Noname manuscript No.
(will be inserted by the editor)

XCSF with tile coding in discontinuous action-value landscapes

Pier Luca Lanzi · Daniele Loiacono

the date of receipt and acceptance should be inserted later

Abstract Tile coding is an effective reinforcement learning method that uses a rather ingenious generalization mechanism based on (i) a carefully designed parameter setting and (ii) the assumption that nearby states in the problem space will correspond to similar payoff predictions in the action-value function. Previously, we extended XCSF with tile coding prediction and compared it to tabular tile coding, showing that (i) XCSF performs as well as parameter-optimized tile coding, while also (ii) evolving individualized parameter settings in each problem subspace. Our comparison was based on a set of well-known reinforcement learning environments (2D Gridworld and the Mountain Car) that involved no action-value discontinuities and so posed no challenge to tabular tile coding.

In this paper, we go a step further and test XCSF with tile coding on a set of problems designed to challenge tile coding by introducing discontinuities in the action value landscape. The new testbed (called MazeWorld) extends 2D Gridworld with impenetrable obstacles, a conceptually simple modification that can dramatically increase the problem complexity for tabular tile coding. We compare four versions of XCSF with tile coding (each adapting a different set of parameters) to tabular tile coding on four problems of increasing complexity. We show that our system (i) needs fewer training problems than standard tile coding to reach an optimal policy; (ii) it can evolve adequate coding parameters in each subspace without any previous knowledge; and that (iii) even when XCSF is not allowed to evolve these parameters, the genetic algorithm will still adapt classifier conditions to properly decompose the problem into subspaces thus being much less sensitive to the parameter settings than tabular tile coding.

P.L. Lanzi
Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy
E-mail: pierluca.lanzi@polimi.it

D. Loiacono
Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy
E-mail: daniele.loiacono@polimi.it

1 Introduction

In Reinforcement Learning [12], an agent learns to solve a problem through *trial-and-error* interactions with an unknown environment. Its goal is to maximize the total amount of future reward it will receive. To achieve such simple goal, the agent has to learn an action-value function $Q(s,a)$ that computes an estimate of the cumulative reward it will receive by performing the action a when the problem is in state s (and then continuing to act using its best known policy). In small problems, the action-value function can be implemented using a *look-up* table, however in large problems look-up tables are infeasible and it is only possible to learn an approximation of the optimal action-value function.

Tile coding [11] is a popular approach to approximate action-value functions in large reinforcement learning problems. It couples a simple linear function approximator to the partitioning of the state space into overlapped areas called *tiles*. The performance of tile coding heavily relies on a proper setting of the parameters that defines the partitioning of problem space: (i) the number of tiling t, i.e., the number of tiles that overlap in any point of the problem space; (ii) the tiling resolution r, i.e., the smallest offset between two partially overlapped tiles. Unfortunately, the best setting for this parameters is problem dependent and might also change during the learning process [10]. Glaubius and Smart [4] note that tabular tile coding relies on Euclidean distance as a source of generalization within the problem space, and this constitutes one of its major limitation.

Learning classifier systems take another approach to the approximation of the action-value function they represent through a set of *condition-action-prediction* rules, called classifiers. Each classifier represents a part of the overall action-value function: classifier conditions identify problem subspaces and associate a *constant prediction value* p [14] or a *prediction function* $p(s,\mathbf{w})$ [15] to the classifier action. The prediction function $p(s,\mathbf{w})$ computes the actual classifier prediction based on the current state s and on a parameter vector $\mathbf{w}$ associated to each classifier; $p(s,\mathbf{w})$ is usually implemented using a linear approximator (i.e. $s\mathbf{w}$), but more complex functions can be used [5,6].

In [8], we extended Wilson's XCSF [15] with tile coding prediction by replacing the original classifier prediction function with a tile coding approximator. We compared our version of XCSF to the tabular tile coding implementation on three well-known reinforcement testbeds, the 2D gridworld [1], the puddle world [1], and the mountain car [11]. Our results showed that XCSF with tile coding prediction can solve reinforcement learning problems which would be too challenging for plain XCSF. In addition, they showed that the evolutionary component of XCSF can partition the problem space effectively to cover different subspaces with the most adequate approximators while also adapting the values of t and r. Finally, our analysis showed that the parameters' adaptation would follow the rules of thumb delineated by human experts [10].

All the problems in [8] involved rather simple action-value functions that posed no difficulty to tile coding. Accordingly, all the results in [8] show that XCSF with tile coding performs as well as tabular tile coding using the best parameters possible and that evolution did not require more training problems than tabular tile coding to adapt the tile coding parameters t and r associated to each classifiers. In this paper, we take a step further and analyze the generalization capabilities of XCSF with tile coding on a set of problems that put a main limitation of

tile coding to test. In fact, tile coding heavily relies on Euclidean distance as the basis for generalization and assumes that nearby states in the problem space will correspond to similar values of the action-value function [4]. To challenge this assumption, we introduced a new testbed, called *MazeWorld*, that extends 2D gridworld [1] with impenetrable obstacles. This relatively simple modification to 2D gridworld introduces discontinuities in the action-value function that challenge the generalization capabilities of tile coding since nearby areas that are separated by (thin) obstacles might now correspond to very different action-value function landscapes. At the same time, XCSF should still be able to evolve local models that properly generalize the target action-value function in each problem subspace. We compared four versions of XCSF with tile coding (each one adapting a different set of tile coding parameters) to plain tile coding using four different *MazeWorld* environments. Our results show that XCSF can adapt the tile coding parameters locally (any one of them) and reach the same performance as the best possible tabular tile coding configuration. Interestingly, even when XCSF is not allowed to adapt any tile coding parameter (not t nor r) and these are set to values which prevent tabular tile coding to solve the problem, XCSF can still work on classifier conditions to decompose the problem space so as to reach a near optimal solution. Furthermore, XCSF with tile coding requires fewer training problems than plain tile coding in almost all the problems and for almost all the parameter settings. Finally, our results also confirm our previous finding [8] showing that in XCSF the tile coding parameter adaptation follows the rules of thumb suggested by human experts [10].

2 Tile Coding

Reinforcement learning (RL) is defined as the problem of an agent that learns through trial-and-error interaction with an environment [12] that provides feedback only by means of a numerical *reward*. At time step t, the agent perceives the current state of the environment s_t and performs an action a_t following its current action selection policy. As a consequence, the agent reaches state s_{t+1} and receives a numerical reward r_{t+1}. The goal of the agent is to maximize the amount of reward received from the environment. To succeed the agent can learn either an action-value function $Q(s_t, a_t)$ that maps the current state action pair into the amount of expected reward or a value function $V(s_t)$ that maps the current state into amount of expected reward.

Reinforcement learning algorithms rely on two rather impractical assumptions: (i) the action-value $Q(s, a)$ (or the value function $V(s_t)$) is represented by a *look-up* table and (ii) the agent visits each state-action pair a large number of times. These assumptions do not apply to problems involving large and continuous state-action spaces. To solve this problem, some form of generalization is necessary, that is, how to represent the action-value $Q(s, a)$ (or the value function $V(s_t)$) compactly and, at the same time, how to reuse collected experience in areas of the problem space scarcely or even never visited.

Generalization in reinforcement learning is usually implemented by methods of function approximation: $Q(s, a)$ (or $V(s_t)$) is represented as a function f parameterized by a vector $\boldsymbol{\theta}$ which is learned online from the interaction between the agent and the environment. Apart from the type of approximator used (e.g.,

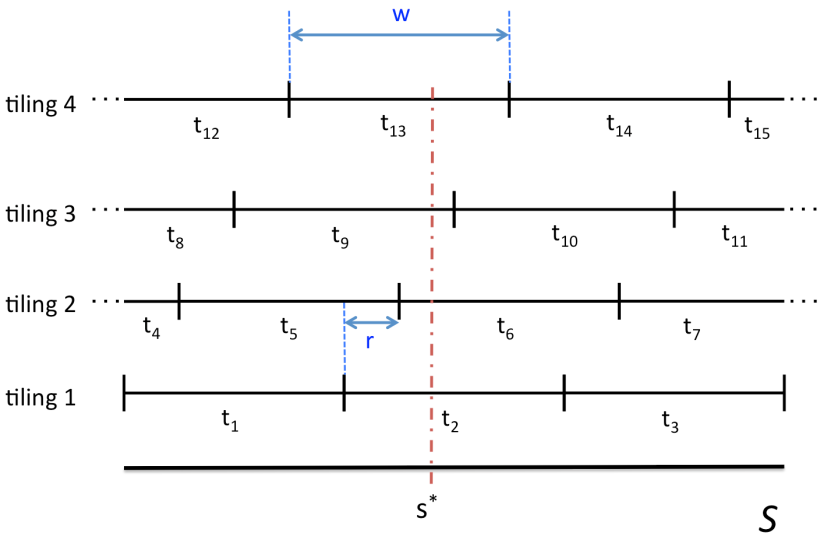

Fig. 1: Example of tile coding on an input space S with a single variable; the input space is covered with 4 tilings, i.e., $t = 4$; each tile covers one third of the input space, i.e., $w = |S|/3$, for a total of 15 tiles; accordingly resolution is $r = w/4$. As an example the input state $s*$ is covered by tiles $\{t_2, t_6, t_9, t_{13}\}$.

linear [11] or neural networks [13]), these methods are also characterized by an *input mapping* function $\phi(s)$ that translates the input space into a feature space more favorable to the approximator (e.g., $\phi(s)$ can be used to map a continuous space into a discrete one).

Tile coding [11] is a popular method of function approximation in reinforcement learning. It exploits an input mapping function $\phi(s)$ to map a continuous state space s into a vector of m binary features $\langle \phi_1(s), \dots \phi_m(s) \rangle$; thus, it computes the value of $Q(s, a)$ as $\phi(s)\boldsymbol{\theta}_a$, where $\boldsymbol{\theta}_a$ is a vector of m parameters associated to action a which are updated with gradient descent.

Function $\phi(s)$ maps the state space into a set of t overlapping *tilings*; each *tiling* partitions the state space into a set of non-overlapping *tiles*; each tile is an hyper-rectangle (i.e., a collection of intervals, one for each state variable) in the state space and it is associated to an element of the parameter vector $\boldsymbol{\theta}$. Given the state s, the component $\phi_i(s)$ of the features vector associated to the i-th tile t_i is computed as,

$$\phi_i(s) = \begin{cases} 0 & \text{if } s \notin t_i, \\ 1 & \text{if } s \in t_i. \end{cases} \tag{1}$$

Tilings are usually positioned to cover the whole input space uniformly: if each tiling consists of tiles of size w and consecutive tilings are displaced by a resolution r, then t ($t = w/r$) tilings are used to represent the input space. Resolution r represents the minimum distance allowed between two states which guarantees that tile coding can associate different values to each state. Figure 1 shows the tile coding of a single variable input space S. The input space S is covered by 4 tilings

($t = 4$); each tile covers a third of the input space ($w = |S|/3$); thus, the input space S is mapped into 15 tiles $\{ t_1, \ldots, t_{15} \}$ with a resolution $r = w/4$. Accordingly, the input state s^* is covered by tiles $\{t_2, t_6, t_9, t_{13}\}$ and, thus, is mapped into the binary vector $\langle 0,1,0,0,0,1,0,0,1,0,0,0,1,0,0\rangle$ in which ones correspond to the active tiles in $s*$ (i.e., those that cover s^*).

Sutton et al. [11] provided limited insight on how to choose the parameters t, w and r. Later, Sherstov and Stone [10] performed an empirical analysis about the effect of different parameters configurations on the performance of tile coding. In [10], they showed that the choice of resolution affects the learning speed as well as the quality of the action-value function approximation whereas the choice of different number of tilings and width, leading to *the same resolution*, affects the learning speed. In fact, a higher resolution (i.e., a lower value of r) allows a better approximation of the action-value function but generally requires a longer learning process; at the same time, a higher number of tilings leads to faster learning at the beginning, but it might be disruptive at the end of the learning process.

3 Problems that Challenge Tile-Coding

Tile coding has been successfully applied to many reinforcement learning problems [11,12], yet it has several limitations. Sherstov and Stone [10] noted that the choice of the parameters r, t and w, plays a key role in tile coding. Firstly, in complex problems and when little domain knowledge is available, finding a suitable parameter setting might be a tough and expensive task. Secondly, the results reported in [10] also suggest that often is not possible to identify a set of optimal parameters but these should be adapted during learning. In [10], they show how the parameters of tile coding should ideally (i) encourage broad generalization in the initial stage of the learning process (i.e., when action-value function is rapidly changing), while (ii) discourage generalization in the final stage of the learning process (i.e., when the action-value function is near to convergence). In addition, Glaubius and Smart [4] suggest that one of the major limitation of tile coding is that it relies on Euclidean distance as a source of generalization within the problem space (i.e., similarity in the problem space is measured on the basis of the Euclidean distance). Accordingly, when this assumption does not hold, due to the topology of the input space induced by the problem dynamics, tile coding may easily lead to a poor approximation of the action-value function.

In our previous work [8], we tested XCSF with tile coding prediction on well known problems taken from the reinforcement learning literature [12], i.e., the 2D gridworld [1], the puddle world [1], and the mountain car [11]. Our experimental results (see [8] for more details) showed that XCSF with tile coding can evolve an optimal solution as fast as *plain tile coding* but can also adapt the tile coding parameters during learning effectively.

However, all the environments in [8] involved a problem space topology that allows a simple generalization based on the Euclidean distance. In fact, as Figure 2 clearly shows, both the 2d gridworld (Figure 2a) and the puddle world (Figure 2b) have a rather smooth value function in that states which are near in the input space have very similar values; the mountain car (Figure 2c) involves a value function with a quite steep slope in few areas of the problem space, but it still allows a rather broad generalization.

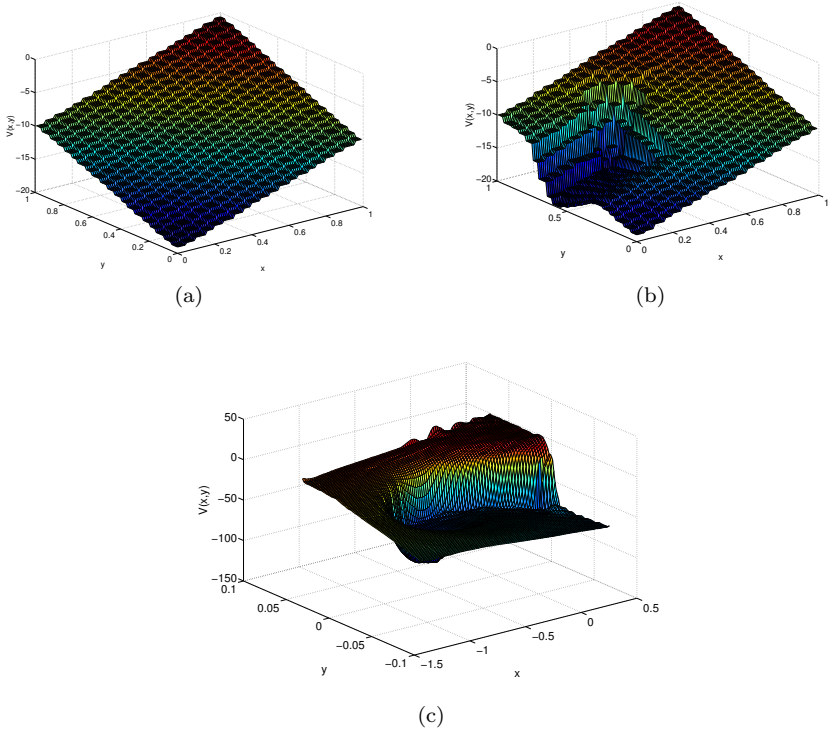

Fig. 2: Value function of typical problems taken from the reinforcement learning literature: (a) the 2D gridworld, (b) the puddle world, and (c) the mountain car.

In this work, we take our previous analysis [8] a step further and we investigate how XCSF with tile coding prediction performs on problems where generalization in the problem space is not based on the Euclidean distance. In particular, we study whether XCSF can exploit its capabilities of decomposing the problem space to provide a reliable generalization in problem with a more complex space topology and how it compares with *plain tile coding*. To this purpose, we introduce a new class of problems, called *MazeWorld*, which extends 2D gridworld testbed by adding one or more impenetrable obstacles in the environment. Similarly to the 2D gridworld, the agent state is defined by a pair of real coordinates $\langle x, y \rangle$ in $[0, 1]^2$ and there are four possible actions, {left, right, up, down}, which correspond to a step of fixed size $s = 0.05$ in one of the main four directions (as in the original 2D gridworld [2]); the agent must reach an area of the problem space defined as *goal*, that is when *both* its coordinates are greater than $1 - s$. In contrast to the 2D gridworld, *MazeWorld* environment contains one or more impenetrable obstacles that cannot be traversed by the agent. Any action that would take the agent outside the domain $[0, 1]^2$ or inside an obstacle will take the agent to the nearest empty position of the grid border. The agent can start *anywhere* but in the goal position or in a position occupied by an obstacle. It is possible to design several instances (or *mazes*) of the

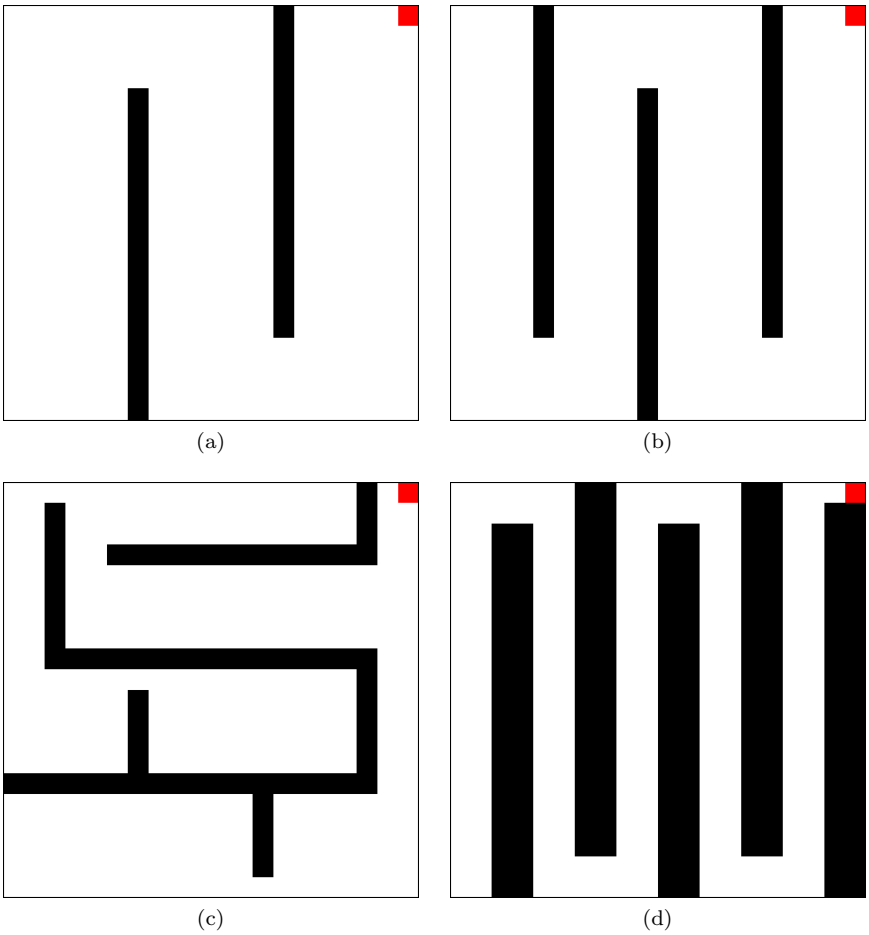

Fig. 3: The four *MazeWorld* problems used for the experimental analysis in this work: (a) `Maze1`, (b) `Maze2`, (c) `Maze3`, and (d) `Maze4`; the black regions are impenetrable obstacles; the red regions are goal areas.

MazeWorld environment by changing the position of the obstacles in the problem space. The instances of the *MazeWorld* environment considered in this work are depicted in Figure 3.

Although with *MazeWorld* we introduce only small changes to the problem definition of the popular 2D gridworld testbed, these changes still lead to rather large differences in the problem space. In fact, as Figure 4 shows, the value function in the *MazeWorld* problems presents several discontinuities that prevent a simple generalization within the problem space and thus challenge a global function approximators like tile coding.

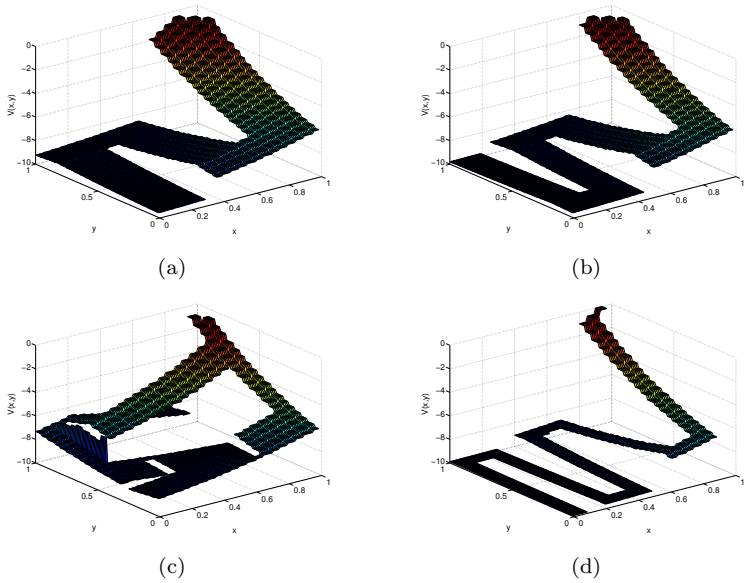

Fig. 4: Optimal value function of the four *MazeWorld* problems used for the experimental analysis in this work: (a) `Maze1`, (b) `Maze2`, (c) `Maze3`, and (d) `Maze4`.

4 XCS with Tile Coding Prediction

XCSF with tile coding prediction extends XCSF [15] by replacing the usual linear prediction with a tile coding approximator. The system has the same structure as XCSF except for (i) two additional parameters, r and t, added to each classifier which define the structure of the tiles over the problem subspace identified by the classifier condition; (ii) the weights vector $\mathbf{w}$ associated to each classifier that has now a weight w_i for each tile in the input domain defined by the condition C of the classifier; (iii) the discovery component that can also evolve the values of r and t.

Classifiers consist of a condition, an action, and six parameters. The condition specifies which input states the classifier matches and it is represented by a concatenation of intervals (l_i, u_i) where the lower and upper bound, l_i and u_i, are real values. The action specifies the action for which the payoff is predicted; in our case, there are four possible actions, one for each direction of movement. The six parameters are: the weight vector $\mathbf{w}$ associated to the classifier's tiling; the number of tilings t and their resolution r; the prediction error ε, that estimates the error affecting the classifier prediction; the fitness F that estimates the accuracy of the classifier prediction; and the numerosity *num*, a counter used to represent different copies of the same classifier;

Performance Component. At time step t, the system builds a *match set* [M] containing the classifiers in the population [P] whose condition matches the current sensory input s_t; 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 [15]. The prediction of all the classifiers in [M] is computed using the classifier's tile coding. First, the tile coding mapping function (Equation 1) is applied to the input state s_t to compute the binary vector $\phi(s_t)$ (Section 2). Then, the classifier prediction is computed as $\phi(s_t)\mathbf{w}$. For each action a_i in [M], the system computes the *system prediction* as in XCSF,

$$P(s_t, a) = \frac{\sum_{cl \in [M]|_a} cl.p(s_t) \times cl.F}{\sum_{cl \in [M]|_a} cl.F} \quad (2)$$

where cl is a classifier, $[M]|_a$ represents the subset of classifiers in [M] with action a, $cl.F$ is the fitness of cl; $cl.p(s_t)$ is the prediction of cl computed in the state s_t as $\phi(s_t)\mathbf{w}$. Next, the system selects an action to perform. The classifiers in [M] that advocate the selected action are put in the current *action set* [A]; the selected action is sent to the environment and a reward r is returned to the system together with the next input state s_{t+1}. The incoming reward r is used to update the weights vector $\mathbf{w}$, the prediction error ε and the fitness F of the classifiers in the action set [A] as done in XCSF [15].

Discovery Component. Similarly to XCSF [15], a genetic algorithm (GA) is applied to the classifiers in the current action set [A] when the average time since the last GA application to the classifiers in [A] exceeds a threshold θ_{ga}. Two offspring classifiers are generated by reproducing, crossing, and mutating the parents. The genetic algorithm can also mutate the new additional parameters r and t. During the learning process these parameters can be either fixed or they can be adapted. Accordingly, we identify four versions of XCSF with tile coding prediction:

- XCSF-C, where the genetic algorithm acts only on classifier condition (as in XCSF) while the number of tilings t and the resolution r are constant.
- XCSF-TC, where the genetic algorithm acts on classifier condition and on the number of tiling t, while the resolution r is constant.
- XCSF-RC, where the genetic algorithm acts on classifier condition and on the resolution r, while the number of tiling t is constant.
- XCSF-RTC, where the genetic algorithm acts on the classifier condition, on the number of tiling t and on the resolution r.

5 Design of Experiments

In this paper, we applied the standard experimental design used in the literature which that we also followed in our first work on XCSF with tile-coding prediction [8]. Each experiment consists of a number of problems that the system must solve. Each problem is either a *learning* problem or a *test* problem. In *learning* problems, XCSF selects actions randomly from those represented in the match set. In *test* problems, XCSF always selects the action with the highest prediction. The genetic algorithm is enabled only during *learning* while it is turned off during *testing*. The covering operator is always enabled, but operates only if needed. Learning problems and test problems alternate. The performance is computed as the average number of steps needed to reach the goal during the last 10 test problems. To speed up the experiments, problems can last at most 1000 steps; when

this limit is reached the problem stops even if the system did not reach the goal. To evaluate the learning speed and the final performance achieved by the different settings and systems, we collected also the average number of steps needed to reach the goal respectively during the first 100 test problems and during the last 100 test problems. All the statistics reported in this paper are averaged over 10 experiments.

Statistical Analysis. To analyze the results reported in this paper, we followed the procedure introduced in [9] for the comparison of performance curves. For each experiment, for every setting we tested, we considered all the performance curves; we sampled the curves and considered only one point every 100 problems; we applied an analysis of variance (ANOVA) [3] on the resulting data to test whether there was some statistically significant difference; finally, we applied four *post hoc tests* [3], Tukey HSD, Scheffé, Bonferroni, and Student-Neumann-Keuls, to find which settings/systems performed significantly different. The same analysis, i.e., the analysis of variance and the four post hoc tests, has been also applied to the average performance of each setting/system during the first 100 test problems and during the last 100 test problems.

6 Experiments

We performed a series of experiments to compare XCSF with tile coding prediction and the plain tabular tile coding on the four *MazeWorld* problems (Figures 3). In particular, we compared the four versions of XCSF we introduced in Section 4 (XCSF-C, XCSF-TC, XCSF-RC, and XCSF-RTC), which differ only in their discovery component.

6.1 Adapting Conditions

In the first set of experiments, we compared tabular tile coding and the simpler version of XCSF with tile coding in which only conditions are evolved during learning (XSCF-C). The parameters of XCSF-C were set following the typical setup used for Gridworld problems [7,8]: $N = 5000$; $\epsilon_0 = 0.05$; $\beta = 0.2$; $\alpha = 0.1$; $\gamma = 0.95$; $\nu = 5$; $\chi = 0.8$, $\mu = 0.04$, $p_{\text{explr}} = 0.1$, $\theta_{del} = 50$, $\theta_{GA} = 50$, and $\delta = 0.1$; GA-subsumption is on with $\theta_{sub} = 50$; while action-set subsumption is off; $m_0 = 0.25$, $r_0 = 0.25$ [15]; for tile coding prediction, the resolution r is 0.05 and different values of number of tilings t have been tested.

Figure 5 compares the performance of XCSF-C (solid dots) with that of tabular tile coding (empty dots) applied to `Maze1`, `Maze2`, `Maze3`, and `Maze4` with $t \in \{1, 16, 256\}$. Overall, the results shows that XCSF-C is slightly faster than tabular tile coding when using the same settings. In particular, Figure 5a shows that the performance of the systems are very close in `Maze1`, while the differences are more evident in `Maze2`, in `Maze3` and in `Maze4` (Figure 5b-d). However, XCSF-C and tabular tile coding achieve almost the same final performance at the end (with few exceptions). Table 1a and Table 1b summarize the average performance of XCSF-C and tabular tile coding in the first 100 test problems and in the last 100 test problems. In `Maze1`, a higher value for t ($t = 64, 256$) leads to a very good performance in the first 100 problems both with tabular tile coding and with

XCSF (Table 1a). In contrast, in the more challenging problems, smaller values of t ($t = 4, 16$) lead to similar or even better performance. In fact, as the complexity of the *MazeWorld* environment increases generalization is more difficult to achieve and higher values of t can lead to overgeneralization in the problem space. Similarly, Table 1b shows that in the more complex environments (`Maze3` and `Maze4`) a higher value of t can even disrupt the performance achieved by tabular tile coding in the final 100 episodes. However, in XCSF the choice of t seems to affect the final performance only slightly (Table 1b). This is not surprising, since XCSF can properly decompose the problem space through the classifiers condition so as to achieve the suitable generalization in different areas.

We performed a statistical analysis (see Section 5) to check whether there are statistically significant differences (i) between tabular tile coding and XCSF; and (ii) between the different values of t considered. The analysis suggests that, in all the problems XCSF-C performs similarly to tabular tile coding using the best settings for both the systems. However, in all the more complex problems (`Maze2`, `Maze3` and `Maze4`) the analysis shows that XCSF-C performs significantly better than tabular tile coding when we use a non optimal setting of t.

Initial Performance. The analysis on the first 100 test problems (Table 1a) shows that XCSF-C is significantly faster than tabular tile coding in all the four mazes, when the same number of tiling t is used for both the systems. The same analysis also reports significant differences with respect to the tiling number t. In `Maze1`, `Maze2`, and `Maze3`, $t = 1$ is significantly slower than $t = \{4, 16, 64, 256\}$, which instead have a similar performance. Instead, in `Maze4`, $t = 256$ is significantly slower than $t = \{1, 4, 16, 64\}$ which perform similarly.

Final Performance. The analysis of the average performance over the last 100 test problems returns unclear results with statistical differences between XCSF-C and tabular tile coding in only two mazes out of four. In `Maze1`, tabular tile coding achieves a final performance that is significantly better than the one achieved by XCSF-C, conversely, in `Maze4`, XCSF-C performs better than tabular tile coding. Concerning the number of tilings t, the analysis of final performance shows that using $t = 1$ always leads to a final performance that is significantly worse than the one achieved with $t = \{4, 16, 64, 256\}$, the only exception being tabular tile coding with $t = 256$ in `Maze4`.

6.2 Adapting Conditions and Number of Tilings

In the second set of experiments, we focused on XCSF-TC, which also adapts the number of tilings t during learning. The parameters of XCSF-TC were set as in the previous set of experiments. Figure 6 compares XCSF-TC (solid dots) to the tabular tile coding (empty dots) with $t \in \{1, 16, 256\}$ applied to the four *MazeWorld* environments. The results show that XCSF-TC learns either as fast as or even slightly faster than tabular tile coding with the best parameter setting in `Maze1`, `Maze2`, and `Maze3`. In contrast, in `Maze4` (Figure 6d), XCSF-TC seems initially slower than tabular tile coding with the best parameter setting (i.e., $t = 16$) although it quickly reaches the same performance.

To test whether the differences among the curves are statistically significant, we performed the usual statistical analysis (Section 5). The analysis suggests that

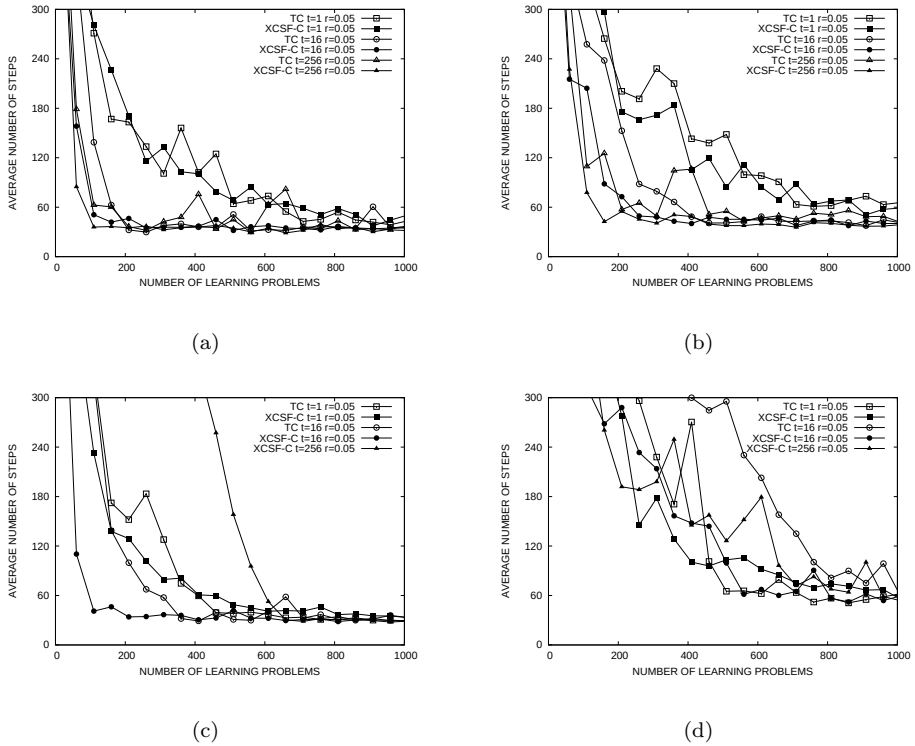

Fig. 5: Performance of XCSF-C and tabular tile coding applied to (a) `Maze1`, (b) `Maze2`, (c) `Maze3`, and (d) `Maze4`. In `Maze3` and in `Maze4` tabular tile coding with $t = 256$ is not reported in the figures due to its poor performance. Curves are averages over 10 runs.

XCSF-TC performs similarly to tabular tile coding with the best setting over the whole learning process in all the problems but `Maze3`, where XCSF-TC performs significantly better. The analysis shows that tabular tile coding with $t = 1$ performs significantly worse than the other settings in the simplest environments, `Maze1` and `Maze2`; instead, in the most complex environments, `Maze3` and `Maze4`, tabular tile coding with $t = 256$ performs significantly worse than the others.

Initial Performance. Table 2a shows that, in the first 100 test problems, XCSF-TC (i) performs slightly better than tabular tile coding in `Maze1` and `Maze2`; (ii) has a performance very close to the best setting of tabular tile coding in `Maze4`; (iii) performs slightly worse than the best settings of tabular tile coding in `Maze3`. The statistical analysis of the performance gives the following outcome. In `Maze1` and in `Maze2` the post-hoc tests [3] identifies three groups: (i) XCSF-TC that performs significantly better than all the settings of tabular tile coding, (ii) tabular tile coding with $t \in 4, 16, 64, 256$ which perform similarly, and (iii) tabular tile coding with $t = 1$ which performs significantly worse than all the other settings and than XCSF-TC. In `Maze3`, no significant differences are reported from the analysis. In

t	Maze1		Maze2	
	TC	XCSF-C	TC	XCSF-C
1	485.90 ± 42.99	440.14 ± 23.00	534.16 ± 23.92	517.78 ± 21.34
4	289.51 ± 25.63	268.80 ± 30.11	394.03 ± 33.62	330.91 ± 25.25
16	294.25 ± 47.93	176.53 ± 23.32	425.29 ± 67.13	285.50 ± 50.63
64	232.84 ± 42.55	135.98 ± 26.48	441.62 ± 76.16	308.98 ± 93.44
256	259.00 ± 47.53	160.34 ± 49.16	388.73 ± 52.68	292.46 ± 51.56
	Maze3		Maze4	
	TC	XCSF-C	TC	XCSF-C
1	515.28 ± 38.95	460.61 ± 29.11	654.03 ± 33.56	581.15 ± 30.74
4	375.49 ± 45.55	255.79 ± 19.16	547.53 ± 52.73	459.59 ± 43.43
16	334.77 ± 67.14	179.20 ± 31.70	596.10 ± 53.32	429.59 ± 68.61
64	510.96 ± 31.00	479.09 ± 46.07	708.06 ± 48.95	617.58 ± 99.67
256	655.60 ± 41.35	573.63 ± 40.64	668.36 ± 101.56	606.86 ± 68.15

(a)

t	Maze1		Maze2	
	TC	XCSF-C	TC	XCSF-C
1	41.81 ± 3.57	50.61 ± 4.53	59.83 ± 14.73	55.61 ± 5.62
4	33.94 ± 2.01	32.91 ± 2.00	40.84 ± 3.03	40.47 ± 2.33
16	38.38 ± 12.06	33.63 ± 1.52	42.73 ± 2.36	39.71 ± 2.57
64	33.50 ± 2.43	33.39 ± 1.79	41.71 ± 3.15	40.00 ± 2.16
256	34.46 ± 2.86	32.40 ± 2.10	47.01 ± 9.34	38.89 ± 2.80
	Maze3		Maze4	
	TC	XCSF-C	TC	XCSF-C
1	30.23 ± 3.08	34.26 ± 2.19	57.38 ± 3.95	66.07 ± 5.74
4	28.63 ± 1.53	29.36 ± 1.45	57.79 ± 4.22	56.13 ± 3.41
16	33.93 ± 3.13	29.61 ± 2.29	82.07 ± 38.91	57.77 ± 4.12
64	36.13 ± 6.43	30.24 ± 1.55	131.03 ± 53.90	63.06 ± 7.53
256	114.64 ± 56.71	29.78 ± 2.06	125.73 ± 56.05	71.10 ± 23.98

(b)

Table 1: XCSF-C and tabular tile coding (TC) applied to `Maze1`, `Maze2`, `Maze3`, and `Maze4`: (a) average performance in the first 100 test problems and (b) average performance in the last 100 test problems. Statistics are averages over 10 runs.

`Maze4`, post-hoc tests identifies three groups: (i) tabular tile coding with $t = 4$ and $t = 16$, that perform better than all the other settings; (ii) then, XCSF-TC and tabular tile coding with $t = 1$ and $t = 64$; (iii) tabular tile coding with $t = 256$, that performs worse than all the other settings.

Final Performance. Table 2b shows that XCSF-TC always reaches a performance similar or better than the one achieved by tabular tile coding with the best setting (the only exception being `Maze4` where the final performance is slightly worse than the one achieved by tabular tile coding with $t = 1$ and $t = 4$). We performed an one-way ANOVA [3] to test whether these differences are statistically significant. In `Maze1`, `Maze2`, and `Maze3` the analysis does not report any statistically significant difference: XCSF-TC and tabular tile coding reaches a similar final performance for all the settings. In `Maze4` tabular tile coding with $t = 256$ achieves a final performance significantly worse than XCSF-TC and tabular tile coding with the other settings.

Finally, Figure 7 shows the average number of tiling t of the classifier evolved by XCSF-TC during the learning process in all the four environments. The more

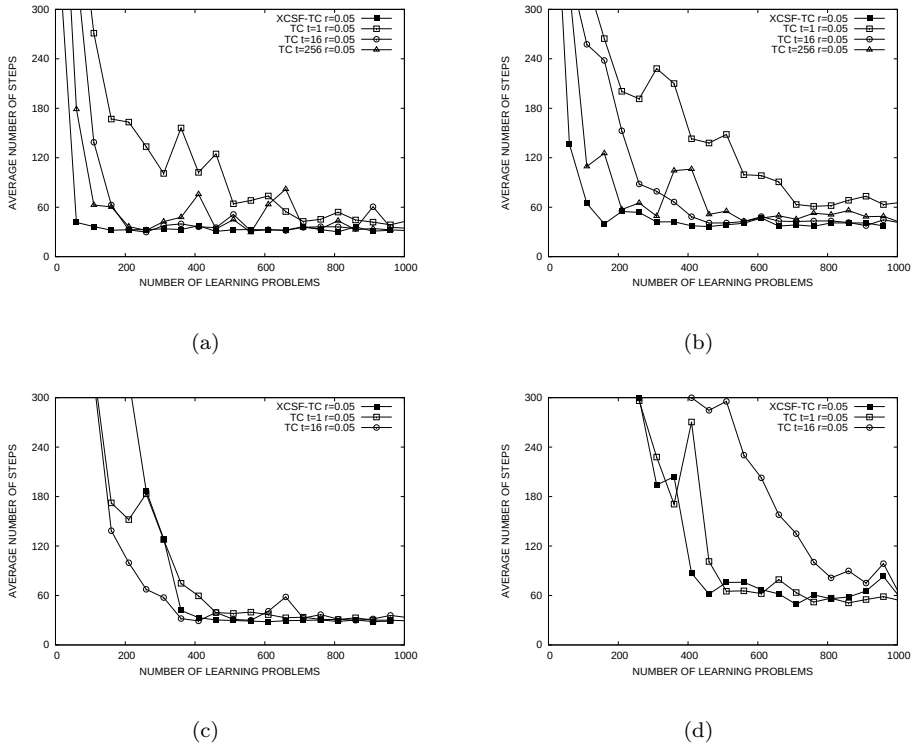

Fig. 6: Performance of XCSF-TC and tabular tile coding (TC) applied to (a) `Maze1`, (b) `Maze2`, (c) `Maze3`, and (d) `Maze4`. In `Maze3` and in `Maze4` tabular tile coding with $t = 256$ is not reported in the figures due to its poor performance. Curves are averages over 10 runs.

complex is the maze the smaller is the number of tiling t evolved; in fact a high number of tiling leads to a broad generalization that is useful only in the simplest environments.

6.3 Adapting Conditions and Tiling Resolution

In the third set of experiments, we tested XCSF-RC which adapts the resolution of the tiling r during the learning process. The parameters of XCSF-RC were set as in the previous experiments except for number of tiling t that was set to 16 for tabular tile coding as well as for XCSF-RC.

Figure 8 compares XCSF-RC (solid dots) to the tabular tile coding (empty dots) with $r \in \{0.0125, 0.025, 0.05\}$ in the five *MazeWorld* environments (tabular tile coding with values of r bigger than 0.05 are not reported due to the poor performances). The figure shows that XCSF-RC learns generally faster than tabular tile coding with the best parameter setting in all the problem except in `Maze3`,

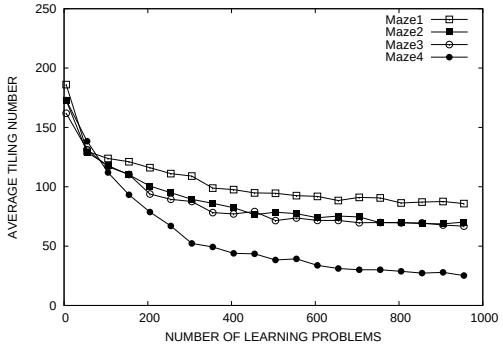

Fig. 7: Average number of tilings evolved by XCSF-TC in `Maze1`, `Maze2`, `Maze3`, and `Maze4`. Curves are averages over 10 runs.

System	`Maze1`	`Maze2`	`Maze3`	`Maze4`
TC $t = 1$	485.90 ± 42.99	534.16 ± 23.92	515.28 ± 38.95	654.03 ± 33.56
TC $t = 4$	289.51 ± 25.63	394.03 ± 33.62	375.49 ± 45.55	547.53 ± 52.73
TC $t = 16$	294.25 ± 47.93	425.29 ± 67.13	334.77 ± 67.14	596.10 ± 53.32
TC $t = 64$	232.84 ± 42.55	441.62 ± 76.16	510.96 ± 31.00	708.06 ± 48.95
TC $t = 256$	259.00 ± 47.53	388.73 ± 52.68	655.60 ± 41.35	668.36 ± 101.56
XCSF-TC	124.16 ± 24.82	211.92 ± 43.65	491.28 ± 59.67	594.08 ± 73.88

(a)

System	`Maze1`	`Maze2`	`Maze3`	`Maze4`
TC $t = 1$	41.81 ± 3.57	59.83 ± 14.73	30.23 ± 3.08	57.38 ± 3.95
TC $t = 4$	33.94 ± 2.01	40.84 ± 3.03	28.63 ± 1.53	57.79 ± 4.22
TC $t = 16$	38.38 ± 12.06	42.73 ± 2.36	33.93 ± 3.13	82.07 ± 38.91
TC $t = 64$	33.50 ± 2.43	41.71 ± 3.15	36.13 ± 6.43	131.03 ± 53.90
TC $t = 256$	34.46 ± 2.86	47.01 ± 9.34	114.64 ± 56.71	125.73 ± 56.05
XCSF-TC	34.69 ± 1.37	38.96 ± 2.20	28.82 ± 1.67	64.95 ± 11.67

(b)

Table 2: XCSF-TC and tabular tile coding (TC) applied to `Maze1`, `Maze2`, `Maze3`, and `Maze4`: (a) average performance in the first 100 test problems and (b) average performance in the last 100 test problems. Statistics are averages over 10 runs.

where it learns slower than tabular tile coding, although it quickly reaches a similar performance.

The statistical analysis of the learning curves suggest that some of these differences are statistically significant. In particular, in `Maze1` tabular tile coding with $r = 0.0125$ performs significantly worse than the other systems/settings which perform similarly; in `Maze2` and in `Maze4` the post-hoc tests identify three different groups: (i) XCSF-RC, that performs better than tabular tile coding with any setting, (ii) then, tabular tile coding with $r = 0.025$ and $r = 0.05$, and (iii) tabular tile coding with $r = 0.0125$ that performs worse than all the others; in `Maze3` the analysis shows that tabular tile coding with $r = 0.025$ and $r = 0.05$ both performs

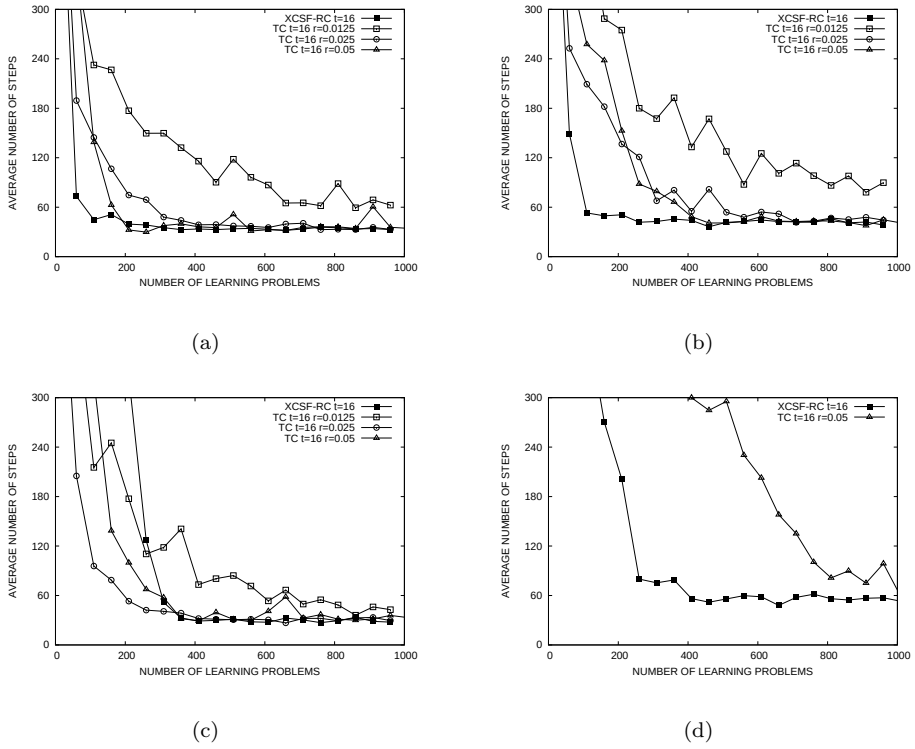

Fig. 8: XCSF-RC and tabular tile coding applied to (a) `Maze1`, (b) `Maze2`, (c) `Maze3`, and (d) `Maze4`. In `Maze4` tabular tile coding with $r = 0.025$ and $r = 0.0125$ are not reported in the figure due to their poor performances. Curves are averages over 10 runs.

significantly better than tabular tile coding with $r = 0.0125$ and XCSF-RC, that instead performs similarly.

Initial Performance. Table 3a reports the average performance of the systems during the first 100 test problems. $r = 0.05$ and $r = 0.025$ are the best settings for tabular tile coding in all the four environments. In particular, in the most complex problem, `Maze4`, $r = 0.025$ allows to tabular tile coding to learn rather faster than the other settings, as this environment does not allow a broad generalization in the problem space. Similarly, XCSF-RC exploits at best the possibility of adapting the tiling resolution in all the environments but `Maze3`, where a coarser resolution (e.g., tabular tile coding with $r = 0.1$ and with $r = 0.2$) appears to be more disruptive. The statistical analysis of the data in Table 3a shows that almost all these differences are statistically significant. In particular, in `Maze1` and in `Maze2`, XCSF-RC requires significantly fewer episodes than tabular tile coding with the best settings; in `Maze3`, tabular tile coding with the best settings ($r = 0.05$ and $r = 0.025$) requires significantly fewer episodes than XCSF-RC; in contrast,

System	Maze1	Maze2	Maze3	Maze4
TC $r = 0.0125$	423.18 ± 29.74	508.64 ± 15.50	455.20 ± 15.05	615.91 ± 21.74
TC $r = 0.025$	249.50 ± 30.32	328.88 ± 18.26	244.67 ± 18.50	479.43 ± 21.47
TC $r = 0.05$	243.26 ± 47.57	406.51 ± 64.94	284.26 ± 82.18	602.71 ± 83.44
TC $r = 0.1$	610.86 ± 111.04	667.23 ± 81.42	723.48 ± 103.32	881.85 ± 25.70
TC $r = 0.2$	643.77 ± 51.88	736.37 ± 51.85	956.40 ± 40.42	873.38 ± 31.55
XCSF-RC	150.94 ± 26.66	224.20 ± 69.40	613.58 ± 86.85	584.26 ± 96.30

(a)

System	Maze1	Maze2	Maze3	Maze4
TC $r = 0.0125$	58.21 ± 5.44	89.59 ± 19.20	39.47 ± 3.17	397.90 ± 29.31
TC $r = 0.025$	33.07 ± 1.37	45.82 ± 4.69	30.58 ± 1.47	270.38 ± 37.68
TC $r = 0.05$	34.36 ± 1.17	41.10 ± 2.69	35.81 ± 5.39	77.38 ± 28.21
TC $r = 0.1$	350.11 ± 108.41	341.47 ± 161.55	517.91 ± 86.51	863.39 ± 42.52
TC $r = 0.2$	624.82 ± 61.21	727.10 ± 45.59	923.87 ± 25.84	850.81 ± 44.71
XCSF-RC	33.59 ± 1.55	38.55 ± 2.88	29.07 ± 1.84	55.16 ± 3.40

(b)

Table 3: XCSF-RC and tabular tile coding (TC) applied to Maze1, Maze2, Maze3, and Maze4: (a) average performance in the first 100 test problems and (b) average performance in the last 100 test problems. Statistics are averages over 10 runs.

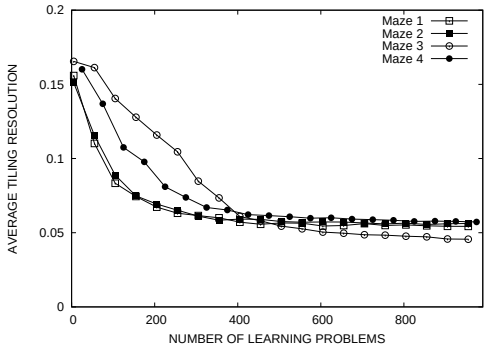

Fig. 9: Average tiling resolution evolved by XCSF-RC in Maze1, Maze2, Maze3, and Maze4. Curves are averages over 10 runs.

in Maze4, the differences between XCSF-RC and tabular tile coding (with $r \in \{0.0125, 0.025, 0.05\}$) are not statistically significant.

Final Performance. Table 3b reports the average performance during the last 100 test problems. The data show that XCSF-RC always reach a performance that is similar (Maze1 and Maze3) or slightly better (Maze2 and Maze4) than the one achieved by tabular tile coding with the best settings; in Maze2 and Maze4 the difference is also statistically significant.

Finally, Figure 9 shows the average tiling resolution r evolved by XCSF-RC over the whole learning process. The results show that the average tiling resolu-

tion decreases over time during the learning process when it reaches a value close to $r = 0.05$ in all the four environments, being just slightly lower in `Maze3`. This *emergent* behavior of XCSF-RC follows the same strategy suggested by Sherstov and Stone in [10]: a larger tiling resolution can be exploited in the early stages to improve the learning speed and then it should be reduced to improve the final performance. It is interesting to note that in XCSF-RC the time required to decrease the average tiling resolution increases in the more complex problems. This suggests that, despite adapting the tiling resolution turns out to be very powerful to improve the learning speed in simple problems, it might actually slow down the learning process when the problem is more challenging. Nevertheless, the same results suggest that XCSF-RC is actually able to adapt the tiling resolution even in the more complex problems effectively without following any specific strategy.

6.4 Adapting All the Parameters at Once

At the end, we tested XCSF-RTC which adapts both the tiling number t and the tiling resolution r. Figure 10 compares XCSF-RTC (solid dots) to XCSF-TC (empty dots) and XCSF-RC (empty boxes) on the same four environments. The results show that XCSF-RTC performs similarly to the other systems in `Maze1`, `Maze2`, and `Maze4`, while it seems to perform worse in `Maze3`. However, all the three versions of XCSF achieve a similar final performance in all the environments. The statistical analysis of the learning curves confirms that the only significant difference is between XCSF-RTC and the other two systems in `Maze3`. This confirms what found in previous experiments: a non optimal setting for the tiling number t and for the tiling resolution r might be rather disruptive in term of performance in some environments (like `Maze3`); accordingly, XCSF-RTC requires a rather large number of learning problems to properly adapt both r and t.

Initial and Final Performance. Table 4 reports the average performance of XCSF-RTC, XCSF-TC, and XCSF-RC in the first and in the last 100 test problems (Table 4a and Table 4b). Both the initial and the final performance of the three versions of XCSF are similar. The statistical analysis reported only one significant difference: XCSF-TC learns significantly faster than XCSF-RTC and than XCSF-RC in `Maze3` (see Table 4a). These results further confirm that the choice of tiling resolution is very critical especially in the `Maze3` environment.

7 Conclusions

We compared XCSF with tile coding prediction and tabular tile coding using a new set of problems (called *MazeWorlds*) designed to challenge tile coding by introducing discontinuities in the action value landscape. *MazeWorlds* extend well-known 2D gridworlds [1] by adding thin impenetrable obstacles, that break broad generalizations in the action-value landscape. Our goal was to investigate whether XCSF could exploit its effective problem decomposition capabilities and provide a more reliable performance than tabular tile coding over such more complex action-value landscapes.

In [8], we considered problems that could be solved optimally by tabular tile coding and XCSF adapted an ensemble of tile coding parameters locally while

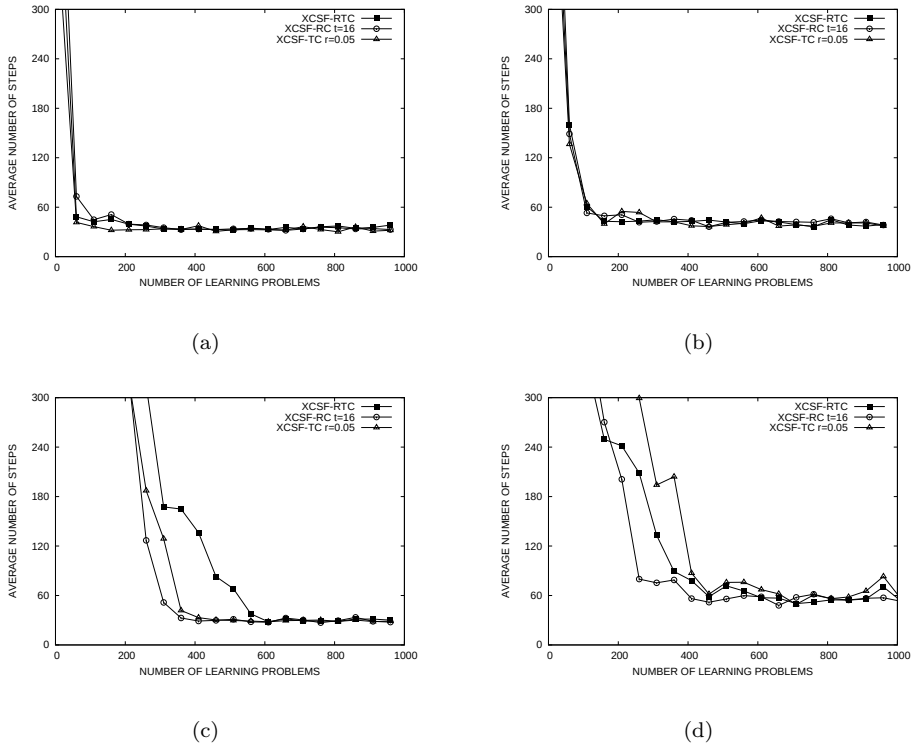

Fig. 10: XCSF-RC, XCSF-TC, and XCSF-RTC applied to (a) `Maze1`, (b) `Maze2`, (c) `Maze3`, and (d) `Maze4`. Curves are averages over 10 runs.

System	`Maze1`	`Maze2`	`Maze3`	`Maze4`
XCSF-TC	124.16 ± 24.82	211.92 ± 43.65	491.28 ± 59.67	594.08 ± 73.88
XCSF-RC	150.94 ± 26.66	224.20 ± 69.40	613.58 ± 86.85	584.26 ± 96.30
XCSF-RTC	140.58 ± 44.87	255.41 ± 51.15	610.90 ± 138.42	496.33 ± 56.83

(a)

System	`Maze1`	`Maze2`	`Maze3`	`Maze4`
XCSF-TC	34.69 ± 1.37	38.96 ± 2.20	28.82 ± 1.67	64.95 ± 11.67
XCSF-RC	33.59 ± 1.55	38.55 ± 2.88	29.07 ± 1.84	55.16 ± 3.40
XCSF-RTC	34.46 ± 1.58	38.33 ± 1.39	29.42 ± 2.22	58.70 ± 4.43

(b)

Table 4: XCSF-RC, XCSF-TC, and XCSF-RTC applied to `Maze1`, `Maze2`, `Maze3`, and `Maze4`: (a) average performance in the first 100 test problems and (b) average performance in the last 100 test problems. Statistics are averages over 10 runs.

also learning how to solve the problem. Accordingly, XCSF with tile coding could perform almost as well as tabular tile coding (both in terms of learning speed and final performance) without requiring prior knowledge of the problem domain which was instead needed to reach optimal performance using tabular tile coding. In this paper, we tackled problems that tabular tile coding may not solve because of the discontinuities in the prediction landscape.

We compared four versions of XCSF with tile coding prediction (XCSF-C, XCSF-TC, XCSF-RC, and XCSF-RTC) to tabular tile coding using four *Maze-World* environments. Each version of XCSF involved a different flavor of parameter adaptation: in XCSF-C the genetic algorithm was only applied to classifiers' conditions; in XCSF-TC and in XCSF-RC evolution also acted on the number of tiling t and the tiling resolution r, respectively; in XCSF-RTC the genetic algorithm was applied on everything at once. Our results suggest that, in terms of number of learning problems, XCSF with tile coding requires significantly less training problems than tabular tile coding in almost all the problems. Moreover, the evolution of the tile coding parameters speeds up learning and does not introduce an overhead. When the adaptation of tiling parameters (t and r) was turned off and the parameters were set to inadequate values (which would make tile coding fail) XCSF outperformed tabular tile coding and it was still capable to learn a better solution by properly partitioning the problem space using classifier conditions. Finally, an analyses we performed on the evolution of the tiling parameters showed XCSF was able to evolve the most adequate parameters in each subproblem and it was always able to learn an optimal solution. Overall, our results confirm that ability of effectively decompose the problem space is a major feature of the learning classifier systems inspired to XCS which should be exploited even more.

All the experiments were performed with a rather basic version of the genetic algorithm and much of what is known about tile coding [10] might be used to implement heuristics to speed up learning in XCSF with tile coding. For example, we initialized the population completely at random, but a heuristic could be added to set better initial values of t and r; similarly, the mutation operator could be modified to increase the likelihood of producing values that encourage generalisation decreases over the course of learning, for instance taking Sherstov and Stone's [10] adaptive method as inspiration.

Acknowledgments

The authors wish to thank the reviewers for their invaluable comments and suggestions regarding possible extensions of the approach using a more competent genetic algorithm.

References

1. Justin A. Boyan and Andrew W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 369–376, Cambridge, MA, 1995. The MIT Press.
2. Justin A. Boyan and Andrew W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors,

Advances in Neural Information Processing Systems 7, pages 369–376, Cambridge, MA, 1995. The MIT Press.
3. S. A. Glantz and B. K. Slinker. *Primer of Applied Regression & Analysis of Variance*. McGraw Hill, 2001. second edition.
4. Robert Glaubius and William D. Smart. Manifold representations for value-function approximation. In Daniela Pucci de Farias, Shie Mannor, Doina Precup, and Georgios Theocharous, editors, *Learning and Planning in Markov Processes — Advances and Challenges: Papers from the 2004 AAAI Workshop*, pages 13–18, June 2004. Available in AAAI Technical Report WS-04-08.
5. Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, and David E. Goldberg. Extending XCSF beyond linear approximation. In *Genetic and Evolutionary Computation – GECCO-2005*, pages 1859–1866, Washington DC, USA, 2005. ACM Press.
6. Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, and David E. Goldberg. XCS with computed prediction for the learning of boolean functions. In *Proceedings of the IEEE Congress on Evolutionary Computation – CEC-2005*, pages 588–595, Edinburgh, UK, September 2005. IEEE.
7. Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, and David E. Goldberg. Xcs with computed prediction in multistep environments. In *Genetic and Evolutionary Computation – GECCO-2005*, pages 1827–1834, Washington DC, USA, 2005. ACM Press.
8. Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, and David E. Goldberg. Classifier prediction based on tile coding. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1497–1504, New York, NY, USA, 2006. ACM Press.
9. Justus H. Piater, Paul R. Cohen, Xiaoqin Zhang, and Michael Atighetchi. A Randomized ANOVA Procedure for Comparing Performance Curves. In *Machine Learning: Proceedings of the Fifteenth International Conference (ICML)*, pages 430–438, Madison, Wisconsin, July 1998. Morgan Kaufmann, San Mateo, CA, USA.
10. Alexander A. Sherstov and Peter Stone. Function approximation via tile coding: Automating parameter choice. In *Proc. Symposium on Abstraction, Reformulation, and Approximation (SARA-05)*, Edinburgh, Scotland, UK, 2005.
11. Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*, pages 1038–1044. The MIT Press, Cambridge, MA., 1996.
12. Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning – An Introduction*. MIT Press, 1998.
13. Gerald Tesauro. TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.
14. Stewart W. Wilson. Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.
15. Stewart W. Wilson. Classifiers that approximate functions. *Natural Computing*, 1(2-3):211–234, 2002.