W-learning: Competition among selfish Q-learners *

Mark Humphrys
University of Cambridge, Computer Laboratory
http://www.cl.cam.ac.uk/users/mh10006 †

April 1995

Abstract

W-learning is a self-organising action-selection scheme for systems with multiple parallel goals, such as autonomous mobile robots. It uses ideas drawn from the subsumption architecture for mobile robots (Brooks), implementing them with the Q-learning algorithm from reinforcement learning (Watkins). Brooks explores the idea of multiple sensing-and-acting agents within a single robot, more than one of which is capable of controlling the robot on its own if allowed. I introduce a model where the agents are not only autonomous, but are in fact engaged in direct competition with each other for control of the robot. Interesting robots are ones where no agent achieves total victory, but rather the state-space is fragmented among different agents. Having the agents operate by Q-learning proves to be a way to implement this, leading to a local, incremental algorithm (W-learning) to resolve competition. I present a sketch proof that this algorithm converges when the world is a discrete, finite Markov decision process. For each state, competition is resolved with the most likely winner of the state being the agent that is most likely to suffer the most if it does not win. In this way, W-learning can be viewed as 'fair' resolution of competition. In the empirical section, I show how W-learning may be used to define spaces of agent-collections whose action selection is learnt rather than hand-designed. This is the kind of solution-space that may be searched with a genetic algorithm.

Keywords: mobile robots, subsumption architecture, action selection, reinforcement learning, Q-learning, multi-module learning, genetic algorithms

1 Autonomous mobile robots

Recent years have seen a new approach to the attempt to build autonomous mobile robots. The new approach has been called *behavior-based* AI, emphasizing intelligence as emerging from ongoing interaction with the world.

*This is University of Cambridge Computer Laboratory technical report no.362, available from http://www.cl.cam.ac.uk/users/mh10006/publications.html (or direct from ftp://ftp.cl.cam.ac.uk/papers/reports/TR362-mh10006-w-learning.ps.gz)

†postal address: University of Cambridge, Computer Laboratory, New Museums Site, Pembroke St., Cambridge CB2 3QG, England. tel: +44 1223 335443. fax: +44 1223 334678 or 334679. email: Mark.Humphrys@cl.cam.ac.uk

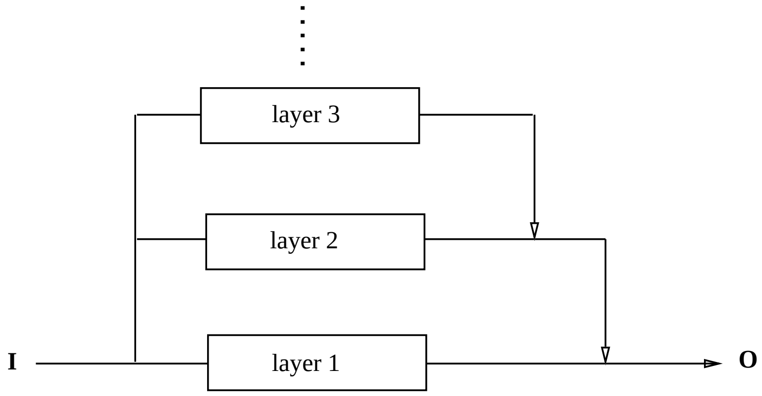

Figure 1: Brooks' horizontal subsumption architecture.

1.1 The subsumption architecture

Brooks [Brooks, 1986, Brooks, 1991] introduces an architecture for building autonomous mobile robots which he calls the *subsumption architecture* (Figure 1).

He builds in layers: layer 1 is a simple complete working system, layers 1-2 together form a complete, more sophisticated system, layers 1-3 together form a complete, even more sophisticated system, and so on. Lower layers do not depend on the existence of higher layers, which may be removed without problem. Higher layers may interfere with the data flow in layers below them - they may 'use' the layers below them for their own purposes. If a higher layer doesn't interfere, lower layers just run as if it wasn't there. The subsumption architecture develops some interesting ideas:

- The concept of default behavior. e.g.the 'Avoid All Things' layer 1 takes control of the robot by default whenever the 'Look For Food' layer 2 is idle.
- Multiple parallel goals. There are multiple candidates competing to be given control of the robot, e.g. control could be given to layer 1, which has its own purposes, or to layer 5, which has different purposes (and may use layers 1-4 to achieve them). Which to give control to may not be an easy decision - one can imagine goals which are directly-competing peers. Multiple parallel goals are seen everywhere in nature, e.g.the conflict between feeding and vigilance in any animal with predators.
- The concept of multiple independent channels connecting sensing to action. Brooks breaks with the traditional AI paradigm of having a 'perception' subsystem, whose job it is to deliver a representation of the world to some central symbolic module where all the 'real' intelligence resides. Instead, he has multiple sensing-to-action channels, working in parallel, each processing sensory information in different ways for its own purposes, some crude, some sophisticated.

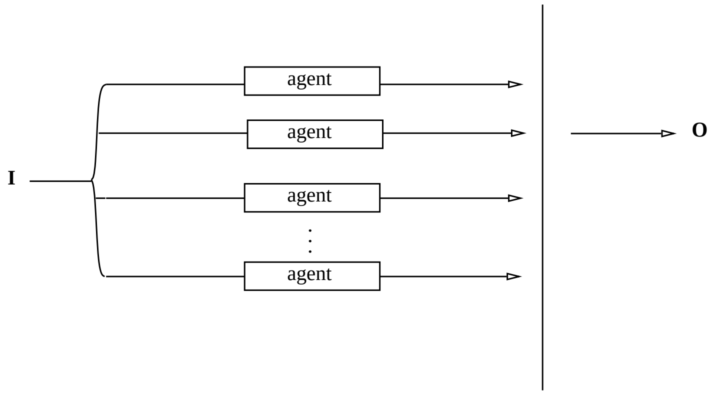

Figure 2: Competition among selfish peer agents in a horizontal architecture. Each agent suggests an action, but only one action is executed. Which agent is obeyed changes dynamically.

1.1.1 The action selection problem

With multiple adaptive sensing-and-acting modules, we have the problem of *action selection*. Modules will suggest actions for the robot, in line with their own perhaps conflicting goals. The robot must decide which action to select for actual execution. Brooks liberates his modules by giving them full sensing-and-acting powers, but does not go as far as letting them compete for control. Instead, action-selection is a job for the programmer.

Brooks' original scheme has been extended (see survey in [Brooks, 1994]), and other schemes have been proposed [Maes, 1989, Blumberg, 1994, Sahota, 1994], but action-selection remains something basically designed rather than self-organised. In an attempt to avoid this problem of design, I introduce a model in which yet further liberation of modules is attempted.

1.2 Competition among selfish agents

I introduce a model (Figure 2) with the following features:

- Make the layers peers, so that each can function in the absence of *all* the others. Now they are fully autonomous sensing-and-acting *agents* [Minsky, 1986], [1] not ordered in any hierarchy, but rather in a loose *collection*.
- Have them compete for control, having to make a case that they should be given it. One will *win*, having its action executed, then they will compete again for the next action to be executed, and so on indefinitely.

[1] I use the word *agent* to emphasise that each is a full autonomous actor in the world - if left alone with a robot body to implement its actions. Under a stricter definition one might claim that the robot is the only agent here, with varying software inside it. For somewhat-autonomous, somewhat-competing modules within a single physical robot, [Brooks, 1986] uses *layer* (though [Brooks, 1994] also uses *process*), [Minsky, 1986] uses *agent* (even though most of his agents do not interact directly with the world), [Blumberg, 1994] uses *activity* and [Sahota, 1994] uses *behavior*.

To be more precise, let the collection consist of agents $A_1, \ldots, A_n$. Time steps are discrete. Each time step, the robot observes the world to be in some state x. Each agent A_i suggests an action $a_i(x)$ that it wants to see executed in this state. The robot chooses precisely one of these actions $a_k(x)$ and executes it. From an agent's point of view, it tells the robot what to do and the robot chooses either to *obey* it or not.

At all times agents are engaged in pursuing their own separate goals. There is no co-operation at all - these are *selfish* agents. Agents have no explicit knowledge of the existence of any other agents. Each, if you like, believes itself to be the entire nervous system. [2]

This model will work if we can find some natural way of resolving competition so that the 'right' agent wins. The basic idea is that an agent will always have an action to suggest (being a complete sensing-and-acting machine), but it will 'care' some times more than others. When no predators are in sight, the 'Avoid Predator' agent will continue to generate perhaps random actions, but it will make no difference to its purposes whether these actions are actually executed or not. When a predator does come into sight however, the 'Avoid Predator' agent must be listened to, and given priority over the default, background agent, 'Wander Around Eating Food'.

A simple scheme would be one where each agent suggests its action with a strength (or *Weight*) W, expressing how important it is to their purposes that they be obeyed at this moment, and the robot executes the action that comes with the largest W (Figure 3).

To be precise, each agent A_i maintains a table of *W-values* $W_i(x)$. Given a state x, each agent A_i suggests some action $a_i(x)$ with weight $W_i(x)$, The robot executes action $a_k(x)$ where:

$$W_k(x) = \max_{i \in 1, \ldots, n} W_i(x)$$

We call A_k the *leader* in the competition for state x at the moment, or the *owner* of x at the moment.

For example, in Figure 3, when the robot is not carrying food, the 'Food' agent tends to be obeyed. When the robot is carrying food, the 'Hide' agent tends to be obeyed. The two agents combine to produce a food-foraging robot.

Note that the 'Hide' agent goes on suggesting the same action with the same W in all situations. It just wants to hide all the time - it has no idea why sometimes it is obeyed, and other times it isn't.

We can draw a map of the state-space, showing for each state x, which agent succeeds in getting its action executed. Clearly, a robot in which one agent achieves total victory (wins the whole state-space) is not very interesting. It will be no surprise what the robot's behavior will be then - it will be the behavior of the agent alone. Rather, interesting robots are ones where the state-space is fragmented among different agents (Figure 4).

As in Brooks' model, interesting robots are ones where control passes to different agents at different times. But unlike Brooks, *no* agent is explicitly aware of the existence of any other. An agent can still 'use' another agent, but

[2] Perhaps there is some evolutionary plausibility in this. Consider that the next step after getting a simple sensor-to-effector link working is not a hierarchy or a co-operative architecture but rather a mutation where, by accident, two links are built, and each of course tries to work the body as if it were alone.

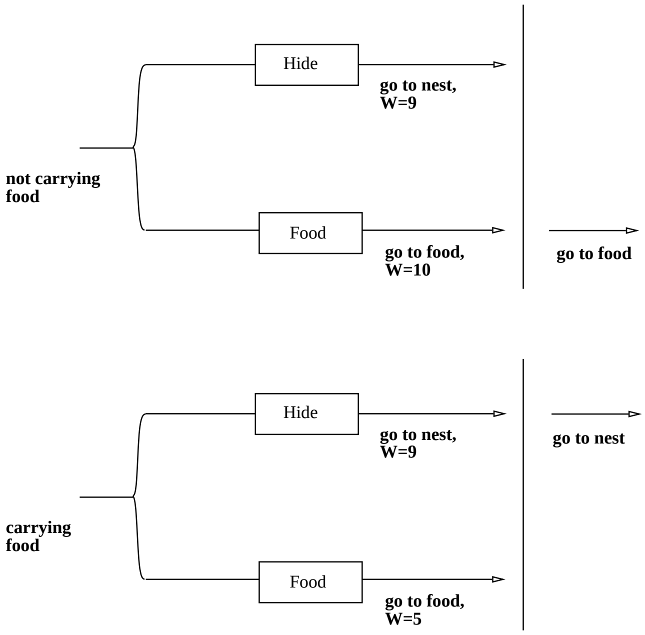

Figure 3: Competition is decided on the basis of W-values. The action with the highest W-value is executed by the robot.

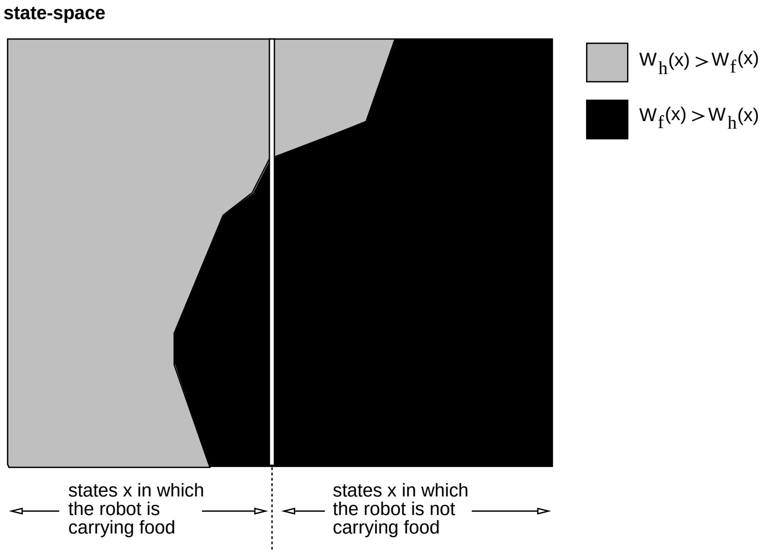

Figure 4: We expect competition to result in fragmentation of the state-space among the different agents. In each state x, the 'Hide' agent suggests some action with weight $W_h(x)$, and the 'Food' agent suggests an action with weight $W_f(x)$. The grey area is the area where $W_h(x) > W_f(x)$, that is, the 'Hide' agent wins all these states. The black area shows the states that the 'Food' agent wins.

not by being explicitly aware of its existence - rather by learning to cede control of appropriate areas of state-space (which the other agent will take over).

1.2.1 W-values as action selection

This is a *winner-take-all* action selection scheme - that is, the winner gets its exact action executed (as opposed to a scheme where the actions of agents are merged).

Blumberg [Blumberg, 1994] uses information sharing between agents, which may lead to a compromise action. Here there is no explicit information sharing.

Also, the division of control is state-based rather than time-based. Blumberg argues the need for a model of fatigue, where a switch of activity becomes more likely the longer an activity goes on. He points out that animals sometimes appear to engage in a form of time-sharing. It is not clear however, that these effects cannot be achieved by a suitable state representation x. If an activity goes on for long enough, some internal component of x (that is, some internal sense, e.g.'hunger') may change, leading to a new x and a potential switch in activity.

For example, consider the conflict between feeding and body maintenance (discussed by Blumberg). Some action selection schemes assign priorities to entire activities, and then worry about how low-priority cleaning is ever going to be able to interrupt feeding. In our scheme, let $x = (e, i)$ be the state, where e is information from external sensors and $i = (f, c)$ is information from internal sensors, where f takes values 2 (very hungry), 1 (hungry) and 0 (not hungry) and c takes values 2 (very dirty), 1 (dirty) and 0 (clean). The 'Food' agent suggests actions with weight $W_f(x)$. The 'Clean' agent suggests actions with weight $W_c(x)$. We should find that for a given e, c:

$$W_f((e,(2,c))) > W_f((e,(1,c))) > W_f((e,(0,c)))$$

and for a given e, f:

$$W_c((e,(f,2))) > W_c((e,(f,1))) > W_c((e,(f,0)))$$

A very strong 'Food', only rarely interrupted by 'Clean', would be represented by, for a given e:

$$\begin{aligned} W_f((e,(2,c))) &> W_f((e,(1,c))) > W_c((e,(f,2))) \\ &> W_f((e,(0,c))) > W_c((e,(f,1))) > W_c((e,(f,0))) \end{aligned}$$

Minsky [Minsky, 1986] warns that too simple forms of state-based switching will be unable to engage in *opportunistic behavior*. His example is of a hungry and thirsty animal. Food is only found in the North, water in the South. The animal treks north, eats, and as soon as its hunger is only partially satisfied, thirst is now at a higher priority, so it starts the long trek south before it has satisfied its hunger. Even before it has got south, it will be starving again. One solution to this would be time-based, where agents get control for some minimum amount of time.

Again, however, time-based switching is not the only answer. Opportunistic behavior is possible with a more sophisticated form of state-based switching, where agents can tell the difference between situations when they are likely to get an immediate payoff and situations when they could only *begin* some sequence of actions which will lead to a payoff later. In Minsky's example, where the thirst agent presents W-values $W_t(x)$, e_0 means no water visible, e_1 means water visible, and i_2 means very thirsty, we would like to see:

$$W_t((e_0, i_2)) < W_t((e_1, i_2))$$

It is not just a vain hope that W-values will be organised like this. As we will see, the agents we are going to use *can* tell the difference between immediate and distant likely payoff, and will present different W-values accordingly.

For agents to be able to generate their own W-values, we need a scheme whereby they attach some kind of numerical 'fitness' value to the actions they wish to take. Previous work in action selection has regarded assigning such values as a problem of design. In the literature, one sees formulas taking weighted sums of various quantities in an attempt to estimate the utility of actions.

In fact, there is a way that these utility values can come for free. Learning methods that automatically assign values to actions are common in the field of reinforcement learning.

2 Reinforcement Learning

A reinforcement learning (RL) agent senses a world, takes actions in it, and receives rewards and punishments from some reward function based on the consequences of the actions it takes. By trial-and-error, the agent learns to take the actions which maximise its rewards.

2.1 Q-learning

Watkins [Watkins, 1989] introduces a method of reinforcement learning which he calls *Q-learning*.

The agent exists within a world that can be modelled as a Markov decision process (MDP). It observes discrete states of the world x ($\in X$, a finite set) and can execute discrete actions a ($\in A$, a finite set). Each discrete time step, it observes state x, takes action a, observes new state y, and receives immediate reward r. Transitions are probabilistic, that is, y and r are drawn from stationary probability distributions $P_{xa}(y)$ and $P_{xa}(r)$, where $P_{xa}(y)$ is the probability that doing a in x will lead to state y and $P_{xa}(r)$ is the probability that doing a in x will generate reward r.

Here, rewards r are bounded by r_{max},r_{min}, where $r_{min} < r_{max}$ ($r_{min} = r_{max}$ would be a system where the reward was the same no matter what action was taken. The agent would always behave randomly and would be of no use or interest).

2.1.1 The task

The agent acts according to a policy π which tells it what action to take in each state x.

The agent is not interested just in immediate rewards, but in the *total discounted reward*. In this measure, rewards received n steps into the future are worth less than rewards received now, by a factor of γ^n where $0 \leq \gamma < 1$:

$$R = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

The *discounting factor* γ defines how much expected future rewards affect decisions now. The expected total discounted reward if we follow policy π forever, starting from x_t, is:

$$\begin{aligned} V^\pi(x_t) = E(R) \quad &= E(r_t) + \gamma E(r_{t+1}) + \gamma^2 E(r_{t+2}) + \cdots \\ &= E(r_t) + \gamma \left[E(r_{t+1}) + \gamma E(r_{t+2}) + \gamma^2 E(r_{t+3}) + \cdots \right] \\ &= E(r_t) + \gamma V^\pi(x_{t+1}) \\ &= \sum_r r P_{x_t a_t}(r) + \gamma \sum_y V^\pi(y) P_{x_t a_t}(y) \end{aligned}$$

$V^\pi(x)$ is called the *value* of state x under policy π. The problem for the agent is to find an optimal policy - one that maximises the total discounted expected reward (there may be more than one policy that does this).

2.1.2 The strategy

The strategy that the Q-learning agent adopts is to build up *Quality-values* (Q-values) for each pair (x, a). In 1-step Q-learning, after each experience, we update:

$$Q(x,a) := (1-\alpha) Q(x,a) + \alpha (r + \gamma \max_{b \in A} Q(y,b)) \tag{1}$$

where the *learning rate* α, $0 \leq \alpha \leq 1$, takes decreasing (with each update) successive values $\alpha_1, \alpha_2, \alpha_3 \ldots$, such that $\sum_{i=1}^{\infty} \alpha_i = \infty$ and $\sum_{i=1}^{\infty} \alpha_i^2 < \infty$. Each pair (x, a) has its own learning rate $\alpha = \alpha(x, a)$. See Lemma B.1.1 (in appendix) for an illustration of how α works.

Note that if the conditions hold, then for any t, $\sum_{i=t}^{\infty} \alpha_i = \infty$ and $\sum_{i=t}^{\infty} \alpha_i^2 < \infty$, so α may start anywhere along the sequence. That is, α may take successive values $\alpha_t, \alpha_{t+1}, \alpha_{t+2} \ldots$ (see Lemma B.1.2).

The scheme used in this work is, where $n(x,a) = 1, 2, 3, \ldots$ is the number of times $Q(x, a)$ has been visited:

$$\begin{aligned} \alpha(x,a) \quad &= \frac{1}{n(x,a)} \\ &= 1, \frac{1}{2}, \frac{1}{3}, \ldots \end{aligned}$$

Since the rewards are bounded, it follows that the Q-values are bounded (Lemma A.2.1).

2.1.3 Convergence of Q-learning

If each pair (x, a) is visited an infinite number of times, then Q-learning converges to a unique set of values $Q(x, a) = Q^*(x, a)$ which define a stationary deterministic optimal policy [Watkins and Dayan, 1992]. Q-learning is asynchronous and sampled - each $Q(x, a)$ is updated one at a time, and the control policy may visit them in any order, so long as it visits them an infinite number of times.

The agent will maximise its total discounted expected reward if it always takes the action with the highest Q^*-value. That is, the optimal policy π^* is defined by $\pi^*(x) = a^*(x)$ where:

$$\begin{aligned} V^*(x) \quad &= Q^*(x, a^*(x)) \\ &= \max_{b \in A} Q^*(x,b) \end{aligned}$$

2.1.4 The control policy for Q-learning

In real life, since we cannot visit each (x,a) an infinite number of times, we can only approximate Q-learning. We could do a large finite amount of random exploration, then exploit our knowledge. But a more practical control policy gets the agent acting adaptively as soon as possible, by starting with high exploration and decreasing it to nothing as time goes on, so that after a while we are only exploring (x,a)'s that have worked out at least moderately well before.

The control policy used in these experiments is a fairly standard one in the field, and was originally suggested by Sutton. The agent suggests actions probabilistically based on their Q-values using a Boltzmann (*soft max*) distribution. Given a state x, it suggests action a with probability:

$$P_x(a) = \frac{e^{\frac{Q(x,a)}{T}}}{\sum_{b \in A} e^{\frac{Q(x,b)}{T}}}$$

The *temperature* T controls the amount of exploration (the probability of suggesting actions other than the one with the highest Q-value). T starts high and declines to zero as time goes on.

2.1.5 The multi-module problem

Most work in RL has focused on single agents, for which a well-developed theory is now in place. In theory, any problem can be seen as just another I/O mapping to be learnt by a single agent. In practice though, the learning methods do not scale up indefinitely, and it is recognised that methods are needed to combine simple agents to solve complex tasks.

A *top-down* approach to multi-module RL systems involves identifying the task and decomposing it into subtasks, each of which can be solved by a single RL agent. Moore [Moore, 1990] does this by hand, but it is clear that this is only feasible with certain problems. Singh [Singh, 1992], and later Tham and Prager [Tham and Prager, 1994], *learn* the decomposition, but only in a class of problems where subtasks combine sequentially to solve the main task. In autonomous mobile robots (and many other systems) we are also likely to be interested in subtasks acting in parallel, and interrupting each other rather than running to completion.

Lin [Lin, 1993] learns the decomposition for potentially parallel, non-terminating subtasks. Each agent learns a different $Q_i(x,a)$ to solve a subtask, and the robot learns $Q(x,i)$, where i is which agent to choose in state x. A problem here is that we have to design a global reward function. While clearly a reward can be given on completion of the global task, interim rewards may be hard to design (similar to the problem of just getting a single monolithic agent to learn the whole thing). To do the job with local reward functions only, we must adopt a *bottom-up* approach.

A bottom-up approach studies the behavior that emerges when multiple RL agents are combined in different ways. Tan [Tan, 1993] studies the benefits of co-operation among agents, where each agent has their own body to control in the world. He focuses on communication of information among agents (analogous, say, to the communication of information among ants). Co-operation is based on explicit interactions with other agents. These interactions are designed in a similar fashion to the design of the agent's sensors.

This type of co-operation is designed in. A more interesting type of co-operation is the involuntary type, which emerges from competition among agents that have a limited ability to get their own way, such as in the model introduced in Section 1.2. This model is a community of agents, without global control, where each makes their own adjustment to the presence of the others, based only on their local needs. The aim is to study the behavior that emerges when they compete - which may or may not involve studying how they solve some (global) task perceived by an observer (and not perceived by the agents).

The existence of numerical Quality-values for actions suggests that using Q-learning agents will be a way to implement the model. A Q-learning agent may not be able to explain *why* it wants to do something, but it certainly knows *how much* it wants to do it (which is something that does not come naturally with a designed agent).

2.2 Competition among selfish Q-learners

I use Q-learning as the mode of operation of the competing selfish agents in my model. Each agent is a Q-learning agent, with its own set of Q-values and more importantly, with its own reward function.

To formalise, each agent A_i receives rewards r_i from a personal distribution $P^i_{xa}(r)$. The distribution $P_{xa}(y)$ is a property of the world - it is common across all agents. Each agent A_i maintains personal Q-values $Q_i(x,a)$ and W-values $W_i(x)$. Given a state x, it suggests an action a_i according to a control policy as in Section 2.1.4, operating on $Q_i(x,a)$. a_i is most likely, as time goes on, to be such that:

$$Q_i(x,a_i) = \max_{b \in A} Q_i(x,b)$$

The robot works as follows. Each time step:

observe x
for (all agents):
  get suggested action a_i with strength $W_i(x)$
find $W_k(x) = \max_{i \in 1,\dots,n} W_i(x)$
execute a_k
observe y
for (all agents):
  get reward r_i
  update Q and/or W

Note that the transition will generate a different reward r_i for each agent. For updating Q, we use normal Q-learning. For updating W, we want somehow to make use of the numerical Q-values. There are a number of possibilities.

2.3 Static W-values

A static measure of W is one which depends only on the agent, not on the collection in which it finds itself. The agent will promote its action with the same strength no matter what (if any) its competition. For example, where a is the suggested action:

$$W(x) = Q(x,a)$$

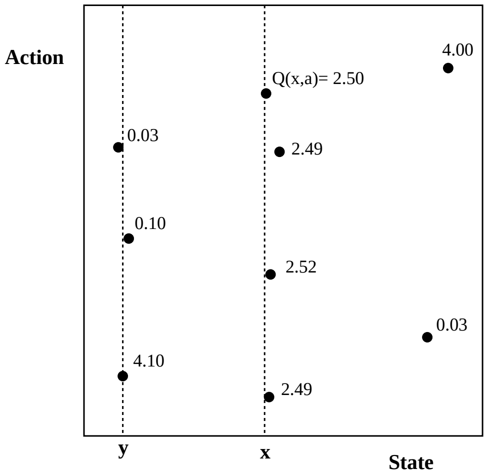

Figure 5: The concept of *importance*. State x is a relatively unimportant state for the agent (no matter what action is taken, the discounted reward will be much the same). State y is a relatively important state (the action taken matters considerably to the discounted reward).

2.3.1 W = importance

A more sophisticated W would represent the difference between taking the action and taking other actions, i.e. how *important* this state is for the agent. The assumption here is that if the agent's W is low and it is not obeyed, some other agent will be, and some other action will be executed. The concept of importance is illustrated in Figure 5.

For example, W could be the difference between a and the worst possible action:

$$W(x) = Q(x, a) - \min_{b \in A} Q(x, b)$$

2.4 Dynamic (learnt) W-values

The problem with a static measure of W is that it fails to take into account what the other agents are doing. If agent A_i is not obeyed, the actions chosen will not be random - they will be actions desirable to other agents. It will depend on the particular collection what these actions are, but they may overlap in places with its own suggested actions. If another agent happens to be promoting the same action as it, then it does not need to be obeyed. Or more subtly, the other agent might be suggesting an action which is almost-perfect for it, while if its exact action succeeded, it would be disastrous for the other agent, which would fight it all the way.

We have two types of states that A_i need not compete for:

Type 1 - A state which is relatively unimportant to it. It doesn't matter much to A_i's discounted reward what action is taken here. In particular, it doesn't matter if some other agent A_k takes an action instead of it.

Type 2 - A state in which it does matter to A_i what action is taken, but where some agent A_k just happens to be suggesting an action which is good for A_i. This may or may not be the action A_i would itself have suggested.

2.4.1 $\mathbf{W} = (\mathcal{P} - \mathcal{A})$

What we really need to express in W is not how important the action is but *what happens when we are not obeyed.* Rather than importance, W should express the difference between predicted reward $\mathcal{P}$ (what is predicted if we are listened to) and actual reward $\mathcal{A}$ (what actually happened). What happens when we are not listened to depends on what the other agents are doing.

Using such a measure of W, an agent will not need explicit knowledge about who it is competing with. It need only have local knowledge - what state x we were in, what action a it suggested, whether it was obeyed or not, what state y we went to, and what reward r that gave it. It will be aware of its competition only indirectly, by the interference they cause. It will be aware of them when they stop its action being obeyed, and will be aware of the y and r caused as a result. The agent will learn W by experience - by actually experiencing what the other agents want to do.

We do not want any global controller analysing all agents and setting their W-values. We want the agents setting their own W-values in an incremental way, using only local information (just like Q-learning). The following algorithm provides such a way.

3 W-learning

Consider Q-learning as the process:

$$\mathcal{P} := (1 - \alpha_Q)\mathcal{P} + \alpha_Q(\mathcal{A})$$

Then W-learning is:

$$W := (1 - \alpha_W)W + \alpha_W(\mathcal{P} - \mathcal{A})$$

For updating the Q-values, only one agent (the leader A_k) suggested the executed action a_k. However, all agents can learn from the transition (under their own different reward functions). [3] We update for all i:

$$Q_i(x, a_k) := (1 - \alpha_Q)Q_i(x, a_k) + \alpha_Q(r_i + \gamma \max_{b \in A} Q_i(y, b)) \tag{2}$$

where $\alpha_Q = \alpha_Q(x, a_k)$. We can do this because Q-learning is asynchronous and sampled (we can learn from the single transition, no matter what came before and no matter what will come after).

For the W-values, we only update the agents that were not obeyed. We update for $i \neq k$:

$$W_i(x) := (1 - \alpha_W)W_i(x) + \alpha_W(Q_i(x, a_i) - (r_i + \gamma \max_{b \in A} Q_i(y, b))) \tag{3}$$

[3] Although communicating a_k to each agent does somewhat compromise the model of agents having local knowledge only.

where $\alpha_W = \alpha_W(x)$ takes successive values $1, \frac{1}{2}, \frac{1}{3}, \ldots$. The reason why we do not update $W_k(x)$ is explained later (Section 4.4). In (object-oriented) pseudo-code, the W-learning system is, every time step:

```
state x := observe();
for ( all i )
 a[i] := A[i].suggestAction(x);
find k
execute ( a[k] );
state y := observe();

for ( all i )
{
 r[i] := A[i].reward(x,y);
 A[i].updateQ ( x, a[k], y, r[i] );
 if (i != k)
  A[i].updateW ( x, a[i], y, r[i] );
}
```

Alternatively [Sutton, 1988], consider Q-learning as the process:

$$\mathcal{P} := \mathcal{P} + \alpha_Q(\mathcal{A} - \mathcal{P})$$

Then W-learning is:

$$W := W + \alpha_W((\mathcal{P} - \mathcal{A}) - W)$$

Note how the agents learn their Q-values together, rather than alone. We must ensure, when learning together, that all agents experience a large number of visits to each of their (x, a).

Since the rewards and Q-values are bounded, it follows that the W-values are bounded (Lemma A.3.1).

Note that this scheme reduces to normal Q-learning when the agent is alone inside the robot. In this case, Q is updated every step with the action it suggested, and W is never updated at all. It doesn't matter what W-values it has - the agent is *always* obeyed.

3.1 Learning Q (somewhat) before learning W

Ideally we would like to say 'learn Q first, then W', but of course it is impossible to learn Q completely in finite time. We could have a long period of learning Q, followed by learning W. Alternatively, the scheme I use here starts learning W while Q is still being learnt:

$$W_i(x) := (1-\alpha_W)W_i(x) + \alpha_W(1-\alpha_Q)^{\omega}(Q_i(x,a_i) - (r_i + \gamma \max_{b \in A} Q_i(y,b))) \quad (4)$$

where $(1 - \alpha_Q) = (1 - \alpha_Q(x, a_i))$ increases with each update of $Q_i(x, a_i)$, and the *delaying rate* $\omega > 0$.

Low ω means letting W converge quickly. High ω means delaying W's convergence until Q is well known. ω can be seen as a parameter to control how 'fair' or scrupulous the adjudication of competition is.

3.2 After Q has been (somewhat) learnt

As Q is learnt:

$$\begin{array}{l} \text{predicted } \mathcal{P} = Q \rightarrow Q^* \\ (1-\alpha_Q) \rightarrow 1 \\ (1-\alpha_Q)^\omega \rightarrow 1 \end{array}$$

and, letting agent A_k be the leader, the update for A_i, $i \neq k$, is approximated by:

$$W_i(x) := (1-\alpha_W)W_i(x) + \alpha_W(V_i^*(x) - (r_i + \gamma V_i^*(y)))$$

In general, Q will be already somewhat learnt while W is doing its learning. Either we delay the learning of W (see previous section) or, alternatively, imagine a dynamically changing collection with agents being continually created and destroyed over time, and the surviving agents adjusting their W-values as the nature of their competition changes. Q is only learnt once, right from the start of the life of the agent, whereas W is relearnt again and again. [4] The skill that A_i learns, expressed in its converged Q-values, remains intact through subsequent competitions for x. Once it learns its action $a_i^*(x)$ it will promote it in all competitions, only varying the strength with which it is promoted (in the long term, we only need to keep $W_i(x)$ values, not $W_i(x,a)$ values). In the long term, any W-competition will settle down into a competition between the optimal actions $a_i^*(x)$. The update for A_i is approximated by:

$$W_i(x) := (1-\alpha_W)W_i(x) + \alpha_W d_{ki}(x)$$

where the random variable $d_{ki}(x)$ is the 'deviation' (difference between predicted $\mathcal{P}$ and actual $\mathcal{A}$) that A_k causes for A_i in state x if both are converged to their respective Q^*.

$d_{ki}(x)$ is a stationary probability distribution because $P^i_{xa}(r)$, $P_{xa}(y)$ are. To be precise:

$$\begin{aligned} E(d_{ki}(x)) &= V_i^*(x) - (E(r_i) + \gamma E(V_i^*(y))) \\ &= V_i^*(x) - \left(\sum_r r P^i_{xa}(r) \quad + \quad \gamma \sum_y V_i^*(y) P_{xa}(y) \right) \end{aligned}$$

where $a = a_k^*(x)$. We expect:

$$E(d_{kk}(x)) = 0$$

(though we do not actually update $W_k(x)$), and we expect for $i \neq k$:

[4] This can be compared to Edelman's biological theory of *Neural Darwinism* [Edelman, 1989, Edelman, 1992], in which the mind is viewed as a dynamic, competing collection of what he calls *neuronal groups*. The idea is appealing - it has been compared to a "rainforest" or dynamic ecosystem inside the head. However, Edelman presents no algorithm to show how it could be implemented. In fact, presenting arguments only about the limitations of various traditional AI models, he draws the extreme conclusion that no computer algorithm could implement his ideas. No mention is made of self-modifying, reinforcement-learning agents embedded in a world (such as the agents described here, but see [Kaelbling, 1993] for a broad survey), to which his criticisms do not apply. Indeed, RL seems to be exactly what Edelman is looking for. A dynamically-changing W-collection would seem to be a rough algorithmic implementation of some of Neural Darwinism's main ideas.

$$E(d_{ki}(x)) \geq 0$$

That is, if obeyed, we expect $\mathcal{A} = \mathcal{P}$. If not obeyed, we expect $\mathcal{A} \leq \mathcal{P}$.

If A_k leads from the start to infinity, then by Lemma B.1.1:

$$\begin{aligned} W_i(x) & \rightarrow E(d_{ki}(x)) \\ & \geq 0 \end{aligned}$$

Of course, it may be interrupted, as some new agent takes the lead. If A_i itself takes the lead, then W-learning stops for it until (if ever) it loses it. If another agent A_l takes the lead, then A_i will suddenly be taking samples from the distribution $d_{li}(x)$. By Lemma B.1.2, if we update forever from *this* point, then $W_i(x)$ eventually converges to the expected value of the new distribution:

$$W_i(x) \rightarrow E(d_{li}(x))$$

and so on. The W-learning algorithm can handle any number of switches of leader. Each time the leader changes, $W_i(x)$ starts converging toward the expected value of the new distribution, until such time (if ever) as there is another change of leader.

The history of $W_i(x)$ will be something like this:

$$\begin{array}{l} \text{leader is } A_k \\ W_i(x) \rightarrow E(d_{ki}(x)) \\ \text{leader changes to } A_l \\ W_i(x) \rightarrow E(d_{li}(x)) \\ A_i \text{ itself becomes the leader} \\ W_i(x) \text{ static} \\ \text{leader changes to } A_m \\ W_i(x) \rightarrow E(d_{mi}(x)) \\ \ldots \end{array}$$

The question is - will competition for the state ever be resolved, or will the leader keep changing forever?

4 Convergence

We have seen that W-learning is bounded, but does it converge? With Q-learning, Watkins showed that there was a unique stable solution, and, more importantly, that Q-learning will actually converge to it. With W-learning, I show that there is at least one and possibly multiple stable solutions, and that W-learning will converge to one of them.

It may seem obvious that W-learning converges, since the learning rate is declining. But that only means W converges if it is sampling from a *stationary* distribution. As long as the leader stays the same, $W_i(x)$ does sample from a stationary distribution, and converges to its expected value. What we need to show is that the leader will not keep changing forever.

4.1 A model for resolution of competition

First consider a matrix:

$$\begin{pmatrix} 0 & d_{21} & d_{31} & \dots & d_{n1} \\ d_{12} & 0 & d_{32} & \dots & d_{n2} \\ d_{13} & d_{23} & 0 & \dots & d_{n3} \\ \dots & & & & \\ d_{1n} & d_{2n} & d_{3n} & \dots & 0 \end{pmatrix}$$

where all $d_{ij} \geq 0$.

Lemma 4.1.1 *Given variables $W_1, \dots, W_n$, the process:*

start with any arbitrary values for $W_1, \dots, W_n$
$i :=$ any column
repeat forever
if $W_i \geq d_{ij} \forall j$ then terminate
else choose any *m such that $d_{im} > W_i$*
$W_m := d_{im}$
$i := m$ and repeat

will terminate within n^2 steps, and we will have found i such that:

$$W_i \geq d_{ij} \forall j$$

Proof: The process goes:

If we don't have the termination condition $W_i \geq d_{ip} \forall p$ then find j s.t:
$d_{ij} > W_i$
$W_j := d_{ij}$
If we don't have the termination condition $W_j \geq d_{jp} \forall p$ then find k s.t:
$d_{jk} > W_j$
$W_k := d_{jk}$
$d_{jk} > d_{ij}$
If we don't have the termination condition $W_k \geq d_{kp} \forall p$ then find l s.t:
$d_{kl} > W_k$
$W_l := d_{kl}$
$d_{kl} > d_{jk} > d_{ij}$
...

We have a strictly increasing sequence here, using up one matrix element at a time, so this process must terminate within n^2 steps.

4.2 Convergence of W-learning

The world is an MDP. State and action spaces are discrete, finite. Time steps are discrete. We have agents $A_1, \dots, A_n$, who experience probabilistic transitions $P_{xa}(y)$, and receive rewards $P^i_{xa}(r)$. For each state x, there is a matrix:

$$\begin{pmatrix} 0 & E(d_{21}(x)) & E(d_{31}(x)) & \dots & E(d_{n1}(x)) \\ E(d_{12}(x)) & 0 & E(d_{32}(x)) & \dots & E(d_{n2}(x)) \\ E(d_{13}(x)) & E(d_{23}(x)) & 0 & \dots & E(d_{n3}(x)) \\ \dots & & & & \\ E(d_{1n}(x)) & E(d_{2n}(x)) & E(d_{3n}(x)) & \dots & 0 \end{pmatrix}$$

where all $E(d_{ij}(x)) \geq 0$.

Theorem 4.2.1 (Convergence Theorem) *If $E(d_{ij}(x))$, $i \neq j$, are all distinct, then W-learning will resolve the competition for this state.*

If this holds for all states x, then W-learning will resolve the competition throughout the whole state-space.

Sketch Proof: Consider the process:

> start with any arbitrary values for $W_1(x), \ldots, W_n(x)$
> $k :=$ the agent such that $W_k(x) = \max_{i \in 1,\ldots,n} W_i(x)$
> repeat forever
> if $W_k(x) \geq E(d_{ki}(x)) \forall i, i \neq k$ then terminate
> else choose *any* l such that:
> $E(d_{kl}(x)) > W_k(x)$
> $W_l(x) := E(d_{kl}(x))$
> $k := l$ and repeat

By Lemma 4.1.1, this process will terminate within n^2 steps, resolving competition with a winner:

$$W_k(x) \geq E(d_{ki}(x)) \forall i, i \neq k$$

Since all $E(d_{ij}(x))$ are distinct, the condition *if* $W_k(x) \geq E(d_{ki}(x))$ is equivalent to the condition *if* $W_k(x) > E(d_{ki}(x))$, as in W-learning.

This is only a sketch proof because of the step $W_l(x) := E(d_{kl}(x))$. W-learning is the same process as above except that $W_l(x) \rightarrow E(d_{kl}(x))$, so there may be statistical variations in any finite sample. As the number of experiences $\rightarrow \infty$ however, the expected values must emerge from the samples and competition will be resolved.

In fact, because of unrepresentative samples, $W_k(x)$ may be overtaken by an agent where actually $E(d_{kl}(x)) < W_k(x)$, but it is expected that as the number of experiences $\rightarrow \infty$, this temporary phenomenon too will vanish.

4.2.1 W-learning in practice

In practice, the condition that $E(d_{ij}(x))$ be all distinct is not really any restriction at all. The $E(d_{ij}(x))$ are real numbers, so the set of problems (two being exactly the same) is infinite but sparse, e.g. for two agents, it is the line $x = y$ in $\Re^2$.

With real-valued rewards (see Section 5), probabilistic transitions, and agents with widely different logic in their reward functions, we expect the $V_i^*(x)$, and hence the $E(d_{ij}(x))$, to be different real numbers, in which case competition will be resolved (if we're really worried about draws, we can just arbitrarily halt competition after some lengthy time).

So, just as we can take any reward function, and Q-learning will eventually converge to the appropriate Q^*, we can put together any collection of agents, and eventually they will divide up state-space between them, based on the deviations they cause each other.

4.3 Multiple possible winners from the $E(d_{ij}(x))$ matrix

Note that for a given $E(d_{ij}(x))$ matrix, there may be more than one possible winner. For example:

$$\begin{pmatrix} 0 & 3 & 0 \\ 0 & 0 & 9 \\ 0 & 0 & 0 \end{pmatrix}$$

Start with all $W_i(x) = 0$. Choose agent A_2's action for execution. Then:

$$W_1(x) := 3$$

Now agent A_1 is in the lead, and:

$$3 > 0, 0, 0$$

Agent A_1 is the winner. However, if we had started by choosing agent A_3's action, then:

$$W_2(x) := 9$$

Now agent A_2 is in the lead, and:

$$9 > 3, 0, 0$$

Agent A_2 is the winner. We have a winner when *somebody* finds a deviation they suffer *somewhere* that is worse than the deviation they cause everyone else.

W-learning does *not* exhaustively search all combinations i, j to find the highest $E(d_{ij}(x))$ in the matrix. It would be impractical to let every agent experience what it is like with every other agent in the lead. W-learning gets down to a winner a lot quicker than that.

Also, it is not simply a question of the highest $E(d_{ij}(x))$ - we aren't interested in the worst deviation an agent could possibly suffer, we are interested in what it is *likely* to suffer if it does not win.

4.4 Scoring $W_k(x)$

Should we score W if obeyed as well? If we do, then:

$$\begin{aligned} W_k(x) & \rightarrow E(d_{kk}(x)) \\ & = 0 \end{aligned}$$

The leader's W is converging to zero, while the other agents' W's are converging to $E(d_{ki}(x)) \geq 0$. They are guaranteed to catch up with it. So there will be no resolution of the competition ever.

We might think it would be nice to try and reduce all weights to the minimum possible; so as soon as you are obeyed, you start reducing your weight. But you can only *find* the minimum by reducing so far that someone else takes over. Hence we will have back and forth competition forever under any such system.

This is why the leader does nothing - it's up to the others to catch up with it. If they can't, we have a resolved competition.

4.5 Scaling, peers and unequal agents

Note that the use of $(\mathcal{P} - \mathcal{A})$ means that the algorithm is not scaled. An agent with rewards 1, -1, 0 will end up with higher W-values than an equivalent agent with rewards 0.1, -0.1, 0 and the same logic, since its absolute $(\mathcal{P} - \mathcal{A})$ differences will be greater. It will win a greater area of state-space in competitions.

This is in fact what we want. We call the agents peers because they compete on the same basis, but we do not want them to be peers in the sense that all their concerns are of equal importance, which is what scaling would make them. We want to be able to express a strong version of the predator-avoider (if predator visible $r = -10$, else $r = 0$) as distinct from a weak version of the same thing (if predator visible $r = -0.1$, else $r = 0$). Adaptive collections are likely to involve well-chosen combinations of weak and strong agents.

5 Empirical Work

We could take selected combinations of agents and see how they resolve their competition. Alternatively, the approach I take is to use the fact that *any* combination of agents will resolve their competition, and do an automated search looking for interesting combinations.

Starting with any n agents with any reward functions (leading to any converged Q-values), W-learning will eventually converge and the agents will have permanently divided up state-space between them. So we can define spaces of possible robot architectures where every point in the space represents a robot that can be built and tried out (for the best exposition of this concept see [Dawkins, 1986, Ch.3]). Such a space can be searched in a manner similar to evolution by natural selection [Langton, 1989].

I use a Genetic Algorithm (GA) (from [Holland, 1975], for an introduction see [Goldberg, 1989]) to search such a space of agent-collections for collections whose W-converged situation is adaptive (by some criterion).

5.1 The Genetic Algorithm

A genotype of the GA encodes a collection of real-valued reward functions, each of which defines a Q-learning agent. An individual is assembled, being a collection of Q-learning agents. At the 'birth' of the individual, agents have random Q-values and W-values. There follows a 'childhood' of Q-learning and W-learning. After a finite time, the individual is tested for fitness according to some criterion. The fitter individuals are chosen for reproduction, which involves crossover and mutation of their genotypes, leading to genotypes encoding new reward functions. A new population of genotypes is constructed - and so on, in the normal GA manner.

Features of this GA:

- Provides one solution to the problem of 'where do the rewards come from?' in RL, by evolving them. It is productive to think of rewards not as something external and informative but as simply some arbitrary internal signals (a similar attitude is taken by Ackley and Littman [Ackley and Littman, 1991], in their evolution-plus-learning experiment).

If an agent generates high rewards for itself when it sees predators, then its converged Q^* will cause it to seek out predators. This may not be a good thing, but that is a separate issue. Adaptive agents are ones that happen to generate suitable internal rewards.

- A concise genotype. Much current work on evolving control systems for robots involves encoding the entire explicit control program in the genotype - see the Artificial Neural Networks (ANN's) of much current Artificial Life work [Collins, 1992, Harvey et al., 1993], or the Genetic Programming (GP) of Koza [Koza, 1991].

  Here, the genotype simply states what is good and what is bad about the world (in each agent's perhaps unadaptive opinion) and nothing else. It says nothing about *how* to make good things happen and how to avoid bad things - agents have to learn this via their own individual, historical experiences. Nor does the genotype state how control is to be shared among the agents - the agents have to learn this by their history of competing with each other.

- One of the major issues with sophisticated ALife encodings such as ANN's is whether one can analyse the solution that evolves, or whether it is just some impenetrable 'spaghetti' that happens to work. Many researchers [Collins, 1992, Sims, 1994, Harvey et al., 1994] have run into the problem of evolving creatures that present a similar challenge of analysis as natural creatures do.

  Here, evolving the rewards rather than the explicit behavior will permit at least some analysis of the solutions found. The classification of the world into good and bad in the genotype immediately tells us what the evolved agent likes and dislikes and allows us classify evolved agents as 'food-seekers', 'predator-avoiders' etc. (though we may find it difficult to see why a particular *combination* of agents is adaptive).

5.2 The Simulated World

The problem I set for my W-learning simulated robot is the conflict between seeking food and avoiding predators on a simple simulated landscape (Figure 6). The world is a toroidal (SIZE x SIZE) gridworld containing a nest, a number (NOFOOD) of static, randomly-distributed pieces of food, and a number (NOPREDATORS) of dynamic, randomly-moving dumb predators. World, evolution and learning are all implemented in C++.

The simulated robot makes a number of runs, each of length STEPSPERRUN timesteps. At the start of each run, the robot, food and predators are placed randomly. Each timestep, the robot can move one square or stay still. When it finds food, it picks it up. It can only carry one piece of food at a time, and it can only drop it at the nest. The task for the robot is to forage food (i.e. find it, and bring it back to the nest) while avoiding the predators.

The world is so simple that we can use discrete states and actions with no generalization. No claims are made for this world - we are constructing here *examples* of how W-learning can work rather than the evidence that it can solve major problems.

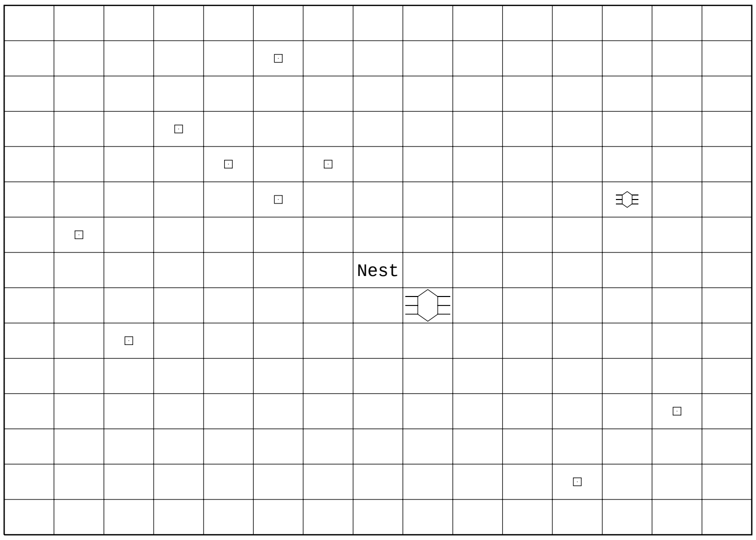

Figure 6: The toroidal gridworld.

The robot senses $x = (i, n, f, p)$. i is whether the robot is carrying food or not, and takes values 0 (not carrying) and 1 (carrying). n is the direction (but not distance) of the nest, and takes values 0-7 (compass directions, see Figure 7), and 8 (when at the nest). f is the direction of the nearest visible food (within a small radius of RADIUS squares around the robot), and takes values 0-8, and 9 (no food within the radius). Similarly, p is the direction of the nearest predator within the radius, and takes values 0-9.

The robot takes actions a, which take values 0-7 (move in that direction) and 8 (stay still).

5.2.1 The fitness function

Encountering a predator doesn't end the run - rather, we leave it to the fitness function to decide how serious encountering a predator is. Over a number of runs, we record the average amount of food foraged per run F, and the average number of predator encounters per run P, and score the fitness:

$$\text{fitness} = C_F F - C_P P$$

where C_F and C_P are some positive constants.

In the experiments we now describe, SIZE=10, NOFOOD=4, NOPREDATORS=1, and RADIUS $= 3.2$. We set STEPSPERRUN=50. We can only see locally, and have no memory, so this more or less guarantees that no strategy, hand-coded or evolved/learnt, will always forage all 4 pieces of food before the run ends. The theoretical maximum fitness $4C_F$ will not therefore be seen.

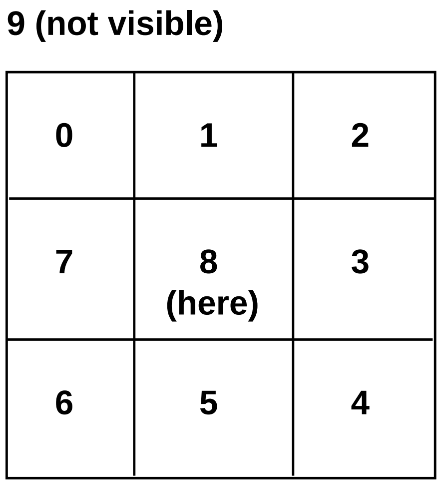

Figure 7: The robot senses the relative direction of things within a small radius around it (with the exception of the nest, whose direction it senses from any distance).

5.3 Hand-coded programs

First, we try to explicitly program the robot to do the task, to get a benchmark against which to compare the performance of evolved versions. We start with $C_F = 100$, $C_P = 1$, and write a program PROG1 which, given a state x, generates action a as follows:

```
programmedAction ( state x )
{
 if (not carrying)
 {
  if (food visible)
   a := directionFood;
  else
   a := randomMove();
 }
 else
  a := directionNest;
}
```

PROG1 takes no action to avoid encountering predators.

As we increase the ratio of C_P to C_F, it begins to pay to have a robot which takes at least some predator-avoiding action. Note that all strategies, hand-coded or evolved/learnt, must deal with the fact that there is no memory. We think up of a new algorithm, PROG2, which works as follows. If PROG1 wants to go in the exact direction of a predator, the direction it suggested is perturbed slightly (by ± 1). The result of any predator-avoidance has to be, of course, a drop in the efficiency of food-foraging.

As we increase C_P further, we need to think of new and stronger predator-avoiding algorithms. PROG3 abandons any attempt at food-foraging altogether when a predator is visible, and moves randomly, hoping to shake it off. If no predators are visible, it behaves like PROG1.

Since robot and predators move simultaneously, and a predator's next movement is unpredictable (even if it wasn't random, we have no memory of its previous motion), the only really sure strategy is moving away from it. PROG4 works like PROG3 except, instead of moving randomly, it moves in the broad opposite direction to the observed predator.

We test each program over 500 runs. As we can see, which algorithm is best depends on the parameters of the fitness function. The best performer for each fitness function is highlighted:

			PROG1	PROG2	PROG3	PROG4
			F P	F P	F P	F P
			3.516 0.488	3.468 0.238	2.844 0.138	2.990 0.006
CF	CP	max fitness:	fitness:	fitness:	fitness:	fitness:
100	1	400	[351.112]	346.562	284.262	298.994
10	1	40	[34.672]	34.442	28.302	29.894
1	1	4	3.028	[3.230]	2.706	2.984
1	10	4	-1.364	1.088	1.464	[2.930]
1	100	4	-45.284	-20.332	-10.956	[2.390]

5.4 Evolved W-collections

We want to define a space of agent-collections which will encompass a very broad range of robot behaviors. Consider the space of collections of 3 agents A_f, A_n, A_p, one of each of these types:

```
FoodSensingAgent
x = (i,f), state-space size 20
reward function:
 reward()
 {
  if (just picked up food) return r1
  else return r2
 }

NestSensingAgent
x = (n), state-space size 9
reward function:
 reward()
 {
  if (just arrived at nest) return r3
  else return r4
 }

PredatorSensingAgent
x = (p), state-space size 10
reward function:
 reward()
 {
  if (just shook off predator (no longer visible)) return r5
  else return r6
 }
```

for different values of $r_1, \ldots, r_6$, in the range $r_{min} = 0$ to $r_{max} = 2$. This defines a huge range of possible robots, including ones containing food-seekers, food-avoiders, predator-seekers, predator-avoiders that are stronger than food-seekers, food-seekers that are stronger than predator-avoiders, and so on. Can all worthwhile robot behaviors be generated here? Or are there useful behaviors which only more complex reward functions and/or combinations of agents can yield? We do not know the answer, but we search this space anyway, hoping that it is big enough to contain solutions that are near-optimal.

The genotype encodes the 6 real numbers as 6 initially-random bitstrings.

Once it has been constructed from the genotype, the robot does 1800 runs during which the agents learn their Q-values and W-values with a temperature (recall Section 2.1.4) that starts high and declines throughout. We use $\gamma = 0.7$ and $\omega = 3$. The robot then does a test of 200 runs at minimum temperature to score the fitness of what has been learnt.

Note that while the robot's $x = (i, n, f, p)$ defines a state-space of size 1800, none of the agents work with this large space, though it remains the 'virtual' field on which they compete.

After only quick evolutionary searches (population size ≤ 60, initially randomised, evolving for ≤ 15 generations), we found the following W-collections. The table shows, for the different fitness functions, the best solution (set of 6 rewards) found by evolution, and in each solution, the division of the full (size 1800) statespace between the agents:

CF	CP	max fitness	best evolved W-collection	r1	r2	r3	r4	r5	r6	percentage ownership Af-An-Ap	F	P	fitness
100	1	400	EVO1	1.73	0.11	0.41	0.26	1.37	1.20	49-35-16	3.650	0.410	[364.590]
10	1	40	EVO1	1.73	0.11	0.41	0.26	1.37	1.20	49-35-16	3.650	0.410	[36.090]
1	1	4	EVO1	1.73	0.11	0.41	0.26	1.37	1.20	49-35-16	3.650	0.410	[3.240]
1	10	4	EVO2	1.93	0.28	0.26	0.07	1.99	0.76	35-16-49	2.990	0.015	[2.840]
1	100	4	EVO3	1.46	0.62	0.22	0.07	1.92	0.12	20-16-63	2.605	0.000	[2.605]

5.4.1 Analysis of EVO1

Our first level of analysis produces no surprises: $r_1 > r_2$, $r_3 > r_4$ and $r_5 > r_6$ throughout. That is, all successful robots are composed of a food-seeker, a nest-seeker, and a predator-avoider.

We start with the 6 decoded rewards, and random Q-values and W-values. Q-learning and W-learning progress, each agent translating what is happening into its own subspace. For example (using the notation `x,a -> y`):

```
    robot.transition (0,8,5,1),(5) -> (1,1,9,9)

     food.transition (0,5),(5) -> (1,9)

     nest.transition (8),(5) -> (1)

 predator.transition (1),(5) -> (9)
```

The next level of analysis looks at who finally wins each state. In EVO1, A_f wins almost the entire space where $i = 0$ (not carrying). In the space where $i = 1$ (carrying), A_n wins if $p = 9$ (no predator visible). Where a predator is visible, the space is split between A_n and A_p.

For example, here are the owners of the area of statespace where $p = 7$. The agents A_f, A_n, A_p are represented by the symbols `o`, `NEST`, `pred` respectively. States which have not (yet) been visited are marked with a dotted line:

```
---- p=7: ----
i=0:
f=0     o       o       o       o       o       o       o       o       o
f=1     o       o       o       o       o       o       o       o       o
f=2     o       o       o       o       o       o       o       o       o
f=3     o       o       o       o       o       o       o       o       o
f=4     o       o       o       o       o       o       o       o       o
f=5     o       o       o       o       o       o       o       o       o
f=6     o       o       o       o       o       o       o       o       o
f=7     o       o       o       o       o       o       o       o       o
f=8     ------------------------------------------------------------------------
f=9     o       o       NEST    o       NEST    o       o       o       o
        n=0     n=1     n=2     n=3     n=4     n=5     n=6     n=7     n=8
i=1:
f=0     NEST    pred    NEST    pred    NEST    NEST    NEST    NEST    --------
f=1     NEST    NEST    NEST    pred    NEST    pred    NEST    NEST    --------
f=2     pred    NEST    NEST    pred    NEST    pred    NEST    pred    --------
f=3     NEST    pred    NEST    NEST    NEST    pred    NEST    pred    --------
f=4     NEST    pred    NEST    NEST    NEST    NEST    NEST    pred    --------
f=5     NEST    pred    NEST    pred    NEST    pred    NEST    NEST    --------
f=6     NEST    pred    NEST    NEST    NEST    pred    NEST    pred    --------
f=7     NEST    pred    NEST    pred    NEST    pred    NEST    NEST    --------
f=8     NEST    NEST    NEST    pred    NEST    pred    NEST    NEST    --------
f=9     NEST    o       NEST    NEST    NEST    pred    NEST    NEST    --------
        n=0     n=1     n=2     n=3     n=4     n=5     n=6     n=7     n=8
```

Here are all the W-values of all the agents, sorted to show who beats who. The W-values $W_f((i,f)), W_n((n)), W_p((p))$ are represented by `food.W(i,f)`, `nest.W(n)`, `predator.W(p)` respectively:

```
     food.W(0,0)        0.195
     food.W(0,1)        0.183
     food.W(0,7)        0.144
     food.W(0,5)        0.114
     food.W(0,2)        0.097
     food.W(0,6)        0.094
     food.W(0,3)        0.089
     food.W(0,9)        0.087
     food.W(0,4)        0.084
     nest.W(6)          0.083
     nest.W(2)          0.074
     nest.W(4)          0.058
     nest.W(0)          0.041
 predator.W(1)          0.037
 predator.W(0)          0.021
 predator.W(2)          0.018
 predator.W(3)          0.016
     nest.W(8)          0.016
 predator.W(7)          0.013
     nest.W(7)          0.012
 predator.W(5)          0.011
     nest.W(1)          0.006
 predator.W(4)          0.004
 predator.W(6)          0.001
     nest.W(3)          0.001
     food.W(0,8)        0.000    (never visited)
     nest.W(5)         -0.000
 predator.W(8)         -0.003
 predator.W(9)         -0.008
     food.W(1,9)       -0.008
     food.W(1,6)       -0.026
     food.W(1,1)       -0.027
```

```
food.W(1,3)        -0.031
food.W(1,5)        -0.032
food.W(1,2)        -0.033
food.W(1,7)        -0.036
food.W(1,0)        -0.036
food.W(1,4)        -0.045
food.W(1,8)        -0.098
```

The next level of analysis is what actions the robot actually ends up executing as a result of this resolution of competition. When not carrying food, A_f is in charge, and it causes the robot to wander, and then head for food when visible. A_n is constantly suggesting that the robot return to the nest, but its W-values are too weak. Then, as soon as $i = 1$, A_f's W-values drop below zero, and A_n finds itself in charge. As soon as it succeeds in taking the robot back to the nest, $i = 0$ and A_f immediately takes over again. In this way the two agents combine to forage food, even though both are pursuing their own agendas.

The final level of analysis is why the W-values turn out the way they do. We can see, for example, that when $i = 1$ (carrying food), A_f is a long way off from getting a reward, since it has to lose the food at the nest first. And it cannot learn how to do this since (n) is not in its statespace. A_f ends up in a state of dependence on A_n, which actually knows better than A_f the action that is best for it.

So why not get rid of A_n and simply supply A_f with the space $x = (i, n, f)$? Because it is more efficient if we can use two agents with statespaces of size 20 and 9 respectively (total memory required = 29) instead of one with a statespace of size 180 (total memory required = 180).

5.4.2 Analysis of EVO2

Turning to EVO2, the contrast is dramatic. Here is that same area of statespace:

```
---- p=7: ----
i=0:
f=0      o        o        pred     o        o        o        o        o        o
f=1      pred     pred     o        pred     pred     pred     pred     pred     pred
f=2      pred     pred     pred     pred     pred     o        pred     pred     pred
f=3      pred     pred     pred     o        pred     pred     pred     o        pred
f=4      o        o        o        o        o        o        pred     o        o
f=5      pred     o        o        pred     pred     pred     pred     pred     o
f=6      o        o        o        o        o        o        o        o        o
f=7      pred     o        o        o        o        pred     o        o        o
f=8      ------------------------------------------------------------------------
f=9      pred     pred     pred     pred     pred     pred     pred     pred     pred
         n=0      n=1      n=2      n=3      n=4      n=5      n=6      n=7      n=8
i=1:
f=0      pred     pred     pred     pred     pred     pred     pred     pred     --------
f=1      pred     pred     pred     pred     pred     pred     pred     pred     --------
f=2      pred     pred     pred     pred     pred     pred     pred     pred     --------
f=3      pred     pred     pred     pred     pred     pred     pred     pred     --------
f=4      pred     pred     pred     pred     pred     pred     pred     pred     --------
f=5      pred     pred     pred     pred     pred     pred     pred     pred     --------
f=6      pred     pred     pred     pred     pred     pred     pred     pred     --------
f=7      pred     pred     pred     pred     pred     pred     pred     pred     --------
f=8      pred     pred     pred     pred     pred     pred     pred     pred     --------
f=9      pred     pred     pred     pred     pred     pred     pred     pred     --------
         n=0      n=1      n=2      n=3      n=4      n=5      n=6      n=7      n=8
```

A_p mainly dominates when a predator is visible in directions 0-7. In particular, in that space A_f loses the crucial state $(0, 9)$. In the special case $p = 8$, all directions are equal as far as A_p is concerned, and A_f and A_n are allowed compete to take the action. When $p = 9$, A_f and A_n fight it out as if A_p wasn't there. They end up combining to forage.

5.5 Discussion

The 6 rewards define a vast space of possible robots, in which, for example, we can find pretty close approximations to PROG1 and PROG4.

As conditions change, we can move continuously through this space by varying the numerical rewards (their size, and the differences between them), so varying each agent's possession of state-space in a continuous manner. This is an alternative to the programmed approach where we have to specify and hand-code perhaps widely different logic for each situation.

To a certain extent also, we have 3 independent and quite simple parts, with much of the complexity arising from all the different ways in which they can interact.

6 Summary and conclusion

We have shown that for any given collection of Q-learners, there is what can be described as a 'natural' action-selection scheme. We avoid the problem of defining the flow of control by having it follow naturally once the collection of agents is specified.

W-learning resolves competition without resorting to devices such as killing off agents that are disobeyed for time t, without any $W \to \infty$, and in fact with normally most $W \ll W_{max}$.

Finally, W-learning is fair resolution of competition - the most likely winner of a state is the agent that is most likely to suffer the highest deviation if it does not win.

We have demonstrated a few interesting examples, but it is clear that the space of competing agent-collections has only barely begun to be explored.

Acknowledgements

I am most indebted to Barney Pell for introducing me to reinforcement learning. Thanks to my supervisor John Daugman, credit to Gavin Rummery for an improvement to the original algorithm, and thanks also to Ralph Becket, Marcel Hernandez, Steve Hodges, Dave Palfrey, Michael K.Sahota and Chen K.Tham for valuable comments.

This research was supported in part by a British Council scholarship and in part by the University of Cambridge Computer Laboratory.

A Bounds

A.1 Bounds with α

Let R be bounded by R_{max},R_{min}. Let X be updated by:

$$X := (1-\alpha)X + \alpha R$$

where the initial value of $\alpha = 1$. Then:

Lemma A.1.1 *X is also bounded by R_{max},R_{min}.*

Proof: Standard result.

A.2 Bounds of Q-values

Lemma A.2.1

$$\begin{array}{l} Q_{max} = \frac{r_{max}}{1-\gamma} \\ Q_{min} = \frac{r_{min}}{1-\gamma} \end{array}$$

Proof: Standard result.
Note that since $r_{min} < r_{max}$, it follows that $Q_{min} < Q_{max}$.

A.3 Bounds of W-values

Lemma A.3.1

$$\begin{array}{l} W_{max} = Q_{max} - Q_{min} \\ W_{min} = -(Q_{max} - Q_{min}) \end{array}$$

Proof: W is updated by:

$$W(x) := (1-\alpha_W)W(x) + \alpha_W \lambda(Q(x,a) - (r + \gamma \max_{b \in A} Q(y,b)))$$

where $0 \leq \lambda \leq 1$, so by Lemma A.1.1:

$$\begin{array}{ll} W_{max} & = (\lambda(Q - (r + \gamma Q)))_{max} \\ & = 1(Q_{max} - (r + \gamma Q)_{min}) \\ & = Q_{max} - Q_{min} \end{array}$$

Similarly:

$$W_{min} = 1(Q_{min} - Q_{max})$$

Note that since $Q_{min} < Q_{max}$, it follows that $W_{min} < 0 < W_{max}$.

B Incremental sampling of random variables

B.1 A single variable

Let $d_1, d_2, d_3, \ldots$ be samples of a stationary random variable d with expected value $E(d)$. Then the following update algorithm provides an elegant way of sampling them. Repeat:

$$W := (1-\alpha)W + \alpha d_i$$

Lemma B.1.1 *If α takes successive values $1, \frac{1}{2}, \frac{1}{3}, \ldots$, then $W \to E(d)$, independent of the initial value of W.*

Proof: This is a standard result, but I include the proof here to illustrate how the learning rate α works. W's updates go:

$$\begin{array}{ll}
W := 0W_{init} + 1d_1 & = d_1 \\
W := \frac{1}{2}d_1 + \frac{1}{2}d_2 & = \frac{1}{2}(d_1 + d_2) \\
W := \frac{2}{3}\frac{1}{2}(d_1 + d_2) + \frac{1}{3}d_3 & = \frac{1}{3}(d_1 + d_2 + d_3) \\
W := \frac{3}{4}\frac{1}{3}(d_1 + d_2 + d_3) + \frac{1}{4}d_4 & = \frac{1}{4}(d_1 + d_2 + d_3 + d_4) \\
\ldots & \\
W = \frac{1}{t}(d_1 + \cdots + d_t) &
\end{array}$$

i.e. W is simply the average of all d_i samples so far. If this continues forever, then $W \to E(d)$.

More generally:

Lemma B.1.2 *If α takes successive values $\frac{1}{t}, \frac{1}{t+1}, \frac{1}{t+2}, \ldots$, then $W \to E(d)$, independent of the initial value of W, and independent of t.*

Proof: Similar to proof of Lemma B.1.1.

One way of looking at this is to consider W_{init} as the average of all samples before time t, samples which are now irrelevant for some reason. We can consider them as samples from a different distribution f:

$$W_{init} = \frac{1}{t-1}(f_1 + \cdots + f_{t-1})$$

Hence:

$$\begin{array}{ll}
W & = \frac{1}{n}(f_1 + \cdots + f_{t-1} + d_t + \cdots + d_n) \\
 & = \frac{1}{n}(f_1 + \cdots + f_{t-1}) + \frac{1}{n}(d_t + \cdots + d_n) \\
 & \to 0 + E(d)
\end{array}$$

as $n \to \infty$.

References

[Ackley and Littman, 1991] Ackley, David and Littman, Michael (1991), Interactions between learning and evolution, in Christopher G.Langton et al., eds., *Artificial Life II.*

[Blumberg, 1994] Blumberg, Bruce (1994), Action-Selection in Hamsterdam: Lessons from Ethology, in Dave Cliff et al., eds., *Proceedings of the Third International Conference on Simulation of Adaptive Behavior (SAB-94).*

[Brooks, 1986] Brooks, Rodney A. (1986), A robust layered control system for a mobile robot, *IEEE Journal of Robotics and Automation* vol.RA-2, no.1, Mar 1986.

[Brooks, 1991] Brooks, Rodney A. (1991), Intelligence without Representation, *Artificial Intelligence* 47:139-160.

[Brooks, 1994] Brooks, Rodney A. (1994), Coherent Behavior from Many Adaptive Processes, in Dave Cliff et al., eds., *Proceedings of the Third International Conference on Simulation of Adaptive Behavior (SAB-94).*

[Collins, 1992] Collins, Robert J. (1992), *Studies in Artificial Evolution*, PhD thesis, UCLA.

[Dawkins, 1986] Dawkins, Richard (1986), *The Blind Watchmaker*, Penguin Books.

[Edelman, 1989] Edelman, Gerald M. (1989), *The Remembered Present: A Biological Theory of Consciousness*, Basic Books.

[Edelman, 1992] Edelman, Gerald M. (1992), *Bright Air, Brilliant Fire: On the Matter of the Mind*, Basic Books.

[Goldberg, 1989] Goldberg, David E. (1989), *Genetic Algorithms: in search, optimization, and machine learning*, Addison-Wesley.

[Harvey et al., 1993] Harvey, Inman; Husbands, Phil; and Cliff, Dave (1993), Issues in Evolutionary Robotics, in Jean-Arcady Meyer et al., eds., *Proceedings of the Second International Conference on Simulation of Adaptive Behavior (SAB-92).*

[Harvey et al., 1994] Harvey, Inman; Husbands, Phil; and Cliff, Dave (1994), Seeing The Light: Artificial Evolution, Real Vision, in Dave Cliff et al., eds., *Proceedings of the Third International Conference on Simulation of Adaptive Behavior (SAB-94).*

[Holland, 1975] Holland, John H. (1975), *Adaptation in Natural and Artificial Systems*, Ann Arbor, Univ. Michigan Press.

[Kaelbling, 1993] Kaelbling, Leslie Pack (1993), *Learning in Embedded Systems*, The MIT Press/Bradford Books.

[Koza, 1991] Koza, John R. (1991), Genetic evolution and co-evolution of computer programs, in Christopher G.Langton et al., eds., *Artificial Life II.*

[Langton, 1989] Langton, Christopher G. (1989), Artificial Life, in Christopher G.Langton, ed., *Artificial Life.*

[Lin, 1993] Lin, Long-Ji (1993), Scaling up Reinforcement Learning for robot control, *Proceedings of the Tenth International Conference on Machine Learning.*

[Maes, 1989] Maes, Pattie (1989), The dynamics of action selection, *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI-89).*

[Minsky, 1986] Minsky, Marvin (1986), *The Society of Mind*, Simon and Schuster, New York.

[Moore, 1990] Moore, Andrew W. (1990), *Efficient Memory-based Learning for Robot Control*, PhD thesis, University of Cambridge, Computer Laboratory.

[Sahota, 1994] Sahota, Michael K. (1994), Action Selection for Robots in Dynamic Environments through Inter-behaviour Bidding, in Dave Cliff et al., eds., *Proceedings of the Third International Conference on Simulation of Adaptive Behavior (SAB-94).*

[Sims, 1994] Sims, Karl (1994), Evolving 3D Morphology and Behavior by Competition, in Rodney A.Brooks and Pattie Maes, eds., *Artificial Life IV.*

[Singh, 1992] Singh, Satinder P. (1992), Transfer of Learning by Composing Solutions of Elemental Sequential Tasks, *Machine Learning* 8:323-339.

[Sutton, 1988] Sutton, Richard S. (1988), Learning to Predict by the Methods of Temporal Differences, *Machine Learning* 3:9-44.

[Tan, 1993] Tan, Ming (1993), Multi-Agent Reinforcement Learning: Independent vs. Cooperative Agents, *Proceedings of the Tenth International Conference on Machine Learning.*

[Tham and Prager, 1994] Tham, Chen K. and Prager, Richard W. (1994), A modular Q-learning architecture for manipulator task decomposition, *Proceedings of the Eleventh International Conference on Machine Learning.*

[Watkins, 1989] Watkins, Christopher J.C.H. (1989), *Learning from delayed rewards*, PhD thesis, University of Cambridge, Psychology Department.

[Watkins and Dayan, 1992] Watkins, Christopher J.C.H. and Dayan, Peter (1992), Technical Note: Q-Learning, *Machine Learning* 8:279-292.