leftarrow Q(s_{t-1},a_{t-1}) + \beta\left(r + \gamma \max_{a\in A} Q(s_t,a) - Q(s_{t-1},a_{t-1})\right)$$

where β is the *learning rate* $(0 \leq \beta \leq 1)$; γ is the *discount factor* [the same of (1)]; r is the reward received for performing a_{t-1} in state s_{t-1}; and s_t is the state the agent encounters after performing a_t in s_t.

B. Example: Woods Environments

To show an example of how Q-learning works, we consider a class of simple although representative problems, namely, *woods environments* [26], involving an agent learning how to reach goal positions in mazes. Woods environments are grid worlds with obstacles (or trees, represented by "T" symbols), goal positions (or food, represented by "F" symbols), and empty positions (represented by blanks, " "). The agent can stay in any of the empty positions, or it is able to move to any adjacent position that is empty. The agent has eight sensors, one for each adjacent position. Sensors are encoded by two bits: obstacles are encoded by 10, goals by 11, empty positions by 00; thus, an agent's sensory input is a string with 16 bits, (2 bits $\times$ 8 positions). There are eight possible actions, one for each possible adjacent position The agent's goal is to learn how to reach the goal position from any empty position. When the agent reaches a goal position (F) the problem ends, and the agent receives a constant reward R equal to 1000. As an example, consider the environment named Woods1, depicted in Fig. 1. Woods1 is a small toroidal grid that consists of 25 positions: 8 positions contain obstacles ("T"); there is one goal position ("F"); there are 16 empty positions, so that the environment has 16 possible states $(s_0,\ldots,s_{15})$. When we apply Q-learning to solve Woods1, with $\beta = 0.2$ and $\gamma = 0.7$, we obtain the Q-table depicted in Fig. 2.

C. Generalization

Tabular Q-learning is simple and easy to implement but it is infeasible for problems of interest because the size of the Q-table (which is $|S| \times |A|$) grows exponentially in the problem

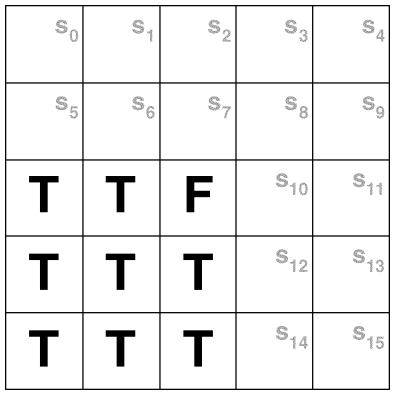

Fig. 1. `Woods1` environment: Each of the 16 empty positions has been labeled with a state $s_j \in \{s_0 \ldots s_{15}\}$.

	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
s_0	490	490	490	700	490	490	490	490
s_1	490	490	700	490	490	490	490	490
s_2	490	490	490	700	700	490	490	490
s_3	490	490	700	1000	700	700	490	490
s_4	490	490	490	700	700	700	490	490
s_5	490	490	700	700	1000	700	700	490
s_6	490	490	490	490	700	700	490	490
s_7	490	490	490	490	700	1000	700	490
s_8	700	490	490	490	700	700	1000	700
s_9	700	490	490	490	490	700	700	1000
s_{10}	700	490	490	490	490	490	490	490
s_{11}	490	490	490	490	490	700	490	490
s_{12}	490	490	490	490	490	700	700	490
s_{13}	490	490	490	490	490	700	700	700
s_{14}	490	490	490	490	490	490	700	700
s_{15}	490	490	490	490	490	490	490	700

Fig. 2. Optimal Q-table obtained applying Q-learning to `Woods1` with the following parameters: $R = 1000, \beta = 0.2, \gamma = 0.7$.

dimensions. This is a major drawback in real applications since the bigger the Q-table: 1) the more the experiences required to converge to a good estimate of the optimal Q-table (Q^*) and 2) the more the memory required to store the table [23]. To cope with the complexity of the tabular representation the agent must be able to *generalize* over its experiences, i.e., to produce a good approximation of the optimal Q-table from a limited number of experiences, using a small amount of storage. In reinforcement learning, generalization is usually implemented by function approximation techniques: the action-value function $Q(\cdot,\cdot)$ is seen as a function that maps state-action pairs onto real numbers (i.e., the expected payoff); gradient descent techniques are used to build a good approximation of function $Q(\cdot,\cdot)$ from on-line experience. For instance, the optimal Q-table for `Woods1` depicted in Fig. 2 can also be represented as a three-dimensional (3-D) function that maps state-action pairs onto payoff values. As the number of states and actions involved increases, such a function will tend to become more and more similar to a surface. The goal of function approximators is to develop a good estimate of such a payoff surface.

An alternative representation of Q-tables, that we will be using further in this paper, is the one shown in Fig. 3. In this case, state-action pairs are enumerated, sorted according to their payoff values, and reported on the abscissa, while payoff values are reported on the ordinate.

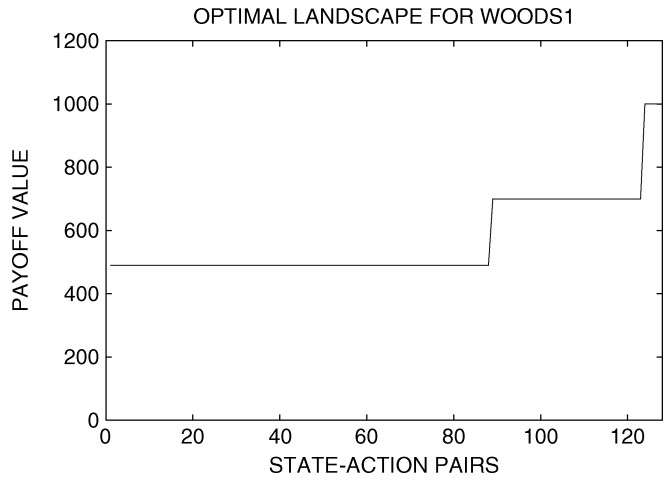

Fig. 3. Optimal Q-table for `Woods1` (Fig. 2) represented as a 2-D landscape.

D. Q-Learning With Gradient Descent

When applying gradient descent to approximate Q-learning, we are actually trying to minimize the error between the desired value that is estimated by "$r+\gamma \max_{a\in A} Q(s_t,a)$" and the current payoff estimate given by $Q(s_{t-1},a_{t-1})$. For instance, in approximate Q-learning with a function approximator parametrized by a weight matrix W using gradient descent, each weight w changes by Δw at each time step t

$$\Delta w = \beta\left(r+\gamma \max_{a\in A} Q(s_t,a) - Q(s_{t-1},a_{t-1})\right)\frac{\partial Q(s_{t-1},a_{t-1})}{\partial w} \tag{2}$$

where β is the learning rate and γ is the discount factor.[1] The change in weights Δw depends both on 1) the difference between the desired payoff value associated to the current state-action pair (i.e., $r+\gamma \max_{a\in A} Q(s_t,a)$) and the current payoff associated to the state-action pair $Q(s_{t-1},a_{t-1})$ and 2) on the gradient component represented by the partial derivative of the current payoff value with respect to the weight; gradient estimates the contribution that the change in w will have on the payoff $Q(s_{t-1},a_{t-1})$ associated to the current state-action pair. Function approximation techniques that update their weights according to the equation above are called *direct algorithms*. Note that (2) provides only the change in the weight values which can be used differently in accordance with the approximation method.

While tabular reinforcement learning methods can be guaranteed to converge, function approximation methods based on direct algorithms, like the previous one, have been shown to be fast but unstable even for problems involving few generalizations [1], [2]. To improve the convergence of function approximation techniques for reinforcement learning applications, another class of techniques, namely, *residual gradient algorithms*, have been developed [2]. Residual gradient algorithms are slower but more stable than direct ones and, most

[1]Note that in the equation, following the notation used in reinforcement learning, $Q(\cdot,\cdot)$ represents the approximated action-value function, not the Q-table [1], [2], [23].

important, they can be guaranteed to converge under adequate assumptions. Residual algorithms extend direct gradient descent approaches, which focuses on the minimization of the difference $(r + \gamma \max_{a \in A} Q(s_t, a) - Q(s_{t-1}, a_{t-1}))$, by adjusting the gradient of the current state with an estimate of the effect of the weight change on the successor state. The weight update Δw for Q-learning implemented with a *residual gradient* approach now becomes the following:

$$\Delta w = \beta \left(r + \gamma \max_{a \in A} Q(s_t, a) - Q(s_{t-1}, a_{t-1}) \right) \times \left[\frac{\partial Q(s_{t-1}, a_{t-1})}{\partial w} - \gamma \frac{\partial}{\partial w} \left(\max_{a \in A} Q(s_t, a) \right) \right] \quad (3)$$

where the partial derivative $(\partial / \partial w)(\max_{a \in A} Q(s_t, a))$ estimates the effect that the current modifications of the weight have on the value of the next state. Note that since this adjustment involves the next state, the discount factor γ must also be taken into account.

Direct approaches and *residual gradient* approaches are combined in *residual algorithms* by using a linear combination of the contributions given by the former approaches to provide a more robust weight update formulation. Let ΔW_d be the update matrix computed by a direct approach, and ΔW_{rg} be the update matrix computed by a residual gradient approach, then, the updated matrix for residual approach ΔW_r is computed as

$$\Delta W_r = (1 - \phi) \Delta W_d + \phi \Delta W_{rg}.$$

By substituting the actual expressions of ΔW_d and ΔW_{rg} for Q-learning with *direct* and *residual gradient* approach, respectively, (see [2] for details), we obtain the following weight update for Q-learning approximated with a *residual approach*:

$$\Delta w = \beta \left(r + \gamma \max_{a \in A} Q(s_t, a) - Q(s_{t-1}, a_{t-1}) \right) \times \left[\frac{\partial Q(s_{t-1}, a_{t-1})}{\partial w} - \phi \gamma \frac{\partial}{\partial w} \left(\max_{a \in A} Q(s_t, a) \right) \right]. \quad (4)$$

Note that the residual version of Q-learning turns out to be basically an extension of the *residual gradient* approach where the contribution on the next state $((\partial / \partial w) \max_{a \in A} Q(s_t, a))$ is weighted by the parameter ϕ. The weighting can be either adaptive, or it can be computed from the weight matrices W_d and W_{rg}; in the latter case, the convergence to the optimal policy is guaranteed [2].

The equations for *residual-gradient* Q-learning and *residual* Q-learning presented so far assume that the environment is represented by a *deterministic* MDP. When this is not the case, the stochastic versions of (3) and (4) are used. These are obtained from the previous equations by substituting the gradient of the value of the (unique) next state $((\partial / \partial w) \max_{a \in A} Q(s_t, a))$ with the value of *one of the possible* states that the agent can experience after performing action a_{t-1} in state s_{t-1}. The version of *residual gradient* Q-learning for stochastic environments is

$$\Delta w = \beta \left(r + \gamma \max_{a \in A} Q(s_t, a) - Q(s_{t-1}, a_{t-1}) \right) \times \left[\frac{\partial Q(s_{t-1}, a_{t-1})}{\partial w} - \gamma \frac{\partial}{\partial w} \max_{a \in A} Q(s', a) \right] \quad (5)$$

while for the residual version of Q-learning the update is

$$\Delta w = \beta \left(r + \gamma \max_{a \in A} Q(s_t, a) - Q(s_{t-1}, a_{t-1}) \right) \times \left[\frac{\partial Q(s_{t-1}, a_{t-1})}{\partial w} - \phi \gamma \frac{\partial}{\partial w} \max_{a \in A} Q(s', a) \right] \quad (6)$$

where s' is one of the states that the agent can reach from s_{t-1} when action a_{t-1} is performed, while s_t is the actual state that the agent reached. The only difference between the deterministic and the stochastic versions of residual gradient and residual Q-learning is in the adjustment component that in stochastic environments takes into account *any* of the possible states reachable from s_t with action a_t.

III. XCS Classifier Systems

XCS [26] is a reinforcement learning method in which generalization is obtained through the evolution of a *population* of condition-action-prediction rules (*classifiers*). With respect to the usual reinforcement learning settings in XCS: 1) the population of classifiers represents the action-value function that solves the target reinforcement learning problem and 2) the classifiers and their parameters roughly correspond to the weight matrix W that is used in function approximation approaches to generalize over the space of possible solutions. This section gives a short introduction to XCS. A detailed algorithmic description can be found in [10].

Classifiers: In XCS, classifiers consist of a condition, an action, and four main parameters: 1) the prediction p, which estimates the average payoff that the system expects when the classifier is used; 2) the prediction error ε, which estimates the average absolute error of the prediction p; 3) the fitness F, which estimates the average relative accuracy of the payoff prediction given by p; and finally 4) the numerosity *num*, which indicates how many copies of classifiers with the same condition and the same action are present in the population.

Performance Component: At each time step, XCS builds a *match set* $[M]$ containing the classifiers in the population $[P]$ whose condition matches the current sensory inputs; if $[M]$ contains less than θ_{mna} actions, *covering* takes place and creates a new classifier that matches the current inputs and has a random action. For each possible action a_i in $[M]$, XCS computes the *system prediction* $P(a_i)$, which estimates the payoff that XCS expects if action a_i is performed. The *system prediction* is computed as the fitness weighted average of the predictions of classifiers in $[M]$, $cl \in [M]$, which advocate action a_i (i.e., $cl.a = a_i$)

$$P(a_i) = \sum_{\frac{cl_k \in [M]|_{a_i}}{\sum_{cl_k \in [M]|_{a_i}} F_k}} p_k \times F_k \quad (7)$$

where $[M]|_{a_i}$ represents the subset of classifiers of $[M]$ with action a_i, p_k identifies the prediction of classifier cl_k, and F_k identifies the fitness of classifier cl_k. Next, XCS selects an action to perform. The classifiers in $[M]$ which advocate the selected action form the current *action set* $[A]$. The selected action is performed in the environment, and a scalar reward r is returned to XCS together with a new input configuration.

Reinforcement Component: When the reward r is received and the match set $[M]$ with respect to the resulting sensory input is formed, the parameters of the classifiers in $[A]$ are updated in the following order [10]: prediction, prediction error, and finally fitness. Prediction p is updated with learning rate $\beta\,(0 \leq \beta \leq 1)$ as follows:

$$p \leftarrow p + \beta\left(r + \gamma \max_{a \in A} P(a) - p\right). \tag{8}$$

Then, the prediction error ε is updated as: $\varepsilon \leftarrow \varepsilon + \beta(|P - p| - \varepsilon)$. Finally, classifier fitness is updated in two steps: first, the *relative accuracy* κ' of the classifiers in $[A]$ is computed; then κ' is used to update the classifier fitness as: $F \leftarrow F + \beta(\kappa' - F)$.

Discovery Component: On a regular basis depending on the parameter θ_{ga}, a genetic algorithm is applied to classifiers in $[A]$. It selects two classifiers with probability *proportional to their fitness*, copies them, and with probability χ performs crossover on the copies; then, with probability μ it mutates each allele. The resulting offspring are inserted into the population and two classifiers are deleted to keep the population size constant.

IV. XCS and Gradient Descent

In this section, we analyze the similarities between tabular Q-learning, Q-learning with gradient descent, and XCS. In general, XCS represents an alternative approach to function approximation problems. However, XCS learning capabilities are based on plain tabular Q-learning, which can work only with a complete and exact representation of the state-action space. In this section we show how gradient descent can be added to XCS learning mechanism through a simple modification.

A. Q-Tables, Gradient Descent, and XCS

When dealing with applications that involve a large number of state-action pairs, function approximators are used to develop accurate approximations of the target action-value function. In Q-learning, the Q-table may be approximated using a function approximator parameterized by a weight matrix W, as shown in Section II. A gradient-based approach essentially uses the gradient component $\partial Q(s_{t-1}, a_{t-1})/\partial w$ to guide the weight update (2). XCS exploits a modification of Q-learning to update the rule base. At each time step t, when the current input is s_t, and the current incoming reward is r, the prediction of each classifier in the action set $[A]_{-1}$, used in the previous time step $(t-1)$, is updated as specified in (8).

While tabular Q-learning and XCS have been compared from a theoretical standpoint [19], we now wish to focus on the analogies between update of tabular Q-learning (2) and the update of classifier prediction in XCS (8). First, we note that in tabular Q-learning at time step t only one update is made for $Q(s_{t-1}, a_{t-1})$, while in XCS more than one classifier is usually involved in the update. In fact, XCS represents each position in the Q-table by a set of classifiers; more precisely, $Q(s_{t-1}, a_{t-1})$ is represented by the classifiers in $[A]_{-1}$, while $Q(s_t, a_t)$ is represented by the classifiers in $[A]$. The *value* of $Q(s_t, a_i)$ in XCS is represented by the system prediction $P(a_i)$ as specified in (7).

Due to its generalization capabilities, XCS is more similar to function approximator approaches than to plain tabular Q-learning. Comparing the weight update for gradient descent (2) and the update for classifier predictions (8), we note that in the latter no term plays the role of the gradient. Classifier prediction update for XCS was directly inspired by *tabular* Q-learning [26]. Accordingly, each term in XCS's prediction update (8) maps directly into a term in the Q-learning update (2). Gradient approaches were not considered so far in XCS. Thus, from a reinforcement learning perspective, XCS exploits a rather elementary approach for learning the action-value function, which is used only for tabular representations in the reinforcement learning literature.

B. Adding Gradient Descent to XCS

To improve the learning capabilities of XCS, we add gradient descent to the equation for the classifier prediction update in XCS. For this purpose, we first need to compute the gradient component $\partial Q(s_{t-1}, a_{t-1})/\partial w$ for XCS, that is, we need to map both the value $Q(s_{t-1}, a_{t-1})$ of the previous state action pair, and the weight w in XCS.

Mapping $Q(s_{t-1}, a_{t-1})$ to XCS: As previously noted, in XCS the value of a specific state-action pair is represented by the system prediction $P(\cdot)$, which is computed as a fitness weighted average of classifier predictions (7). More precisely, the value of $Q(s_{t-1}, a_{t-1})$ corresponds to the system prediction $P(a_{t-1})$ associated with the classifiers in $[A]_{-1}$ that is computed as follows:

$$Q(s_{t-1}, a_{t-1}) = P(a_{t-1}) = \frac{\sum_{cl_j \in [A]_{-1}} p_j \times F_j}{\sum_{cl_j \in [A]_{-1}} F_j} \tag{9}$$

where p_j and F_j are, respectively, the prediction and the fitness of classifier cl_j.

Mapping Weights to XCS: In the usual function approximation methods, the weights w of a weight matrix W are used to combine the function arguments so as to produce an approximated value. Learning classifier systems consider the rules that are active and combine their predictions (their *strength*) to obtain an overall estimate of the reward that should be expected. Different models have different strategies to combine classifiers' predictions, but all the existing models *exploit* classifier prediction (*strength*) to estimate future expectation. In other words, different models have different ways of computing the *value* of state-action pairs but all these models use classifier prediction to compute this value. From this perspective, in learning classifier systems, we can regard classifier prediction as the major actor that plays the role of the weights in function approximation approaches. It is worth to note that by taking classifier prediction as the equivalent of weight we make a very general assumption that allows us to extend the gradient approach proposed in this paper to other models of classifier systems such as ZCS [25] or SB-XCS [15], as discussed in Appendix B.

We now estimate the gradient component for a classifier $cl_k \in [A]_{-1}$ by computing the partial derivate of $Q(s_{t-1}, a_{t-1})$ with respect to the prediction p_k of classifier cl_k

$$\frac{\partial Q(s_{t-1}, a_{t-1})}{\partial w} = \frac{\partial}{\partial p_k}\left[\frac{\sum_{cl_j \in [A]_{-1}} p_j \times F_j}{\sum_{cl_j \in [A]_{-1}} F_j}\right] = \frac{1}{\sum_{cl_j \in [A]_{-1}} F_j}\frac{\partial}{\partial p_k}\left[\sum_{cl_j \in [A]_{-1}} p_j \times F_j\right] = \frac{F_k}{\sum_{cl_j \in [A]_{-1}} F_j}. \quad (10)$$

For each classifier $cl_k \in [A]_{-1}$, the component corresponding to gradient descent can be computed as the ratio between its fitness F_k and the sum of fitnesses of classifiers in the same action set ($\sum_{cl_j \in [A]_{-1}} F_j$).

Reinforcement Component With Gradient Descent: The version of XCS with gradient descent works as the original XCS except for the update of classifier prediction. When the parameters of classifiers in $[A]_{-1}$ are updated, first the sum $F_{[A]_{-1}}$ of classifiers' fitness in $[A]_{-1}$, is computed as

$$F_{[A]_{-1}} = \sum_{cl_j \in [A]_{-1}} F_j.$$

Then, for each classifier $cl_k \in [A]_{-1}$, its prediction p_k is updated as

$$p_k \leftarrow p_k + \beta\left(r + \gamma \max_{a \in A} P(a) - p_k\right)\frac{F_k}{F_{[A]_{-1}}}. \quad (11)$$

The other parameters, prediction error and classifier fitness are updated as usual (Section III). In the remainder of this paper, we refer to the version of XCS with the prediction updated based on gradient descent as XCSG.

C. Effect of Gradient Component on Prediction Update

The introduction of gradient descent in XCS results in an additional term to the original prediction update provided by [26]. Before moving to experimental results, it is interesting to see how the additional gradient term $(F_k/F_{[A]_{-1}})$ influences the update of classifier predictions.

We can look at the gradient contribution in many different ways. Probably the most intuitive way is to consider the gradient term $(F_k/F_{[A]_{-1}})$ as a straightforward extension of system prediction computation to prediction update: since the system prediction is a fitness-weighted average, (8) should be fitness-weighted. The additional gradient term does exactly this; $F_k/F_{[A]_{-1}}$ biases the prediction update of classifiers with respect to their contribution to the calculation of system prediction. Note, however, that this consideration is limited in scope since it actually applies to XCS only, but the gradient computation can be added to any model of classifier system for which it is possible to compute $\partial Q(s_{t-1}, a_{t-1})/\partial w$.

The gradient in (11) also represents an adaptive way to adjust the learning rate separately for each classifier. Given a classifier cl_k, if its contribution to system prediction is small (i.e., $F_k/F_{[A]_{-1}}$ is small), then the update of its prediction value will be based on a small learning rate equal to $\beta F_k/F_{[A]_{-1}}$; if the contribution of classifier cl_k to system prediction is high, then the update of its prediction value will be based on an actual learning rate $\beta F_k/F_{[A]_{-1}}$, very near to the original β. In terms of accurate and inaccurate classifiers this means that 1) inaccurate (overgeneral) classifiers (with a small value of $F_k/F_{[A]_{-1}}$) will tend to have a more stable prediction value since their actual learning rate will be only a small fraction of β and 2) accurate (maximally general) classifiers will approach their actual prediction value faster than the inaccurate ones. Stability is assured due to the increased accuracy. Accordingly, we can also view gradient as a way to make the information about classifier predictions more reliable. Oscillating (inaccurate) classifiers will have a more reliable prediction value through small learning rate values; accurate classifiers will have a reliable prediction value since they are more accurate. From a reinforcement learning approach, gradient update in XCS makes the payoff surface more reliable to provide more accurate information to the generalization mechanism of XCS. As a side effect, we believe that the evolutionary component of XCS can be more effective when gradient is adopted since the information about the accuracy of the classifiers is more reliable.

V. Experimental Setting

The experiments reported in this paper were performed following the standard settings used in the literature [26]. Each experiment consists of a number of problems that the system must solve. When the system solves the problem correctly, it receives a constant reward equal to 1000; otherwise, it receives a constant reward equal to 0. Each problem is either a *learning* problem or a *test* problem. In *learning* problems, the system selects actions randomly from those represented in the match set. In *test* problems, the system always selects the action with highest prediction. The genetic algorithm is enabled only during *learning* problems, and it is turned off during *test* problems. The covering operator is always enabled, but operates only if needed. Learning problems and test problems alternate. In multistep problems, such as in woods environments, the performance is computed as the average number of steps needed to reach goal or food positions during the last 50 test problems. To speed up the experiments, problems can last at most 1500 steps; when this limit is reached the problem stops even if the system did not reach the goal. All the statistics reported in this paper are averaged over 20 experiments. All the experiments reported have been conducted on `xcslib` [20] and have also been duplicated on Butz's XCS implementation [6].

VI. Experiments in Multistep Problems

We apply XCS with gradient descent (XCSG) to a set of well-known multistep problems to test whether the addition of gradient descent in the prediction update improves XCS performance. We consider four environments: `Woods1`, `Maze5`, `Maze6`, and `Woods14` (`Woods1` in Fig. 1; the remainder in Fig. 4).

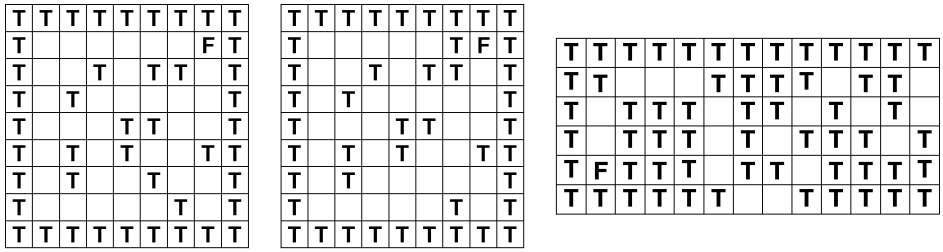

Fig. 4. (a) Maze5 environment. (b) Maze6 environment. (c) Woods14 environment.

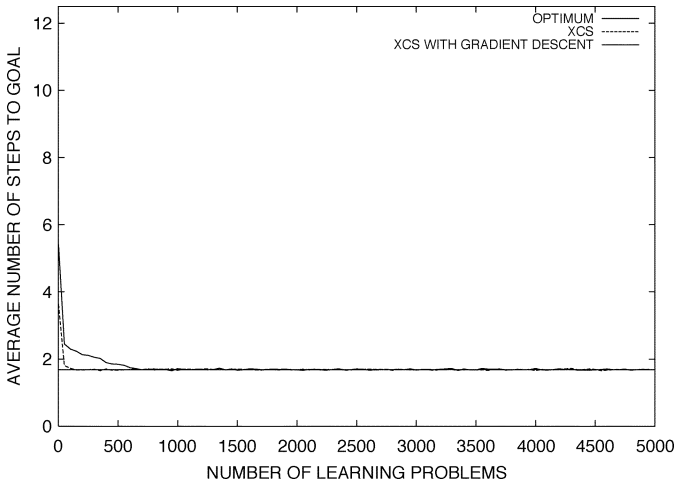

Fig. 5. Performance of XCS (dashed line) and XCSG (solid line) in Woods1.

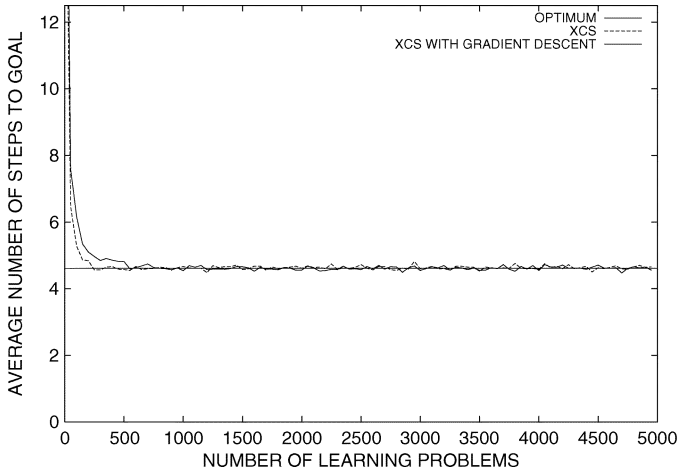

Fig. 6. Performance of XCS (dashed line) and XCSG (solid line) in Maze5 when $N = 3000$.

A. Woods1 *Environment*

First, we apply XCSG to Woods1 and compare XCSG performance with that of XCS. Since Woods1 is very simple, we do not expect much difference in the performance of the two systems. Parameters are set as follows: the population size N is 1600 classifiers, $P_{\#} = 0.6$, $\beta = 0.2$, $\gamma = 0.7$, $\chi = 0.8$, $\mu = 0.04$, $\theta_{nma} = 8$, $p_{\text{explr}} = 1.0$, $\theta_{\text{GA}} = 25$, $\varepsilon_0 = 10$, $\theta_{\text{del}} = 20$, $doGASubsumption = 0$, $doASSubsumption = 0$. Fig. 5 reports the performance of XCS (dashed line) and XCSG (solid line) in Woods1; as expected, there is almost no difference between the two versions; the problem is very simple, and they both reach optimality very easily. However, if we look closely at the very beginning of the experiments, we note that XCS is actually somewhat faster than XCSG.

B. Maze5 *Environment*

Maze5 is more difficult for XCS since only few generalizations are possible [16]. Parameters are set as follows: the population size N is 3000 classifiers, $P_{\#} = 0.3$, $\beta = 0.2$, $\gamma = 0.7$, $\chi = 0.8$, $\mu = 0.01$, $\theta_{nma} = 8$, $p_{\text{explr}} = 1.0$, $\theta_{\text{GA}} = 25$, $\varepsilon_0 = 5$, $\theta_{\text{del}} = 20$, $doGASubsumption = 0$, $doASSubsumption = 0$. Fig. 6 reports the performance of XCS (dashed line) and XCSG (solid line) in Maze5. As can be noted, both XCS and XCSG reach optimal performance and, as in previous experiments, XCS learns faster than XCSG. However, if we reduce the number of classifiers in the population to 2500 (Fig. 7), XCS does not reach optimal performance *in all the runs* anymore, whereas XCSG continues to learn an optimal behavioral policy reliably. The analysis of single runs shows that while in most of the runs XCS reaches either optimal (in 8 out of 20 runs) or near-optimal performance (in 10 out of 20 runs), there are two runs in which XCS does not actually converge even near to the optimum. As an overall result, XCS average performance is quite far from the optimum. In contrast, XCSG reaches optimal performance stably in all 20 runs. Finally, we compare XCS and XCSG in Maze5 when the mutation rate μ is increased from 0.01 to 0.04—effectively introducing higher

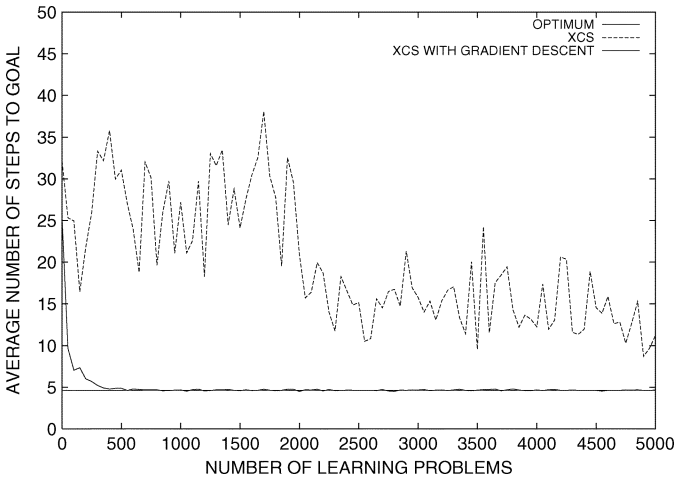

Fig. 7. Performance of XCS (dashed line) and XCSG (solid line) in Maze5 when $N = 2500$.

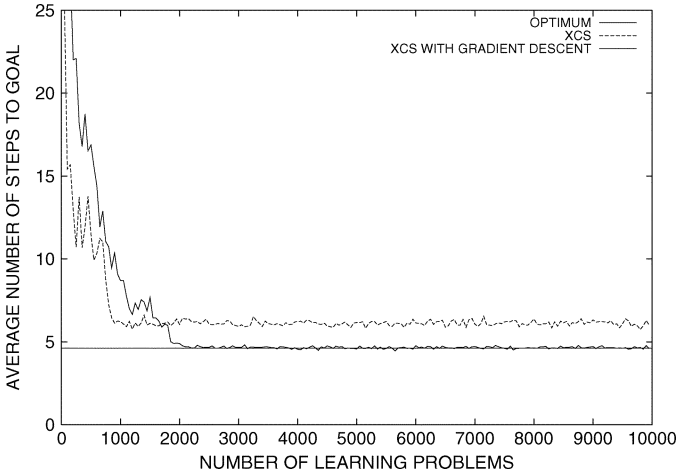

Fig. 8. Performance of XCS (dashed line) and XCSG (solid line) in `Maze5` when $N = 3000$ and $\mu = 0.04$.

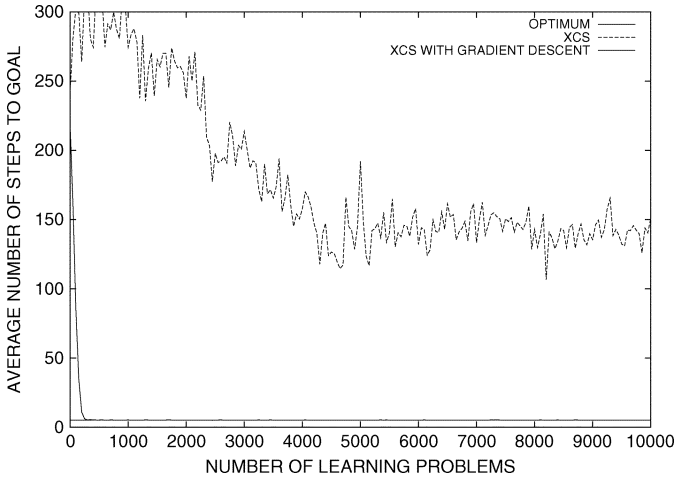

Fig. 9. Performance of XCS (dashed line) and XCSG (solid line) in `Maze6`.

specificity and variance to classifier conditions [8]. Fig. 8 reports the performance of XCS (dashed line) and XCSG (solid line) in `Maze5` when $\mu = 0.04$; all the other parameters are set as in the first experiment. Also, in this case, XCSG performs better than XCS. The analysis of single runs shows that XCS reaches optimal performance in only two of the 20 runs (i.e., in 10% of the runs), while in the other 18 runs XCS performance converges to around 7 steps (optimum is 4.611 steps). In contrast, XCSG reaches optimal performance in 18 of the 20 runs, that is, in 90% of the cases; in the remaining two runs. XCSG performance converges very near to the optimum, around 4.75 steps.

C. `Maze6` *Environment*

`Maze6` differs from `Maze5` just in two positions, but it is much more difficult since the reward position is more hidden, resulting in longer random walks and, thus, in a sparser reception of reinforcement [16]. Parameters are set as follows: the population size N is 3000 classifiers, $P_{\#} = 0.3$, $\beta = 0.2$, $\gamma = 0.7$, $\chi = 0.8$, $\mu = 0.01$, $\theta_{nma} = 8$, $p_{\text{explr}} = 1.0$, $\theta_{GA} = 100$, $\varepsilon_0 = 1$, $\theta_{\text{del}} = 20$, $doGASubsumption = 0$, $doASSubsumption = 0$. Fig. 9 reports the performance of

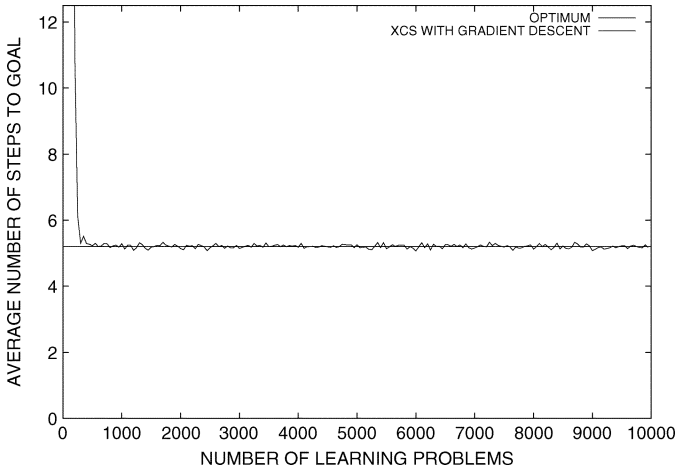

Fig. 10. Performance of XCSG in `Maze6` shown with a larger scale than Fig. 9.

XCS (dashed line) and XCSG (solid line) in `Maze6`; note that XCS performance is quite far from the optimum, while XCSG reaches the optimum rapidly and stably. The analysis of single runs shows that XCS does not reach optimal performance in most of the runs. Thus, on average, XCS performance in `Maze6` is much worse than performance in `Maze5`. In contrast, XCSG always reaches full optimality, as shown in the more detailed Fig. 10. This suggests that as the problem complexity increases, the improvement in XCS performance provided by gradient descent becomes more and more relevant.

In Section IV, we argued that the additional gradient term in the update of classifier prediction can be viewed also as a way to adjust the learning rate separately for each classifier. In particular, we suggested that one of the main effects of gradient in XCSG is to reduce the learning rate of inaccurate classifiers, so as to make their prediction more stable, while keeping a learning rate near to β for accurate classifiers (Section IV-C). To verify that the gradient approach does not only result in a smaller learning rate for the reward prediction, we now compare the performance of XCSG with that of a modified version of XCS in which the update of classifier prediction is based on a learning rate β_P smaller than the actual learning rate β used for the update of the classifier prediction error and classifier fitness. Fig. 11 reports the performance of XCS (heavy dashed line), XCSG (light dashed line), and the modified XCS with $\beta_P = 0.01$ in `Maze6`. Although the small learning rate β_P does improve the performance of XCS in `Maze6`, the overall improvement is far from the one achieved by XCSG. Thus, the general decrease in the learning rate of reward predictions is not sufficient to improve XCS's performance but the gradient approach in XCSG, which selectively decreases the learning rate of inaccurate classifiers, is necessary.

D. `Woods14` *Environment*

`Woods14` is a corridor of 18 positions leading to a goal position. It is considered a quite difficult problem and as far as we know XCS cannot solve it unless *specify* or *teletransportation* is used; some successful experiments reported in [5] show that a modification of Wilson's ZCS with very high learning rates can reach optimal performance. Parameters are set as follows:

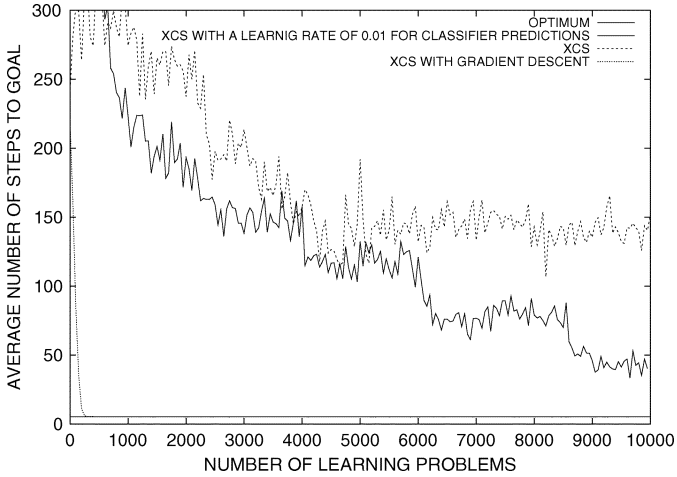

Fig. 11. Performance of XCSG with $\beta_P = 0.01$ (solid line), XCS (upper dashed line), and XCSG (lower dashed line) in Maze6.

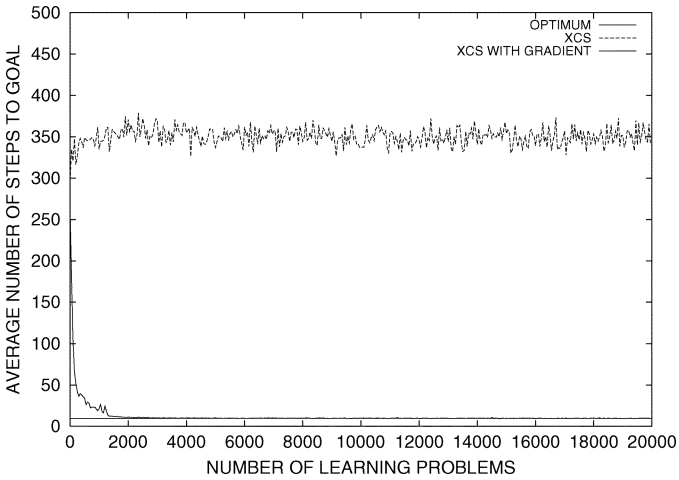

Fig. 12. Performance of XCSG (solid line) and XCS (dashed line) in Woods14 when $N = 4000$.

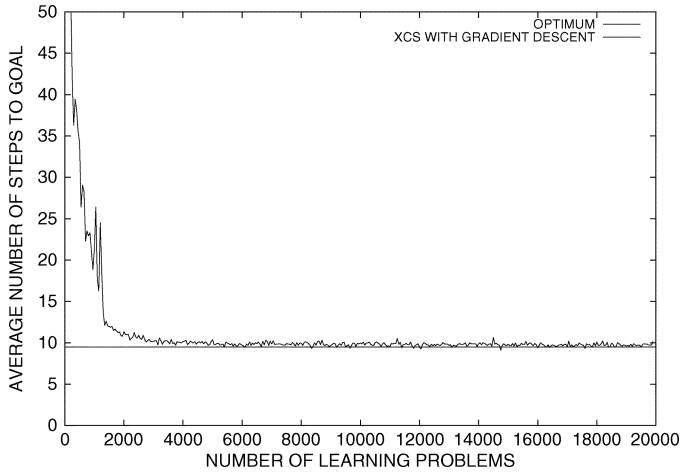

Fig. 13. Performance of XCSG (solid line) in Woods14 shown with a larger scale than Fig. 12.

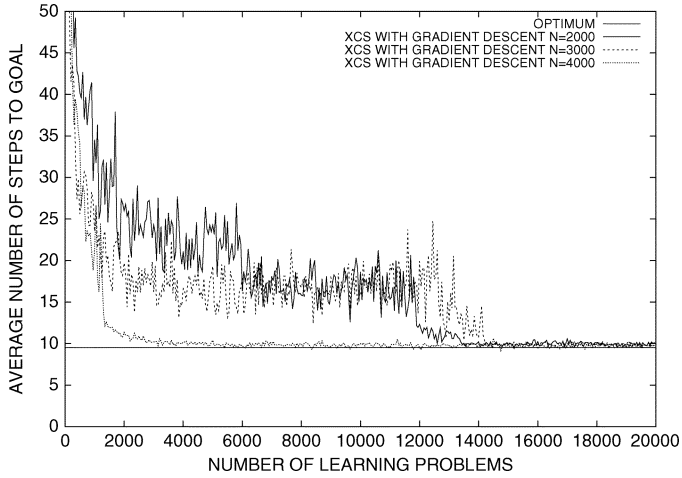

Fig. 14. Performance of XCSG in Woods14 when $N = 4000$ (light dashed line), $N = 3000$ (medium dashed line), and $N = 2000$ (solid line).

the population size N is 4000 classifiers, $P_{\#} = 0.3$, $\beta = 0.2$, $\gamma = 0.7$, $\chi = 0.8$, $\mu = 0.01$, $\theta_{nma} = 8$, $p_{\text{explr}} = 0.3$, $\theta_{\text{GA}} = 400$, $\varepsilon_0 = 0.05$, $\theta_{\text{del}} = 20$, $doGASubsumption = 0$, $doASSubsumption = 0$. Because of the number of steps involved, the prediction error ε_0 must be set small enough to allow XCS to distinguish among small payoff values.[2] To speed up the experiments, we reduced the maximum number of steps per problems: problems can now last at most for 500 steps instead of 1500 steps as in the previous experiments (Section V). Fig. 12 reports the performances of XCS (dashed line) and XCSG (solid line) in Woods14. The performance of XCS remains far from optimal in all of the 20 runs. In contrast, XCSG reaches the optimum rapidly and stably. The analysis of single runs shows that in 18 of the 20 runs reported in Fig. 12 XCSG was able to reach optimal performance. Two runs stayed near optimal. Fig. 13 reports XCSG performance in Woods14 with a larger scale.

We repeat the previous comparison with different values of population size parameter N. Fig. 14 reports the performance of XCSG in Woods14 when $N = 4000$ (light dashed line), $N = 3000$ (medium dashed line), and $N = 2000$ (solid line). Population size surely has an impact on the learning speed of XCSG: the smaller the population, the more problems need to be presented before convergence. Once all runs have converged, the performance is very similar; however, an analysis of every run shows that XCSG reaches the highest number of optimal policies when $N = 4000$ (18 runs out of 20). In the other cases, additional runs were nonoptimal (4 and 8 out of 20 for the case of $N = 3000$ and $N = 2000$, respectively). It is worth noting that both the one-sample t-test for unequal variance and the nonparametric Wilcoxon test [12] show that the difference between the above XCSG performances and the optimal performance (9.5) is not statistically significant for a confidence level of 95%. Fig. 15(a) and (b) reports the comparison of XCS and XCSG performance in Woods14 when $N = 3000$ and $N = 2000$.

In the previous section, we suggested that gradient addition to XCS may be viewed as a way to provide more reliable information about the classifier predictions to the discovery component. To test our hypothesis, we repeat the first experiment on Woods14 with different rates of genetic algorithm activations, that is, different values of θ_{ga}. Small θ_{ga} values result in a frequent activation of the genetic algorithm, whereas a large

[2]To be able to reach optimal performance XCS must be able to distinguish between two payoff values: $1000 \times \gamma^{18}$ and $1000 \times \gamma^{19}$; accordingly, ε_0 must be smaller than $1000(1-\gamma)\gamma^{18}/2 = 0.245$.

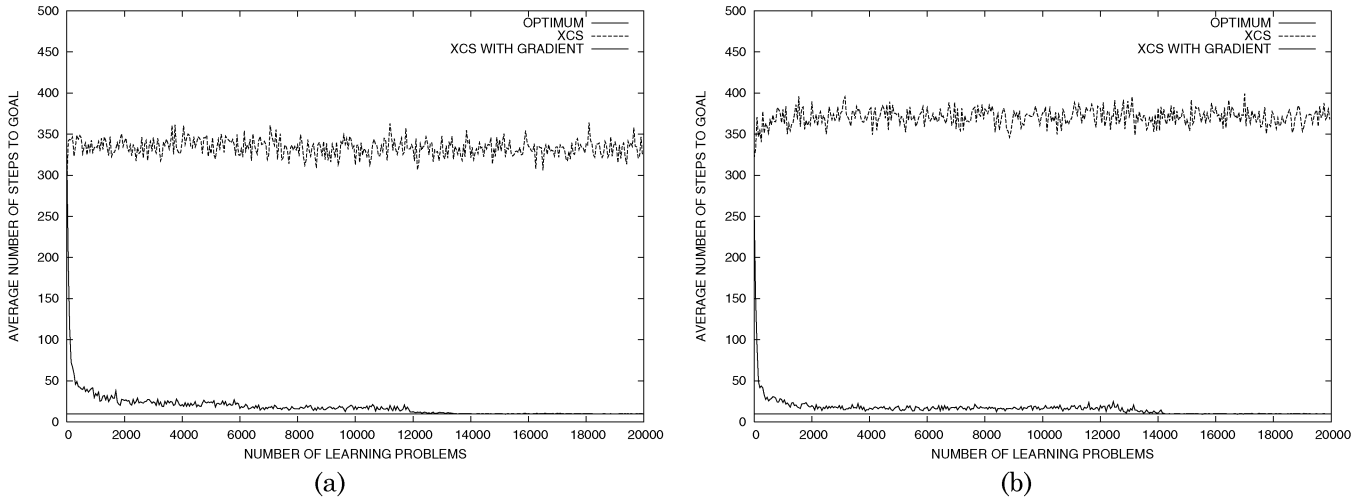

Fig. 15. Performance of XCSG (solid line) and XCS (dashed line) in `Woods14` when (a) $N = 2000$ and (b) when $N = 3000$.

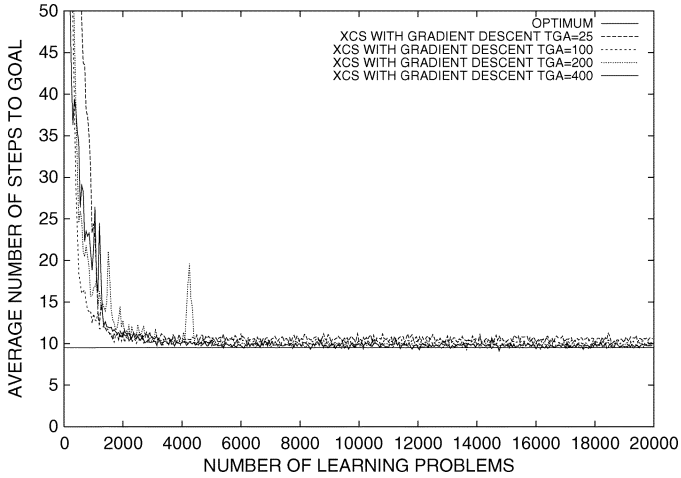

Fig. 16. Performance of XCSG in `Woods14` when $\theta_{\mathrm{ga}} = 25$, $\theta_{\mathrm{ga}} = 100$, $\theta_{\mathrm{ga}} = 200$, and $\theta_{\mathrm{ga}} = 400$.

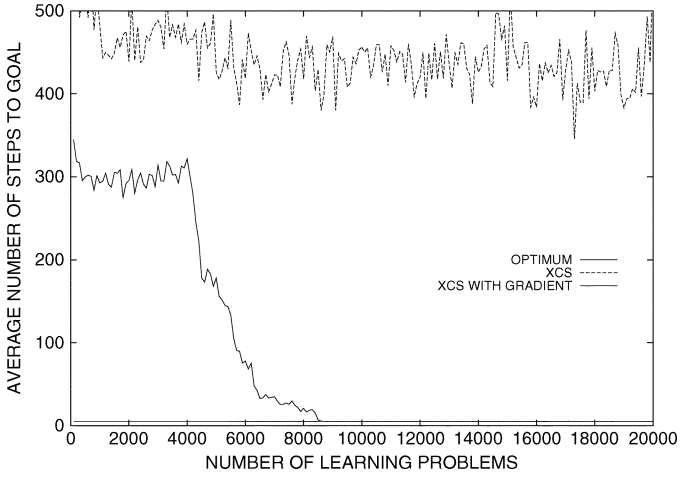

Fig. 17. Performance of XCSG in `Maze6` when $N = 5000$, $P_{\#} = 0.8$, and 30 irrelevant bits are added to the inputs. Curves are averages over 20 runs.

θ_{ga} results in a less frequent activation. Accordingly, a less frequent application enables the reinforcement component to develop more accurate parameter estimates which in turn enables a better genetic selection. Fig. 16 reports the performance of XCSG in `Woods14` when $\theta_{\mathrm{ga}} = 25$, $\theta_{\mathrm{ga}} = 100$, $\theta_{\mathrm{ga}} = 200$, and $\theta_{\mathrm{ga}} = 400$. Interestingly, we can exactly observe the balancing effect of θ_{ga}: with a value of 25, the genetic algorithm is applied too frequently delaying the evolution of an accurate policy. A value of $\theta_{\mathrm{ga}} = 100$ seems most effective resulting in fastest learning. On the long run, all settings converge to near-optimal performance. The largest θ_{ga} value appears closest to the optimum but the difference is minor.

E. Learning With Irrelevant Inputs

Finally, we test the learning capabilities of XCSG when many irrelevant inputs are present. For this purpose, we consider again `Maze6`, and we modify the problem by adding 30 irrelevant bits to the usual 16 input bits. The irrelevant bits are set independently equally probable to zero or one in each perceptual input. We apply XCS and XCSG to `Maze6` with inputs consisting of 46 bits (16 relevant, 30 irrelevant) and the following parameter settings: $N = 5000$, $P_{\#} = 0.8$, $\chi = 1.0$ (crossover is always applied), and uniform crossover is applied instead of the usual single point one [10]; all the other parameters are set as in the previous experiments. We apply uniform crossover in this experiment to avoid any possible bias due to the position of irrelevant input bits.

Fig. 17 compares the performance of XCS and XCSG against the optimal performance. As can be noted, notwithstanding the presence of a large amount of irrelevant bits, XCSG can still converge to the optimal solution, while XCS cannot. In fact, with the additional 30 irrelevant bits, XCS performance worsens compared with the previous experiment (Fig. 9). It is also worth noting that this problem is very hard to solve with tabular Q-learning techniques since the presence of the irrelevant bits blows up the size of the Q-table to $36 \times 2^{30} \times 8$ entries. Even with an optimal exploration of 500 steps per problem, Q-learning needs more than 618×10^6 problems to encounter each entry in the Q-table at least once. Clearly, XCS outperforms tabular Q-learning by far.

VII. XCS With Residual Gradient Descent

Direct methods, like the gradient descent approach added to XCS, have been shown to be fast but often unstable (for details, see [1] and [2]). Residual gradient methods improve the stability of direct methods by adjusting the gradient component with an estimate of the effect of the weight change on the next visited state. In Section II, we showed that the weight update Δw for Q-learning with a *residual gradient* approach is

$$\Delta w = \beta \left(r + \gamma \max_{a \in A} Q(s_t, a) - Q(s_{t-1}, a_{t-1}) \right) \times \left[\frac{\partial Q(s_{t-1}, a_{t-1})}{\partial w} - \gamma \frac{\partial}{\partial w} \left(\max_{a \in A} Q(s_t, a) \right) \right]$$

where $(\partial/\partial w)(\max_{a \in A} Q(s_t, a))$ estimates the effect that the current modifications of the weight have on the Q-value of the next state.

To extend XCS with residual gradient descent, we must estimate the additional component for XCS. As noted in Section IV, the term $Q(s_t, a)$ in the equation above corresponds to the system prediction of action a in XCS. Thus, the additional residual component in XCS should be computed as

$$\frac{\partial}{\partial w} \left(\max_{a \in A} P(a) \right).$$

Let $\hat{a}$ be the action corresponding to the highest system prediction

$$\hat{a} = \arg\max_{a \in A} P(a)$$

we remind from Section IV that, according to our approach, in LCS's classifier prediction (*strength*) plays the role of the weight w. Thus, for each classifier $cl_k \in [A]_{-1}$, we have

$$\begin{aligned} \frac{\partial}{\partial w} \left(\max_{a \in A} Q(s_t, a) \right) &= \frac{\partial}{\partial p_k} \left(\max_{a \in A} P(a) \right) \\ &= \frac{\partial}{\partial p_k} P(\hat{a}) \\ &= \frac{\partial}{\partial p_k} \left(\frac{\sum_{cl_j \in [M]|_{\hat{a}}} p_j \times F_j}{\sum_{cl_j \in [M]|_{\hat{a}}} F_j} \right) \end{aligned} \quad (12)$$

where $[M]|_{\hat{a}}$ is the subset of classifiers in $[M]$ that advocate action $\hat{a}$. To finally compute the value, we have to distinguish between two cases. If classifier $cl_k \in [A]_{-1}$ appears also in $[M]|_{\hat{a}}$, then we have

$$\frac{\partial}{\partial w} \left(\max_{a \in A} Q(s_t, a) \right) = \frac{F_k}{\sum_{cl_j \in [M]|_{\hat{a}}} F_j}.$$

Otherwise, if classifier $cl_k \in [A]_{-1}$ *does not* appear in $[M]|_{\hat{a}}$, then the residual in (12) is a constant with respect to p_k and, therefore, its partial derivate is zero

$$\frac{\partial}{\partial w} \left(\max_{a \in A} Q(s_t, a) \right) = 0.$$

To summarize, at time step t, for each classifier $cl_k \in [A]_{-1}$ the additional residual gradient component res_k is computed as

$$res_k = \begin{cases} \dfrac{F_k}{\sum_{cl_j \in [M]|_{\hat{a}}} F_j}, & \text{if } cl_k \in [M]|_{\hat{a}} \\ 0, & \text{otherwise} \end{cases} \quad (13)$$

and its prediction is updated as

$$p_k \leftarrow p_k + \beta \left(r + \gamma \max_{a \in A} P(a) - p_k \right) \left[\frac{F_k}{F_{[A]_{-1}}} - \gamma res_k \right]. \quad (14)$$

The formulation of the *residual gradient approach* for XCS can be directly extended to the general *residual approach* by introducing the parameter ϕ that biases the contribution of the weight change of the next state [see (4)]. In this case, the update for prediction of classifiers in $[A]_{-1}$ becomes

$$p_k \leftarrow p_k + \beta \left(r + \gamma \max_{a \in A} P(a) - p_k \right) \left[\frac{F_k}{F_{[A]_{-1}}} - \phi \gamma res_k \right]. \quad (15)$$

Note that when using function approximator approaches, the value of ϕ can be computed from the current weight matrices so as to guarantee convergence. In contrast, in XCS this is not possible since the overall convergence actually depends on the interaction of the reinforcement component with the discovery component.

Reinforcement Component With Residual Gradient: To update the prediction with residual gradient, the action $\hat{a}$ that corresponds to the highest system prediction is computed and the action set $[\hat{A}]$ containing all the classifiers in $[M]$ with action $\hat{a}$ is created. Next, the parameters of classifiers in $[A]_{-1}$ are updated using the sum $\hat{F}$ of classifier fitnesses in $[A]_{-1}$

$$F_{[A]_{-1}} = \sum_{cl_j \in [A]_{-1}} F_j$$

and the sum $F_{[\hat{A}]}$ of classifiers in $[\hat{A}]$

$$F_{[\hat{A}]} = \sum_{cl_j \in [\hat{A}]} F_j.$$

Finally, the prediction p_k of each classifier $cl_k \in [A]_{-1}$ that is also present in $[\hat{A}]$ is updated

$$p_k \leftarrow p_k + \beta \left(r + \gamma \max_{a \in A} P(a) - p_k \right) \left(\frac{F_k}{F_{[A]_{-1}}} - \gamma \frac{F_k}{F_{[\hat{A}]}} \right). \quad (16)$$

Otherwise, if cl_k is not in $[\hat{A}]$, classifier prediction is updated as previously done in the gradient approach

$$p_k \leftarrow p_k + \beta \left(r + \gamma \max_{a \in A} P(a) - p_k \right) \frac{F_k}{F_{[A]_{-1}}}.$$

Prediction error and classifier fitness are updated as usual (Section III). An algorithmic description of the modifications made to XCSRG is given in Appendix A. In the remainder of this paper, we will refer to the version of XCS in which

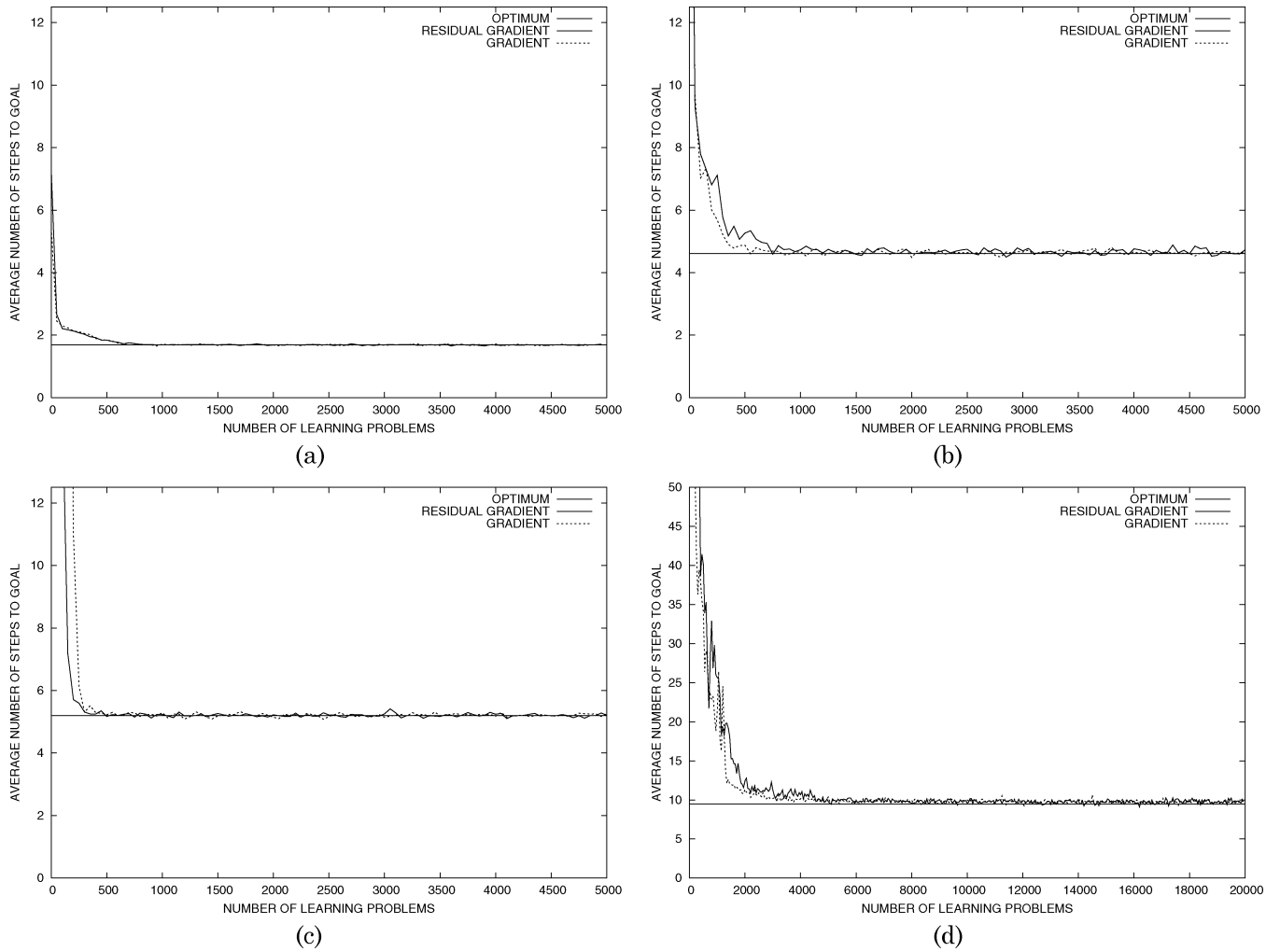

Fig. 18. Performance of XCSG (dashed line) and XCSRG (solid line). (a) `Woods1`. (b) `Maze5`. (c) `Maze6`. (d) `Woods14`.

the prediction is updated based on residual gradient descent as XCSRG. Note that XCSRG corresponds to a general residual approach with parameter $\phi = 1$.

VIII. Experiments With Residual Gradient

We now compare the performance of XCS with gradient descent (XCSG) with that of XCS with residual gradient descent (XCSRG) on the same problems considered in Section VI. Fig. 18 reports the performance of XCSG and XCSRG in (a) `Woods1`, (b) `Maze5`, (c) `Maze6`, and (d) `Woods14`. All the reported experiments use the parameter settings used in Section VI; in the experiments with `Maze5`, we use the slightly smaller population size $N = 2500$. The performance of XCS with gradient (XCSG) is always reported with a dashed line; the performance of XCS with residual gradient (XCSRG), as well as the optimal performance is reported with a solid line. As the figures show, there is almost no difference between the performance of the direct (gradient) approach and the residual gradient approach; the experiments on `Maze5` show that the performance of the residual gradient approach (XCSRG) is slightly worse than that of the direct gradient approach. However, the `Maze6` comparison show that the residual gradient approach is slightly faster.

We extend the previous results by comparing the performance of XCSG and XCSRG when the usual roulette wheel selection strategy is replaced by the more effective tournament selection strategy [9] in the discovery component. Figs. 19–22 compare the performances of XCSG and XCSRG with roulette wheel (dashed lines) and tournament selection (solid lines). The reported curves suggest that tournament selection might result in slightly faster convergence, but in general it appears that such a difference is not very much relevant in terms of reaching problem solutions in these maze problems.

Tournament selection was introduced to XCS to achieve a more reliable and more stable fitness pressure from the overgeneral side toward maximal accuracy. Thus, the advantage of tournament selection should become stronger in environments in which the problem needs to be solved starting from a higher generality in the initial population (controlled by $P_{\#}$) requiring a stronger pressure toward accuracy. In this case, the combination of the gradient-based update method and tournament selection should be most advantageous in that the gradient-based reinforcement learning approach provides most accurate fitness estimates and tournament selection exploits the estimates most effectively.

It remains to be explained why the residual gradient approach does not result in any significant performance improvement. In

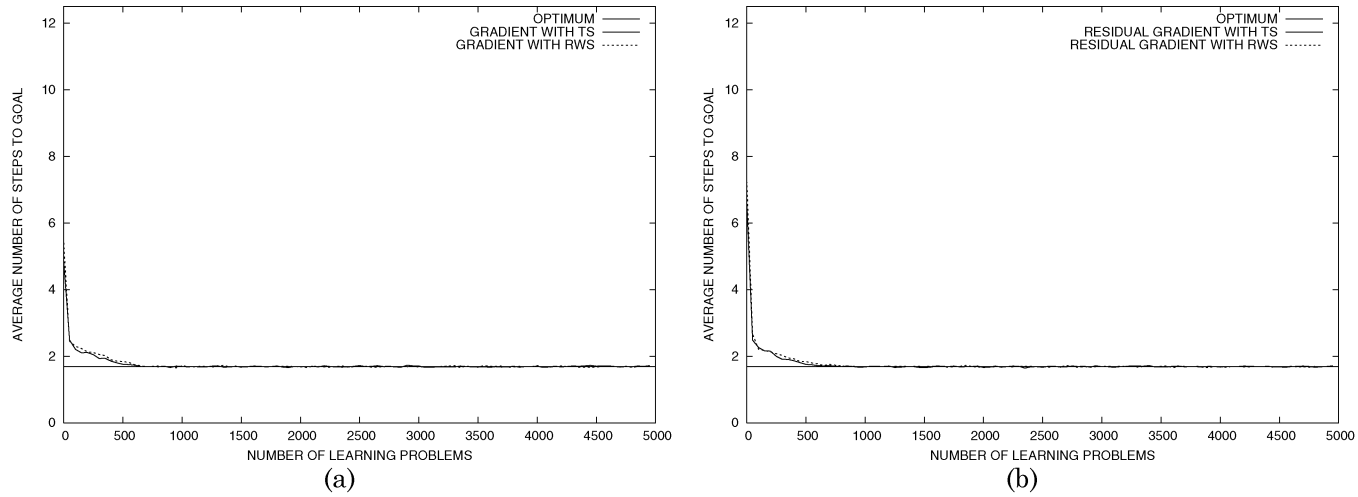

Fig. 19. (a) Performance of XCSG in `Woods1` when offspring selection is based on roulette wheel (dashed line) and tournament selection (solid line). (b) Performance of XCSRG in`Woods1` when offspring selection is based on roulette wheel (dashed line) and tournament selection (solid line).

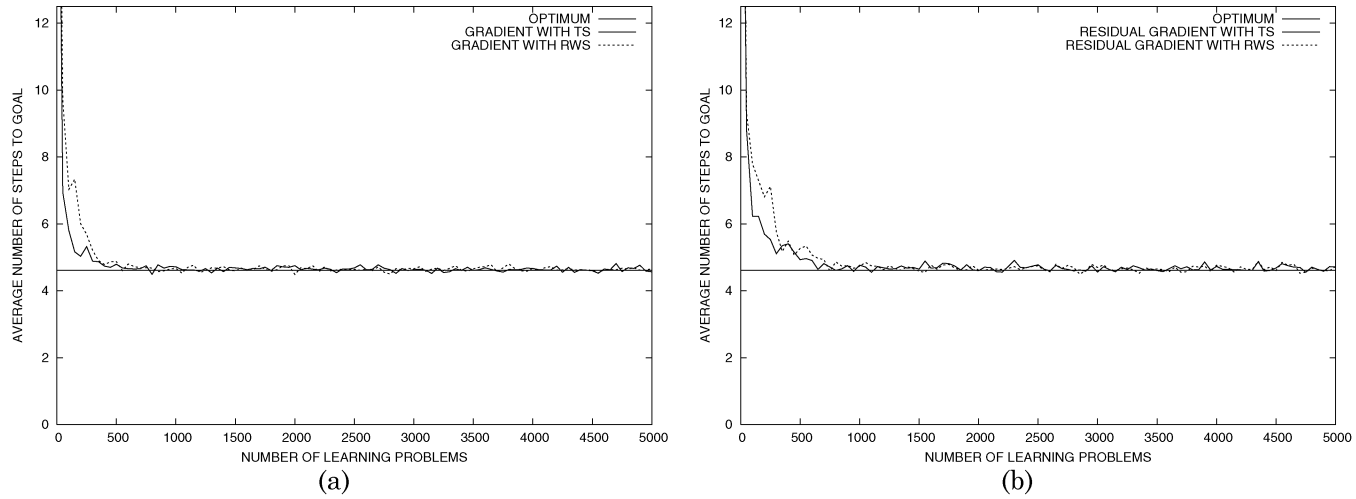

Fig. 20. (a) Performance of XCSG in `Maze5` when offspring selection is based on roulette wheel (dashed line) and tournament selection (solid line). (b) Performance of XCSRG in`Maze5` when offspring selection is based on roulette wheel (dashed line) and tournament selection (solid line).

neural networks, the stabilization due to the residual gradient approach appears successful since neurons are often continuously active. Classifiers in XCS, on the other hand, are evolved to represent one reward level accurately. Usually, only few overgeneral classifiers are active in successive reward niches. Thus, since the condition part is responsible for a proper distinction between successive niches, the stabilization is accomplished by the condition part instead of the residual approach. Since the generation of the condition part is the responsibility of the genetic learning component in XCS, the residual approach appears hardly necessary. Nonetheless, for a proper early distinction of successive niches the residual gradient approach might still be beneficial in environments in which many generalizations are possible or in which successive niches are very hard to distinguish by the conditions.

IX. Evolution of Payoff Landscapes

We now analyze the results reported in Section VI from a reinforcement learning viewpoint by focusing on the analysis of the evolved action-value functions instead of the raw performance. We focus on the comparison between XCS and XCSG. Since payoff landscapes of XCSRG look similar to those of XCSG, they are not considered herein. For our purpose, we use the two-dimensional (2-D) representation of action-value functions, introduced in Section II, that is defined as follows. Given an action-value function, all possible state-action pairs are sorted according to their predicted payoff, then they are enumerated, and reported on the abscissa; payoff values are reported on the ordinate (see Fig. 3 for an example). More precisely, to build the payoff landscape from a population of classifiers P, for every possible state s and for every possible action a, we extract the set $A(s,a)$ of all the classifiers whose condition matches s and whose action is a. Then, the payoff associated to the state-action pair $\langle s,a\rangle$ is computed as the system prediction associated to the classifier set $A(s,a)$ [see (7)]. In the following, we will refer to this type of plot as *payoff landscape*. All the data discussed here are from the experiments reported in Section VI.

Fig. 23(a) and (c) compares the optimal payoff landscape for `Woods1` obtained by applying Q-learning with the two land-

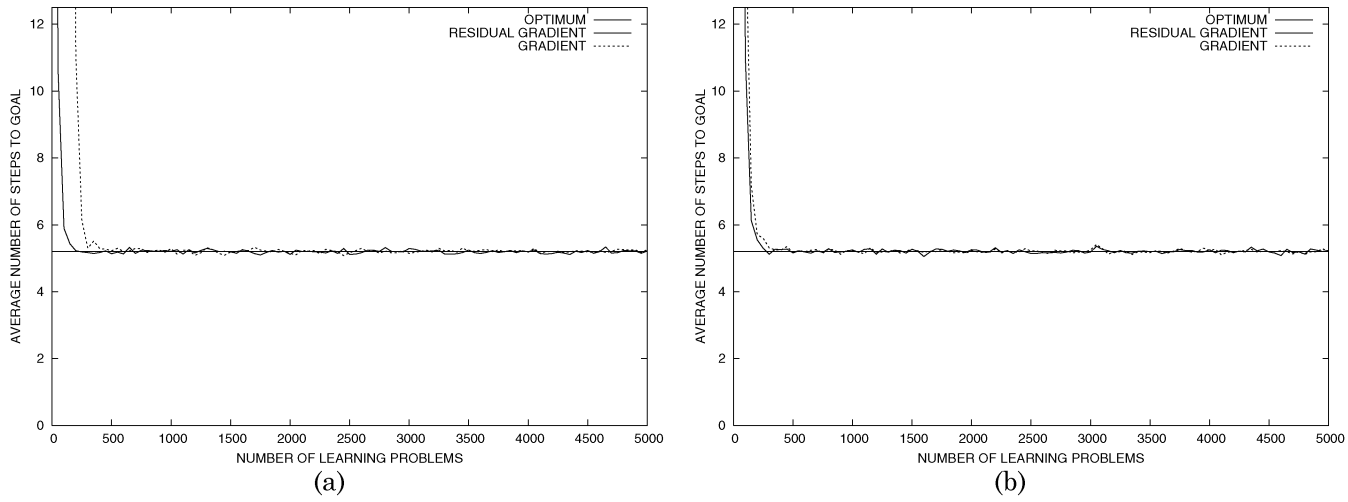

Fig. 21. (a) Performance of XCSG in `Maze6` when offspring selection is based on roulette wheel (dashed line) and tournament selection (solid line). (b) Performance of XCSRG in`Maze6` when offspring selection is based on roulette wheel (dashed line) and tournament selection (solid line).

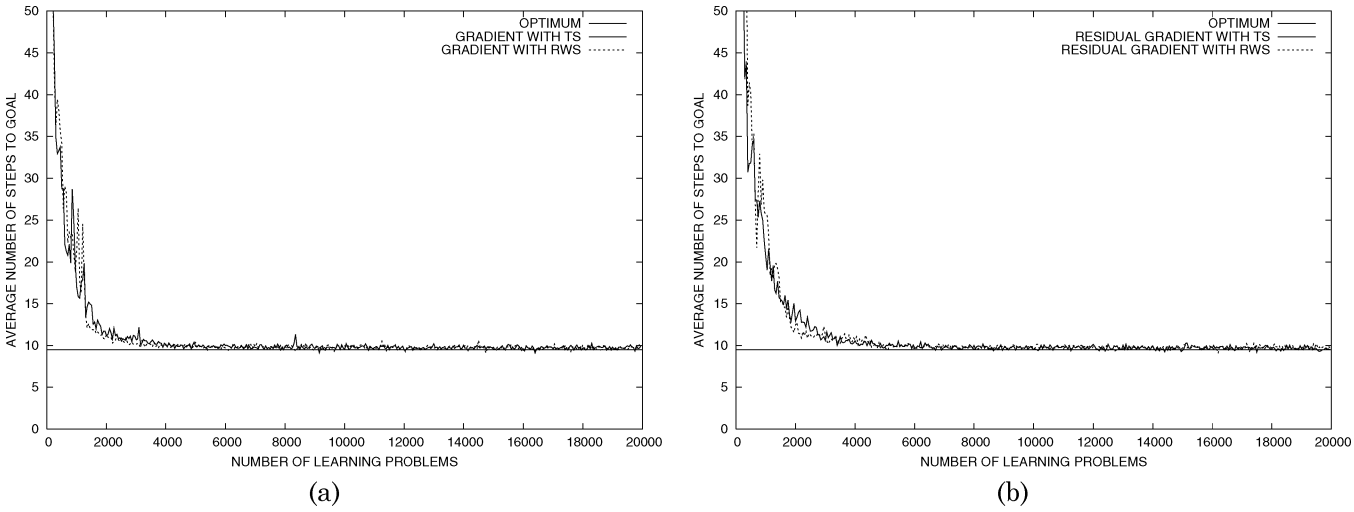

Fig. 22. (a) Performance of XCSG in `Woods14` when offspring selection is based on roulette wheel (dashed line) and the tournament selection (solid line). (b) Performance of XCSRG in`Woods14` when offspring selection is based on roulette wheel (dashed line) and tournament selection (solid line).

scapes obtained, respectively: 1) by averaging the 20 final landscapes evolved by XCS [Fig. 23(a)] and 2) by averaging the 20 final landscapes evolved by XCSG [Fig. 23(c)].[3] Fig. 23(b) and (d) reports the same result with additional error bars representing the standard error affecting the reported averages. In this simple problem, both XCS and XCSG develop an accurate representation of the target payoff landscape. The average landscapes evolved are very near to the target landscape developed by Q-learning and the error associated to the evolved landscape is practically null [Fig. 23(a) and (b)].

To have a better perspective of how payoff landscapes are evolved, it is possible to extract the currently evolved landscape at fixed time intervals (for instance, every 50 learning problems) and to report them on a 3-D plot. Figs. 24(a) and 25(a) depict the 3-D representations of how XCS [Fig. 24(a)] and XCSG [Fig. 25(a)] evolve the payoff landscape for `Woods1`. The two plots have been built by extracting one payoff landscape every 50 learning problems. The axis labeled "*problems*" identifies the learning time, one point corresponds to 50 learning problems; the axis labeled "*state-action pairs*" corresponds to the abscissa in previous 2-D plots. Note that in the early problems the payoff landscape evolved by XCSG is more noisy than that evolved by XCS. In fact, while XCS almost immediately reaches the target optimal landscape, XCSG needs around 500 problems before reaching optimality. Another perspective on the evolution of payoff landscapes is provided by Figs. 24(b) and 25(b) that represent the projection of the 3-D plots on the surface parallel to the vertical axis. With this approach, different payoff values are represented by different shades of gray; vertical lines separate the different levels of the payoff landscape. As already noted in the previous 3-D plots, the many different levels of gray at the bottom of Fig. 25(b) demonstrate that XCSG takes a little bit longer than XCS before reaching the target landscape.

Fig. 26(a) and (b) represents the averages of the 20 landscapes evolved by XCS and XCSG for `Maze5` when $N = 3000$ and $\mu = 0.04$, corresponding to the performance plots in Fig. 8. As discussed in Section VI, XCS achieves optimal performance with this settings in two runs only; XCSG achieves optimal

[3]To make the plots coherent, we use the same enumeration of state-action pairs both for Q-learning, XCS, and XCSG.

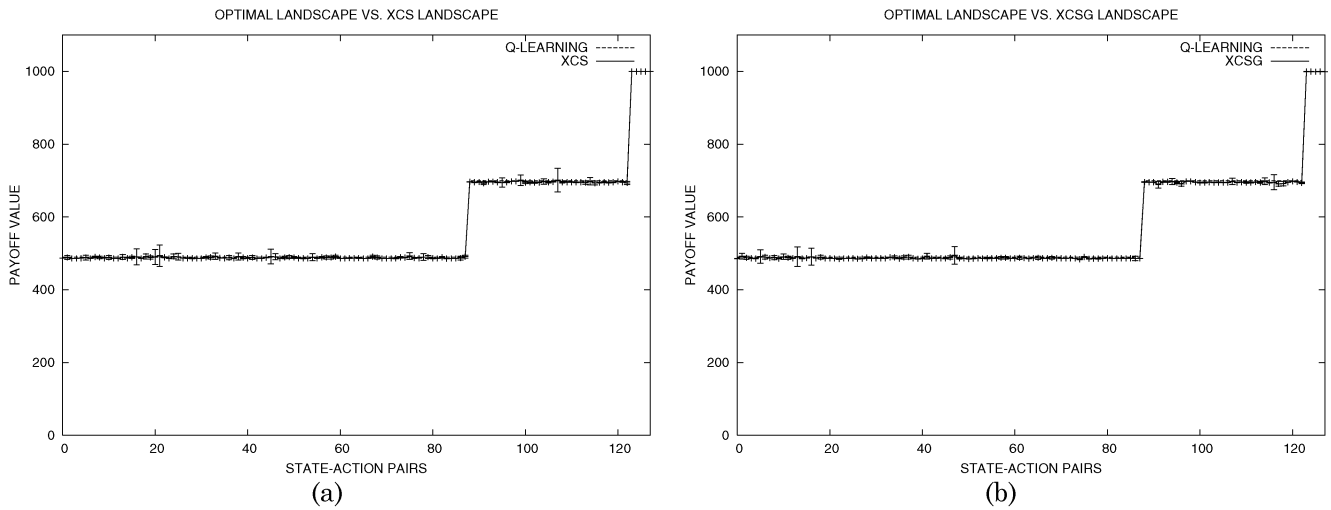

Fig. 23. XCS and XCSG in the `Woods1` environment. (a) Payoff landscape produced by XCS (solid line) with error bars representing the standard error. (b) Payoff landscape produced by XCSG (solid line) with error bars representing the standard error. Optimal payoff landscape produced with Q-learning is represented by a dashed line.

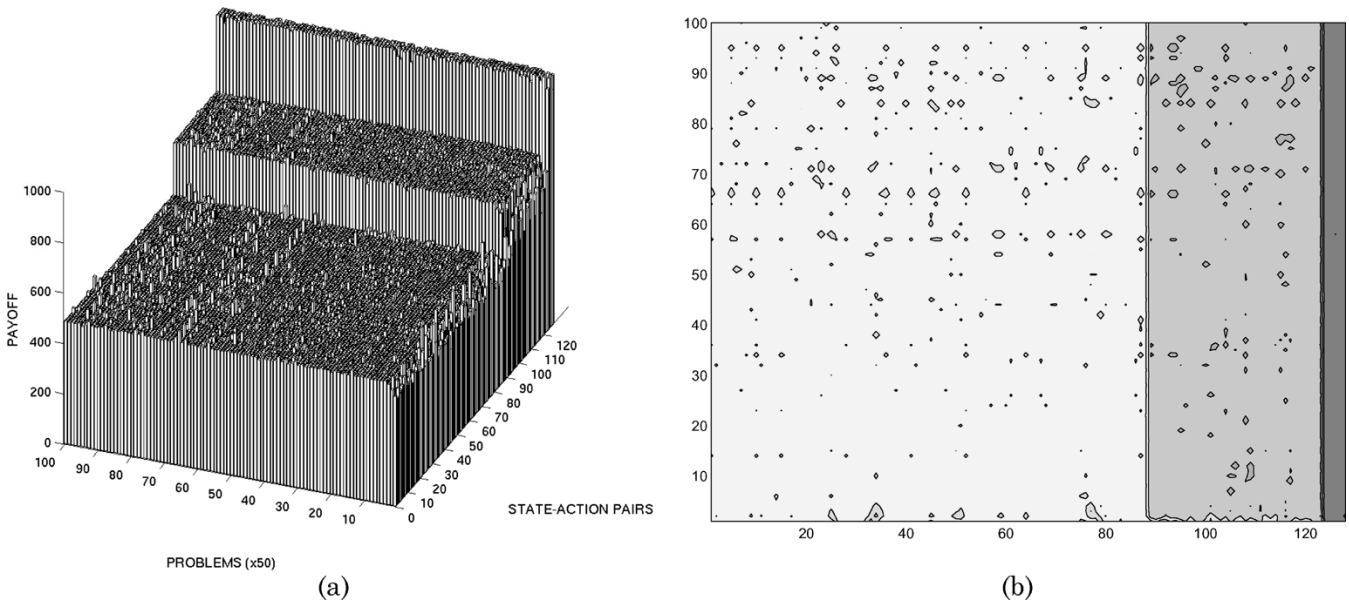

Fig. 24. Evolution of the payoff landscape for XCS in `Woods1`.

performance in 18 runs, while two runs are very near to the optimum. This situation is well depicted by the averages of the evolved landscapes in Fig. 26(a) and (b). Fig. 26(a) shows that with these settings the payoff landscapes evolved by XCS are not able to approximate the lower values of the optimal payoff landscape correctly. In fact, we note that the averaged evolved landscapes do not distinguish the first three steps of the optimal payoff landscape. In contrast, the averaged payoff landscapes evolved by XCSG approximate the optimal payoff quite accurately. The additionally depicted standard errors in Fig. 26(a) and (b) confirms these results; the payoff landscapes evolved by XCS are affected by a large standard error, while those evolved by XCSG are more accurate in that they are affected by a very small standard error. Note the large error bar in Fig. 26(a) around the state-action pair 95, which indicates that XCS has been trying to develop a generalization with a state-action of a very different payoff level.

Fig. 27(a) and (b) represents the averages of the 20 landscapes evolved by XCS and XCSG for `Maze6` when $N = 3000$ and $\mu = 0.01$, corresponding to the performance plots in Fig. 9. Fig. 27(a) shows that with these settings the payoff landscapes evolved by XCS is near to the optimum but the approximation is affected by a large error as shown by the error bars. The large errors are mainly caused by the runs where XCS cannot converge even to a near-optimal solution. In contrast, the average payoff landscape produced by XCSG approximate the optimal payoff landscape well [Fig. 27(b)] and, most importantly, accurately as demonstrated by the very small error bars. Fig. 28(a) and (c) reports the evolution of the payoff landscapes for a single run with XCS [Fig. 28(a)], in which XCS does not converge to an optimal policy, and XCSG [Fig. 29(a)]. In such a typical run, XCS cannot distinguish any of the problem payoff levels, no steps emerge in Fig. 28(a) during evolution, as also confirmed by the 2-D view in Fig. 27(b). In contrast, XCSG can distinguish all

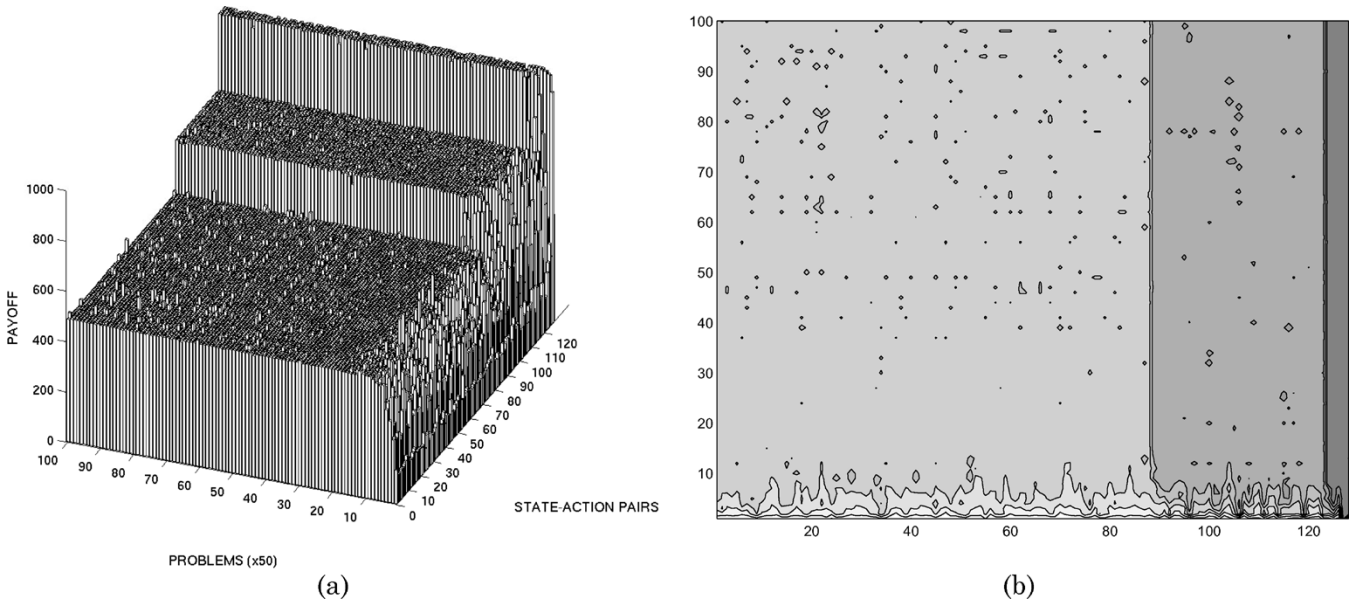

Fig. 25. Evolution of the payoff landscape for XCSG in `Woods1`.

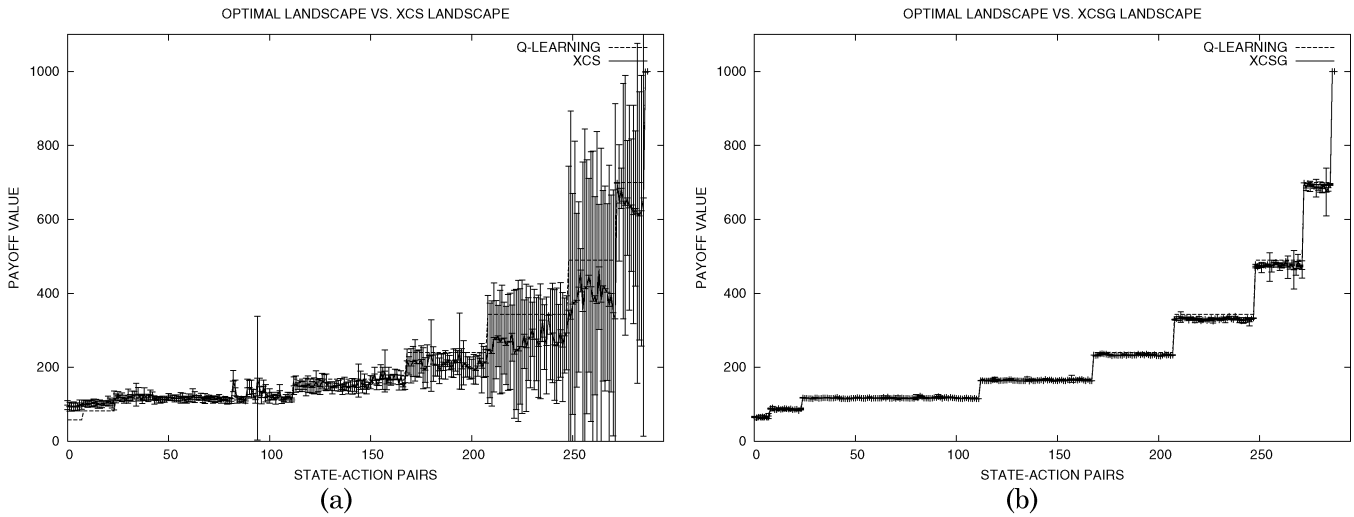

Fig. 26. XCS and XCSG in the `Maze5` environment. (a) Payoff landscape produced by XCS (solid line) with error bars representing the standard error. (b) Payoff landscape produced by XCSG (solid line) with error bars representing the standard error. Optimal payoff landscape produced with Q-learning is represented by a dashed line.

the payoff levels as demonstrated by the steps in Fig. 29(a) and the vertical lines in Fig. 29(b) that represent the various steps of the 2-D plot in Fig. 27(b).

Finally, Fig. 30(a) and (b) shows the averages and standard errors of the 20 landscapes evolved by XCS and XCSG for `Woods14` when $N = 4000$ and $\theta_{\mathrm{ga}} = 300$, corresponding to the performance plots in Fig. 12. With these settings, XCS cannot generate a near optimal strategy to solve `Woods14` even once; XCSG achieves optimal performance in 18 runs, while in the remaining two runs XCSG develops policies that are near-optimal. Fig. 30(a) shows that XCS is able to evolve a payoff approximation which can clearly distinguish the higher payoff levels, which correspond to the positions close to the goal, but it hardly distinguishes the lower payoff levels. In particular, we note peaks all over the state-action pairs from 0 to 100, which indicate that XCS evolved some overgeneralization which associates the same payoff level to all those state-action pairs. In addition, we note that the error is lower on state-action pairs corresponding to high payoff levels, where XCS is actually able to distinguish among different levels, and higher on lower payoff levels. XCSG, on the other hand, is able to represent the optimal payoff landscape very accurately, as demonstrated by the average landscape and the error bars in Fig. 30(b). It is interesting to have a closer look at the payoff values corresponding to the position farthest from the goal. Fig. 31 reports, with a larger scale, the initial part of the average payoff landscape shown in Fig. 30. The distinction between the two payoff levels is very small and overgeneralizations that lead to a reward overestimation seem likely to occur, which explains the nonoptimality of two out of 20 runs in this setting.

X. Conclusion

In this paper, we have added gradient descent methods to XCS. We have compared XCS with XCS with direct gradient (XCSG) and XCS with residual gradient (XCSRG) on a

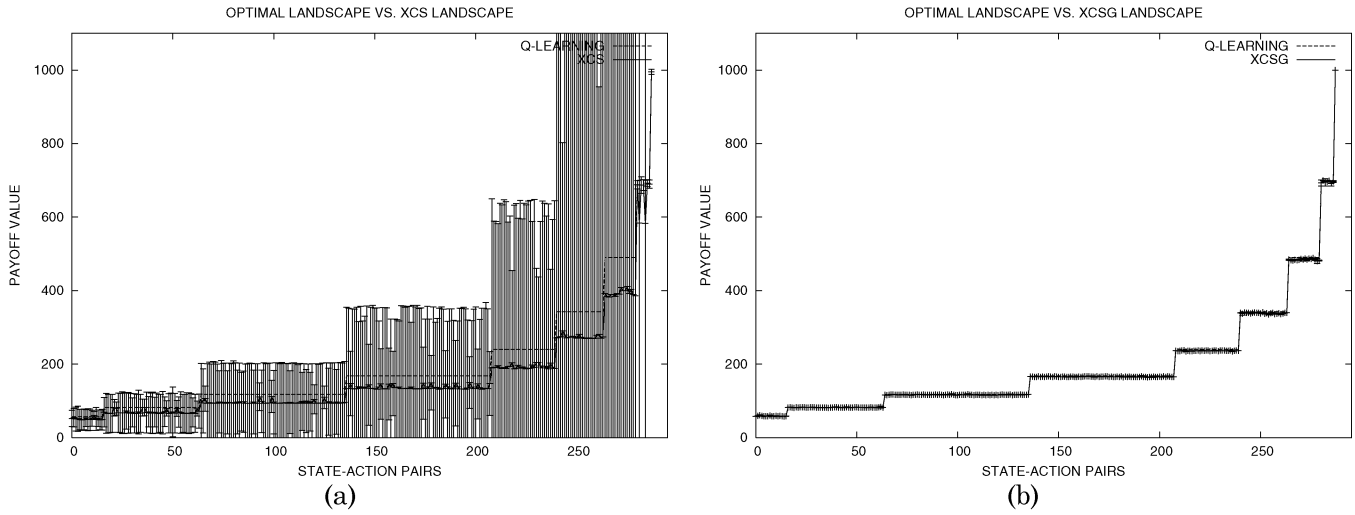

Fig. 27. XCS and XCSG in the `Maze6` environment. (a) Payoff landscape produced by XCS (solid line) with error bars representing the standard error. (b) Payoff landscape produced by XCSG (solid line) with error bars representing the standard error. Optimal payoff landscape produced with Q-learning is represented by a dashed line.

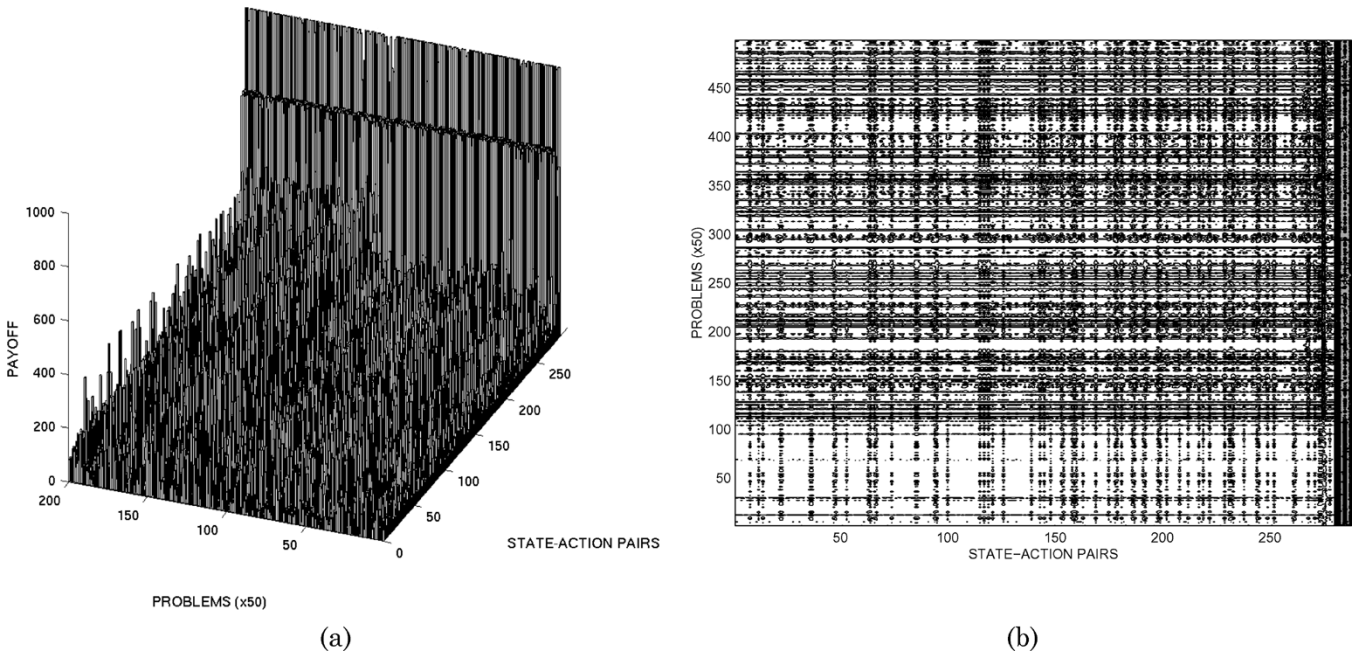

Fig. 28. Evolution of the payoff landscape for XCS in `Maze6`.

number of well-known multistep problems. Overall, we noted that *when* XCS can reach optimal performance, XCSG and XCSRG converge slightly slower to the optimum. This affects only a limited number of initial steps, though, and later in the run, XCSG and XCSRG perform as well as XCS. More importantly, we showed that learning in XCS becomes much more robust due to the addition of gradient descent techniques. Both versions of XCS with gradient descent can reliably solve typically difficult multistep problems in which reward is sparse, resulting in long random walks (`Maze6`), and in which reward is distant, requiring far backpropagation (`Woods14`). We have also analyzed the performance of XCS and XCSG (XCSRG behaves similarly) from a reinforcement learning perspective by studying the evolving payoff landscapes. The analyses we present show that the addition of gradient descent to XCS allows the evolution of payoff landscapes that are more accurate and less noisy than those evolved by plain XCS. Additionally, the results in `Maze6` with 30 additional randomly changing attributes showed that XCSG is able to strongly outperform tabular Q-learning due to its strong generalization capabilities. Overall, the experimental study conducted on woods and maze environments showed improved performance, robustness, as well as the successful and more stable approximation of the optimal payoff function due to the incorporated gradient-based update approach.

We would like to emphasize that our approach is general and might be applied to any other model of learning classifier system for which it is possible to compute a gradient term (i.e., the partial derivative $\partial Q(s_{t-1}, a_{t-1})/\partial w$ discussed in Section IV). For instance, gradient descent can easily be added

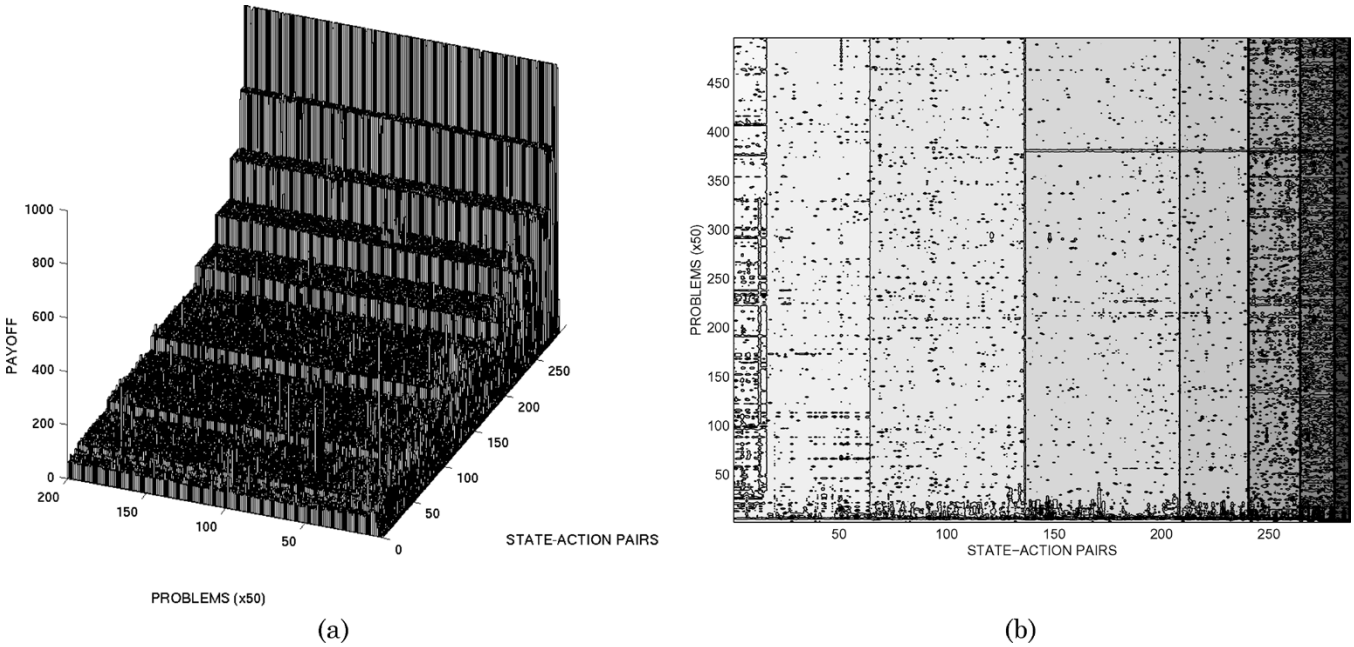

Fig. 29. Evolution of the payoff landscape for XCSG in `Maze6`.

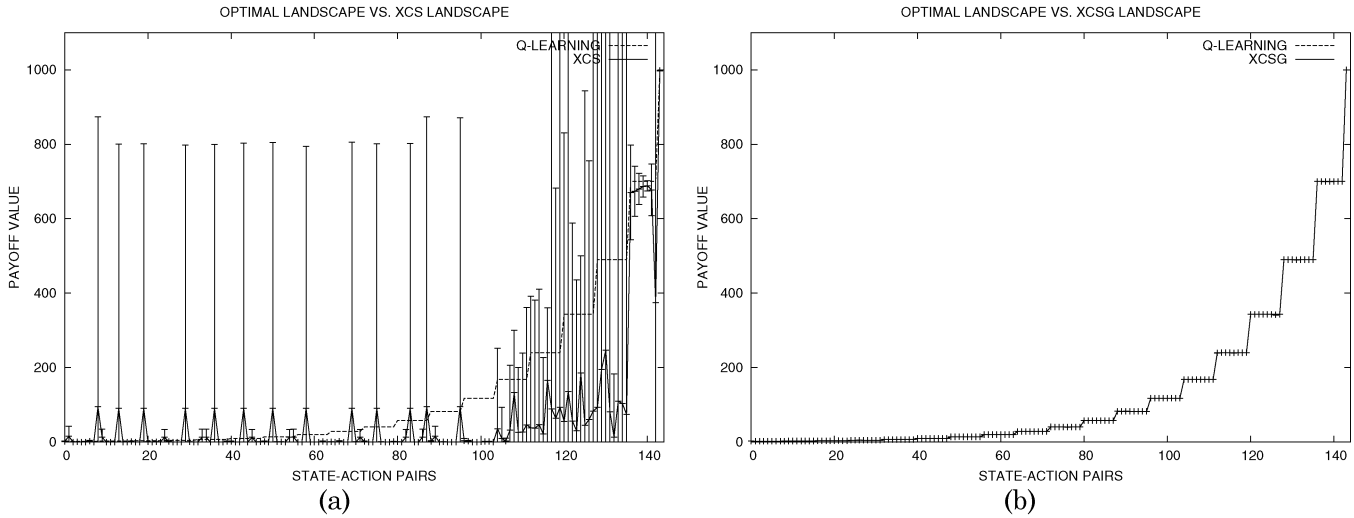

Fig. 30. XCS and XCSG in the `Woods14` environment. (a) Payoff landscape produced by XCS (solid line) with error bars representing the standard error. (b) Payoff landscape produced by XCSG (solid line) with error bars representing the standard error. Optimal payoff landscape produced with Q-learning is represented by a dashed line.

to Kovacs' SB-XCS [15]. In SB-XCS, the system prediction is computed as the numerosity-weighted average of classifier predictions. Accordingly, the gradient term in SB-XCS turns out to be the ratio between the classifier numerosity and the number of classifiers in the action set (see Appendix B for details). Our approach also covers those models of classifier systems where at each time step, only the prediction of one classifier is used to predict the expected payoff, and it is also the only rewarded classifier, such as in [13]. In such models, since only one classifier is activated and rewarded, the gradient component turns out to be exactly one, that is, the gradient does not modify the original update.

The generality of our approach also emphasizes the strong relation between LCSs and tabular reinforcement learning methods, as well as other online generalizing reinforcement learning approaches such as neural-based methods. In fact, XCS appears to lie somewhere in between tabular-based reinforcement learning and neural-based reinforcement learning. In tabular-based reinforcement learning and in particular in tabular Q-learning, each Q-value is represented by *one* entry in the Q-table. In neural-based reinforcement learning, each Q-value is approximated by the activity in the *whole* neural network. XCS lies in between because it estimates Q-values by a *subset* of classifiers.

Similar to neural networks, LCSs are function approximation mechanisms (e.g., [29]). However, the rule-based representation establishes another means of function approximation. Function approximation is done by detecting subspaces whose function values can be effectively approximated by the chosen approximation mechanism. In the simplest case, the approximation is a

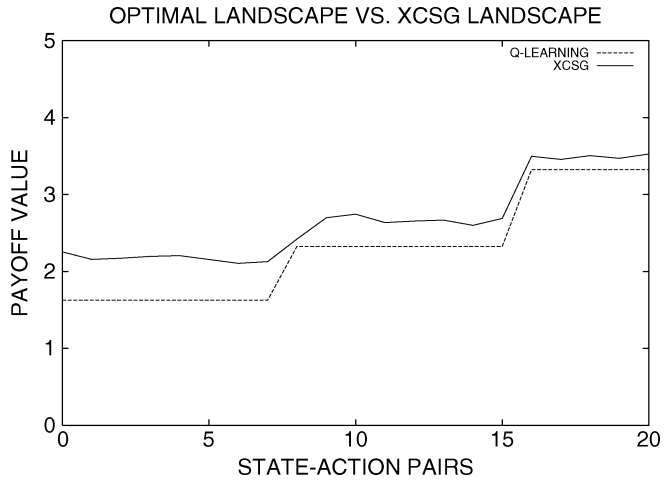

Fig. 31. Detail of the payoff landscape produced by XCSG (solid line) for the positions far from the goal position. Optimal payoff landscape produced with Q-learning.

constant as in this paper but other methods, such as linear approximation, are under investigation [31].

The learning mechanisms in LCSs then evolve a suitable problem space partitioning interactively. While the genetic component generates novel partitions out of currently most promising ones, the approximation method estimates the prediction value in each partition and rates the relative goodness of each partition. This observation also offers an explanation of why residual gradient methods did not improve XCSs learning performance significantly. Since the distinction of successive Q-values is the responsibility of the condition part and, thus, the genetic component, the addition of the residual to the approximation method can, if at all, only be effective early in a run since later the evolved conditions take care of the partitioning of the search space. Problems in which clear-cut value encapsulations are not possible but the subspaces are overlapping are under investigation.

This paper establishes a basis for many possible future investigations in the realm of XCS, as well as LCSs in general. First, the impact of gradient descent needs to be investigated when single-step problems are considered. In multistep problems, we plan to study the impact of gradient descent in XCS when the environment is stochastic, such as those considered elsewhere [21]. We also plan to devise more difficult multistep problems to challenge XCSG and XCSRG to be able to study the impact of the additional residual component in XCSRG and the impact of tournament selection as opposed to roulette wheel selection. Also, the achieved capabilities may be transfered to the real-valued XCS system introduced in [28].

It should also be noted that this paper focused on improving the robustness of XCS learning. Although learning speed was affected slightly negatively, the addition of the gradient-based technique is independent of the basic reinforcement learning mechanism. Thus, speed-up techniques such as the addition of eligibility traces or the application of $Q(\lambda)$ techniques [14], [23] should be straight forward to incorporate. Another approach lies in the combination of XCSG with a model-based learner, such as an anticipatory learning classifier system (ALCS), along the lines of Sutton's Dyna architecture [22], [23]. Butz showed that XCS can be modified to serve as a state value learner in such a combination [7]. The addition of the gradient component is expected to yield additional learning improvements and stability.

The established relation to reinforcement learning and neural-based function approximation techniques, as well as our knowledge about the role and functioning of the evolutionary method in XCS promise to lead to a fundamental theory on XCS's learning capabilities. Future research should aim at showing in what problems a rule-based function approximation approach in combination with an evolutionary-based search mechanism for problem subspaces is advantageous in comparison to neural-based techniques.

APPENDIX A
ALGORITHMIC DESCRIPTION

In this section, following the notation of [10], we present the algorithmic description for the prediction update of XCS with gradient descent (Section IV) and XCS with residual gradient descent (Section VII). In the code, we used the dot notation for identifying the different elements of classifiers. Given a classifier cl, $cl.p$ indicates its prediction, $cl.\varepsilon$ indicates its prediction error, $cl.F$ indicates its fitness, $cl.as$ indicates its estimate of the action set size, and $cl.exp$ indicates its experience.

A. XCS With Gradient Descent

To add direct gradient descent (Section IV) to XCS, the procedure for classifier update is modified as follows (see [10]).

UPDATE SET $([A], \mathrm{P}, [P])$:
1) $F_{[A]} = 0$
2) // compute the sum of fitness for classifiers in the set
3) for each classifier cl in $[A]$
4) $F_{[A]} = F_{[A]} + cl.F$
5) for each classifier cl in $[A]$
6) $cl.exp++$
7) //update prediction $cl.p$ with gradient $cl.F/F_{[A]}$
8) $cl.p \leftarrow cl.p + \beta(\mathrm{P} - cl.p)(cl.F/F_{[A]})$
9) //update prediction error $cl.\varepsilon$
10) if $(cl.exp < 1/\beta)$
11) $cl.\varepsilon \leftarrow cl.\varepsilon + (|\mathrm{P} - cl.p| - cl.\varepsilon)/cl.exp$
12) else
13) $cl.\varepsilon \leftarrow cl.\varepsilon + \beta(|\mathrm{P} - cl.p| - cl.\varepsilon)$
14) //update action set size estimate $cl.as$
15) if $(cl.exp < 1/\beta)$
16) $cl.as \leftarrow cl.as + (\sum_{c\in[A]} c.num - cl.as)/cl.exp$
17) else
18) $cl.as \leftarrow cl.as + \beta(\sum_{c\in[A]} c.num - c.as)$
19) UPDATE FITNESS in set $[A]$
20) if $(doActionSetSubsumption)$
21) DO ACTION SET SUBSUMPTION in $[A]$ updating $[P]$

B. XCS With Residual Gradient Descent

To add residual gradient to XCS, we need to modify the procedure *RUN EXPERIMENT* that takes care of the main performance cycle, and we have also to add a new procedure *UPDATE SET WITH RESIDUAL* that takes care of updating the parameters of classifiers in $[A]_1$ according to the residual component computed from $[\hat{A}]$ (see Section VII for details).

RUN EXPERIMENT():
1) do{
2) $\sigma \leftarrow env$: get situation
3) GENERATE MATCH SET $[M]$ out of $[P]$ using σ
4) GENERATE PREDICTION ARRAY PA out of $[M]$
5) $act \leftarrow$ SELECT ACTION according to PA
6) // select the best action for computing the residual for $[A]_1$
7) $\hat{a} \leftarrow$ SELECT BEST ACTION FROM PA
8) GENERATE ACTION SET $[A]$ out of $[M]$ according to act
9) env: execute action act
10) $\rho \leftarrow rp$: get reward
11) if($[A]_{-1}$ is not empty)
12) GENERATE ACTION SET $[\hat{A}]$ out of $[M]$ according to $\hat{a}$
13) $\mathrm{P} \leftarrow \rho_{-1} + \gamma \max(PA)$
14) UPDATE SET WITH RESIDUAL $[A]_{-1}$ using P, $[\hat{A}]$ possibly deleting in $[P]$
15) RUN GA in $[A]_{-1}$ considering σ_{-1} inserting in $[P]$
16) if(rp: eop)
17) $\mathrm{P} \leftarrow \rho$
18) UPDATE SET $[A]$ using P possibly deleting in $[P]$
19) RUN GA in $[A]$ considering σ inserting in $[P]$
20) empty $[A]_{-1}$
21) else
22) $[A]_{-1} \leftarrow [A]$
23) $\rho_{-1} \leftarrow \rho$
24) $\sigma_{-1} \leftarrow \sigma$
25) }while(termination criteria are not met)

UPDATE SET WITH RESIDUAL $([A], [\hat{A}], \mathrm{P}, [P])$:
1) // compute the sum of fitness for classifiers in the set $[\hat{A}]$
2) $F_{[\hat{A}]} = 0$
3) for each classifier cl in $[\hat{A}]$
4) $F_{[\hat{A}]} = F_{[\hat{A}]} + cl.F$
5) // compute the sum of fitness for classifiers in the set $[A]$
6) $F_{[A]} = 0$
7) for each classifier cl in $[A]$
8) $F_{[A]} = F_{[A]} + cl.F$
9) for each classifier cl in $[A]$
10) $cl.exp++$
11) //compute the residual contribution res
12) $res = 0$
13) if $(cl \in [\hat{A}])$
14) $res = \gamma cl.F/F_{[\hat{A}]}$
15) //update prediction $cl.p$ with gradient $cl.F/F_{[A]}$
16) $cl.p \leftarrow cl.p + \beta(\mathrm{P} - cl.p)(cl.F/F_{[A]} - res)$
17) //update prediction error $cl.\varepsilon$
18) if $(cl.exp < 1/\beta)$
19) $cl.\varepsilon \leftarrow cl.\varepsilon + (|\mathrm{P} - cl.p| - cl.\varepsilon)/cl.exp$
20) else
21) $cl.\varepsilon \leftarrow cl.\varepsilon + \beta(|\mathrm{P} - cl.p| - cl.\varepsilon)$
22) //update action set size estimate $cl.as$
23) if $(cl.exp < 1/\beta)$
24) $cl.as \leftarrow cl.as + (\sum_{c \in [A]} c.num - cl.as)/cl.exp$
25) else
26) $cl.as \leftarrow cl.as + (\sum_{c \in [A]} c.num - c.as)$
27) UPDATE FITNESS in set $[A]$
28) if $(doActionSetSubsumption)$
29) DO ACTION SET SUBSUMPTION in $[A]$ updating $[P]$

APPENDIX B
ADDING GRADIENT TO SB-XCS

Kovacs' SB-XCS [15] is a modification of Wilson's XCS [26] in which classifier fitness is based on classifier prediction (or *strength*) as opposed to the classifier's fitness based relative accuracy used in XCS. SB-XCS resembles many features of the original XCS but it differs from it in the following main points.

- In SB-XCS, classifier prediction p is used both for estimating action prediction $P(a)$ and for estimating the classifier fitness.
- In SB-XCS, system prediction is computed as a numerosity-weighted average of classifier predictions, as opposed to the fitness-weighted average used in XCS. In particular (7) (Section III), used in XCS to estimate the prediction associated to action a_i, in SB-XCS is replaced by

$$P(a_i) = \frac{\sum_{cl_j \in [M]|_{a_i}} p_j \times num_j}{\sum_{cl_j \in [M]|_{a_i}} num_j}$$

  where $[M]|_{a_i}$ represents the subset of classifiers of $[M]$ with action a_i, num_j is the numerosity of classifier cl_j.
- SB-XCS uses tournament selection [9] instead of the original roulette wheel selection to overcome the initial strong generalization pressure (see Kovacs [15]).

To add gradient to SB-XCS, we have to compute the gradient component $\partial Q(s_{t-1}, a_{t-1})/\partial w$. Following the same approach

discussed in Section IV, we can compute such a contribution for SB-XCS as follows:

$$\begin{aligned}\frac{\partial Q(s_{t-1},a_{t-1})}{\partial w} &= \frac{\partial}{\partial p_k}\left[\frac{\sum_{cl_j\in[A]_{-1}} p_j \times num_j}{\sum_{cl_j\in[A]_{-1}} num_j}\right] \\ &= \frac{1}{\sum_{cl_j\in[A]_{-1}} num_j}\left[\frac{\sum_{cl_j\in[A]_{-1}} p_j \times num_j}{\sum_{cl_j\in[A]_{-1}} num_j}\right] \\ &= \frac{1}{\sum_{cl_j\in[A]_{-1}} num_j}\frac{\partial}{\partial p_k}\left[\sum_{cl_j\in[A]_{-1}} p_j \times num_j\right] \\ &= \frac{num_k}{\sum_{cl_j\in[A]_{-1}} num_j}. \end{aligned} \tag{17}$$

Thus, the gradient component in SB-XCS turns out to be an additional term in the prediction update that weights the prediction update contribution according to the percentage of classifiers in the niche.

ACKNOWLEDGMENT

M. Butz and P. L. Lanzi are more than grateful to X. Llora, M. Pelikan, and K. Sastry for their help and the useful discussions. P. L. Lanzi wishes to thank M. Colombetti and S. Ceri for invaluable support; M. Butz and P. L. Lanzi also wish to thank D. E. Goldberg for invaluable discussions and inspiration.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research, the National Science Foundation, or the U.S. Government.

REFERENCES

[1] L. C. Baird, "Residual algorithms: Reinforcement learning with function approximation," in *Proc. 12th Int. Conf. Mach. Learn.*, Jul. 1995, pp. 30–37.

[2] ——, "Reinforcement learning through gradient descent," Ph.D. dissertation, School of Comput. Sci., Carnegie Mellon Univ., Pittsburgh, PA, 1999.

[3] A. M. Barry, "The stability of long action chains in XCS," *Soft Comput.—A Fusion Foundations, Methodologies, Applicat.*, vol. 6, no. 3–4, pp. 183–199, 2002.

[4] E. Bernadó, X. Llorà, and J. M. Garrell, "XCS and GALE: A comparative study of two learning classifier systems with six other learning algorithms on classification tasks," in *Proc. 4th Int. Workshop Learn. Classifier Syst.*, 2001, pp. 337–341.

[5] L. Bull and J. Hurst, "ZCS redux," *Evol. Comput.*, vol. 10, no. 2, pp. 185–205, 2003.

[6] M. V. Butz, "XCS (+ tournament selection) classifier system implementation in C, Version 1.2," Illinois Genetic Algorithms Lab., Univ. Illinois, Urbana, IL, Tech. Rep. 2 003 023, 2003.

[7] M. V. Butz and D. E. Goldberg, "Generalized state values in an anticipatory learning classifier system," in *Anticipatory Behavior in Adaptive Learning Systems: Foundations, Theories, and Systems*, M. V. Butz, O. Sigaud, and P. Gérard, Eds. Berlin, Germany: Springer-Verlag, 2003, pp. 282–301.

[8] M. V. Butz, T. Kovacs, P. L. Lanzi, and S. W. Wilson, "Toward a theory of generalization and learning in XCS," *IEEE Trans. Evol. Computat.*, vol. 8, no. 1, pp. 28–46, Feb. 2004.

[9] M. V. Butz, K. Sastry, and D. E. Goldberg, "Tournament selection: Stable fitness pressure in XCS," in *Lecture Notes in Computer Science*, E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, Eds. Chicago, IL, Jul. 12–16, 2003, vol. 2724, Proc. Genetic and Evol. Comput., pp. 1857–1869.

[10] M. V. Butz and S. W. Wilson, "An algorithmic description of XCS," *Soft Comput.—A Fusion of Foundations, Methodologies, Applicat.*, vol. 6, no. 3–4, pp. 144–153, 2002.

[11] P. W. Dixon, D. W. Corne, and M. J. Oates, "A preliminary investigation of modified XCS as a generic data mining tool," in *Advances in Learning Classifier Systems*, P. L. Lanzi, W. Stolzmann, and S. W. Wilson, Eds. Berlin, Germany: Springer-Verlag, 2002, vol. 2321, LNAI, pp. 133–150.

[12] S. A. Glantz and B. K. Slinker, *Primer of Applied Regression & Analysis of Variance*, 2nd ed. New York: McGraw-Hill, 2001.

[13] D. E. Goldberg, "Computer-aided gas pipeline operation using genetic algorithms and rule learning," Ph.D. dissertation, Univ. Michigan, Ann Arbor, MI, 1983.

[14] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: a survey," *J. Artif. Intell. Res.*, vol. 4, pp. 237–285, 1996.

[15] T. Kovacs, *Strength or Accuracy: Credit Assignment in Learning Classifier Systems*. Berlin, Germany: Springer-Verlag, 2003.

[16] P. L. Lanzi, "An analysis of generalization in the XCS classifier system," *Evol. Comput.*, vol. 7, no. 2, pp. 125–149, 1999a.

[17] ——, "Extending the representation of classifier conditions, Part I: From binary to Messy coding," in *Proc. Genetic Evol. Comput. Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., 1999b, pp. 337–344.

[18] ——, "Mining interesting knowledge from data with the XCS classifier system," in *Proc. Genetic Evol. Comput. Conf.*, L. Spector, E. D. Goodman, A. Wu, W. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, Eds., Jul. 7–11, 2001, pp. 958–965.

[19] ——, "Learning classifier systems from a reinforcement learning perspective," *Soft Comput.—A Fusion of Foundations, Methodologies, Applicat.*, vol. 6, no. 3–4, pp. 162–170, 2002a.

[20] ——, (2002b) The XCS library. [Online]. Available: http://xcslib.sourceforge.net

[21] P. L. Lanzi and M. Colombetti, "An extension to the XCS classifier system for stochastic environments," in *Proc. Genetic Evol. Comput. Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., Orlando, FL, Jul. 1999, pp. 353–360.

[22] R. S. Sutton, "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming," in *Proc. 7th Int. Conf. Mach. Learn.*, 1990, pp. 216–224.

[23] R. S. Sutton and A. G. Barto, *Reinforcement Learning—An Introduction*. Cambridge, MA: MIT Press, 1998.

[24] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Cambridge, U.K., 1989.

[25] S. W. Wilson, "ZCS: A zeroth level classifier system," *Evol. Comput.*, vol. 2, no. 1, pp. 1–18, 1994. [Online]. Available: http://prediction-dynamics.com/.

[26] ——, "Classifier fitness based on accuracy," *Evol. Comput.*, vol. 3, no. 2, pp. 149–175, 1995. [Online]. Available: http://prediction-dynamics.com/.

[27] ——, "Generalization in the XCS classifier system," in *Proc. 3rd Ann. Conf. Genetic Program.*, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, Eds., 1998, pp. 665–674. [Online]. Available: http://prediction-dynamics.com/.

[28] ——, "Get real! XCS with continuous-valued inputs," in *Learning Classifier Systems. From Foundations to Applications*, P. L. Lanzi, W. Stolzmann, and S. W. Wilson, Eds. Berlin, Germany: Springer-Verlag, 2000, vol. 1813, LNAI, pp. 209–219.

[29] ——, "Function approximation with a classifier system," in *Proc. Genetic Evol. Comput. Conf.*, L. Spector, E. D. Goodman, A. Wu, W. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, Eds., Jul. 7–11, 2001a, pp. 974–981.

[30] ——, "Mining oblique data with XCS," in *Lecture Notes in Computer Science*, P. L. Lanzi, W. Stolzmann, and S. W. Wilson, Eds. Berlin, Germany: Springer, 2001b, vol. 1996, Proc. Int. Workshop Learn. Classifier Syst., pp. 158–176.

[31] ——, "Classifiers that approximate functions," *Natural Comput.*, vol. 1, pp. 211–234, 2002.

Martin V. Butz received the Diploma degree in computer science with a minor in psychology (Honors) from the Bayerische Julius-Maximilians Universität Würzburg, Würzburg, Germany, in August 2001, and the Ph.D. degree from the University of Illinois at Urbana–Champaign, Urbana, with Prof. D. E. Goldberg as his supervisor in October 2004. His thesis focuses on a detailed analysis of learning classifier systems and the XCS system in particular, also emphasizing the relation to cognitive systems.

Since the completion of his Ph.D. degree, he is studying the influences of anticipatory mechanisms on behavioral control and cognition at the Department of Cognitive Psychology, Universität Würzburg, in collaboration with several European partners supported by the European Commission.

David E. Goldberg received the B.S.E., M.S.E., and Ph.D. degrees in civil engineering from the University of Michigan, Ann Arbor, in 1975, 1976, and 1983, respectively.

He is the Jerry S. Dobrovolny Distinguished Professor of Entrepreneurial Engineering at the University of Illinois at Urbana–Champaign (UIUC), Urbana, and Director of the Illinois Genetic Algorithms Laboratory (IlliGAL, http://www-illigal.ge.uiuc.edu/). From 1976 to 1980, he held a number of positions at Stoner Associates, Carlisle, PA, including Project Engineer and Marketing Manager. Following his doctoral studies, he joined the Engineering Mechanics Faculty, University of Alabama, Tuscaloosa, in 1984, and he moved to the University of Illinois–Champaign in 1990. His book *Genetic Algorithms in Search, Optimization and Machine Learning* (Reading, PA: Addison-Wesley, 1989) is one of the most widely cited texts in computer science according to *Citeseer*. His most recent book *The Design of Innovation: Lessons from and for Competent Genetic Algorithms* (http://www-doi.ge.uiuc.edu/), shows how to design scalable genetic algorithms and how such procedures are similar to certain processes of human innovation. His research focuses on the design, analysis, and application of genetic algorithms—computer procedures based on the mechanics of natural genetics and selection, and other innovating machines.

Professor Goldberg was a 1985 recipient of a U.S. National Science Foundation Presidential Young Investigator Award, and in 1995 he was named an Associate of the Center for Advanced Study at UIUC. He was founding Chairman of the International Society for Genetic and Evolutionary Computation (http://www.isgec.org/).

Pier Luca Lanzi was born in Turin, Italy, in 1967. He received the Laurea degree in computer science from the Università degli Studi di Udine, Udine, Italy, in 1994, and the Ph.D. degree in computer and automation engineering from the Politecnico di Milano, Milan, Italy, in 1999.

He is an Associate Professor in the Department of Electronics and Information, Politecnico di Milano. He is member of the Editorial Board of the *Evolutionary Computation Journal*. His research areas include evolutionary computation, reinforcement learning, and machine learning. He is interested in applications to data mining and autonomous agents.