Multiagent Reinforcement Learning Using Function Approximation

Osman Abul, Faruk Polat, and Reda Alhajj

***Abstract*—Learning in a partially observable and nonstationary environment is still one of the challenging problems in the area of multiagent (MA) learning. Reinforcement learning is a generic method that suits the needs of MA learning in many aspects. This paper presents two new multiagent based domain independent coordination mechanisms for reinforcement learning; multiple agents do not require explicit communication among themselves to learn coordinated behavior. The first coordination mechanism is *perceptual coordination mechanism*, where other agents are included in state descriptions and coordination information is learned from state transitions. The second is *observing coordination mechanism*, which also includes other agents in state descriptions and additionally the rewards of nearby agents are observed from the environment. The observed rewards and agent's own reward are used to construct an optimal policy. This way, the latter mechanism tends to increase region-wide joint rewards. The selected experimented domain is *adversarial food-collecting world* (AFCW), which can be configured both as single and multiagent environments. Function approximation and generalization techniques are used because of the huge state space. Experimental results show the effectiveness of these mechanisms.**

***Index Terms*—Adaptive behavior, multiagent learning, reinforcement learning.**

I. Introduction

MULTIAGENT systems form a particular type of distributed artificial intelligence systems. In fact, many real-world problems such as engineering design, intelligent search, medical diagnosis, robotics, etc require multiple agents [1]. Multiagent systems are different from single agent systems in the sense that there is no global control and globally consistent knowledge. In multiagent systems, data and control are distributed. This way, limitations on the processing power of a single agent are eliminated. Distribution also brings up inherent advantages of distributed systems, such as scalability, fault-tolerance, parallelism, etc.

Multiagent systems have been successfully utilized for *reinforcement learning* (RL), which is a learning technique that requires almost nothing about the dynamics of the environment to learn about. It is based on learning from trial and error as in the early ages of people. An agent with its goal embedded in an environment learns how to transform any environmental state into another that contains its goal. An agent that has the ability of doing this with minimal human supervision is called *autonomous* [2]. Autonomous agents learn from their environment by receiving reinforcement signals after interacting with the environment. Learning from an environment is robust because agents are directly affected from the dynamics of the environment.

How agents acquire and maintain knowledge is an important issue in RL. When the state space of the task is small and discrete, the lookup table method is generally preferred. However, in case that the state space is huge or continuous, lookup tables do not scale up or are inappropriate. Function approximation and generalization methods seem to be more feasible solutions. Several researchers have studied these methods and algorithms for lookup tables have been modified to suit them [3], [4]. Unfortunately, optimal convergence of function approximation implementation of RL algorithms has not been proven yet [5].

This paper presents two new domain independent coordination mechanisms. The selected experimented domain is *adversarial food-collecting world* (AFCW), which can be configured both as single and multiagent environments. AFCW is a survival task for agents in a hostile environment. In order to survive, agents need to learn collecting food pieces and escaping from their adversaries. In multiagent settings of AFCW, there are cases that require cooperation among agents. From this perspective, multiagent AFCW is a cooperative learning task. The first coordination mechanism presented in this paper is *perceptual coordination mechanism* (PCM) and the second is *observing coordination mechanism* (OCM). In PCM, other agents are included in state descriptions and coordination information is learned from state transitions. The second mechanism, OCM, also includes other agents in state descriptions and additionally the rewards of nearby agents are observed from the environment. The observed rewards and agent's own reward are used to construct an optimal policy. This way, OCM tends to increase region-wide joint rewards. Experimental results show the effectiveness of these methods.

We have developed various learning agents for AFCW. All our agents use multilayer feed-forward perceptron (MLP) based function approximation. MLP based agents learn using connectionist reinforcement learning algorithms: $Q(0)$, Q-learning with eligibility traces, Sarsa with eligibility traces, and $Q(\lambda)$ [6]–[8]. These algorithms are implemented on two different MLP architectures, single network MLP and multiple network MLP. These architectures determine how the utility of actions is represented. Domain knowledge from AFCW is exploited and agents using this knowledge are implemented.

The rest of the paper is organized as follows. The function approximation and generalization methods used in this work are

Manuscript received January 1, 2000.

O. Abul is with the Department of Computer Engineering, Middle East Technical University, Ankara, Turkey.

F. Polat is with the Department of Computer Engineering, Middle East Technical University, Ankara, Turkey (e-mail: polat@ceng.metu.edu.tr).

R. Alhajj is with the Department of Math and Computer Science, American University of Sharjah, Sharjah, United Arab Emirates (e-mail: ralhajj@aus.ac.ae).

Publisher Item Identifier S 1094-6977(00)11209-X.

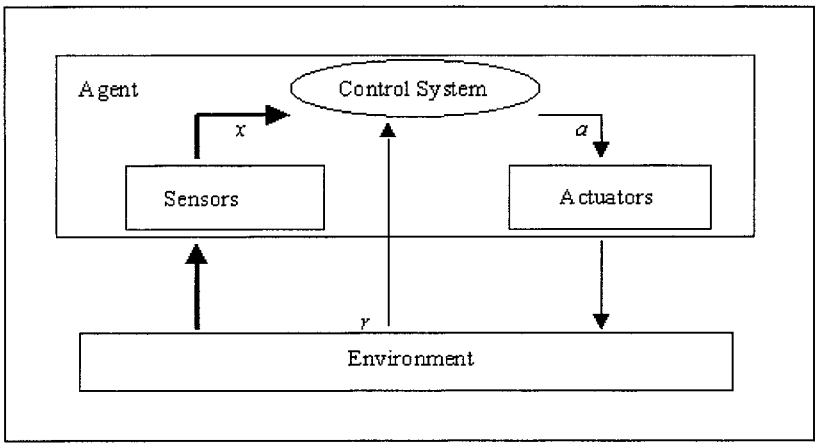

Fig. 1. Reinforcement learning framework.

introduced in Section II, after a review of the theory of RL and the basics of implemented RL algorithms. A simulated multidimensional dynamic environment, AFCW is presented in Section III, to be used as a platform for experiments throughout this study. The proposed action coordination methods are described in Section IV. The experiments and the results achieved for both single and multiagent AFCWs are given in Section V. Conclusions and future research directions are included in Section VI.

II. Reinforcement Learning

RL requires learning from interactions in an environment in order to achieve certain goals [5], [9], [10]. The entity interacting with its environment by actions is called *agent* (or controller). At each time step, an agent observes its environment and selects the next actions based on that observation. In the next time step, the agent obtains the new observation that may reflect the effects of its previous action and a *payoff* value indicating the quality of the selected action. Shown in Fig. 1 is the generic RL framework, which consists of an environment and an agent.

An agent is composed of three main components, namely sensors, actuators and the control system. An agent observes the environment through its sensors and presents an indication of that (denoted x) to the control system. The control system in turn decides on the action to be taken (denoted a) and reports that action to its actuators. This action changes the environment. Upon this change, the environment reports a reward or punishment value (denoted r) back to the agent. Based on this cycle, the control system tries to learn optimal action selection (optimal policy) in the agent's environment. The dynamics of the environment is called its model. If the model of the environment is known *a priori*, then producing *off-line* solutions is possible as in dynamic programming. Most of the research in AI focused on this assumption. Later, researchers developed methods that do not require a model to act in an environment. *Indirect adaptive control* methods incrementally construct a model and then use the model to design solutions. On the other hand, *direct adaptive control* methods do not estimate any model to design a solution. Actually, the direct adaptive control is the main motivation behind RL.

Markov decision process (MDP) is the most widely accepted mathematical model of RL; also called *multistage decision task* [9], [11], [12]. An MDP is a quadruple of $\langle X, A, T, R\rangle$, where X is a finite set of all possible states of the environment, A is a finite set of all possible actions in each state, T is a function from state-action pairs to states, $T : X \times A \rightarrow X$, and R is an expected value of reward (reinforcement) function from state-action pairs to numeric values, $R : E\{X \times A \rightarrow \Re\}$.

In MDP, agents are assumed to observe full description of the states sufficient to make optimal decisions. Usually, this is not the case in real life because there may be hidden knowledge, insufficient sensors, etc. In such cases, agents either try to extract the exact state of the environment using the current observation and past information, or they use the current observation as the state of the environment. Research in *partially observable* MDPs (POMDPs) addresses this problem [13], [14]. In POMDPs, agents use *belief states* to resolve the state from the current observation. Theoretically, belief states give enough statistics about the past [13].

Dynamic programming (DP) methods require complete environment dynamics to solve MDPs. If dynamics of the environment are not known *a priori*, one can use Monte Carlo (MC) methods to solve MDPs. MC methods [15] require only experiences (sample state transitions and rewards). Policy evaluation and improvement in DP and MC are similar, i.e., the current policy is evaluated for visited state-action pairs and improved for visited states. DP methods update their estimates based on other estimates. Unlike DP methods, MC methods update their estimates based on experiences. Temporal Difference (TD) methods, on the other hand, combine these two approaches, i.e., use other estimates and experiences to update their estimates. The two mostly commonly used and understood TD methods are Q-learning and Sarsa.

Q-learning algorithm [8] directly computes the approximation of optimal action-value function independent of the policy followed. It is off-policy and its convergence is proven under the assumption that each state-action pair is visited infinitely often. It has the following update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

where

a	learning factor;
γ	discount factor;
r_t	reward at time t;
$Q(s_t, a_t)$	value of acting by a in state s at time t.

The Sarsa algorithm [6] is *on-policy* version of the *Q*-Learning, where the *max* operator is dropped from the update equation and action value of the next state-action is used. The update equation of *Sarsa* is

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

The *Sarsa* algorithm is on-policy because current backup depends on the next action selected. Note the difference from Q-learning is that the next action selected from the next state is also required in the update. The two algorithms Q-learning and Sarsa update the value of previous state-action pair by using current TD-error. On the other hand, MC methods update state values based on entire episode. $\mathrm{TD}(\lambda)$ methods $(0 \leq \lambda \leq 1)$ are compromise between them. In other words, 1-step methods and MC methods are specialized versions of $\mathrm{TD}(\lambda)$ when $\lambda = 0$ and $\lambda = 1$, respectively. In $\mathrm{TD}(\lambda)$, the value of λ adjusts the weighted value of current TD propagated to all past states or state-action pairs depending on the method used. The intuition behind this idea is that the current TD does not depend only on previous state and previous action, but also on the past states and actions. Note that $\mathrm{TD}(\lambda)$ methods solve the credit assignment problem by back-propagating TD errors to all state-action pairs in an episode.

In the naive implementation of $\mathrm{TD}(\lambda)$, all past state-action pairs are stored and when a TD error occurs, all values of the past state-action pairs are updated by weighting this TD error. This implementation is impractical when episodes are very long. The *eligibility traces* (ET) implementation, on the other hand, does not require storing of all past state-action pairs. It is an incremental technique and can be also used with nonepisodic tasks. In ET implementation of Sarsa, an eligibility value for each state-action pair is stored. Before the episode starts, all the eligibility values are set to zero and, while trials continue, they are decayed by $\lambda\gamma$. The eligibility trace value of the current state-action pair is incremented by 1. Then all Q-values of the state-action pairs are updated using their own eligibility trace value and the current TD. However, $\mathrm{TD}(\lambda)$ method is not directly applicable to Q-learning because it is off-line.

Finally, one can expect from $\mathrm{TD}(\lambda)$ methods faster convergence due to the propagation of current TD to all past state-action pairs. $\mathrm{TD}(\lambda)$ methods, unfortunately, require all Q-values of state-action pairs updated in every step. Cichosz [17] presented a solution for this problem called *truncated temporal difference* (TTD) method, which stores fixed state-action pairs (fixed window to the past). At each time step, the value of the least recent entry is updated and the window is shifted. This way, great computational saving is achieved.

A. Function Approximation

So far, we have assumed that state and state-action values are stored in a look-up table. The main drawback of look-up tables is their scaling problem. In case we have a task that has huge state space, it is unlikely to store all states in a limited memory and to visit each state in reasonable time. The solution is to generalize visited states to unvisited ones as in supervised learning. This is known as the *generalization of states*. We can also represent state or state-action utilities by functions instead of look-up tables. This is called *function approximation*. Function approximation and generalization become crucial parts of a learning system when state-space is too large to be stored in look-up tables. Function approximation and generalization methods are well studied by the RL community [3], [4], [6], [16], [17].

Neural networks [18], [19] are widely used in function approximation and generalization. The Q-learning, Sarsa, adaptive heuristic critic (AHC), and other RL methods can be implemented using MLPs. Q-learning and Sarsa have very similar implementation when look-up tables and MLPs are used. The update mechanism of the back-propagation algorithm is based on updating weights in the direction of error-gradients. Error-gradients also determine the size of the update as well as the direction. However, the weights in MLP based RL are updated using *output gradients* of the current input. The output gradients determine the direction of the weight update for each individual weight. The size of the update is determined by the temporal prediction difference. The output gradients can be computed similar to error-gradients, i.e., by partial derivative of output terms instead of error terms on weights.

In general, there are two alternative methods to update the weights, namely *off-line* and *on-line*. In the off-line method, sequential experiences are stored during a trial. After the trial ends, they are presented to the network in temporally backward order, hence called *backward replay*. Clearly, this approach only applies to episodic tasks and does not apply to continual tasks. This approach may require a great deal of memory to store experiences when the length of the trial is very long. On the other hand, on-line methods do not have any of the problems mentioned above. On-line methods learn from an individual experience and forget about it. Because an episode of AFCW can be long, we have adopted the use of eligibility trace mechanism, which is an on-line method.

The eligibility trace mechanism can also be applied to MLP based function approximation. This makes the $\mathrm{TD}(\lambda)$ algorithms also applicable for general λ. In the eligibility traces implementation, each weight needs an associated eligibility trace value. So, the storage required for eligibility traces is two-fold. Rummery [6] discussed the eligibility traces implementation of Q-learning, Sarsa and $Q(\lambda)$. He noted that when multiple predictions are used to update weights, each weight would require multiple eligibility value, one per prediction. However, in Q-learning and Sarsa only one prediction is used at each time step. So, 1 eligibility per weight is sufficient. He also noted that Q-learning does not add up TD-errors correctly unless greedy action is taken in each time step. To apply eligibility traces mechanism to Q-learning, we should reset all eligibilities when an exploratory (nongreedy) action is selected. The two algorithms Q-learning with $\mathrm{TD}(\lambda)$ are combined in $Q(\lambda)$, which does not need setting to zero eligibilities when nongreedy actions are taken.

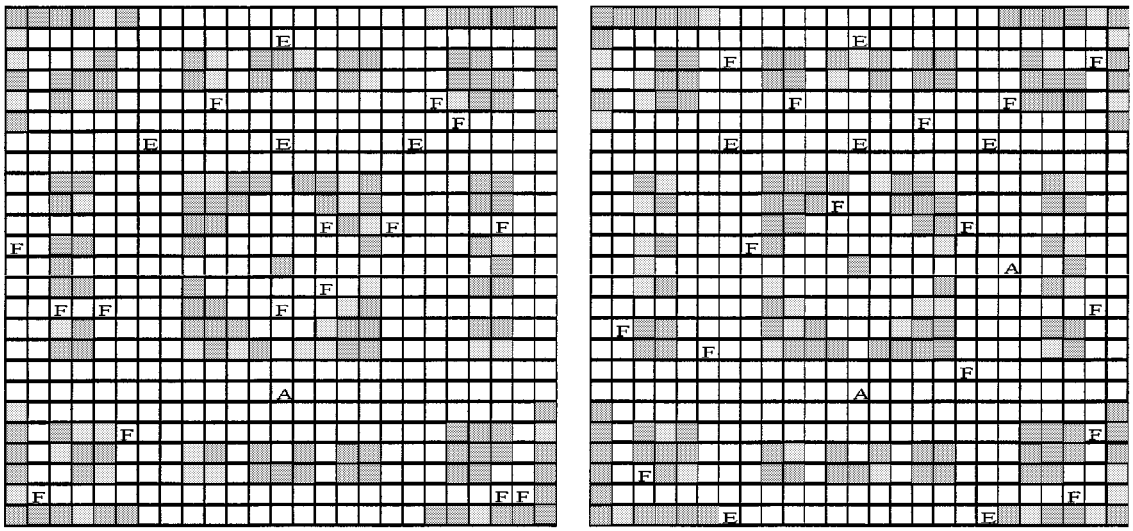

Fig. 2. Sample single-agent (left) and multiagent (right) AFCW environments.

III. AFCW Environment

We have used AFCW[1] as a domain for our study. AFCW is a grid based survival environment (25×25). It contains four kinds of entities, namely *agents*, *enemies*, *food pieces* and *obstacles*. In this environment, agents are learning entities. They try to collect food pieces while escaping from their adversaries (enemies), On the other hand, enemies try to catch and kill agents. Obstacles are employed to disturb the homogeneity of the environment. Agents and enemies are active (mobile) entities, while food pieces and obstacles are passive (stationary) entities.

The AFCW task is multiagent based because there are more than one decision making entities. In his work, Lin [16] fixed the number of agents to one and the number of enemies to four. His work treats enemies as intelligent, but nonlearning entities. In other words, enemies have fixed (nonadaptive) behaviors. This treatment makes the task a single-agent task because enemies can be modeled as a part of the environment. In our work, we have treated enemies in the same way, but allow the environment to have more than one learning agent. Thus, AFCW is categorized as a multiagent learning task. Obviously, AFCW covers Lin's simulated dynamic environment as a special case.

Shown in Fig. 2 are examples for single and multiagent AFCW environments. There are four kinds of entities: *agents* (denoted by the symbol A), *enemies* (denoted by the symbol E), *food pieces* (denoted by the symbol F) and *obstacles* (denoted by dark squares). Every agent has limited energy and energy decreases after each move. Contrary to agents, enemies have unlimited energy. A particular food piece item disappears from the environment when it is captured by an agent. The energy of an agent increases after it captures a food piece item. An agent dies if it goes out of energy or gets captured by an enemy. On the other hand, an agent wins when no food piece item remains in the environment. So, an agent must collect food pieces while escaping from enemies to survive. Therefore, AFCW is a survival task. The moves (actions) in the domain are

- enemies have five action alternatives at each time step: going one step north, west, south, east or staying still;
- agents can make four different moves at each time step: going one step north, west, south, or east;
- moving off the environment is considered to be illegal;
- enemy movements toward a food piece, an obstacle and another enemy are considered to be illegal;
- agent movements toward a position occupied by an obstacle or another agent are considered to be illegal.

Illegal movements are not allowed to execute. In case of an illegal movement, the entity making the movement remains at its current location. Illegal movement of one entity may cause the movement of other entities to be illegal too. So, they both are not allowed to move. In AFCW domain, each cell can be occupied by only one entity at any time. The denial of this rule is also treated as illegal movement. In case an agent and an enemy occupy the same cell, the agent dies and the cell is given to the enemy. When an agent and an enemy make cross-movements, the agent dies. When an agent moves to a cell that contains a food piece, the food piece disappears and the agent's energy level is increased.

Enemies look around 80% of the time to detect agents within their visual field (ten cells in each direction), the remaining 20% of the time they stay still. In case there is no agent in their visual field, enemies make random movements, hoping to find one. Otherwise, they find the closest agent (by using the Manhattan distance measure) and tend to move toward it probabilistically.

This domain has obvious similarities to natural domains in which animals must learn to find and collect (eat) food pieces. Such tasks are of interest to researchers investigating animal learning and animal behavior. However, we do not claim that the techniques used in this study accurately model the way animals might learn or behave in this and similar situations. Actually, this domain was chosen for several reasons. First, it provides an interesting learning task where an agent must not only learn to move to a target, but to a target that leaves it in a good position

[1]AFCW is similar to the dynamic environment introduced in [16] with slight modifications.

with respect to future opportunities. Second, it is suitable to any number of agents, the only limit is the size of the environment. This allows the study of RL in a multiagent system. Finally, it provides opportunities for cooperative behavior.

A. Agent Modeling in AFCW

Learning agents need *perceptual*, *cognitive*, and *effectual skills* to interact with their environment. The way of modeling (representing) these skills directly affects agents' performance in the environment. In RL, usually perceptual and effectual skills are considered as part of the environment while cognitive skills are considered as part of the agent. In AFCW, each agent is assumed to have a number of sensors to gather information about the state of the environment. In other words, there are four sensory arrays (maps) for agents, namely *food piece map*, *enemy map*, *obstacle map*, and *agent map*. Each sensor in each array only senses the corresponding entity type in a fixed location relative to the agent. So, each sensor is activated only if there are particular entities in particular locations. In order to remember the previous action taken, an agent stores four-bits where only one is active (set) at any time step. Additionally, one more bit is used as indicator of the result of the previous action, i.e., whether the agent collided or not. This information is internal to the agents.

The assumption of having each array containing sensors for every cell is unrealistic because of huge dimensions, such as 25×25. That is why we should fix the number of sensors independent of the environment dimensions. There are four types of sensors: X, Y, O and Z, where X, Y and O types of sensors are coarse coded, as they are sparse and can be activated by five, nine and thirteen nearby cells, respectively. Each sensor can be activated by more than one entity of the same type and each entity can activate more than one sensor. The Z type of sensors is exact coded and can be activated only by particular cells. There are 52 food, 32 enemy, 40 obstacle and 20 agent sensors. Totally, agents have 149 sensors (including five internal ones) when using agent maps and 129 sensors otherwise.

There are two types of output representations: *local* and *global*. In local representation, agents declare their actions relative to their orientation, e.g., go right. In global representation agents do not have orientation and declare actions as either move north, south, west or east. We have adopted the global action representation in this study. Global action representation has action symmetry as a result of the symmetry [16]. If one changes the input representation 90°, 180°, and 270°, the environment becomes as if there was only one action available (say moving always to the North).

As stated above, the cognition of agents is not part of the environment and this is left totally to the agents. Agents can use their input as state or extract features from this input and use these features as state constituents or include any other information (e.g. history) to form a state descriptor. Finally, agents are given $+1$ unit of reinforcement when capturing a food piece, -1 when captured by enemies and 0 otherwise.

B. Analysis of the AFCW

An agent in AFCW must pursue multiple goals to survive. In general, an agent must learn how to approach food, escape from enemies, avoid obstacles, identify dangerous situations and seek food pieces when there is no food piece in its sight. These goals are for single-agent case settings. In multiagent case, additionally every agent must learn how (and when) to collaborate with other agents in achieving their goals.

AFCW is not a trivial task because it has the following properties: multiple goals, high-dimensional input, dynamic environment, partial observability and history sensitivity. The environment is *dynamic* because there are many entities that can change the state of the world. For instance, one agent can consume a particular food piece while another agent approaches to it.

As stated before, if the dimensions of the environment increase, then it becomes intractable to store the learned knowledge. To overcome this scalability problem, we must adhere to the partial information principle. Since the number of sensors is restricted, many cells will never be sensed in a big environment. This is due to the *partial observability*. So, AFCW is a POMDP task in the sense that agents observe the environment partially. However, POMDPs are generally studied in environments where agents get ambiguous observations (perceptual aliasing) and all the states are enumerated. This does not apply to AFCW because agents do not know about environmental states. Perceptual aliasing can occur in AFCW due to coarse coding. In AFCW, history information from the recent past is stored. This information includes the most recent action and whether it caused a collision or not. It is internal to agents and can be viewed as a degenerate use of belief states.

The task is *history sensitive*, because agents must remember about the past locations. In some cases, an agent sees food pieces and there are chasing enemies that prevent the agent from reaching the pieces. The best thing for the agent to do is to remember the locations of these food pieces and to collect these pieces only when escaping from enemies.

There are other domains similar to AFCW. For example, Cichosz [17] used food-collecting task for his experiments. The food-collecting task is a simplified version of the AFCW. In food-collecting task, there is no enemy and there is always one learning agent. So, food-collecting task is a single-agent task. The environment is stationary because there is only one moving object (the learning agent). The food-collecting task assumes that the positions of food pieces are the same for all trials. This property makes its state size equal to the number of cells. Another related domain is the game of Pacman. Pacman is a computer game and resembles the single-agent case of AFCW. In Pacman, a user directs the agent to capture food pieces and escape from enemies. Other RL domains and applications are described in [20]–[26].

C. AFCW Simulator

We have implemented a visual simulator and environment editor for AFCW. Using our implementation, it is possible to create any environment that can be used for learning/testing purposes. The simulator is capable of creating entities and executing their actions concurrently. When started the simulator, the user selects one of the previously created environments and specifies the number of re-presentations (passes) of the environment and the mode (learning or testing).

The simulator creates entities first. At each time step, it gives the actual state of the environment to agents and enemies. Agents extract their sensor readings and declare their action choices. Then, the simulator notifies agents about the reinforcement values of their selected actions. The simulator removes killed agents from the environment. After the trial ends, the next environment is constructed and a new trial begins. Note that agents are restarted and their learned knowledge is preserved between trials.

IV. Action Coordination in AFCW

A multiagent system consists of a number of autonomous agents. As they are autonomous agents, they try to maximize their own utility over time. In the case of one agent's action affects the utility of others (*strategic interdependence*), the behavior of each agent must be conditioned on the behaviors of others for optimal control. Agents may model other agents explicitly/implicitly or they can treat them as part of the environment (only included in state descriptions).

Multiagent AFCW is a coordination task because the optimal behavior of an agent depends on other agents. Agents should cooperate for maximizing their own and system-wide utilities. We have distinguished two main types of cooperation; one from the *rules* of AFCW and the other from the *nature* of AFCW. In case two agents want to move to the same cell, they both will stay in their previous locations (by the rule of AFCW) and both agents will lose energy. Note that this situation also happens when agents move toward a cell which contains a food piece. This case illustrates that self-interested agents may score poor.

In single-agent AFCW, the agent does not need to use agent sensors because it will always read the value zero (indicating no agent) from all of the 20 agent sensors. However, in multiagent AFCW, agents should use agent sensors to see other nearby agents. This way, it is likely to behave in a coordinated way. Here, exploring different parts of the environment is better than moving together. This way, the number of actions taken by the agents decreases. Another good coordination strategy is that some agents can deal with enemies while others collect food pieces freely. We have also noted that an agent can be forced by enemies to enter into a closed corridor. In that situation, another agent can approach these enemies and force enemies to move toward itself, causing the other agent to be free.

In the literature of multiagent learning, there are many coordination schemes that involve modeling, observing, communicating other agents [27]–[34]. Some coordination schemes may not be possible due to the domain. For example, there may be no time to communicate in reactive domains. Explicit agent modeling is generally studied when joint state information and joint reward function are known. In this case, agents try to model actions to be taken by others. Because of the partial observability in AFCW, each agent only sees the environment from its perspective. Although the state-spaces of the agents are all the same, the agents sense different state information from the same environmental state. There is also no joint reward function in AFCW. Therefore, agent modeling and stochastic game theory are not directly applicable to AFCW. We have developed two new coordination mechanisms among agents, namely *perceptual coordination mechanism* (PCM) and *Observing coordination mechanism* (OCM).

A. Perceptual Coordination Mechanism

In PCM, agents treat other agents as part of the environment and they only include them in state information. This way, the behaviors of other agents are embedded in state transitions. This works well when other agents have fixed or slowly changing behaviors. Coordination information in this setting is gathered from the environment by reward mechanism. Note that this kind of learning is the same as single-agent reinforcement learning because agents are unaware of others in their cognition.

To implement this mechanism in AFCW, we have used agent sensors to sense the nearby agents. Assume that there are two agents, A and B with agent sensors. Because of the symmetry, if A sees (does not see) B then B sees (does not see) A. In case only A learns, B can be modeled no different than enemies. The reason is that the environment is stationary from the point of view of A. In this case, A can learn the optimal coordination strategy directly from the environment. PCM suits well to this case. If B learns too, the environment becomes nonstationary for A. In this case, A can use PCM assuming that B is either nonlearner or changes its behavior slowly.

By PCM, coordination among agents is possible because agents' strategies are embedded in state transitions. Note that when agents do not use agent sensors, every agent becomes a self-interested (rational) agent. In this case, agents within the environment are learning independent of each other, i.e., without sharing sensory inputs or policies. As a result, other agents appear as additional components within the environment, which has a dynamic and unpredictable behavior.

B. Observing Coordination Mechanism

This mechanism extends the PCM to cognitive level. In OCM, like PCM, other agents are included in state information. Additionally, rewards and actions of other agents in the *visual field* (these agents are called seen agents) are observed from the environment. Visual field is a circle of fixed radius (six in our experiments) around each agent. When an agent employs OCM, it uses the information of the rewards of other agents. One way of using this information is calculating and weighting the average reward of seen agents and then integrating the result with agent's own reward. It is obvious that in this setting agents are aware of others. They try to explicitly learn the coordination strategy, without explicitly modeling others.

The idea behind OCM is that the sum of the rewards of agents in a region is their joint actions' utility. Agents will learn coordination if every agent in the region tries to increase the latter sum. Assume that there are two agents, A and B. If only A learns, the OCM works well because A will learn to do its best given the fixed behavior of B. In case A and B move toward a food piece, they both will receive 0 reward. If this situation occurs later, B will again move toward the food piece (if it is rational), A may make an exploratory move elsewhere. In this case, the total reward is one. From this observation, A will conclude that it does not need to move toward a food piece next to B. This mechanism also works when B is nonlearner and not rational. In that case, A will move toward the food. If B learns too, both A and B

will learn that moving toward a food piece is not always good. A deadlock seems to happen when both A and B learn not moving toward a food piece while they are both adjacent to it. However, the deadlock can be eliminated by exploratory actions. From this example and the already stated considerations, we can conclude that OCM can satisfy coordination requirements arising from the rules and nature of AFCW.

OCM has been implemented in two different ways. In the first implementation, each agent has a single policy network and uses the reinforcement function of

$$r_{\text{coop}} = \alpha r + (1-\alpha)\frac{\sum r_a}{|a|}$$

where r and r_a are the reinforcement value of the agent and the reinforcement value of seen agents, respectively. The value $\alpha(0 \leq \alpha \leq 1)$ adjusts the importance of reinforcements.

In the second implementation, each agent has two policy networks: one for its rational behavior and the other for its cooperative behavior. For rational policy network, each agent uses its own reinforcement function. For cooperative behavior policy network, each agent uses the reinforcement function formula. An agent needs to mix the two policy networks into one for both learning (training) and acting (testing). While acting, each agent looks for other agents in its visual field. If there is at least one, then it uses the cooperative policy. Otherwise, it uses the rational policy. While learning, the selection of mixed behavior is more crucial as learning and exploration take place.

As a result, we have used two different policy selection strategies. In the first, an agent always selects actions using its rational policy to generate experiences. This implies that an agent can use $\text{TD}(\lambda)$, for general λ, for its rational policy network because TD errors add up correctly. The generated experiences are also used to update the cooperative policy network if there is at least one agent in the visual field. For this network, general TD methods can not be used, but rather only $\text{TD}(0)$ methods can be used. In the second policy selection strategy, when an agent observes any agent in its visual field, the cooperative policy is used to generate experience (the experience is also used to update the rational policy); if not, the rational policy is used (the experience is not used to update cooperative policy). Both policy networks are restricted to use $\text{TD}(0)$ methods because of the mixed action selection process.

V. Experiments and Results

We have implemented several agents that use different architectures and algorithms. Table I contains names of the implemented agents along with their learning architecture and algorithm. Each agent name is self-descriptive. All of the agents can be used in multi and single-agent AFCW settings, but PCM, 1BpnnOCM and 2BpnnOCM are designed for the multiagent case. Multiagent AFCW agents, including PCM, 1BpnnOCM and 2BpnnOCM can be derived from any MLP based agent in Table I.

All the MLPs used in the algorithms are three-layer feed forward networks. The input layer contains 129 binary neurons, 52 for food pieces map, 40 for obstacles map, 32 for enemies map, 4 for previous action, and 1 to keep whether the previous move of the agent was legal or not. In multiagent case, the use of agent map is optional and when used, agents have additional 20 binary input dimensions. So, the input layer becomes 149 binary neurons. The number of the hidden layer neurons is used as a parameter for tuning (default 20). The number of output layer neurons varies depending on the architecture selected.

For each of the single-agent and multiagent AFCW, we have created 200 training environments and 50 test environments randomly. The created environments are stored and used in all the experiments. This way, fair comparison of agents is possible. We have saved learned knowledge of agents in every third pass (3×200 trials) and tested on 100 test environments (50 of them are the first 50 training environments). The results are averaged over ten runs for all experiments. In all of the experiments, MLP network weights are initialized with random values between -1 and 1, and neurons compute their output with unipolar sigmoid function. All single-agent (multiagent) environments differ from the environment given in Fig. 2 in locations of food pieces. In other words, only food pieces are placed randomly in each environment.

In the next subsections, we present the obtained experimental results. First, we concentrate on the performance results of the agents in Table I for single-agent AFCW in order to have idea about the effectiveness of their learning algorithms and agent architectures. Then, we look at the performance results of our mechanisms for multiagent AFCW.

A. Single-Agent AFCW Experiments

There are various performance measures of the agent in a single-agent case, such as *average life time* measured in number (denoted #) of steps, *average number of food pieces collected*, *number of wins*, etc. The most important measure is the average number of food pieces collected. The number of wins is also important because it also measures the agent's exploration for all regions of the environment. Although the task of an agent is to survive long, the measure of average life time may be a bad statistics about its performance. For example, an agent may make small loops and be unseen by enemies by chance. We have measured several additional statistics (e.g., numbers of being killed by enemies). However, only the average number of food pieces collected and the average percentage of wins are used as performance measures.

In this experiment, we have tested the performance of one and four MLP agents. Exploration strategy (Boltzman), number of hidden units (20) and learning factors (0.8) are same for all agents. Therefore, this experiment compares the learning algorithms. The values of λ and γ are fixed to 0.9 for 1Bpnn Q, 1BpnnSarsa, 1Bpnn $Q(\lambda)$, SymmetricSarsa, 4Bpnn Q, 4BpnnSarsa, and 4Bpnn $Q(\lambda)$.

The average number of food pieces collected by 1Bpnn agents is plotted in Fig. 3. The percent of wins is not given because all agents achieved less than 5%. The results of one MLP symmetric agents and four MLP agents are given in Figs. 4 and 5, respectively. The comparison of Fig. 3 with Fig. 5(a) yields that 4Bpnn agents performed noticeably better. The agents 1Bpnn $Q(0)$, 1Bpnn Q, 1BpnnSarsa and 1Bpnn $Q(\lambda)$ use a single MLP with four outputs, one per action. So, the hidden units are shared among the actions. In other words, weights of

TABLE I
AGENT ARCHITECTURES AND ALGORITHMS

Agent	Architecture	Algorithm
4BpnnQ(0)	Four 3-layer MLP with 1 output neuron	Standard 1-step connectionist Q-learning
4BpnnQ	Four 3-layer MLP with 1 output neuron	Connectionist Q-learning with eligibility traces
4BpnnSarsa	Four 3-layer MLP with 1 output neuron	Connectionist Sarsa with eligibility traces
4BpnnQ(λ)	Four 3-layer MLP with 1 output neuron	Connectionist Q(λ) with eligibility traces
1BpnnQ(0)	One 3-layer MLP with 4 output neurons	Standard 1-step connectionist Q-learning
1BpnnQ	One 3-layer MLP with 4 output neurons	Connectionist Q-learning with eligibility traces
1BpnnSarsa	One 3-layer MLP with 4 output neurons	Connectionist Sarsa with eligibility traces
1BpnnQ(λ)	One 3-layer MLP with 4 output neurons	Connectionist Q(λ) with eligibility traces
SymmetricQ(0)	One 3-layer MLP with 1 output neuron	Standard 1-step connectionist Q-learning
SymmetricSarsa	One 3-layer MLP with 1 output neuron	Connectionist Sarsa with eligibility traces

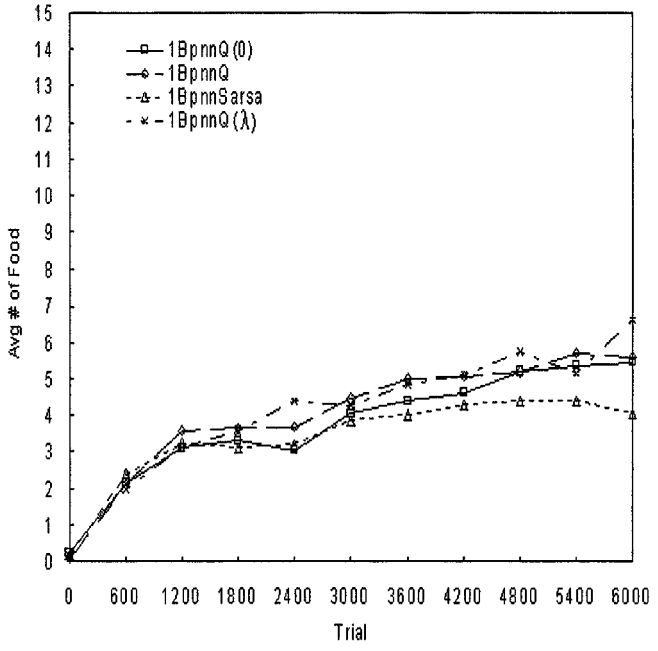

Fig. 3. Results of 1 MLP agents.

the hidden units are always updated with current TD regardless of the action selected. This is why the performance of 1Bpnn agents is low compared to the performance of 4Bpnn agents.

The results in Fig. 5 show that 4Bpnn Q and 4Bpnn $Q(0)$ agents outperformed the 4BpnnSarsa and 4Bpnn $Q(\lambda)$. In lookup table implementations, agents using eligibility trace mechanism converge faster. From our results, it is unlikely to deduce the same property of eligibility trace mechanism. We consider that it is because of function approximation.

The performance of 4Bpnn agents are not as good as $Symmetric\,Q(0)$, because $Symmetric\,Q(0)$ uses action symmetry (domain knowledge). The comparison of all the results show that $Symmetric\,Q(0)$ is the best among all 1 and 4 MLP agents with selected settings. We have also tried other settings by changing the number of hidden units, learning factors, discount factors, etc and observed that $Symmetric\,Q(0)$ is always the best. This is not surprising because due to the action symmetry, it always behaves as if there is only one action available to the agent.

B. Multiagent AFCW Experiments

In multiagent AFCW experiments, we used the same performance measures used with single-agent AFCW. Concerning the experimental settings, the agents are $Symmetric\,Q(0)$, the number of hidden units is 20, the momentum factor is 0, exploration type is Boltzman ($1/T =$ from 20 up), and learning rate is $0.8 \rightarrow 0.2$ and discount factor is 0.9. $Symmetric\,Q(0)$ is selected for being the best for the single-agent AFCW case. So, our Rational, PCM and OCM agents are $Symmetric\,Q(0)$. The average number of food pieces collected by each agent and the overall average number of food pieces collected are the basic performance measures. Because all the statistics are considered from the perspectives of agents (not the environment),

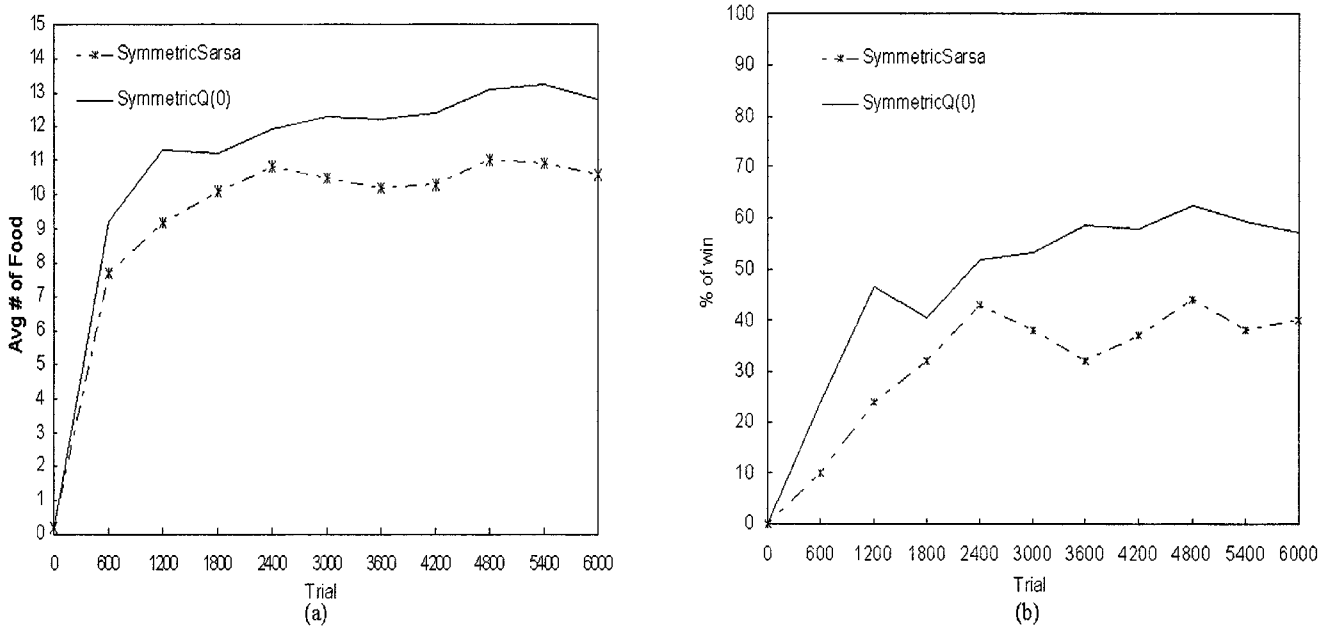

Fig. 4. Results of 1 MLP symmetric agents.

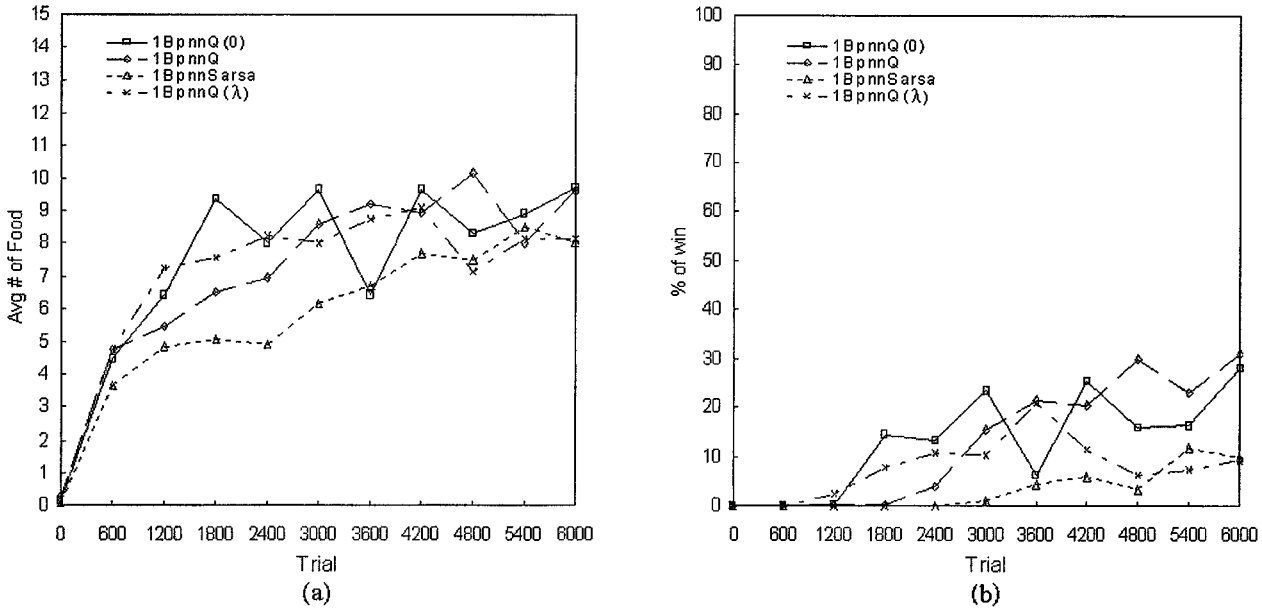

Fig. 5. Results of 4 MLP agents.

we also give the average percent of wins of agents. The actual percentage of agents' wins from the perspective of the environment is obviously higher than the results presented. For example, when agent A dies and agent B collects all the remaining food pieces, only B gets the win status. In the next trial, when B dies and A wins, then A gets the win status. From these two trials, statistics of the environment show that agents win with 100%. However from the agents' perspective, A wins with 50% and B wins with 50%. So, they win with 50% on the average too.

C. Rational and PCM Agents

Rational agents do not have agent sensors and have 129 input dimensions. Therefore, they do not see and model other learning agents. On the other hand, PCM agents see other agents and have 149 input dimensions. Shown in Fig. 6(a) and (b) are the average number of food pieces collected by each agent and their cumulative for Rational and PCM agents, respectively. In Fig. 6(a) both agents are rational, while in Fig. 6(b) they are PCM agents, i.e., they use agent sensors. The average percent of wins by Rational and PCM agents are plotted in Fig. 7.

The results show that both rational and PCM agents are converged and their results are very close. Initially, we have expected that the performance of PCM agents will be better, but the obtained results show that it is not. From this, we conclude that strategic dependence in AFCW happens rarely. In our visual simulations (both in single and multiagent), we have observed that enemies sometimes help the agents to explore different parts of the environment because they change the state information of

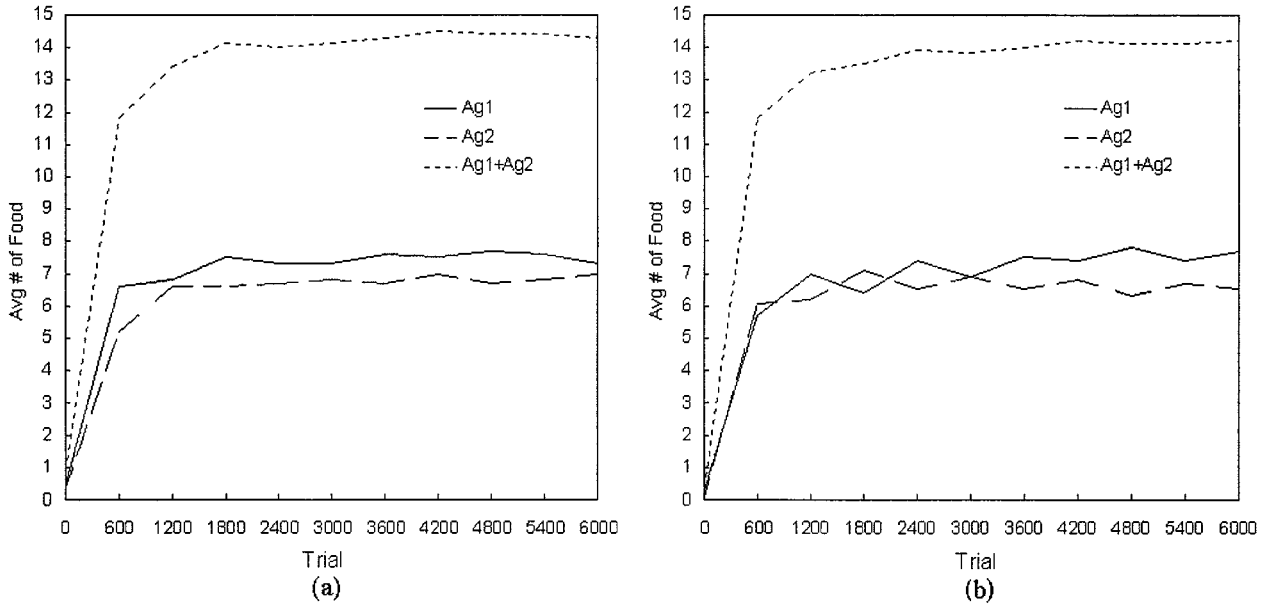

Fig. 6. Results of Rational and PCM agents (average # of food pieces collected).

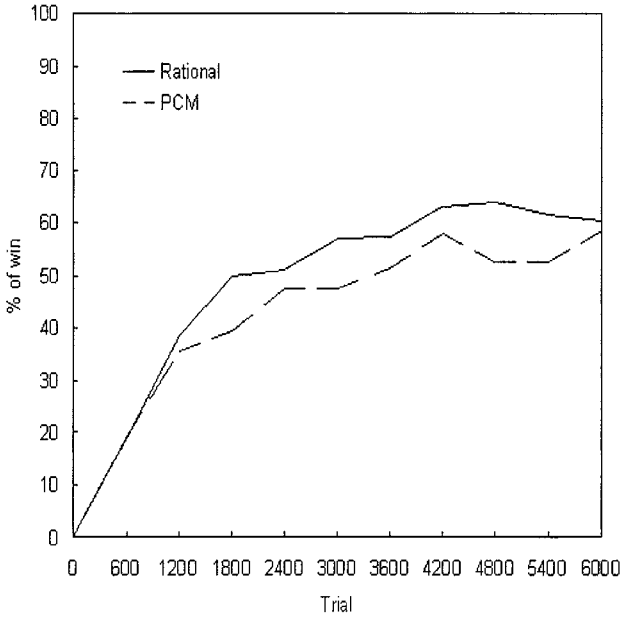

Fig. 7. Results of Rational and PCM agents (% of wins).

the agents. If enemies were removed from the environment, possibly rational agents will score poor. Consider the case that two agents try to collect the same food piece and there is no enemy in the environment. In this case, agents will do the same action until they run out of energy because there is nothing changing state information of both agents. PCM agents can possibly learn how to behave in such patterns.

D. OCM Agents

The average number of food pieces collected by each agent and their cumulative for 1BpnnOCM and 2BpnnOCM agents are shown in Fig. 8. In Fig. 8(a), both agents are 1BpnnOCM with $\alpha = 0.5$. In Fig. 8(b), both are 1BpnnOCM agents with $\alpha = 0.75$. In Fig. 8(c), both are 2BpnnOCM agents with $\alpha = 0.5$. In Fig. 8(d), both are 2BpnnOCM mixed exploring agents with $\alpha = 0.5$. Finally, the average percent of wins for all the settings are plotted in Fig. 9.

The results show that all of the agents converged. Performance of 1BpnnOCM seems a bit better than that of 2BpnnOCM. The comparison of Fig. 8(a) and (b) yields that equal treatment $(\alpha = 0.5)$ of rational and cooperative behaviors produced fair number of food pieces collected by both agents from the beginning of learning. The comparison also yields that agents with $\alpha = 0.5$ converged faster than agents with $\alpha = 0.75$. The performance of 1BpnnOCM agents is above that of 2BpnnOCM. This can be explained by the fact that there are 2 networks in 2BpnnOCM agents and the cooperative policy network is only updated when agents get closer to each other. If this occurs rarely, the network is updated with less number of experiences. In other words, learning a cooperative policy may require more experiences. In 2BpnnOCM agents, experiences are generated by either network and used in both networks. This may lead to the situation of playing bad actions.

The 2BpnnOCM-mixed agents created the problem of selecting mixed behavior while training. There is no such problem in 2BpnnOCM agents because only the rational behavior policy produces the experiences during the training. A similar problem also happens while testing because action selection is mixed, i.e., in a trial in some states rational policy network is used for greedy action selection and in some states cooperative policy network is used for greedy action selection. Despite these problems, the obtained results show the success of agents.

VI. Discussion and Conclusions

A modular visual simulator for AFCW domain has been designed and implemented. An interface for agents is determined and any agent implementing this interface can act in AFCW environments. A series of RL based learning agents have been implemented and trained separately. We have implemented the MLP based RL algorithms, namely $Q(0), Q$ with eligibility traces, Sarsa with eligibility traces and $Q(\lambda)$. It is observed

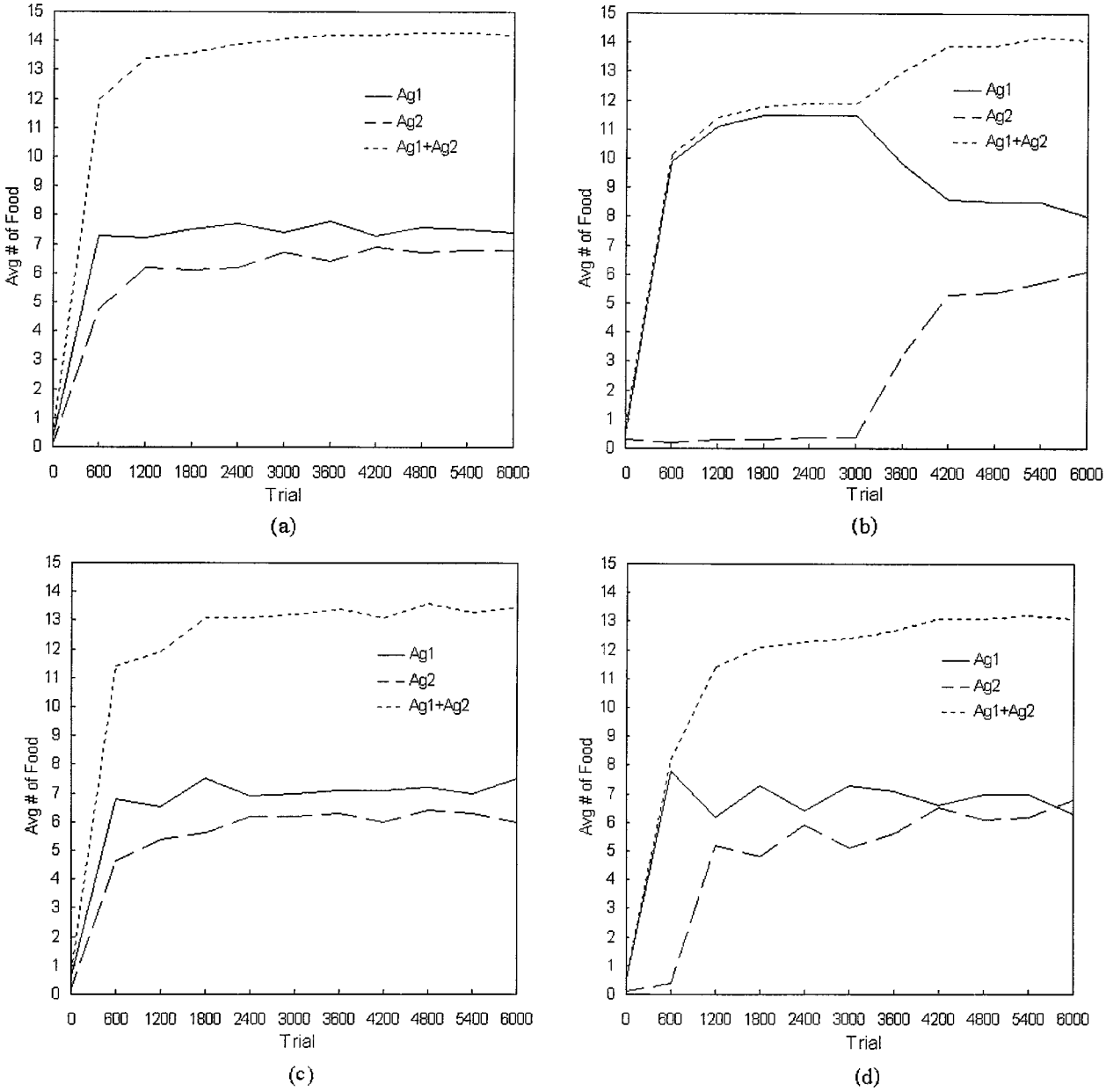

Fig. 8. Results of OCM agents (average # of food pieces collected).

that the experimental results of these algorithms (with the same settings) are very close. Basically, there are two ways of implementing MLP based RL algorithms. In the first, multiple networks exist, one for each action. In the second, there is only one network having multiple outputs, one for each action. In the former, utility of one action does not affect that of others, but in the latter it does. The experiments show that agents using the former approach outperform agents using the latter approach.

The input space of AFCW is symmetric, so agents can use this domain knowledge to boost their performance. We have implemented the agents using this property and shown that their experimental results are remarkably better than that of other agents. The algorithms using the eligibility trace mechanism usually converge noticeably earlier in lookup table implementations. We have observed that algorithms with eligibility traces and 1-step algorithms exhibited similar performances because of the function approximation.

Single agent AFCW results show the importance of domain knowledge in the connectionist RL algorithms, since symmetric agents scored better. The results also show the success of 4-MLP agents over 1-MLP agents. Another result is that algorithms using eligibility trace mechanisms neither improved the performance nor the convergence speed.

For multiagent AFCW, two coordination mechanisms are proposed and experimented. The PCM mechanism allows agents to include other agents in state descriptions. The OCM mechanism integrates the rewards of other agents with agent's own reward value to reach team-wide optimality. The OCM mechanism includes other agents in agent's state description too. The experimental results showed that both PCM and OCM agents performed well and these two mechanisms are effective.

We have considered experience exchange through communication mechanism among the agents. However, learning from experiences of other agents may cause harm to agents when

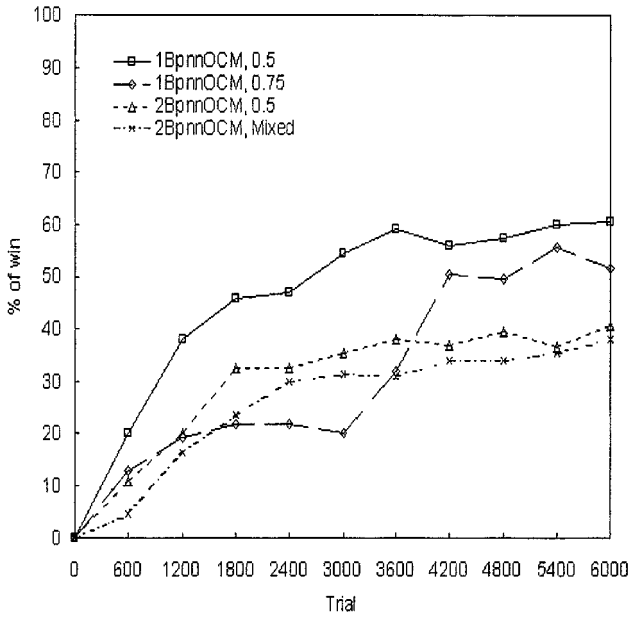

Fig. 9. Results of 1BpnnOCM and 2BpnnOCM agents (% of wins).

agents use eligibility traces mechanisms. Additionally, communication brings up a burden on agents. Obviously, if possible, learning without communication is preferable.

In our multiagent experiments, there are two agents and both are of the same type, i.e., homogeneous. Currently, we are investigating how the performance of the proposed cooperation mechanisms will be affected by introducing heterogeneity into the system, i.e., by constructing different type of agents. For example, 1BpnnOCM-2BpnnOCM, 2BpnnOCM-2BpnnOCM(mixed), 1BpnnOCM-Rational etc. can be experimented. Finally, AFCW can be used as a platform to implement and test various approaches and algorithms. In this work we have assumed perfect sensors and actuators. In order to simulate the real world better the imperfect sensors and actuators can be modeled in AFCW too. This is another area we are currently investigating.

REFERENCES

[1] F. Polat and A. Guvenir, "A conflict resolution based decentralized multi-agent problem solving model," in *Artificial Social Systems*. Berlin, Germany: Springer-Verlag, 1994. LNAI 130.

[2] S. Benson, "Reacting, Planning and Learning in an Autonomous Agent," Ph.D. thesis, Comput. Sci. Dept., Stanford Univ., Stanford, CA, 1995.

[3] L. Baird, "Residual algorithms: Reinforcement learning with function approximation," in *Proc. Int. Conf. Machine Learning*, 1995.

[4] R. S. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," in *Advances in Neural Information Processing Systems*, 1996.

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

[6] G. A. Rummery, "Problem Solving with Reinforcement Learning," Ph.D. dissertation, Eng. Dept., Cambridge Univ., Cambridge, U.K., 1995.

[7] T. W. Sandholm and R. H. Crites, "On multiagent Q-learning in a semi-competitive domain," in *Adaption and Learning in Multi-Agent Systems*, G. Weiss and S. Sen, Eds. Berlin, Germany: Springer-Verlag, 1996.

[8] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, pp. 279–292, 1992.

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

[10] P. Langley, *Elements of Machine Learning*. San Mateo, CA: Morgan Kaufmann, 1996.

[11] J. W. Sheppard, "Multi-Agent Reinforcement Learning in Markov Games," Ph.D. dissertation, John Hopkins Univ., Baltimore, MD, 1997.

[12] S. P. Singh, "Learning to Solve Markov Decision Processes," Ph.D. dissertation, Dept. Comput. Sci., Univ. Mass., Boston, 1994.

[13] L. P. Kaelbling *et al.*, "Planning and acting in partially observable stochastic domains," *Artif. Intell.*, vol. 101, 1998.

[14] N. Meuleau *et al.*, "Learning finite-state controllers for partially observable environments," in *Proc. Int. Conf. Uncertainty Artificial Intelligence*, 1999.

[15] D. J. C. MacKay, "Introduction to Monte Carlo methods," in *Learning in Graphical Models*, M. I. Jordan, Ed. Cambridge, MA: MIT Press, 1999, pp. 175–204.

[16] L. J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Mach. Learn.*, vol. 8, pp. 293–321, 1992.

[17] P. Cichosz, "Reinforcement learning by truncating temporal differences," Ph.D. dissertation, Dept. Electron. Inform. Technol., Warsaw Univ. Technol., Warsaw, Poland, 1997.

[18] J. A. Boyan, "Modular neural networks for learning context-dependent game strategies," M.Sc. thesis, Dept. Eng., Cambridge Univ., Cambridge, U.K., 1992.

[19] M. J. Zurada, *Introduction to Artificial Neural Systems*. New York: West, 1993.

[20] R. H. Crites and A. G. Barto, "Improving elevator performance using reinforcement learning," in *Advances in Neural Information Processing Systems*. Cambridge, MA: MIT Press, 1996, vol. 8, pp. 1017–1023.

[21] L. Gambardella and M. Dorigo, "Ant-Q: A reinforcement learning approach to the traveling salesman problem," *IEEE Trans. Syst., Man, Cybern. B*, vol. 26, no. 1, pp. 29–41, 1995.

[22] J. Hu and M. P. Wellman, "Multiagent reinforcement learning and stochastic games," *Games Econ. Behav.*, 1998.

[23] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," in *Proc. Int. Conf. Machine Learning*, 1994.

[24] V. Miagkikh and W. Punch, "Global search in combinatorial optimization using reinforcement learning algorithms," in *Proc. 1999 Congr. Evolutionary Computation*, vol. 1, 1999, pp. 189–196.

[25] T. W. Sandholm and R. H. Crites, "Multiagent reinforcement learning in the iterated prisoner's dilemma," *Biosystems*, vol. 37, pp. 147–166.

[26] S. P. Singh and D. Bertsekas, "Reinforcement learning for dynamic channel allocation in cellular telephone systems," in *Proc. Advanced Neural Information Processing Systems*, 1996, pp. 974–980.

[27] M. Asada *et al.*, "Agents that learn from other competitive agents," in *Proc. Machine Learning Workshop Agents That Learn from Other Agents*, 1995.

[28] F. Polat, "A negotiation platform for cooperating multi-agent systems," *Int. J. Concurrent Eng.*, no. 3, 1993.

[29] D. C. Parkes and L. H. Ungar, "Learning and adaption in multiagent systems," in *Proc. AAAI Multiagent Learning Workshop*, 1997.

[30] A. Schaerf, Y. Shoham, and M. Tennenholtz, "Adaptive load balancing: A study in multi-agent learning," *J. Artif. Intell. Res.*, vol. 2, pp. 475–500, 1995.

[31] S. Sen and M. Sekaran, "Multiagent coordination with learning classifier systems," in *Proc. IJCAI Workshop Adaptation Learning Multi-Agent Systems*, 1995, pp. 84–89.

[32] S. Sen and T. Haynes, "Co-adaptation in a team," *Int. J. Comput. Intell. Organ.*, vol. 1, no. 4, 1997.

[33] M. Tan, "Multi-agent reinforcement learning: Independent vs. cooperative agents," in *Proc. Int. Conf. Machine Learning*, 1993, pp. 330–337.

[34] G. Weiss, "Learning to coordinate actions in multi-agent systems," in *Proc. Nat. Conf. AI*, 1994.

[35] J. Carbonell *et al.*, "Integrating planning and learning: The PRODIGY architecture," *J. Theoret. Exper. Artif. Intell.*, vol. 7, no. 1, 1995.

[36] C. V. Goldman and J. S. Rosenschein, "Mutually supervised learning in multiagent systems," in *Proceedings of Adaptation and Learning in Multi-Agent Systems IJCAI95 Workshop, Lecture Notes in Artificial Intelligence*, G. Weiss and S. Sen, Eds. Berlin, Germany: Springer Verlag, 1995, vol. 1042.

[37] N. Ono and K. Fukumoto, "Multi-agent reinforcement learning: A modular approach," in *Proc. Int. Conf. Multi-Agent Systems*, 1996, pp. 252–258.

[38] M. Veloso and W. Uther, *Adversarial Reinforcement Learning*, 1997.

Osman Abul received the B.S. degree in 1996 from Hacettepe University, Ankara, Turkey, and the M.S. degree in 1999, from Middle East Technical University, Ankara, where he is currently pursuing the Ph.D. degree.

He is a Senior Software Engineer with the Microwave & System Technologies Division, Aselsan, Inc., Ankara.

Faruk Polat received the B.S. degree in computer engineering from the Middle East Technical University, Ankara, Turkey, in 1987 and the M.S. and Ph.D. degrees in computer engineering from Bilkent University, Ankara, in 1989 and 1993 respectively.

He is an Associate Professor with the Department of Computer Engineering, Middle East Technical University, Ankara. He conducted research as a Visiting NATO Science Scholar at the University of Minnesota, Minneapolis, in 1992–1993. His research interests include multiagent systems, artificial intelligence, and object-oriented data models.

Reda Alhajj received the B.Sc. degree in computer engineering in 1988 from Middle East Technical University, Ankara, Turkey, and the M.Sc. and Ph.D. degrees in computer engineering and information sciences from Bilkent University, Ankara, in 1990 and 1993, respectively.

In 1995, he was promoted to Associate Professor by the Turkish Institute for Higher Education. He served as an Assistant Professor with the College of Computers and Information Sciences, King Saud University, Saudi Arabia, from 1993 to 1996. He then spent one year at Bilkent University and two years at Sultan Qaboos University, Oman. Currently, he is an Associate Professor with the Department of Math and Computer Science, American University of Sharjah, United Arab Emirates. He has published more than 50 papers in refereed international journals and conference proceedings. His primary work and research interests are in the areas of object-oriented databases, data warehouses and view maintenance, multiagent systems, schema unification and reengineering of legacy databases, parallel algorithms, and pattern recognition of hand-written Arabic characters and Hindi numerals.