Appears in the *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94).*

Incorporating Advice into Agents that Learn from Reinforcements*

Richard Maclin **Jude W. Shavlik**
Computer Sciences Dept., University of Wisconsin
1210 West Dayton Street
Madison, WI 53706
Email: {maclin,shavlik}@cs.wisc.edu

Abstract

Learning from reinforcements is a promising approach for creating intelligent agents. However, reinforcement learning usually requires a large number of training episodes. We present an approach that addresses this shortcoming by allowing a connectionist Q-learner to accept advice given, at any time and in a natural manner, by an external observer. In our approach, the advice-giver watches the learner and occasionally makes suggestions, expressed as instructions in a simple programming language. Based on techniques from knowledge-based neural networks, these programs are inserted directly into the agent's utility function. Subsequent reinforcement learning further integrates and refines the advice. We present empirical evidence that shows our approach leads to statistically-significant gains in expected reward. Importantly, the advice improves the expected reward regardless of the stage of training at which it is given.

Introduction

A successful and increasingly popular method for creating intelligent agents is to have them learn from reinforcements (Barto, Sutton, & Watkins 1990; Lin 1992; Mahadevan & Connell 1992). However, these approaches suffer from their need for large numbers of training episodes. While several approaches for speeding up reinforcement learning have been proposed, a largely unexplored approach is to design a learner that can also accept advice from an external observer. We present and evaluate an approach for creating advice-taking learners.

To illustrate the general idea of advice-taking, imagine that you are watching an agent learning to play some video game. Assume you notice that frequently the agent loses because it goes into a "box canyon" in search of food and then gets trapped by its opponents. One would like to give the learner advice such as "do not go into box canyons when opponents are in sight." Importantly, the external observer should be able to provide its advice in some quasi-natural language, using terms about the specific task domain. In addition, the advice-giver should be oblivious to the details of whichever internal representation and learning algorithm the agent is using.

Recognition of the value of advice-taking has a long history in AI. The general idea of an agent accepting advice was first proposed about 35 years ago by McCarthy (1958). Over a decade ago, Mostow (1982) developed a program that accepted and "operationalized" high-level advice about how to better play the card game Hearts. More recently Gordon and Subramanian (1994) created a system that deductively compiles high-level advice into concrete actions, which are then refined using genetic algorithms. However, the problem of making use of general advice has been largely neglected.

In the next section, we present a framework for using advice with reinforcement learners. The subsequent section presents experiments that investigate the value of our approach. Finally, we list possible extensions to our work, further describe its relation to other research, and present some conclusions.

The General Framework

In this section we describe our approach for creating a reinforcement learner that can accept advice. We use *connectionist Q-learning* (Sutton 1988; Watkins 1989) as our form of reinforcement learning (RL).

Figure 1 shows the general structure of a reinforcement learner, augmented (in bold) with our advice-taking extensions. In RL, the learner senses the current world state, chooses an action to execute, and occasionally receives rewards and punishments. Based on these reinforcements from the environment, the task of the learner is to improve its action-choosing module such that it increases the amount of rewards it receives. In our augmentation, an observer watches the learner

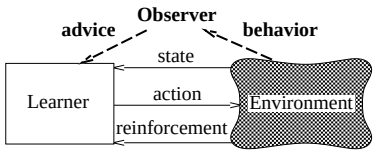


Figure 1: RL with an external advisor.

*This research was partially supported by ONR Grant N00014-93-1-0998 and NSF Grant IRI-9002413.

and periodically provides advice, which is then incorporated into the action-choosing module (the advice is refined based on subsequent experience).

In Q-learning (Watkins 1989) the action-choosing module is a *utility function* that maps states and actions to a numeric value. The utility value of a particular state and action is the predicted future (discounted) reward that will be achieved if that action is taken by the agent in that state. Given a perfect version of this function, the optimal plan is to simply choose, in each state that is reached, the action with the largest utility.

To learn a utility function, a Q-learner starts out with a randomly chosen utility function and explores its environment. As the agent explores, it continually makes predictions about the reward it expects and then updates its utility function by comparing the reward it actually receives to its prediction. In *connectionist* Q-learning, the utility function is implemented as a neural network, whose inputs describe the current state and whose outputs are the utility of each action.

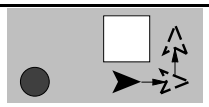
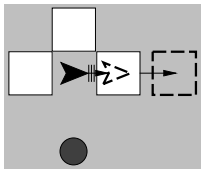
We now return to the task of advice-taking. Hayes-Roth, Klahr, and Mostow (1981) (also see pg. 345–349 of Cohen & Feigenbaum 1982) described the steps involved in taking advice. In the following subsections, we state their steps and discuss how we propose each should be achieved in the context of RL.

Step 1. Request the advice. Instead of having the learner request advice, we allow the external observer to provide advice whenever the observer feels it is appropriate. There are two reasons for this: (i) it places less of a burden on the observer; and (ii) it is an open question how to create the best mechanism for having an RL agent recognize (and express) its need for advice. Other approaches to providing advice to RL agents are discussed later.

Step 2. Convert the advice to an internal representation. Due to the complexities of natural language processing, we require that the external observer express its advice using a simple programming language and a list of acceptable task-specific terms. We then parse the advice, using traditional methods from programming-language compilers.

Step 3. Convert the advice into a usable form. Using techniques from *knowledge compilation*, a learner can convert ("operationalize") high-level advice into a (usually larger) collection of directly interpretable statements (see Gordon & Subramanian 1994; Mostow 1982). In many task domains, the advice-giver may wish to use natural, but imprecise, terms such as "near" and "many." A compiler for such terms will be needed for each general environment. Our compiler is based on the methods proposed by Berenji and Khedkar (1992) for representing fuzzy-logic terms in neural networks. Note that during training the initial definitions of these terms can be refined, possibly in context-dependent ways.

Table 1: Samples of advice in our advice language.

Advice	Pictorial Version
IF An Enemy IS (Near ∧ West) ∧ An Obstacle IS (Near ∧ North) THEN MULTIACTION MoveEast MoveNorth END;	
WHEN Surrounded ∧ OKtoPushEast ∧ An Enemy IS Near REPEAT PushEast MoveEast UNTIL ¬ OKtoPushEast ∨ ¬ Surrounded	

Step 4. Integrate the reformulated advice into the agent's current knowledge base. We use ideas from *knowledge-based neural networks* to directly install the operationalized advice into the connectionist representation of the utility function. In one such approach, KBANN (Towell, Shavlik, & Noordewier 1990), a set of propositional rules are re-represented as a neural network. KBANN converts a ruleset into a network by mapping the "target concepts" of the ruleset to output units and creating hidden units that represent the intermediate conclusions. It connects units with highly weighted links and sets unit biases (thresholds) in such a manner that the (non-input) units emulate AND or OR gates, as appropriate.

We extend the KBANN approach to the mapping of (simple) programs, as explained below. Unlike previous applications of knowledge-based neural networks, we allow rules to be installed *incrementally* into networks. That is, previous approaches first reformulated a ruleset then refined it using backpropagation. We allow new rules (i.e., advice) to be inserted into the network at any time during learning.

Table 1 shows some sample advice one might provide to an agent learning to play a video game. We will use it to illustrate the process of integrating advice into a neural network. The left column contains advice in our programming language, and the right shows the effects of the advice. A grammar for our advice language appears elsewhere (Maclin & Shavlik 1994).

We have made three extensions to the standard KBANN algorithm: (i), we allow advice that contains multi-step plans; (ii), advice can contain loops; (iii), advice can refer to previously defined terms. In all three cases incorporating advice involves adding hidden units representing the advice to the existing neural network, as shown in Figure 2. Note that the inputs and outputs to the network remain unchanged; the advice only changes how the function from states to the utility of actions is calculated.

As an example of a multi-step plan, consider the first entry in Table 1. Figure 3 shows the network additions that represent this advice. We first create a hidden unit (labeled A) that represents the conjunction of (i) an

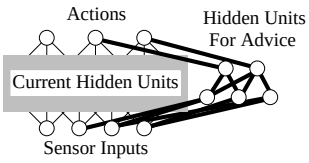


Figure 2: Advice is added to the neural network by adding hidden units that correspond to the advice.

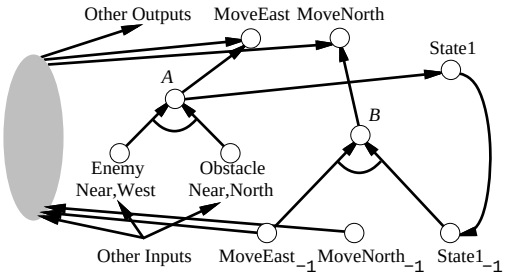


Figure 3: Translation of the first piece of advice. The ellipse at left represents the original hidden units. Arcs show units and weights set to make a conjunctive unit. We also add, as is typical in knowledge-based networks, zero-weighted links (not shown) to other parts of the current network. These links support subsequent refinement.

enemy being near and west[1] and (ii) an obstacle being adjacent and north. We then connect this unit to the action $MoveEast$, which is an existing output unit (recall that the utility function maps states to values of actions); this constitutes the first step of the two-step plan. We also connect unit A to a newly-added hidden unit called $State1$ that records when unit A was active in the previous state. We next connect $State1$ to a new input unit called $State1_{-1}$. This *recurrent* unit becomes active ("true") when $State1$ was active for the previous input (we need a recurrent unit to implement multi-step plans). Finally, we construct a unit (labeled B) that is active when $State1_{-1}$ is true and the previous action was a eastward move (the input includes the previous action taken in addition to the current sensor values). When active, unit B suggests moving north – the second step of the plan.

We assign high weights to the arcs coming out of units A and B. This means that when either unit is active, the total weighted input to the corresponding output unit will be increased, thereby increasing the utility value for that action. Notice that during subsequent training the weight (and the definition) of a piece of advice may be substantially altered.

The second piece of advice in Table 1 also contains a multi-step plan, but this time it is embedded in a REPEAT. Figure 4 shows the resulting additions to the network for this advice. The key to translating this construct is that there are two ways to invoke the two-step plan. The plan executes when the WHEN condition

[1] A unit recognizing this concept, "enemy near and west," is creating using a technique similar that in Berenji and Khedkar (1992); for more details see Maclin and Shavlik (1994).

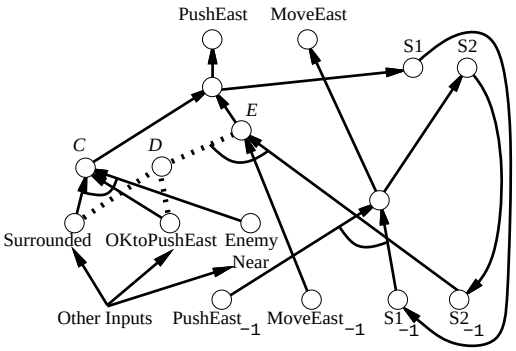


Figure 4: Translation of the second piece of advice. Dotted lines show negative weights. As with all translations, the units shown are added to the existing network.

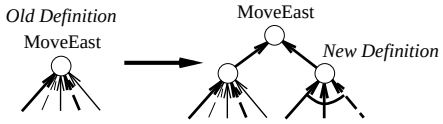


Figure 5: Incorporating the definition of a term that already exists.

is true (unit C) and also when the plan was just run and the UNTIL condition is false. Unit D is active when the UNTIL condition is met, while unit E is active when the UNTIL is unsatisfied and the agent's two previous actions were pushing and then moving east.

A final issue for our algorithm is dealing with advice that involves previously defined terms. This frequently occurs, since advice generally indicates new situations in which to perform existing actions. Figure 5 shows how we address this issue. We add a new definition of an existing term by first creating the representation of the added definition and making a copy of the unit representing the existing definition. We create a new unit, which becomes the term's new definition, representing the disjunction of the old and new definitions.[2] This process is analogous to how KBANN processes multiple rules with the same consequent.

Once we insert the advice into the RL agent, it returns to exploring its environment, thereby integrating and refining the advice. This is a key step because we cannot determine the optimal weights to use for the new piece of advice; instead we use RL to fine tune it.

Step 5. Judge the value of the advice. We currently rely on Q-learning to "wash out" poor advice. One can also envision that in some circumstances – such as a game-learner that can play against itself (Tesauro 1992) or when an agent builds an internal world model (Sutton 1991) – it would be straightforward to empirically evaluate the new advice. It would also be possible to allow the observer to retract or counteract bad advice.

[2] The process in Figure 5 would be used when adding the network fragments shown in Figures 3 and 4, assuming the advice came after the learner began exploring and learning.

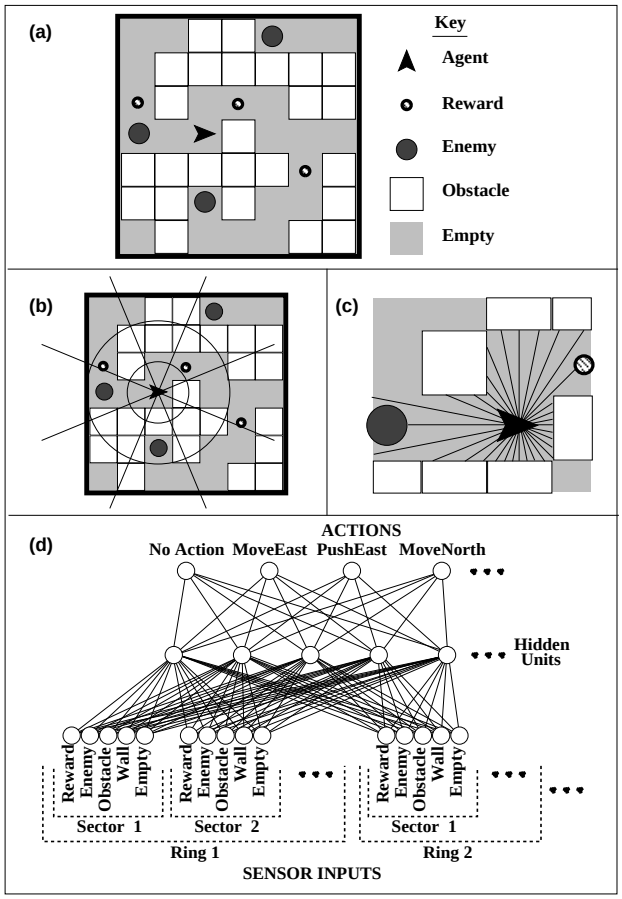


Figure 6: Our test environment: (a) sample configuration; (b) sample division of the environment into sectors; (c) distances measured by the agent's sensors; (d) a neural network that computes the utility of actions.

Experimental Study

We next empirically judge the value of our approach for providing advice to an RL agent.

Testbed

Figure 6a illustrates our test environment. Our task is similar to those explored by Agre and Chapman (1987) and Lin (1992). The agent can perform nine actions: *moving* and *pushing* in the directions East, North, West and South; and *doing nothing*. Pushing moves the obstacles in the environment – when the agent is next to an obstacle and pushes it, the obstacle slides until it encounters another obstacle or the board edge.

The agent receives reinforcement signals when: (i) an enemy eliminates the agent by touching the agent (–1.0); (ii) the agent collects one of the reward objects (+0.7); and (iii) the agent destroys an enemy by pushing an obstacle into it (+0.9). Each enemy moves randomly unless the agent is in sight, in which case it moves toward the agent.

We do not assume a global view of the environment, but instead use an agent-centered sensor model. It is based on partitioning the world into a set of sectors around the agent (see Figure 6b). The agent calculates the percentage of each sector that is occupied by each type of object – reward, enemy, obstacle, or wall. These percentages constitute the input to the neural network (Figure 6d). To calculate the sector occupancy, we assume the agent is able to measure the distance to the nearest occluding object along a fixed set of angles around the agent (Figure 6c). This means that the agent is only able to represent the objects in direct line-of-sight from the agent. Further details of our world model appear in Maclin and Shavlik (1994).

Methodology

We train the agents for a fixed number of *episodes* for each experiment. An episode consists of placing the agent into a randomly generated, initial environment, and then allowing it to explore until it is captured or a threshold of 500 steps is reached. Each of our environments contains a 7x7 grid with approximately 15 obstacles, 3 enemy agents, and 10 rewards. We use three randomly-generated sequences of initial environments as a basis for the training episodes. We train 10 randomly initialized networks on each of the three sequences of environments; hence, we report the averaged results of 30 neural networks. We estimate the average total reinforcement (the average sum of the reinforcements received by the agent)[3] by freezing the network and measuring the average reinforcement on a testset of 100 randomly-generated environments.

We chose parameters for our Q-learning algorithm that are similar to those investigated by Lin (1992). The learning rate for the network is 0.15, with a discount factor of 0.9. To establish a baseline system, we experimented with various numbers of hidden units, settling on 15 since that number resulted in the best average reinforcement for the baseline system.

After choosing an initial network topology, we then spent time acting as a advisor to our system, observing the behavior of the agent at various times. Based on these observations, we wrote several collections of advice. For use in our experiments, we chose four sets of advice (see Appendix), two that use multi-step plans (referred to as *ElimEnemies* and *Surrounded*), and two that do not (*SimpleMoves* and *NonLocalMoves*).

Results and Discussion

For our first experiment, we evaluate the hypothesis that our system can in fact take advantage of advice. After 1000 episodes of initial learning, we measure the value of (independently) providing each of the four sets of advice. We train the system for 2000 episodes after adding the advice and then measure testset rein-

[3] We report the average total reinforcement rather than the average discounted reinforcement because this is the standard for the RL community. Graphs of the average *discounted* reward are qualitatively similar to those shown in the next section.

Table 2: Testset results for the baseline and the four different types of advice; each of the gains (over the baseline) in average total reinforcement for the four sets of advice is statistically significant at the $p < 0.01$ level (i.e., with 99% confidence).

Advice Added	Average Total Reinforcement
None (baseline)	1.32
SimpleMoves	1.92
NonLocalMoves	2.01
ElimEnemies	1.87
Surrounded	1.72

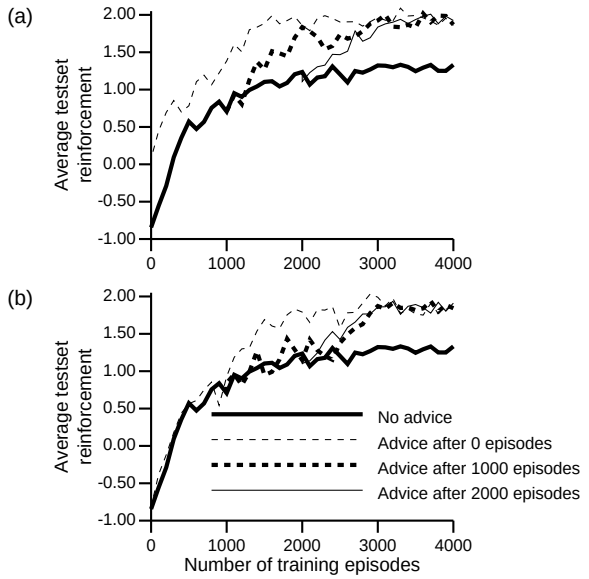


Figure 7: Testset results of (a) *SimpleMoves* and (b) *ElimEnemies* advice.

forcement. (The baseline is trained for 3000 episodes). Table 2 reports the average testset reinforcement; all gains over the baseline system are significant.

In our second experiment we investigate the hypothesis that the observer can beneficially provide advice at any time during training. To test this, we insert the four sets of advice at different points in training (after 0, 1000, and 2000 episodes). for the *SimpleMoves* and *ElimEnemies* advice respectively. These graphs indicate the learner does indeed converge to approximately the same expected reinforcement no matter when the advice is presented. It is also important to note that the effect of the advice may not be immediate – the agent may have to refine the advice over a number of training episodes. Results for the other pieces of advice are qualitatively similar to those shown in Figure 7.

Each of our pieces of advice addresses specific subtasks: collecting rewards (*SimpleMoves* and *NonLocalMoves*); eliminating enemies (*ElimEnemies*); and avoiding enemies, thus surviving longer (*SimpleMoves*, *NonLocalMoves*, and *Surrounded*). Hence, it is natural to ask how well each piece of advice meets its intent. Table 3 reports statistics on the components of the reinforcement. These statistics show that the pieces of advice do indeed lead to the expected improvements.

Table 3: Mean number of enemies captured, rewards collected, and number of actions taken for the experiments summarized in Table 2.

Advice Added	Enemies	Rewards	Survival Time
None (baseline)	0.15	3.09	32.7
SimpleMoves	0.28	3.79	39.6
NonLocalMoves	0.26	3.95	39.1
ElimEnemies	0.44	3.50	38.3
Surrounded	0.30	3.48	46.2

Future and Related Work

There are two tasks we intend to address in the near term. Our current experiments only demonstrate the value of giving a single piece of advice. We plan to empirically study the effect of providing multiple pieces of advice at different times during training. We also intend to evaluate the use of "replay" (i.e., periodic retraining on remembered pairs of states and reinforcements), a method that has been shown to greatly reduce the number of training examples needed to learn a policy function (Lin 1992).

There are a number of research efforts that are related to our work. Clouse and Utgoff (1992), Lin (1992), and Whitehead (1991) developed methods in which an advisor provides feedback to the learner – the advisor evaluates the chosen action or suggests an appropriate action. Lin (1993) also investigated a teaching method where the input to the RL system includes some of the previous input values. Thrun and Mitchell (1993) investigated RL agents that can make use of prior knowledge in the form of neural networks trained to predict the results of actions. These methods address the issue of reducing the number of training examples needed in RL; but, unlike our approach, they do not allow an observer to provide general advice.

Our work, which extends knowledge-based neural networks to a new task and shows that "domain theories" can be supplied piecemeal, is similar to our earlier work with the FSKBANN system (Maclin & Shavlik 1993). FSKBANN extended KBANN to deal with *state* units, but it does not create *new* state units.

Gordon and Subramanian (1994) developed a system similar to ours. Their agent accepts high-level advice of the form IF *conditions* THEN ACHIEVE *goal*. It operationalizes these rules using its background knowledge about goal achievement. The resulting rules are then incrementally refined using genetic algorithms, an alternate method for learning from the reinforcements an environment provides.

Finally, some additional research closely relates to our approach for instructing an agent. Nilsson (1994) developed a simple language for instructing robots, while Siegelman (1994) proposed, but has not yet evaluated, alternate techniques for converting programs expressed in a general-purpose, high-level language into recurrent neural networks.

Conclusions

We present an approach that allows an reinforcement learning agent to take advantage of suggestions provided by an external observer. The observer communicates advice using a simple programming language, one that does not require the observer to have any knowledge of the agent's internal workings. The advice is directly installed into a neural network that represents the agent's utility function, and then refined. Our experiments demonstrate the validity of this advice-taking approach.

Acknowledgements

We wish to thank C. Allex, M. Craven, D. Gordon, and S. Thrun for helpful comments on this paper.

Appendix – Four Sample Pieces of Advice

The four pieces of advice used in our experiments appear below. To make it easier to specify advice that applies in any direction, we defined the special term *dir*. During parsing, *dir* is expanded by replacing each rule containing it with four rules, one for each direction. Similarly we defined a set of four terms {*ahead, back, side1, side2*}. Any rule using these terms leads to *eight* rules – two for each case where *ahead* is East, North, West and South and *back* is appropriately set. There are two for each case of *ahead* and *back* because *side1* and *side2* can have two sets of values for a given value of *ahead* (e.g. if *ahead* is North, *side1* could be East and *side2* West, or vice-versa).

SimpleMoves

If An Obstacle is (NextTo ∧ *dir*) Then OkPush*dir*;
If No Obstacle is (NextTo ∧ *dir*) ∧ No Wall is (NextTo ∧ *dir*)
  Then OkMove*dir*;
If An Enemy is (Near ∧ ¬ *dir*) ∧ OkMove*dir* Then Move*dir*;
If A Reward is (Near ∧ *dir*) ∧ No Enemy is (Near ∧ *dir*) ∧
  OkMove*dir* Then Move*dir*;
If An Enemy is (Near ∧ *dir*) ∧ OkPush*dir* Then Push*dir*;

NonLocalMoves

If No Obstacle is (NextTo ∧ *dir*) ∧ No Wall is (NextTo ∧ *dir*)
  Then OkMove*dir*;
If Many Enemy are (¬ *dir*) ∧ No Enemy is (Near ∧ *dir*) ∧
  OkMove*dir* Then Move*dir*;
If An Enemy is (*dir* ∧ {Medium ∨ Far}) ∧ No Enemy is (*dir* ∧ Near)
  ∧ A Reward is (*dir* ∧ Near) ∧ OkMove*dir* Then Move*dir*;

ElimEnemies

If No Obstacle is (NextTo ∧ *dir*) ∧ No Wall is (NextTo ∧ *dir*)
  Then OkMove*dir*;
If An Enemy is (Near ∧ *back*) ∧ An Obstacle is (NextTo ∧ *side1*) ∧
  OkMove*ahead* Then MultiAction Move*ahead* Move*side1*
  Move*side1* Move*back* Push*side2* End;

Surrounded

If An Obstacle is (NextTo ∧ *dir*) Then OkPush*dir*;
If An Enemy is (Near ∧ *dir*) ∨ A Wall is (NextTo ∧ *dir*) ∨
  An Obstacle is (NextTo ∧ *dir*) Then Blocked*dir*;
If BlockedEast ∧ BlockedNorth ∧ BlockedSouth ∧ BlockedWest
  Then Surrounded;
When Surrounded ∧ OkPush*dir* ∧ An Enemy is Near
  Repeat Push*dir* Move*dir* Until ¬ OkPush*dir*;

References

Agre, P., & Chapman, D. 1987. Pengi: An implementation of a theory of activity. *AAAI-87*, 268–272.

Barto, A., Sutton, R., & Watkins, C. 1990. Learning and sequential decision making. In Gabriel, M., & Moore, J., eds., *Learning and Computational Neuroscience*. MIT Press.

Berenji, H., & Khedkar, P. 1992. Learning and tuning fuzzy logic controllers through reinforcements. *IEEE Trans. on Neural Networks* 3:724–740.

Clouse, J., & Utgoff, P. 1992. A teaching method for reinforcement learning. *Proc. 9th Intl. ML Conf.*, 92–101.

Cohen, P., & Feigenbaum, E. 1982. *The Handbook of Artificial Intelligence, Vol. 3*. William Kaufmann.

Gordon, D., & Subramanian, D. 1994. A multistrategy learning scheme for agent knowledge acquisition. *Informatica* 17:331–346.

Hayes-Roth, F., Klahr, P., & Mostow, D. J. 1981. Advice-taking and knowledge refinement: An iterative view of skill acquisition. In Anderson, J., ed., *Cognitive Skills and their Acquisition*. Lawrence Erlbaum.

Lin, L. 1992. Self-improving reactive agents based on reinforcement learning, planning, and teaching. *Machine Learning* 8:293–321.

Lin, L. 1993. Scaling up reinforcement learning for robot control. *Proc. 10th Intl. ML Conf.*, 182–189.

Maclin, R., & Shavlik, J. 1993. Using knowledge-based neural networks to improve algorithms. *Machine Learning* 11:195–215.

Maclin, R., & Shavlik, J. 1994. Incorporating advice into agents that learn from reinforcements. Technical Report 1227, CS Dept., Univ. of Wisconsin-Madison.

Mahadevan, S., & Connell, J. 1992. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence* 55:311–365.

McCarthy, J. 1958. Programs with common sense. *Proc. Symp. on the Mech. of Thought Processes, Vol. 1*, 77–84.

Mostow, D. J. 1982. Transforming declarative advice into effective procedures: A heuristic search example. In Michalski, R., Carbonell, J., & Mitchell, T., eds., *Machine Learning: An AI Approach, Vol. 1*. Tioga Press.

Nilsson, N. 1994. Teleo-reactive programs for agent control. *J. of Artificial Intelligence Research* 1:139–158.

Siegelmann, H. 1994. Neural programming language. *AAAI-94*, this volume.

Sutton, R. 1988. Learning to predict by the methods of temporal differences. *Machine Learning* 3:9–44.

Sutton, R. 1991. Reinforcement learning architectures for animats. In Meyer, J., & Wilson, S., eds., *From Animals to Animats*. MIT Press.

Tesauro, G. 1992. Practical issues in temporal difference learning. *Machine Learning* 8:257–277.

Thrun, S., & Mitchell, T. 1993. Integrating inductive neural network learning and explanation-based learning. *IJCAI-93*, 930–936.

Towell, G., Shavlik, J., & Noordewier, M. 1990. Refinement of approximate domain theories by knowledge-based neural networks. *AAAI-90*, 861–866.

Watkins, C. 1989. *Learning from Delayed Rewards*. Ph.D. Dissertation, King's College, Cambridge.

Whitehead, S. 1991. A complexity analysis of cooperative mechanisms in reinforcement learning. *AAAI-91*, 607–613.