1 Introduction

Reinforcement learning is defined by Kaelbling, Littman and Moore [6] as 'the problem faced by an agent that must learn behaviour through trial-and-error interactions with a dynamic environment'. Mathematically, the reinforcement learning problem has been formalized as a Markov Decision Process (a process where the probability of the agent moving from one state to another, given its choice of action, is independent of the history of the system prior to reaching that state). The mathematics of Markov processes has been extensively studied, one significant result, Bellman [3], being that an algorithm based on dynamic programming converges to an optimal policy if the Markov process is stationary (a stationary Markov process is one in which the state transition probabilities, given the agent's choice of action, do not change over time).

The present work is concerned with the problem of reinforcement learning in multi-agent systems. This poses two major problems from a theoretical perspective. First, as has been noted elsewhere, e.g. Ono and Fukumoto [8], the simple-minded method of representing the agent's policy as a table (a fully specified function from states to actions), leads to difficulty in generalization, i.e. applying the results of learning in one state to different, but similar, states. One solution to this problem is to use a more compact representation of the policy, such as a neural network. Experimental studies in a number of domains from the field of Artificial Life have shown, that despite the lack of guarantee, standard learning algorithms do converge on some useful policy (it is difficult to be sure the resulting policy is optimal). A more significant problem from the theoretical point of view is that due to the very nature of truly distributed multi-agent systems, the Markov processes they embody are nonstationary, meaning that the standard learning algorithms based on dynamic programming are not guaranteed to converge. This is because, from the perspective of a single agent, as the other agents learn, they adjust their policies for choosing actions (this is the point of the learning!), hence the state transition probabilities change correspondingly.

It is important to realize that the problem for learning caused by the nonstationarity of the Markov process due to the presence of multiple agents exists only when the system is what we have referred to as 'truly' distributed, by which we mean that the agents do not have access to one another's policies. This can easily be seen by observing that non-truly distributed multi-agent systems are equivalent to single agent systems: the collection of agents can be construed as one single "aggregated" agent whose policy is the Cartesian product of the policies of the individual agents, i.e. the aggregated agent chooses one "composite" action which is one action for each of the individual agents in the system. In this way, there is no theoretical obstacle to using standard learning algorithms, and furthermore, it is obvious that n distributed, but omniscient, agents in a multi-agent system can learn effectively n times faster than a single agent, c.f. Whitehead [15], since they effectively have n times as many experiences to draw on. However, there are practical problems arising from the size of the resultant search space through which the learning algorithm must navigate: Ono and Fukumoto [8] call this a 'monolithic' reinforcement learning problem to emphasize the point. Thus one strand of work in multi-agent reinforcement learning is concerned with discovering compact representation schemes for the utility functions, then showing that standard algorithms still enable the agents to learn.

The present work investigates a multi-agent system in which the agents are situated in a simple artificial environment. The environment consists of a 2-dimensional grid on which "food"-type resources are placed at random locations. The agents must move to these food elements in order to "eat" them. The world can be made dynamic by allowing food to spontaneously appear and disappear. The task of the agent is to consume as much food as possible, i.e. to choose actions that maximize the total reward it can receive in the long term. Note that it is easy to specify this goal for the agent, but it is not obvious which action is optimal at each step. It is therefore difficult to apply supervised learning in this domain.

We use and compare two reinforcement learning algorithms: the Adaptive Heuristic Critic (AHC) algorithm, Barto, Sutton and Anderson [2], and the Q-learning algorithm, Watkins [14]. The AHC method attempts to learn the value or utility of being in a given state of the world. This utility is just how beneficial that state is to the agent, i.e. is a measure of the total reward it will receive in the future. Once these utilities are known, the agent can then perform actions that lead to beneficial states. Q-learning attempts to learn the value or utility of performing each possible action in a given state of the world. Then, given the current state of the world, the agent can pick the action with the highest Q-value. Both algorithms require some method of storing these functions, mapping each state or, in the case of Q-learning, each state-action pair, to a real number representing the utility. As the current domain contains a very large number of states making lookup tables impractical (in our simulations there are typically over 10^{36} possible states), we use neural networks to learn the various functions. This also enables generalization between similar states, which is certainly appropriate, and perhaps even necessary, for effective learning in this domain; if two states are identical except for, say, a single, distant piece of food, the utilities of those states will be very similar.

The interesting feature of this domain is that the actions of other agents can affect an agent's potential for receiving rewards. This can occur due to interference, competition or cooperation. Hence an agent that can learn a strategy which takes account of the actions of the other agents should outperform an agent that cannot do so. The main contribution of this paper is to present a simple method, in the spirit of reinforcement learning, which enables an agent to learn by "observation" of the other agents, and hence to learn a superior policy. The method is called *vicarious reinforcement*. The idea is simple: an agent, as well as receiving rewards from its own actions, receives a portion of the reward given to "nearby" agents. At first sight, this may seem to discourage competition between agents and hence decrease performance; but, as we shall demonstrate experimentally, agents that learn using vicarious rewards outperform more selfish agents. Intuitively, it appears that a group of agents with vicarious reinforcement are learning to cooperate by avoiding conflict. What is interesting is that this actually benefits each of them in the long run. This need not be the case: a purely selfish agent may also learn to maximize reward by minimizing interference, possibly taking advantage of other agents' cooperative behaviour. Something more is going on with vicarious reward. It appears that in order to attain optimal performance in the system, each agent is required to act in a "socially responsible" way, at times sacrificing the maximum personal reward in order to benefit other agents. The crucial point is that if *all* agents cooperate in this way, *each* agent benefits in the long run. So it seems that vicarious reinforcement assists an agent only when it is part of a community of agents which adopt this learning strategy.

This is an issue of local optimality versus global optimality. In this context, a locally optimal action is an action that maximizes the long-term reward for the agent performing it, while a globally optimal action is one that maximizes the total long-term reward received by all agents in the system. If each agent attempts to locally optimize their strategies, the resulting system may be globally sub-optimal, i.e. all agents may be worse off, as local optimization does not take into account the effect of actions on others. The main result of this paper is that vicarious rewards enable agents to learn a superior global strategy using only local information (whether this is a globally optimal strategy is difficult to determine). It also appears that a crucial factor enabling the learning is the spatial distribution of the agents and the “locality” of utility information. What we mean is that, intuitively, the feedback provided by vicarious reinforcement is useful to the extent that information about “nearby” agents is important to incorporate in an agent’s utility function, while feedback from “distant” agents has very little value to the agent. On this reasoning, learning with vicarious rewards works because it gives a better, and at the same time, useful, approximation to the centralized, monolithic reinforcement learning problem than a distributed multi-agent system based only on local rewards. Such an explanation seems to underlie the work of Tan [13], who investigates various ways that agents can “share” experiences whilst learning by reinforcement, resulting in improved performance; see also Clark, Irwig and Wobcke [5].

The remainder of this paper is organized as follows. In section 2, we define, at an abstract level, the different agent architectures used in our experiments. In section 3, we describe the experimental design used in the current work, specifying the domain and the various types of agents used in our simulations. We present and discuss the results of the simulations in section 4, and conclude in section 5.

2 Agent Architectures

In this section, we describe the agent architectures and learning algorithms used in our experiments. We focus on the abstract properties of the agent types, deferring discussion of the specific details of particular agents types until section 3. We use two different types of agent architecture: one based on the Adaptive Heuristic Critic (AHC) algorithm and one based on the Q-learning algorithm. Both architectures use neural networks to represent utility functions, and both use back-propagation, Rumelhart, Hinton and Williams [9], for learning this function. In addition, the AHC agent has a second neural network representing a mapping from states to actions, and this network is trained using Complementary Reinforcement Back-Propagation (CRBP), Ackley and Littman [1]. We now give a more detailed description of these architectures.

2.1 Adaptive Heuristic Critic

The Adaptive Heuristic Critic (AHC) algorithm, presented by Barto, Sutton and Anderson [2], consists of two components: a *critic* component and an *action* component. The action component is used to determine what action to perform in a given state, i.e. it implements the current *policy* of the agent. Given that the agent is currently executing policy π, the critic component learns the value function V_π using reinforcement signals received by the agent whilst carrying out its policy. The value function used here is the expected infinite discounted sum of rewards of being in a particular state. More precisely, the value of a state s given policy π is defined as follows, where the discount factor γ, a real number such that $0 < \gamma < 1$, is a measure of how immediate rewards are weighed against potential future rewards, and E denotes expected value:

$$V_\pi(s) = E\left(\sum_{t=0}^{\infty} \gamma^t r_t\right)$$

Assuming that the critic component has learned the function V_π for the current policy π, the action component can learn an improved policy, π', that maximizes $V_\pi(s)$. The action component learns using feedback from the critic component, which provides an estimate of the discounted reinforcement that can eventually be obtained from the environment given the current state. To see how this estimate allows an improved policy to be developed, consider the case where the agent is in state s, and its current policy is to perform action a_1, which would lead to state s'. The value of being in state s' is given by $V_\pi(s')$. Now suppose the agent finds through trial and error that there is another state, s'', which is reached from state s by performing action a_2, and that $V_\pi(s'') > V_\pi(s')$. Performing action a_2 when in state s is clearly better than performing action a_1, and the agent can modify its policy accordingly. In this way, the action component learns an improved policy π' by maximizing the value computed by the critic component. The critic component then computes the value function for the new improved policy, $V_{\pi'}$, and the cycle repeats. Williams and Baird [16] have shown that this method eventually converges with probability 1 to the optimal policy, assuming some simple conditions. In practice, the critic component and action component often learn simultaneously, rather than alternating as described above. With this variation, the system is not guaranteed to converge to the optimal policy.

2.1.1 Critic Component

The critic component learns using the TD(0) (Temporal Differences) method described in Sutton [11]. Suppose an agent in state s performs action a, receiving a reward r and ending up in state s'. This provides some information about the value or utility of being in state s, $V(s)$. If we know the value of being in state s', $V(s')$, and if action a performed in state s always leads to state s', we can simply set $V(s) = r + \gamma V(s')$. This is because, by definition, the value of being in state s is equal to any immediate external reward plus the discounted value of the next state. However, the Markov Decision Process model assumes that an action performed in some state may lead to different states on different occasions, according to the state transition probability function T. Furthermore, the critic's current estimate of $V(s')$ may not be accurate. In light of these considerations, the critic takes account of this new information using the update rule:

$$V(s) := V(s) + \alpha(r + \gamma V(s') - V(s))$$

Here α is a learning rate, γ is the discount rate, and both values are between 0 and 1. After applying this update rule, the estimate of a state's value is shifted towards $r + \gamma V(s')$; the degree to which it is shifted is determined by the value of α. Since s' is sampled from the distribution of states with probability $T(s, a, s')$, it can be shown that, assuming α is slowly decreased and the policy is held constant, the TD(0) method will converge to the correct value function with probability 1, Sutton [11]. It should be noted that this update rule allows the value function to be learned using a supervised learning algorithm or a simple lookup table, as the rule provides the desired output for some state of the world.

2.1.2 Action Component

The critic's output (the value of being in some state) can easily be used to generate a value which represents whether or not some action is better than the action proposed by the current policy. This value can then be used as a reinforcement signal for the action component. Consider the general case where an agent in state s performs action a, receiving an external reward r and ending up in state s'. If the action forms part of the policy for which the value function was learned, we expect that $V(s) = r + \gamma V(s')$. If the agent has tried some action that was not dictated by the policy, it may be that the agent finds itself in a better position than predicted by the value function, i.e. $V(s) < r + \gamma V(s')$. Similarly, if the new action turns out to be worse than expected, we have $V(s) > r + \gamma V(s')$. We can use this information to define a reinforcement signal ρ:

$$\rho = r + \gamma V(s') - V(s)$$

When $\rho = 0$, the action did as well as expected, and no new information is gained. When $\rho > 0$, the action was better that that given by the current policy. When $\rho < 0$, the action is worse than the action given by the current policy with regard to the current value function.

We use Complementary Reinforcement Back-Propagation (CRBP), Ackley and Littman [1], to improve the current policy using the feedback given by the reinforcement signal. CRBP works by transforming the problem of reinforcement learning from immediate rewards into one of supervised learning, which is then accomplished using standard back-propagation. Supervised learning requires the desired output to be specified, and CRBP generates this desired output using a simple heuristic. The first step is to take the binary output vector from the action network. The CRBP method assumes that if the reinforcement signal is positive, the desired output is identical to the generated output vector. However, if the reinforcement signal is negative, the desired output is assumed to be the *complement* of this output vector, multiplied by a dampening factor. Back-propagation can then be used to train the action network to minimize the error function so specified.

Figure 1 shows the architecture of our Adaptive Heuristic Critic (AHC) agent, which uses neural networks to represent the action and critic functions. The networks are trained using back-propagation, with CRBP used to determine feedback for the action network.

2.2 Q-learning

Q-learning is a method developed by Watkins [14] that uses techniques similar to those used in the TD(0) method. With the AHC algorithm, the value function $V^*(s)$ is defined as the representing the utility of being in state s, assuming that the optimal policy is being used. With Q-learning, $Q^*(s,a)$ is defined to be the value of taking action a in state s, given that all future actions will be optimal. Notice that if $Q^*(s,a)$ is known, an optimal policy can be determined by, in every state s, choosing the action a that maximizes $Q^*(s,a)$. This eliminates the need for separate critic and action components as used in the AHC algorithm. The Q-learning update rule is analogous to the TD(0) update rule:

$$Q(s,a) := Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

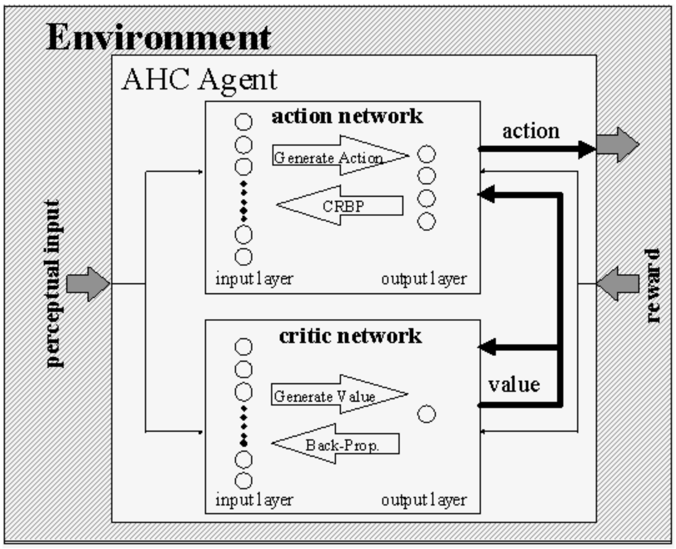

Figure 1: Architecture of an agent that uses neural networks and CRBP to implement the AHC algorithm.

Once again, α is the learning rate, γ the discount rate, and both values are between 0 and 1. It has been shown by Watkins [14] that if each action is executed in each state an infinite number of times, and α is slowly decreased over time, the function will converge with probability 1 to the optimal Q^* function. As for the TD(0) method, this function can be learned using a simple lookup table or a supervised learning algorithm. The Q-learning algorithm appears to be more popular than the AHC algorithm, in part because it is easier to implement. It is also claimed to be not as sensitive to the manner in which the agent explores new actions, Watkins [14]. On the other hand, the separation of the action component and critic component maintained by the AHC algorithm may be advantageous when extending the architecture, as done by Sutton [12]. It seems more intuitive to reason about the value of being in some state, than about completely separate values of performing different actions in some state. It is not clear in which situations it is best to use which algorithm.

Figure 2 shows the architecture of an agent that learns using the Q-learning algorithm. It consists of several neural networks, each learning the value of performing a particular action in a given state. The Q-learning update rule provides the desired output, allowing the networks to be trained using standard back-propagation.

3 Experimental Design

In this section, we present the details of the experimental design used in our study of multi-agent reinforcement learning. We describe our domain, the simulation procedures, and the different types of agents used.

3.1 Domain and Simulations

The domain used is based on a simple variation of the Tileworld described by Kinny and Georgeff [7]. Our world consists of a two-dimensional grid of squares, with the edges of the grid "wrapped around" to form a torus.

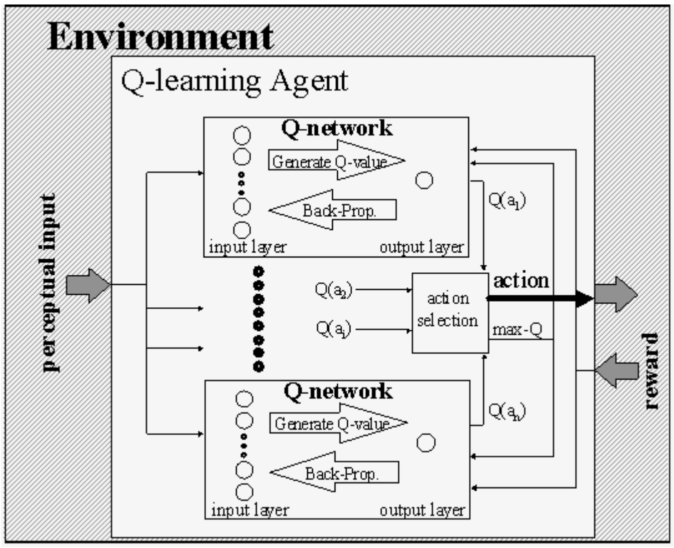

Figure 2: Architecture of an agent that uses neural networks to implement the Q-learning algorithm.

Some squares contain a "food"-type resource, which agents must "eat" in order to score points. An agent eats a piece of food simply by moving to the same location. Agents can move one square at a time, either north, south, east or west, but cannot move onto a square occupied by another agent. The world can be made dynamic by allowing food to spontaneously appear and disappear. It is an important feature of this world that any food that is eaten or spontaneously disappears is replenished at a random position, therefore there is no inherent limitation on an agent's performance due solely to the scarcity of resources; the number of pieces of food is a constant of the simulation. The domain is illustrated in Figure 3.

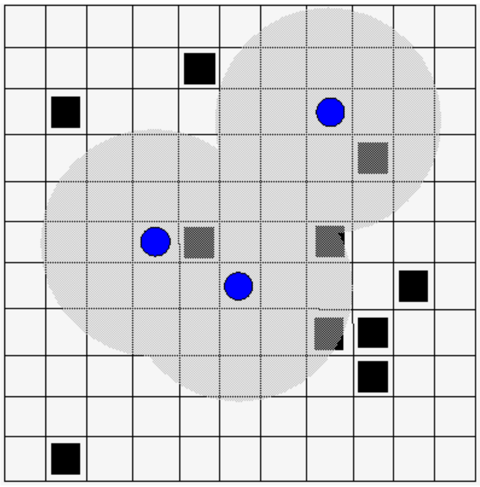

Figure 3: Illustration of the Tileworld domain, where the circles are agents and the squares are pieces of food.

This domain has obvious similarities to natural domains in which animals must learn to find and collect or eat food. Such tasks are of interest to those studying animal learning and animal behaviour. However, we do not claim that the techniques used in this work accurately model the way animals might learn or behave in this type of situation. The domain was chosen for the following reasons:

- it provides an interesting learning task – an agent must not only learn to move to a target, but to a target that leaves it in a good position with respect to future opportunities;
- it can easily be made dynamic by allowing food to spontaneously appear and disappear;
- it is suited to any number of agents (limited only by the size of the grid), allowing the study of reinforcement learning in a multi-agent system;
- it provides opportunities for cooperative behaviour, e.g. opting for a slightly worse action in order to benefit another agent, allowing the investigation into the emergence of cooperative strategies.

All simulations consist of the world grid described above, a collection of agents, and an external clock. At each clock tick or *cycle*, each agent is given a chance to perform one action (i.e. move one square). The world is maintained at the start of each cycle by a special agent known as the *controller*. The controller takes two parameters: the number of pieces of food F, and the food vanishing rate v. These values are both constants of a simulation. The vanishing rate is a measure of how dynamic the world is. It is defined as the probability that a given piece of food will spontaneously disappear during a given cycle. A vanishing rate of 0 means that no food will disappear unless eaten by an agent. A vanishing rate of 1 means that every piece of food will disappear and reappear at a random position each cycle. It is the task of the controller agent to randomly delete pieces of food according to the vanishing rate, and ensure that there are F pieces of food in the world at the start of each cycle. Because this occurs at the start of each cycle, the order in which the other agents act is randomized to ensure that no agent is treated unfairly by being positioned consistently early or late in the cycle.

3.2 Types of Agent

Three types of agent are used in these experiments. The first is a baseline agent that acts according to a hand-coded greedy strategy. The two other types of agent learn their action policies using two different reinforcement learning algorithms: Adaptive Heuristic Critic (AHC) and Q-learning. All agents perceive a limited area of the world, defined as their *visual field*. A visual field of $d \times d$ squares means that the agent perceives the $d \times d$ portion of the world grid with itself at the centre. Recall that the formal model of reinforcement learning assumes a Markov process. That is, it is assumed that the result of performing some action given some state of the world in no way depended on the agent's previous actions or states. This assumption does not hold if the agent only partially observes the world grid, i.e. if the world grid is larger than the agent's visual field, as is the case in some of our experiments. There are strategies that can be used to learn in partially observable environments, Kaelbling, Littman and Moore [6]. Here, we simply assume that, providing the visual field is not too small and food not too scarce, any effect on future reinforcement of the "hidden" part of the world is negligible.

3.2.1 Baseline Greedy Agent

This agent acts according to a hand-coded greedy strategy. At every cycle, the agent locates the closest piece of food within its visual field and moves one step towards it. If there is no food within the visual field, or the move cannot be made due to the presence of another agent, a random move is made. This agent is designed to provide a comparison to the performance of the agents with learned strategies. It is not an optimal strategy; it is short-sighted, in that it does not consider any actions beyond eating the next piece of food. It may be better in the long run to target a piece of food that is further away if it leaves the agent in a better position to eat other food. To perform better than this simple strategy, an agent must not only learn to move efficiently towards a piece of food at which point it receives a reward, but must take into account the potential for still more distant rewards.

3.2.2 AHC Agent

This agent learns using the Adaptive Heuristic Critic architecture presented by Barto, Sutton and Anderson [2]. As described above, simple lookup tables are impractical in the current domain due to the large number of world states. For a $d \times d$ visual field, there are 2^{d^2} possible states, assuming that the agent only notices pieces of food and not other agents. In these simulations, an agent's visual field was typically set to 11×11 squares, resulting in 2.7×10^{36} possible states. Therefore, some way of generalizing between states is required. The domain lends itself to such generalization, as similar states will be of similar value to the agent and will require a similar action. For this reason, each component is implemented as a single layer, feed-forward, fully connected neural network. Single layer neural networks appear to be sufficient for learning the basic task the agents are required to perform, and require less computation time than networks with hidden layers.

The critic network takes as input the current state of the world, and produces a scalar output estimating the value of being in that state (the expected infinite discounted sum of rewards). Each square on the grid that is within the visual field of the agent corresponds to an input node, requiring d^2 input nodes for visual field of $d \times d$. The activation of the input node is set to 1 if there is a piece of food on that square, and 0 otherwise. If the agent also has the ability to perceive the location of other agents, each square on the grid corresponds to two input nodes – one representing whether or not there is food on the square, and one representing whether there is another agent on the square. This requires $2d^2$ input nodes. The network is trained by back-propagation, using the TD(0) update rule to provide the desired output. The implementation of the critic network was based on pseudo-C code supplied by Bonde and Sutton [4].

The action network also takes as input the current state of the world, and has four output nodes, each corresponding to a different action. Recall that the four actions available to an agent in this domain are to move one square north, south, east or west. The output of this network is used to determine which action the agent will take. The action corresponding to the output node with the maximum activation represents the agent's current policy given the current state of the world. However, always selecting this action would prevent the agent from exploring new and possibly better actions, which is a crucial part of the reinforcement learning process. Therefore, the action is determined probabilistically from the network output while the agent is learning. Given output activations y_1, y_2, y_3 and y_4, the probability of selecting action a_i, $p(a_i)$, is given by:

$$p(a_i) = \frac{y_i}{y_1 + y_2 + y_3 + y_4}$$

When testing the performance of the agent, the action corresponding to the output node with maximum activation is always selected, as exploration is no longer required.

The action network is trained using complementary reinforcement back-propagation (CRBP). CRBP is applied to the current problem as follows. The output activations of the action network give a binary output vector o representing the action to be taken. The value of o_i should be 1 if action a_i is selected, and 0 otherwise. As the agent can only take one action at each step, exactly one element of o should be equal to 1. An immediate reinforcement signal for this action, ρ, is calculated using the output from the critic network. When ρ is positive, o is treated as the desired output, so the error for output node i is $e_i = o_i - y_i$. When ρ is negative, the CRBP heuristic defines the desired output as the complement of o, and the error for output node i as $e_i = \eta(1 - o_i - y_i)$, where η is a dampening factor designed to reduce the effect of negative reinforcement. The errors are then back-propagated in the standard manner.

In these experiments, the reward r was 0.2 if the agent ate a piece of food and 0 otherwise. The back-propagation learning rate δ was set to $1/n$ where n is the number of input nodes in the neural network, and the discount rate γ was set to 0.9 as suggested by Bonde and Sutton [4]. Moreover, the back-propagation learning rate makes the learning rate α in the TD update rule redundant, therefore α was set to a value of 1. The parameter η used in generating the CRBP errors for negative reinforcement was set at 0.2, experimentally determined to perform well for the case of a single agent in a static world. This has the effect of attaching greater significance to positive reinforcement than to negative reinforcement, which is appropriate considering the heuristic nature of the errors generated in the case of negative reinforcement, Ackley and Littman [1].

3.2.3 Q-learning Agent

The Q-learning agent consists of four single layer, feed-forward neural networks. Each network corresponds to a different action. A network takes the current state of the world as input, and produces as output the value of performing that action. This value is an estimate of the infinite discounted sum of rewards. The networks are trained by back-propagation, using the Q-learning update rule to provide the desired output. Given the output of the four networks, the agent must select an action. As discussed above for the AHC agent, consistently selecting the action with the best Q-value would prevent the agent from exploring new and possibly better strategies. However, using the Q-values as probabilities, as is done with the values of the AHC agent's output network, is inappropriate. This is because the Q-values of the various actions are very similar, and using these similar values as probabilities would result in agents that act in a largely random way. To avoid this, probabilities are generated from the Q-values using a Boltzmann distribution, which favours actions with higher value estimates, e.g. Sandholm and Crites [10]. The probability of selecting action a_i, $p(a_i)$, is given by:

$$p(a_i) = \frac{e^{Q(s,a_i)/T}}{\sum_a e^{Q(s,a)/T}}$$

Here T is a *temperature* parameter that controls the amount of exploration. In these experiments, T was set to a value of 0.05, experimentally determined to perform well for the case of a single agent in a static world. As for the AHC agent, the discount rate γ was set to 0.9, the learning rate for the back-propagation procedure δ was set to $1/n$ where n is the number of input nodes in the neural network, and the redundant Q-learning update parameter α was set to 1. Agents received a reward r from their environment of 0.2 when they ate a piece of food, and 0 otherwise.

4 Experiments and Discussion

In this section, we present and discuss the results of our simulations. The section consists of two parts corresponding to the two aims of this work. In section 4.1, we apply standard reinforcement learning to the present domain, which also serves to validate the experimental setup for the following section. We compare the performance of the Adaptive Heuristic Critic (AHC) and Q-learning (QL) agents to that of the baseline Greedy agent, and explore how performance is affected by the number of agents in the world, whether the agents can "see" each other, and the degree to which the world is dynamic. In section 4.2, we discuss the reasons that selfish agents, i.e. agents that learn to maximize their own food intake, may not learn the globally optimal strategy in the present domain. We then present the results of simulations in which the agents are encouraged to learn to behave cooperatively by receiving vicarious reinforcement, and experimentally demonstrate that these agents perform better than selfish agents.

4.1 Standard Reinforcement Learning

These simulations examine how the AHC and Q-learning agents perform when several agents coexist in the world. As noted above, because of the presence of other agents and because of the agents' limited field of vision, standard learning algorithms are not guaranteed to converge. The problem is exaggerated if agents cannot "see" each other, i.e. if an agent's perception of the state of the world does not include the locations of other agents. Knowing the locations of other agents provides some information as to which pieces of food are likely to disappear due to other agents eating them. We expect an improvement in performance because if this information does not form part of the agent's state, the state transition probabilities appear to vary dramatically as the other agents move around in the world. We also explore how the performance of agents changes as the world becomes dynamic. Note that for a single agent in a dynamic world, the state transition probabilities do *not* change over time; however, the task of learning how to behave is more challenging in a dynamic world than in a static one.

These simulations were conducted in a 50×50 world grid, with 250 food elements and various vanishing rates v. Each simulation consisted of one or more agents acting in this world, however all agents in a given simulation were of the same type. The agents had a visual field of 11×11 squares. In each simulation, agents were allowed to learn for 3 000 000 cycles. Their policies were then fixed, and their performance was tested over 100 000 cycles. Performance was measured as the average number of food elements eaten over 10 000 cycles *per agent*.

Figures 4 and 5 plot average performance against the number of agents in the world when the world is static ($v = 0$). The graphs show the performance of the three different types of agent: the Greedy agent, the Adaptive

Heuristic Critic (AHC) agent, and the Q-learning (QL) agent. Figure 4 shows the performance of learning agents that are not aware of the presence of other agents. That is, an agent's perception of the world consists only of the location of food elements within its visual field. Figure 5 shows the case where each learning agent perceives not only the location of food elements, but also the location of other agents within its visual field. We call the former agents *non-aware* and the latter *aware*.

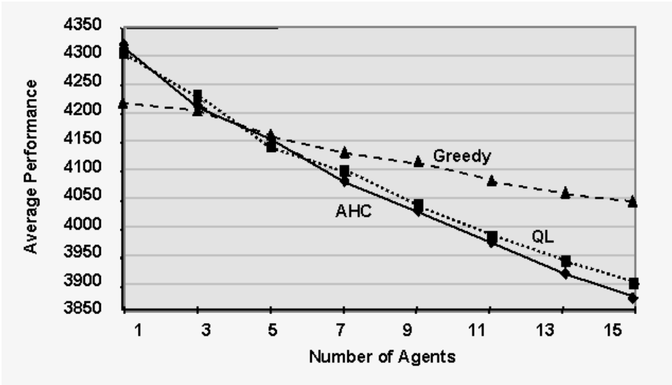

Figure 4: Average performance of non-aware agents in a static world for various numbers of agents in the world.

When there is only a single agent in the world, the AHC and QL agents perform substantially better than the Greedy agent. As expected, there is no difference between the performance of the aware and non-aware agents when only a single agents exists in the world, as in this case being aware provides no additional information. As the number of agents in the world starts to increase, the average performance starts to decrease for all types of agents. This is not surprising; while using several agents may increase the overall performance of the system, each individual agent's performance may suffer due to interference and competition from others.

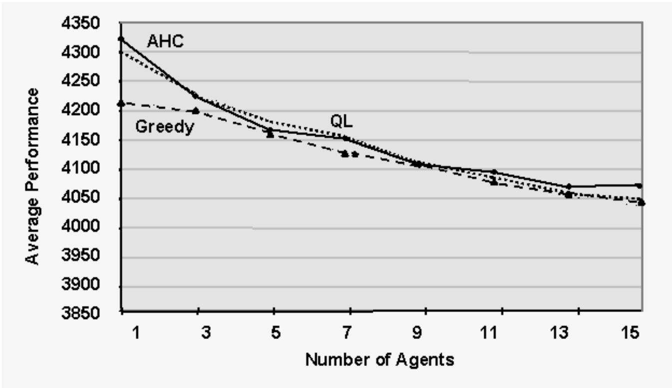

Figure 5: Average performance of aware agents in a static world for various numbers of agents in the world.

Figure 5 shows the performance of the aware AHC and QL agents, as well as the Greedy agent, as the number of agents in the world increases. In contrast to results for the non-aware agents shown in Figure 4, the aware AHC and QL agents perform as well as, if not better than, the Greedy agent.

As aware agents include the location of other agents in their perception of the state of the world, the variation of state transition probabilities is less dramatic. The aware AHC agent nearly always performs better than the Greedy agent. A one-tailed binomial test shows that over this range of settings, the superiority of the AHC agent is statistically significant ($P < 0.05$). The QL agent appears to perform similarly to the AHC agent for a small number of agents, but as the number of agents increases beyond 9 its performance drops, and seems to be similar to that of the Greedy agent. Further tests are required to establish whether this apparent trend is significant, and whether the performance of the QL agent can be improved by changing its parameters, in particular the Boltzmann temperature T, which determines the degree to which the agent explores new actions.

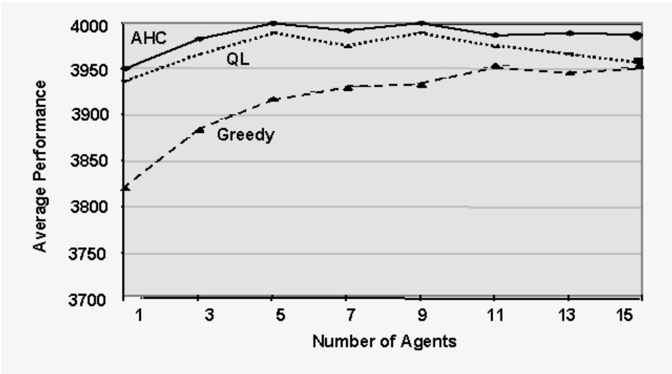

Figure 6: Average performance of aware agents in a dynamic world ($v = 0.01$) for various numbers of agents in the world.

Figures 6 and 7 plot the performance of the aware AHC and QL agents, as well as the Greedy agent, in dynamic worlds. Recall that the vanishing rate v is defined as the probability that a given food element will "spontaneously" disappear each cycle, and is a measure of how dynamic the world is. The vanishing rates used for the simulations in Figures 6 and 7 were 0.01 and 0.02 respectively.

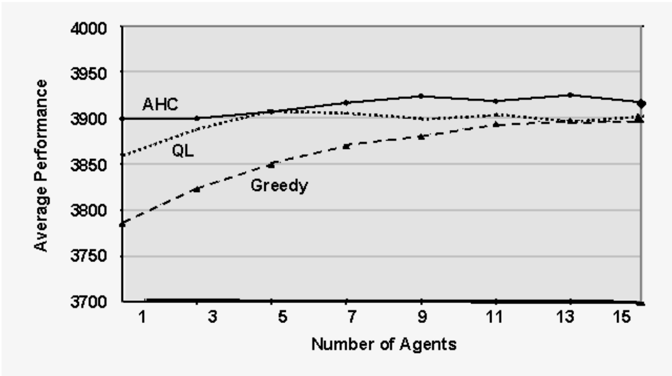

Figure 7: Average performance of aware agents in a dynamic world ($v = 0.02$) for various numbers of agents in the world.

It is a somewhat surprising result that even for Greedy agents, the presence of other agents in the world actually enhances the average performance per agent. That is, as the number of agents in the world increases, so does the average performance of an agent. This type of phenomenon is discussed for reactive agents by Clark, Irwig and Wobcke [5], and has to do with the replenishment of food elements consumed by other agents, in addition to a reduction in the impact of competition due to the dynamic nature of the world. Both graphs show that the learning agents generally perform better than the Greedy agent in a slightly dynamic world, with the AHC agent performing best of all. In particular, the performance of the QL agent tends to decrease when many agents are present. This trend may not be significant, but the AHC agent does appear to be the more successful agent in these domains. Comparing Figures 6 and 7, we see that the differences between the learning agents and the Greedy agent are not as great in the more dynamic world. This is not surprising, as in a very dynamic world, it is not obvious that a better strategy than the greedy one exists; if there is a high probability that food will disappear, it makes sense to take advantage of any food that happens to be close.

Overall, these results show that the AHC agent performs better than both the Greedy agent and the QL agent in static and slightly dynamic worlds. The AHC has a particular advantage over QL agents when many agents are present or when the world is dynamic. This could be due to the fact that it only learns a single value function rather than four separate ones, which may be an advantage when attempting to generalize between states; every new experience contributes to this single function. As above, further tests are required to show whether the performance of the QL agent can be improved by changing its parameter settings, such as the Boltzmann temperature T, or by modifying the structure of the neural networks used to represent the Q functions. As mentioned in section 3.2, parameters in both types of agent were experimentally determined to perform well for a single agent in a static world.

4.2 Vicarious Reinforcement

In this section, we examine whether it is possible for agents acting selfishly to develop an optimal strategy in the present multi-agent domain. As discussed above, even when agents are aware of the locations of other agents, the state transition probabilities vary over time due to changes in the behaviour of the other agents as they learn. Hence the AHC and Q-learning algorithms are no longer guaranteed to find an optimal policy. Another way of looking at this problem is that the state transition probabilities, from a single agent's perspective, depend on the current policies of the other agents, and this information is not contained in the agent's perception of the world, hence is not available to the learning procedure.

To better understand possible reasons that agents may not learn an optimal policy, we turn to ideas from game theory. Consider several agents in the world, each attempting to maximize its long-term reward. The system is in Nash equilibrium if no agent can improve its policy with respect to the policies of other agents. That is, if the policies of all other agents are fixed, each agent can do no better than its current policy. A system of learning agents will tend towards such a state, since whenever an agent can improve its policy, it will do so. However, when each agent adopts a strategy that produces globally optimal behaviour, the system may not be in equilibrium. This is because a globally optimal state might be one in which each agent sometimes sacrifices a personal reward in order that another agent benefits, so increasing the total reward obtained by the collection of agents. If such a situation is possible, a globally optimal state will not be an equilibrium because, fixing the policies of all other agents, a single selfish agent can improve its own policy by never sacrificing any personal reward, but still taking advantage of the sacrifices of others. Moreover, if every agent does this, the resultant system will not be globally optimal.

By way of illustration, consider the following scenario, which can occur in the present domain, where the circles are agents and the squares are food:

Let us follow the agents over the next four cycles. If the agents act selfishly, agent 1 will eat both pieces of food to its right before agent 2 can reach them, and agent 2 will not be able to eat any food. Therefore, two pieces of food get eaten overall. However, if agent 1 sacrifices the optimal personal reward and both agents consistently move to the left in these states, a total of three pieces of food will be eaten in four steps. Imagine that this game is repeated many times, with the agents alternating starting positions. Then, if both agents cooperate, they will on average eat $1\frac{1}{2}$ pieces of food each per repetition, while if both act selfishly they will on average eat only 1 piece of food each per repetition. Although each agent maximizes its reward if both cooperate, selfish agents will not consistently cooperate, as the globally optimal strategy is not an equilibrium. This type of scenario is analogous to the prisoner's dilemma. The main point is that for globally optimal strategies to occur, both agents must at times act non-selfishly, the result being that in the long run both agents benefit.

In order to encourage cooperation, we present the notion of *vicarious reinforcement*, where an agent is not only rewarded for eating food, but is also rewarded when an agent within its visual field eats food. In the above example, if agents learned to maximize the total reward received by both agents, the globally optimal strategy *would* be an equilibrium. To avoid the situation where agents learn to stay near other agents in order to receive additional reward, we calculate an agent's reinforcement signal as follows:

$$r = \frac{p \cdot r_p + \sum_{i=1}^{n} r_i}{p+n}$$

Here p is the agent's *personal weighting*, r_p is the personal reward due to the agent eating food, agents 1 to n are in the agent's visual field, and r_i is the personal reward received by agent i. The agent's personal weighting controls how much emphasis is placed on the agent's personal reward compared to that of other agents. For example, a personal weighting of 1 means that an agent treats its own reward as being of equal importance to that generated by each other nearby agent, whereas a personal weighting of 2 means that an agent views its personal reward as twice as important as that of each nearby agent. Assuming agents perform at about the same level, the denominator ensures that an agent does not benefit merely by staying close to another agent.

To test the effectiveness of vicarious reinforcement, we initially ran simulations of two AHC agents in a 9×9 world grid containing 8 food elements. Agents could see the whole world, i.e. they had visual fields of 9×9 squares. Both agents in each simulation had the same personal weightings, and all other simulation details were as for the simulations described in section 4.1.

Figure 8 plots the average performance of agents against the personal weighting used. Each point on the graph is an average over three simula-

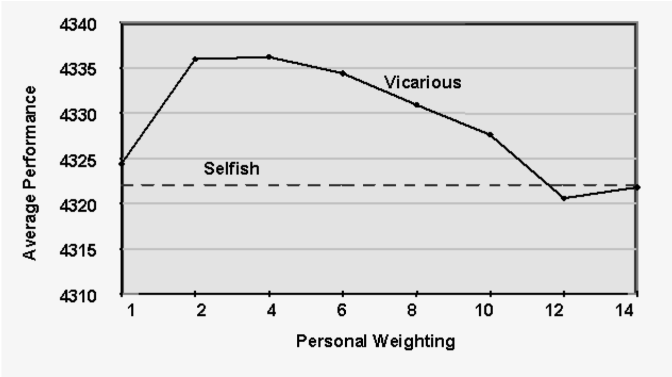

Figure 8: Average performance of agents with vicarious reinforcement with various personal weightings compared to selfish agents. There were 2 agents with 9×9 visual fields in a 9×9 static world.

tions. The graph also shows the performance of selfish agents, i.e. agents that receive no vicarious reinforcement. When the personal weighting was high, the performance was similar to that of the selfish agents. This is expected, as the higher the personal weighting, the more selfish the agent. For a personal weighting of 1, the performance seems to be slightly better than for selfish agents. However, the best performance seemed to occur for personal weightings between 2 and 6. It is not surprising that a personal weighting of 1 produced worse performance than slightly higher personal weightings; the agent only has direct control over its own actions, and so it is easier to learn if more weight is given to the direct results of these actions. The trend in this graph seems clear, however the absolute increase in performance is small even for the most beneficial personal weightings (around 15 extra food elements in 10 000 cycles).

To test the hypothesis that the agents with vacarious reinforcement perform better than selfish agents, we ran 15 simulations using agents with a personal weighting of 6, and 15 simulations using selfish agents. A one-tailed t-test strongly contradicts the hypothesis that the mean performances of the two types of agent are equal ($P < 0.0001$). This provides strong evidence that vicarious reinforcement with a personal weighting of 6 improves the performance of the agents in these simulations. This statistically significant advantage holds even when the world is dynamic. Figure 9 shows average performance for different vanishing rates. Each point on the graph represents 15 simulations, and for each vanishing rate a one-tailed t-test strongly contradicted the hypothesis that the mean performances of the selfish and agents with vicarious reinforcement were equal ($P < 0.0001$ in all cases).

In the above simulations, the agents perceived the entire world. We also tested the effect of vicarious reinforcement for the more realistic case of a partially observable world. Simulations were run in a static 18×18 world grid containing 32 food elements (maintaining the same food density as for previous simulations), and the agents had 9×9 visual fields. Note that vicarious reinforcement is received only from other agents within an agent's visual field. Figure 10 plots the average performance of both selfish and vicarious AHC agents against the number of agents in the world. Agents with vicarious reinforcement had a personal weighting of 6, and once again, all agents in a given simulation were of the same type. The graph shows that the agents with vicarious reinforcement consistently performed better than

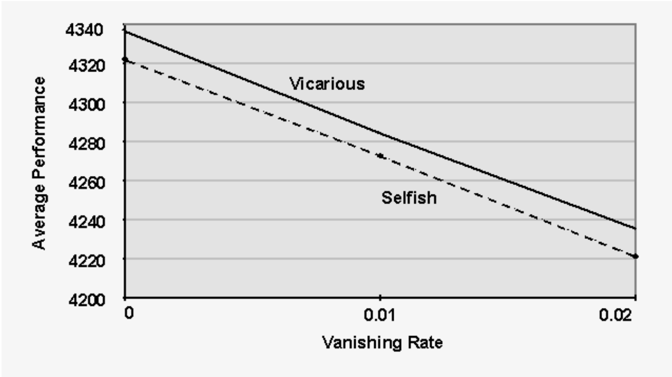

Figure 9: Average performance of agents with vicarious reinforcement compared to selfish agents for various vanishing rates. There were 2 agents with 9×9 visual fields in a 9×9 dynamic world.

the selfish agents. Again, the absolute difference in performance is small, but the direction in which they differ is consistent. A one-tailed binomial test shows that over this range of settings, the agent with vicarious reinforcement is superior ($P = 0.02$). The difference in performance appears to decrease as the number of agents in the world becomes large. If this trend is significant, it may be due to the fact that when many agents are present, less weight is given to an agent's personal reward compared to the total reward of others. If this is so, increasing an agent's personal weighting when many agents are present may further enhance performance.

The results presented in this section show that vicarious reinforcement improves performance in the present domain. We believe that the reasons behind the improvement are general to multi-agent reinforcement learning problems with spatially distributed agents where conflict avoidance improves performance. However, the degree of improvement in this domain was small, and there appear to be two possible reasons for this. First, this domain may not require a large degree of cooperative behaviour in order for agents to perform well. Second, it may be that the single layer neural networks used to represent the various functions are not capable of learning the functions required for more cooperative behaviour.

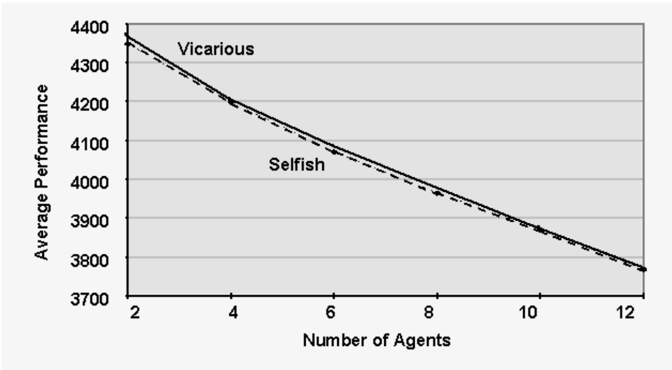

Figure 10: Average performance of agents with vicarious reinforcement compared to selfish agents with 9×9 visual fields in a static 18×18 world for various numbers of agents.

5 Conclusion

This work presents the idea of multi-agent reinforcement learning with vicarious rewards to address the lack of guarantee of convergence of standard dynamic programming algorithms due to the presence of multiple agents. Experiments were conducted in a simple dynamic domain based on the Tileworld, and both the Adaptive Heuristic Critic (AHC) and Q-learning algorithms were used, with single layer feed-forward artificial neural networks for representing the various functions required by these algorithms, thus enabling generalization between states.

An initial series of experiments validated the experimental framework. It was shown that in a static world containing a single agent, both the AHC and Q-learning agents learned to perform substantially better than a hand-coded Greedy agent. When the learning agents could perceive the locations of nearby agents in the world, Q-learning agents seemed to perform as well as Greedy agents, while AHC agents performed better. In a slightly dynamic world, Q-learning agents generally performed better than Greedy agents, and AHC agents performed best of all. Thus AHC agents were better suited than Q-learning agents to this domain. This may be due to the fact that the AHC algorithm learns a single value function rather than separate value functions for each action. When generalizing over a large number of states, this may be an advantage, as information from every experience in the world contributes to the single value function estimate.

In general, it is not sufficient for each agent to act selfishly in order to find globally optimal strategies in multi-agent systems. We showed experimentally that AHC agents learning with vicarious reinforcement perform better than selfish agents in this domain. This is true in static as well as dynamic worlds, for various numbers of agents, and irrespective of whether the agents can observe the entire world. Learning with vicarious rewards works in this domain because of two important factors: (i) the measure of overall performance is the average (or equivalently, the total) performance of the group members, and (ii) the spatial distribution of agents means that the effect on the utility functions of the actions of "distant" agents is much less than that of "nearby" agents. Under these conditions, the use of vicarious rewards means that the overall system better approximates the standard single-agent reinforcement learning problem than the system consisting purely of selfish agents with only local reinforcement.

Acknowledgements

This work was carried out in the Basser Department of Computer Science at the University of Sydney. We would like to thank Malcolm Clark and Ed Golja for their collaboration on the development of the Tileworld simulator, and Les Irwig for advice on statistical techniques. Thanks also to Richard Sutton for making available the code for the learning algorithms. The first author was supported by a scholarship from Object-Oriented Pty. Ltd. The second author was supported by the Australian Research Council.

References

[1] Ackley, D.H. & Littman, M.L. (1990) 'Generalization and Scaling in Reinforcement Learning.' in Touretzky, D.S. (Ed.) *Advances in Neural Information Processing Systems 2.* Morgan Kaufmann, San Mateo, CA.

[2] Barto, A.G., Sutton, R.S. & Anderson, C.W. (1983) 'Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems.' *IEEE Transactions on Systems, Man, and Cybernetics (SMC-13)*, 834–846.

[3] Bellman, R.E. (1957) *Dynamic Programming.* Princeton University Press, Princeton, NJ.

[4] Bonde, A. & Sutton, R.S. (1993) ftp://ftp.cs.umass.edu/pub/anw/pub/sutton/td-backprop-pseudo-code

[5] Clark, M.P., Irwig, K.G. & Wobcke, W.R. (1997) 'Emergent Properties of Teams of Agents in the Tileworld.' in Cavedon, L., Rao, A.S. & Wobcke, W.R. (Eds) *Intelligent Agent Systems.* Springer-Verlag, Berlin.

[6] Kaelbling, L.P., Littman, M.L. & Moore, A.W. (1996) 'Reinforcement Learning: A Survey.' *Journal of Artificial Intelligence Research*, **4**, 237–285.

[7] Kinny, D.N. & Georgeff, M.P. (1991) 'Commitment and Effectiveness of Situated Agents.' *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, 82–88.

[8] Ono, N. & Fukumoto, K. (1996) 'Multi-agent Reinforcement Learning: A Modular Approach.' *Proceedings of the Second International Conference on Multi-Agent Systems*, 252–258.

[9] Rumelhart, D.E., Hinton, G.E. & Williams, R.J. (1986) 'Learning Internal Representations by Error Propagation.' in Rumelhart, D.E., McClelland, J.L. & the PDP Research Group (Eds) *Parallel Distributed Processing. Volume 1.* MIT Press, Cambridge, MA.

[10] Sandholm, T.W. & Crites, R.H. (1996) 'On Multiagent Q-Learning in a Semi-competitive Domain.' in Weiss, G. & Sen, S. (Eds) *Adaption and Learning in Multi-Agent Systems.* Springer-Verlag, Berlin.

[11] Sutton, R.S. (1988) 'Learning to Predict by the Methods of Temporal Differences.' *Machine Learning*, **3**, 9–44.

[12] Sutton, R.S. (1990) 'First Results with Dyna, an Integrated Architecture for Learning, Planning and Reacting.' in Miller, III, W.T., Sutton, R.S. & Werbos, P.J. (Eds) *Neural Networks for Control.* MIT Press, Cambridge, MA.

[13] Tan, M. (1993) 'Multi-Agent Reinforcement Learning: Independent vs. Cooperative Agents.' *Proceedings of the Tenth International Conference on Machine Learning*, 330–337.

[14] Watkins, C.J.C.H. (1989) *Learning with Delayed Rewards.* Ph.D. Thesis, Department of Experimental Psychology, University of Cambridge.

[15] Whitehead, S.D. (1991) 'A Complexity Analysis of Cooperative Mechanisms in Reinforcement Learning.' *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, 607–613.

[16] Williams, R.J. & Baird, III, L.C. (1993) 'Analysis of Some Incremental Variants of Policy Iteration: First Steps Toward Understanding Actor-Critic Learning Systems.' Technical Report NU-CCS-93-14, College of Computer Science, Northeastern University.

Appendix. Learning Algorithms

In this pseudo-code, variables in **bold** are vectors, whereas those in *italic* have scalar values. The function get_current_state() returns the current state of the world as perceived by the agent. The function move(*action*) moves the agent in the world according the specified action, and returns the external scalar reward received as a result of the action. The function neural_net(**input**) is assumed to return the output vector of the relevant neural network given the input vector supplied as the argument. The function back-prop(**errors**, **net-input**, **net-output**) is assumed to back-propagate errors and modify the weights accordingly.

```
ahc_agent()
    state = get_current_state();
    // reward r is undefined the first time these are called
    state-eval = critic_net(state, r);
    action = action_net(state, state-eval, r);
    r = move(action);

critic_net(current-state, r)
    // Input: current-state and reward received at this state
    // Returns: estimation of the value of this state
    if (t > 0)
        y = neural_net(current-state);  // evaluation of current state
        e = r + γ·y − oldy;             // TD error for previous state
        back_prop(e, oldstate, oldy);   // improve value estimation
    oldy = neural_net(current-state);   // new value of current state
    oldstate = current-state;
    return(oldy);                       // return value of current state

action_net(current-state, eval, r)
    // Input: current-state, value of this state and reward received
    // Returns: integer representing which action to perform
    if (t > 0)
        ρ = r + γ·eval − oldeval;       // reinforcement for previous move
        // generate errors according to CRBP heuristic
        if (ρ > 0)
            e = o − y;
        else
            e = η·(1 − o − y);
        back_prop(e, oldstate, y);
    y = neural_net(current-state);      // output of action network
    oldeval = eval;
    o = form_action(y);                 // form vector probabilistically
    return(i such that o_i = 1);        // return action to take
```

```
q-learning_agent()
    state = get_current_state();
    for all i in (N, S, E, W)
        q[i] = q-net[i].neural_net(state);  // calculate Q-values
    action = pick_action(q);                // using Boltzmann distribution
    r = move(action);
    if (t > 0)
        // learn from previous action
        q-net[oldaction].learn(r, max_i(q[i]), oldstate, oldq[oldaction]);
        q[oldaction] = q-net[oldaction].neural_net(oldstate);
    oldaction = action;
    oldq = q;
    oldstate = state;

learn(r, q, state, oldq)
    // Input: reward, current q estimate, previous state and old q estimate
    e = r + γ·q − oldq;                     // Q-learning update rule
    back_prop(e, state, oldq);
```