Learning Strategies for Mid-Level Robot Control:
Some Preliminary Considerations and Experiments

Nils J. Nilsson
Robotics Laboratory
Department of Computer Science
Stanford University
Stanford, CA 94305

http://robotics.stanford.edu/users/nilsson/bio.html

nilsson@cs.stanford.edu

Draft of May 11, 2000

ABSTRACT

Versatile robots will need to be programmed, of course. But beyond explicit programming by a programmer, they will need to be able to plan how to perform new tasks and how to perform old tasks under new circumstances. They will also need to be able to learn.

In this article, I concentrate on two types of learning, namely supervised learning and reinforcement learning of robot control programs. I argue also that it would be useful for all of these programs, those explicitly programmed, those planned, and those learned, to be expressed in a common language. I propose what I think is a good candidate for such a language, namely the formalism of teleo-reactive (T-R) programs. Most of the article deals with the matter of learning T-R programs. I assert that such programs are PAC learnable and then describe some techniques for learning them and the results of some preliminary learning experiments. The work on learning T-R programs is in a very early stage, but I think enough has been started to warrant further development and experimentation. For that reason I make this article available on the web, but I caution readers about the tentative nature of this work. I solicit comments and suggestions at: nilsson@cs.stanford.edu.

I. Three-Level Robot Architectures

Architectures for the control of robots and other agents are often stratified into three levels. Working up from the motors and sensors, the *servo level* is in direct sensory control of effectors and uses various conventional and advanced control-theory mechanisms---sometimes implemented directly in hardware circuitry. Next, what I call the *teleo-reactive level* organizes the sequencing of servo-level actions so that they robustly react to unforeseen and changing environmental conditions in a goal-directed manner. Control at this level is usually implemented as computer programs that attempt to satisfy sub-goals specified by the level above. The top level, the *strategic level*, creates plans to satisfy user-specified goals. One of the earliest examples of this three-level control architecture was that used in Shakey, the SRI robot (Nilsson, 1984). There are several other examples as well (Connell, 1992).

There are various ways of implementing control at these levels---some of which support adaptive and learning abilities. I am concerned here primarily with the middle, teleo-reactive, level and with techniques by which programs at this level can learn. Among the proposals for teleo-reactive control are conventional computer programs with interrupt and sensor-polling mechanisms, so-called "behavior-based" control programs, neural networks (usually implemented on computers), finite-state machines using explicit state tables, and production-rule-like systems, such as the so-called "teleo-reactive" programs (Nilsson, 1994).

Some sort of adaptivity or machine learning seems desirable, possibly even required, for robust performance in dynamic, unpredictable environments. Two major kinds of learning regimes have been utilized. One is *supervised learning*, in which each datum in a specially gathered collection of sensory input data is paired with an action response known to be appropriate for that particular datum. This set of input/response pairs is called the *training set*. Learning is accomplished by adjusting the control mechanism so that it produces (either exactly or approximately) the correct action for each input in the training set.

The other type, *reinforcement learning*, involves giving occasional positive or negative "rewards" to the agent while it is actually performing a task. The learning process attempts to modify the control system in such a way that long-term rewards are maximized (without necessarily knowing for any input what is the guaranteed best action).

In one kind of supervised learning, the controller attempts to mimic the input/output behavior of a "teacher" who is skilled in the performance of the task being learned. This type is sometimes called *behavioral cloning* (Michie, *et al*., 1990; Sammut, *et al*., 1992; Urbancic & Bratko, 1994). A familiar example is the automobile-steering system called ALVINN (Pomerleau, 1993). There, a neural network connected to a television camera is trained to mimic the behavior of a human steering an automobile along various kinds of roads.

Perhaps the most compelling examples of reinforcement learning are the various versions of TD-Gammon, a program that learns to play backgammon (Tesauro, 1995). After playing several hundred thousand backgammon games in which rewards related to whether or not the game is won or lost are given, TD-gammon learns to play at or near world-championship level. Another example of reinforcement learning applied to a practical problem is a program for cell phone routing (Singh & Bertsekas, 1997).

II. The Programming, Teaching, Learning (PTL) Model

Although machine learning methods are important for adapting robot control programs to their environments, they by themselves are probably not sufficient for synthesis of effective programs from a blank slate. I believe that efforts by human programmers at various stages of the process will continue to be important---initially to produce a preliminary program and later to improve or correct programs already modified by some amount of learning. (Some of my ideas along these lines have been stimulated by discussions with Sebastian Thrun.)

The *programming par*t of what I call the PTL model involves a human programmer attempting to program the robot to perform its suite of tasks. The *teaching part* involves another human, a teacher, who shows the robot what is required (perhaps by “driving” it through various tasks). This showing produces a training set, which can then be used by supervised learning methods to clone the behavior of the teacher. The *learning part* shapes behavior during on-the-job reinforcement learning, guided by rewards given by a human user, a human teacher, and/or by the environment itself. Although not dealt with in this article, a complete system will also need, at the strategic level, a *planning part* to create mid-level programs for achieving user-specified goals. [A system called TRAIL was able to learn the preconditions and effects of low-level robot actions. It then used these learned descriptions in a STRIPS-like automatic planning system to create mid-level robot control programs (Benson, 1996).]

I envision that the four methods, programming, teaching, learning, and planning might be interspersed in arbitrary orders. It will be important therefore for the language(s) in which programs are constructed and modified to be languages in which programs are easy for humans to write and understand and ones that are compatible with machine learning and planning methods. I believe these requirements rule out, for example, C code and neural networks, however useful they might be in other applications.

III. Perceptual Imperfections

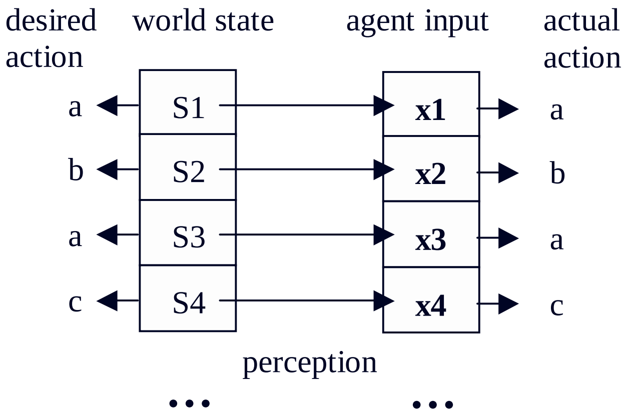
Robot learning must cope with various perceptual imperfections. Before moving on to discuss learning methods themselves, I first describe some perceptual difficulties. Effective robot control at the teleo-reactive level requires perceptual processing of sensor data in order to determine the state of the environment. Suppose, in so far as a given set of specific robot tasks is concerned, the robot’s world can be in any one of a set of states $\{S_i\}$. Suppose the robot’s perceptual apparatus transforms a world state, S, through a mapping, $\mathscr{P}$, to an input vector, $\mathbf{x}$. That is, so far as the robot is concerned, its knowledge of its world is given entirely by a vector of features, $\mathbf{x} = (x_1, x_2, . . ., x_n)$. (I sometimes abbreviate and call $\mathbf{x}$ the agent input even though the actual input is first processed by $\mathscr{P}$.)

Two kinds of imperfections in the perceptual mapping, $\mathscr{P}$, concern us. Because of random noise, $\mathscr{P}$ might be a one-to-many mapping, in which case a given world state might at different times be transformed into different input vectors. Or, because of inadequate sensory apparatus, $\mathscr{P}$ might be a many-to-one mapping, in which case several different world states might be transformed into the same input vector. This latter imperfection is called *perceptual aliasing*.

(One way to mitigate against perceptual aliasing is to keep a record in memory of a string of preceding input vectors; often, different world states are entered via different state sequences, and these different sequences may give rise to different perceptual histories.)

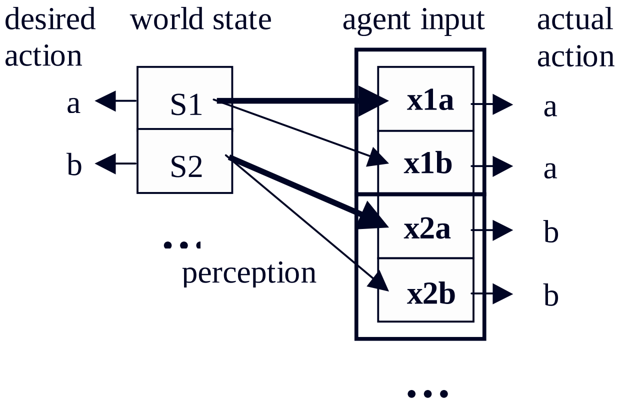
We can distinguish six interesting cases in which noise and perceptual aliasing influence the relationship between the action desired in a given world state and the actual action taken by an agent in the state it perceives. I describe these cases with the help of some diagrams.

Case 1 (no noise; no perceptual aliasing):



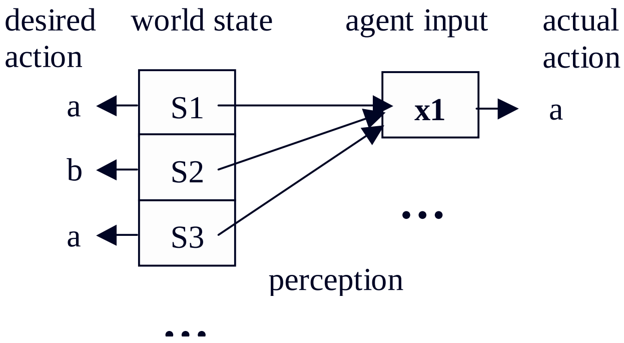
Here, each world state is faithfully represented by a distinct input vector so that the actual actions to be associated with inputs can match the desired actions. This is the ideal case. Note that different world states can have the same desired actions. (Taken in different world states, the same action may achieve different effects.)

Case 2 (minor noise; no perceptual aliasing):



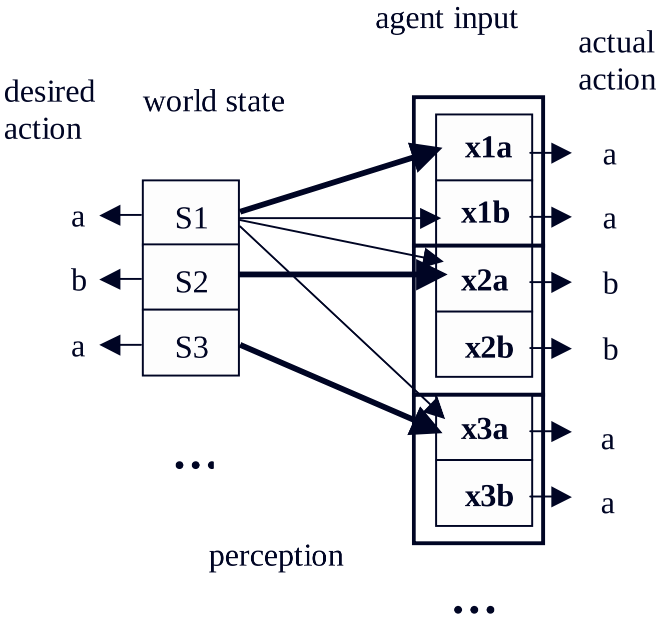
Here, each state is nominally perceived as a distinct input (represented by the dark arrows in the diagram), but noise sometimes causes the state to be perceived as an input only slightly different from the nominal one. We assume in this case that the noise is not so great as to cause the agent to mistake one world state for another. For such minor noise, the actual agent action can be the same as the desired action.

Cases 3 and 4 (perceptual aliasing; no noise):



In this example, perceptual aliasing conflates three different world states to produce the same agent input. In case 3, S1 and S2 have different desired actions, but since the agent cannot make this distinction it will sometimes execute an inappropriate action. In case 4, although S1 and S3 are conflated, the same action is called for, which is the action the agent correctly executes.

Cases 5 and 6 (major noise occasionally simulates perceptual aliasing):



Here, although each state is nominally differentiated by the agent's perceptual system (the dark arrows), major noise sometimes causes one world state to be mis-recognized as another. Just as in the case of perceptual aliasing, there are two different outcomes: in one (case 5), mis-recognition of S1 as S2 evokes an inappropriate action, and in the other (case 6), mis-recognition of S1 as S3 leads to the correct action. Unlike case 3, however, if mis-recognition is infrequent, case 5 will occur only occasionally, which might be tolerable.

In a dynamic world in which the agent takes a sequence of sensor readings, several adjacent ones can be averaged to reduce the effects of noise. Some of the case 5 mis-recognitions might then be eliminated but at the expense of reduced perceptual acuity. We will see examples of the difficulties these various imperfections cause in some learning experiments to be described later.

IV. Teleo-Reactive (T-R) Programs

A. The T-R Formalism

A *teleo-reactive (T-R)* program is an agent control program that robustly directs the agent toward a goal in a manner that continuously takes into account changing perceptions of the environment. T-R programs were introduced in two papers by Nilsson (Nilsson 1992, Nilsson 1994). In its simplest form, a T-R program consists of an ordered list of production rules:

$K_1 \rightarrow a_1$
. . .
$K_i \rightarrow a_i$
. . .
$K_m \rightarrow a_m$

The K_i are conditions on perceptual inputs (and possibly also on a stored model of the world), and the a_i are actions on the world (or that change the model). In typical usage, the condition K_1 is a goal condition, which is what the program is designed to achieve, and the action a_1 is the null action.

A T-R program is interpreted in a manner roughly similar to the way in which ordered production systems are interpreted: the list of rules is scanned from the top for the first rule whose condition part is satisfied, and the corresponding action is then executed. A T-R program is usually designed so that for each rule $K_i \rightarrow a_i$, K_i is the regression, through action a_i, of some particular condition higher in the list. That is, K_i is the weakest condition such that the execution of action a_i under ordinary circumstances will achieve some particular condition, say K_j, higher in the list (that is, with $j < i$). T-R programs designed in this way are said to have the *regression property*.

We assume that the set of conditions K_i covers *most* of the situations that might arise in the course of achieving the goal K_1. (Note that we do not require that the program be a *universal plan*, i.e. one covering all possible situations.) If an action fails, due to an execution error, noise, or the interference of some outside agent, the program will nevertheless typically continue working toward the goal in an efficient way. This robustness of execution is one of the advantages of T-R programs.

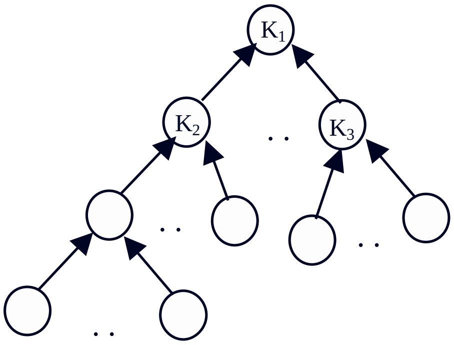
T-R programs differ substantively from conventional production systems, however, in that actions in T-R programs can be durative rather than discrete. A *durative* action is one that can continue indefinitely. For example, a mobile robot might be capable of executing the durative action move, which propels the robot ahead (say at constant speed). Such an action contrasts with a discrete one, such as move forward one meter. In a T-R program, a durative action continues only so long as its corresponding condition remains the highest true condition in the list. When the highest true condition changes, the current executing action immediately changes correspondingly. Thus, unlike ordinary production systems, the conditions must be continuously evaluated; the action associated with the *currently* highest true condition is *always* the one being executed. An action terminates when its associated condition ceases to be the highest true condition.

The regression condition for T-R programs must therefore be rephrased for durative actions: For each rule $K_i \rightarrow a_i$, K_i is the weakest condition such that continuous execution of the action a_i (under ordinary circumstances) eventually achieves some particular condition, say K_j, with $j < i$. (The fact that K_i is the weakest such condition implies that, under ordinary circumstances, it remains true until K_j is achieved.)

In a general T-R program, the conditions K_i may have free variables that are bound when the T-R program is called to achieve a particular ground instance of K_1. These bindings are then applied to all the free variables in the other conditions and actions in the program. Actions in a T-R program may be primitive, they may be sets of actions executed simultaneously, or they may themselves be T-R programs. Thus, recursive T-R programs are possible. (See Nilsson, 1992 for examples.)

When an action in a T-R program is itself a T-R program, it is important to emphasize that the usual computer science control structure does *not* apply. The conditions of *all* of the nested T-R programs in the hierarchy are *always* continuously being evaluated! The action associated with the highest true condition in the highest program in the stack of “called” programs is the one that is evoked. Thus, any program can always regain control from any of those that it causes to be called---essentially interrupting any durative action in progress. This responsiveness to the current perceived state of the environment is another one of the advantages of T-R programs.

Sometimes it is useful to represent a T-R program as a tree, called a T-R tree, as shown below:



Suppose two rules in a T-R program are $K_i \rightarrow a_i$ and $K_j \rightarrow a_j$ with $j < i$ and with K_i the regression of K_j through action a_i. Then we have nodes in the T-R tree corresponding to K_i and K_j and an arc labeled by a_i directed from K_i to K_j. That is, when K_i is the shallowest true node in the tree, execution of its corresponding action, a_i, should achieve K_j. The root node is labeled with the goal condition and is called the *goal node*. When two or more nodes have the same parent, there are correspondingly two or more ways in which to achieve the parent's condition.

Continuous execution of a T-R tree would be achieved by a continuous computation of the shallowest true node and execution of its corresponding action. (Ties among equally shallow *True* nodes can be broken by some arbitrary but fixed tie-breaking rule.) We call the shallowest true node in a T-R tree the *active node*.

The "backward-from-the-goal" approach to writing T-R programs makes them relatively easy to write and understand, as experience has shown.

B. T-R programs and Decision Lists

Decision lists are a class of Boolean functions described by Rivest (Rivest, 1987). In particular, the class *k*-DL(*n*) consists of those functions that can be written in the form:

$K_1 \rightarrow v_1$
. . .
$K_i \rightarrow v_i$
. . .
$K_m \rightarrow v_m$

where:

1) each K_i (for $i = 1, \ldots, m-1$) is a Boolean term over n variables consisting of at most k literals, and $K_m = T$ (having value *True*). (A *term* is a conjunction of literals, and a *literal* is a Boolean variable or its complement, having value *True* or *False*.)

and

2) each v_i is either *True* or *False*.

The value of a *k*-DL(*n*) function represented in this fashion is that v_i corresponding to the *first* K_i in the list having value *True*. Note that if none of the K_i up to and including K_{m-1} has value *True*, the function itself will have value v_m.

T-R programs over n variables whose conditions are Boolean terms having at most k literals are thus a generalization of the class *k*-DL(n), a generalization in which the v_i may have $q > 2$ different values. Let us use the notation *k*-TR(*n*,*q*) to represent this class of T-R programs. Note that *k*-TR(*n*,2) = *k*-DL(*n*).

V. Learnability of T-R Programs

Since it appears that T-R programs are not difficult for humans to write and understand, I now come to the topic of machine-learning of T-R programs. First I want to make some remarks stemming from the fact that T-R programs whose conditions operate on binary inputs are multi-output generalizations of decision lists. Rivest has shown that the class *k*-DL(*n*) of decision lists is polynomially PAC learnable (Rivest, 1987). To do so, it is sufficient to prove that:

1) the size of the class of *k*-DL(*n*) = $O(2^{n^t})$, where *n* is the dimensionality of the input and *t* is some constant,

and,

2) one can identify in polynomial time a member of the class *k*-DL(*n*) that is consistent with the training set.

The first requirement was shown to be satisfied using a simple, worst-case counting argument, and the second was shown by construction using a greedy algorithm.

It is straightforward to show by analogous arguments that both requirements are also met by the class *k*-TR(*n*, *q*). Therefore, this class is also polynomially PAC learnable.

Even though much experimental evidence suggests that PAC learnability of a class of functions is not necessarily predictive of whether or not that class can be usefully and practically learned, the fact that this subclass of T-R programs is polynomially PAC learnable is a point in their favor.

VI. The Squish Algorithm for Supervisory Learning of T-R Programs

George John (John, 1994) proposed an algorithm he called *Squish* for learning a T-R program to mimic the performance of a teacher (behavioral cloning). Some limited experimental testing of this algorithm has been performed---some using simulated robots and some using a physical robot. These experiments will be described shortly.

Squish works as follows. An agent is “steered” by a teacher in the performance of some task. By steering, I mean that the teacher, observing the agent and the agent’s environment, controls the agent’s actions until it achieves the goal (or one of the goals) defined by the task. Squish collects the perceptual/action history of this experience. To do so, the string of perceptual input vectors is sampled (at some rate appropriate to the task), and the action selected by the teacher at each sample point is noted. Several such histories are collected.

The result of this stage will be a collection of strings such as the following:

$\mathbf{x}_{11}a_{11}\mathbf{x}_{12}a_{12}\mathbf{x}_{13}a_{13} \ldots \mathbf{x}_{1n}a_{1n}\mathbf{x}_{Gn}$

. . .

$\mathbf{x}_{i1}a_{i1}\mathbf{x}_{i2}a_{i2}\mathbf{x}_{i3}a_{i3} \ldots \mathbf{x}_{im}a_{im}\mathbf{x}_{Gi}$

. . .

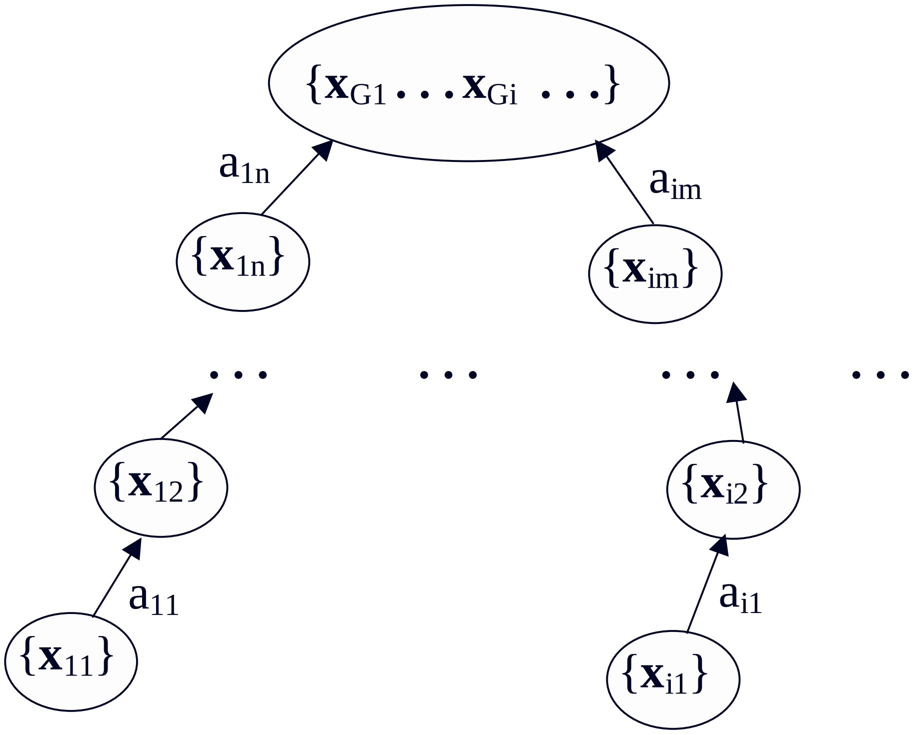
Each $\mathbf{x}_{ij}$ is a vector of inputs (obtained by perceptual processing by the agent), and each a_{kl} is the action selected by the teacher for the input vector preceding that action in the string. The vectors $\mathbf{x}_{Gi}$ are inputs that satisfy the goal condition for the task.

Note that each such string can be thought of as a T-R program of the form:

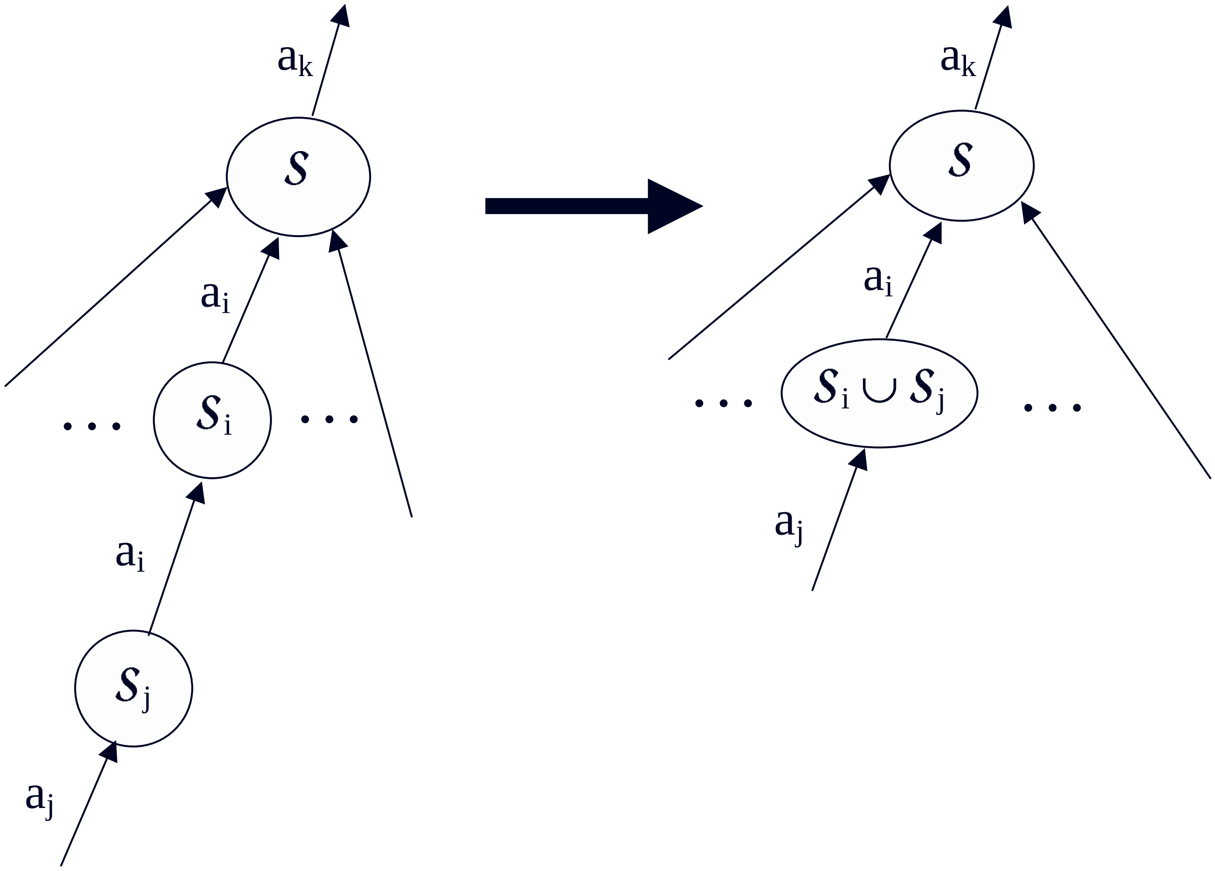
$\{\mathbf{x}_{Gi}\} \rightarrow$ Nil
$\{\mathbf{x}_{im}\} \rightarrow a_{im}$
. . .
$\{\mathbf{x}_{i1}\} \rightarrow a_{i1}$

where the singleton sets $\{\mathbf{x}_{im}\}$ represent conditions satisfied only if the input vector is, in fact, a member of the set.

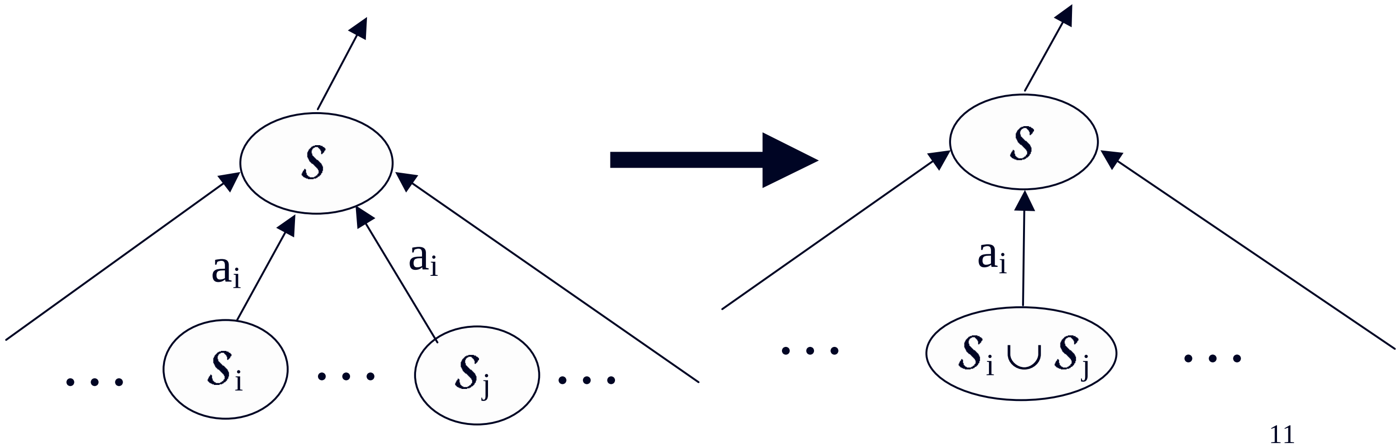
Since T-R programs can take the form of trees, we can combine all of the learning sequences into a T-R tree as shown below:



Of course the program represented by such a tree could evoke actions only for those exact inputs that occurred during the teaching process. That limitation (as well as the potentially large size of the tree) motivates the remaining stages of the Squish algorithm. First, we collapse (squish) chains of identical actions. (For these, obviously, the same action endured through multiple samplings of the input.) We illustrate this process by the diagram below:



Next, beginning with the top node and proceeding recursively, we look for any immediate successors of a node that evoke the same action. These siblings are combined into a single node labeled by the union of the sets labeling the siblings. We illustrate this process by the diagram below:



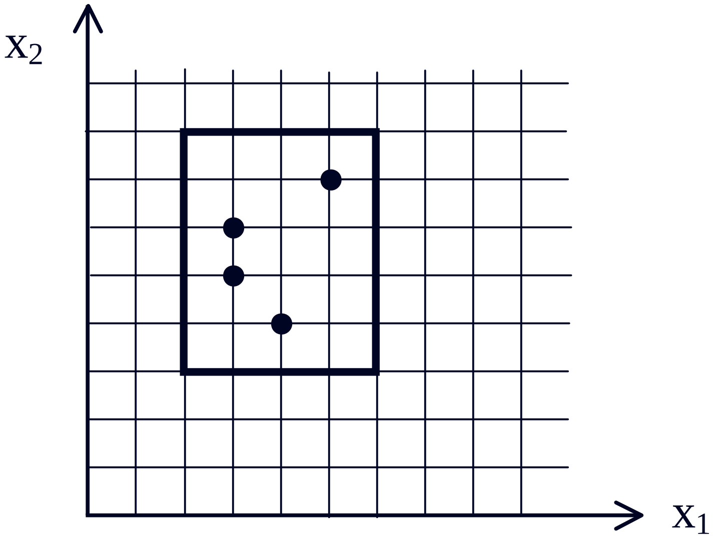
Finally, no more collapsing of these sorts can be done, and we are left with a tree whose nodes are labeled by sets of input vectors and whose arcs are labeled by actions.

Still, the conditions at the nodes are satisfied only by the members of the sets labeling those nodes; there is no generalization to similar inputs. To deal with this defect, we use machine learning methods to replace each set by a more general condition that is satisfied by all (or most) members of the set. (Perhaps it would be appropriate to relax “all” to “most” in the presence of noise.)

There are at least three ways in which this generalization might be accomplished. In the first, a connected region of multi-dimensional space slightly bigger, say, (or perhaps smaller in the case of noise) than the convex hull of the members of the set is defined by bounding surfaces---perhaps hyperplanes parallel to the coordinate axes. If such hyperplanes are used, the condition of being in the region can be given by a conjunction of expressions defining intervals on the input components. Such a condition would presumably be easy to understand by a human programmer inspecting the result of the learning process. A two-dimensional example might be illuminating. Suppose the inputs in a certain set are:

(3,5), (3,6), (5,7), and (4,4)

Each input lies within the box illustrated below:



The conditions associated with this set of four inputs would be:

2 ≤ x1 ≤ 6, and
3 ≤ x2 ≤ 8

In this manner a T-R program consisting of interval-based conditions with their associated actions is the final output of the teacher-guided learning process.

In another method of generalizing the condition at a node, the inputs labeling a node of the tree are identified as positive instances, and the inputs at all those nodes not ancestral to that node are labeled as negative instances. Then, one of a variety of machine learning methods can be used to build a classifier that discriminates between positive and negative instances for each node in the T-R tree. If the conditions are to be easily understood by human programmers, one might learn a decision tree whose nodes are intervals on the various input parameters. The condition implemented by a decision tree can readily be put in the form of a conjunction of interval tests. Alternatively, one could use a neural-net-based classifier. (John's original suggestion was to use a maximum-likelihood classifier.)

Another method for generalization uses a "nearest-neighbor" calculation. First, in each node any repeated vectors are eliminated. A new input vector triggers that node having a vector that is closest to the new input vector (in a squared-difference sense)---giving preference to nodes higher in the tree in case of a tie.

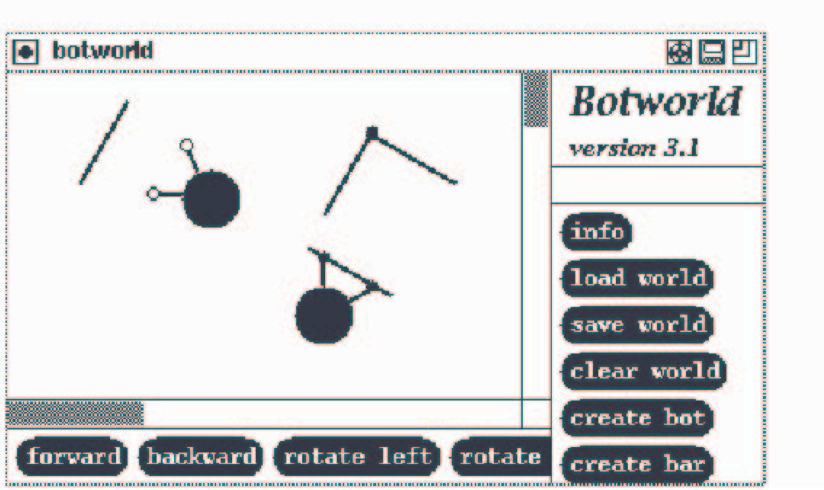
VII. Experiments with Squish

A. Experiments with Simulated Robots

1. The task and experimental set-up

John used Squish (with a maximum-likelihood classifier) to have a robot learn how to grab an object in a simulated two-dimensional world called *Botworld* (Benson, 1993). In this simulated world, there was no perceptual aliasing.

The Botworld environment has had several instantiations. In John's experiments, Botworld appeared as in the screen shot below:



The round objects are simulated robots, called “bots,” which can move forward, turn, and grab and hold a “bar” with their “arms” as shown in the figure. Using the buttons on the graphical interface, a teacher could drive the bot during training sessions.

For the learning experiments to be described, John endowed the bot with the following perceptual predicates:

Grabbing: Has value *True* if and only if the bot is holding the bar)

At-bar: Has value *True* if and only if the bot is in the right position to grab a bar (it must be at just the right distance from the bar)

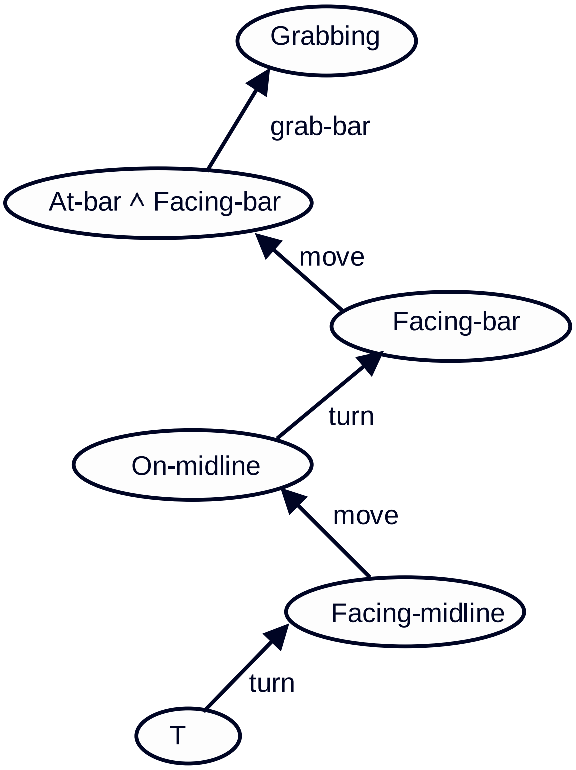
Facing-bar: Has value *True* if and only if the bot is facing the bar)

On-midline: Has value *True* if and only if the bot is on the imaginary line that is the perpendicular bisector of the bar)

Facing-midline: Has value *True* if and only if the bot is facing a certain "preparatory area" segment of the midline)

Because a bot's heading and position were represented by real numbers, all of these predicates (except Grabbing) involved tolerance intervals.

The bot had two durative actions, namely turn and move and one “ballistic” action, namely grab-bar. A T-R tree for bar grabbing using these actions and these perceptual predicates is shown below:



2. Learning experiments

The bot was "driven" to grab a bar a few times in order to generate a training set. The input vectors were composed of the values of the five perceptual predicates as the bot was driven. Squish was used to generate a T-R tree, and a maximum-likelihood classifier was established at each node. The vectors at a node were the positive instances for that node, and all of the vectors at nodes lower in the tree were the negative instances for that node.

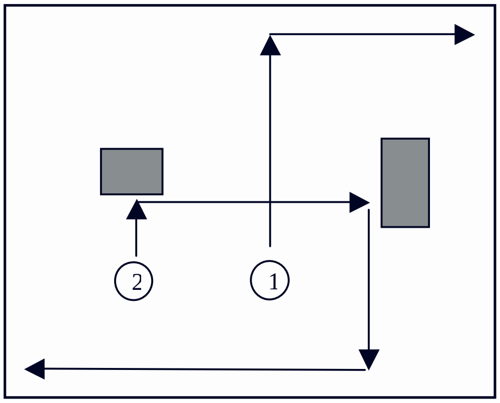
According to John (unpublished private communication): ". . . it did work most of the time, meaning that afterwards it could drive itself (to grab the bar), but this was only if I drove the bot using my knowledge of which features it (the lisp code) could observe, and only if I was pretty careful to drive it well. It would break if the driver wasn't very good. This is a common problem in programming by demonstration---how to get the driver or demonstrator to understand the features that the learning algorithm can observe, so that the instruction can be productive."

B. Experiments with a Nomad Robot

1. The task and experimental set-up

In the next set of experiments, Thomas Willeke (Willeke, 1998) wrote a T-R program for a real robot to enable it to perform the simple task of "corner-finding." The task involved moving forward perpendicular to one of the walls of a rectangular enclosure, turning 90 degrees to the right whenever the robot's motion was impeded by a wall or an obstacle. The robot continued these actions until it sensed that it was in one of the corners of its enclosure.

The robot used was a Nomad 150 from Nomadic Technologies, Inc. (See http://www.robots.com/n150.htm for full technical details.) The Nomad 150 is a wheeled cylindrical base whose only external sensors are sixteen sonar transceivers evenly positioned around its circumference. Thus, the input vector, **x**, is a 16-dimensional vector whose components are sonar-measured distances to objects and walls. These vectors have different typical forms that can be used to distinguish situations such as: I am in (relatively) free space, there is a wall or obstacle in front of me, there is a wall on my left (or right) side, and I am in a corner. The experimental set-up and desired behaviors are shown below:



When starting from position 1, for example, the robot moves forward until it comes close to a wall. It then turns to the right and proceeds to a corner. When starting in position 2, it moves until it comes close to the object, turns right, proceeds to the other object, turns right, proceeds to the wall, turns right and proceeds to a corner.

It may be argued that, since the desired action (turn right) when blocked by an obstacle is the same as the desired action when blocked by a wall, these two states (although different in the world) are the same so far as the robot is concerned. For some classes of control programs it does no harm to conflate these states (as the Nomad robot does). But since, in general, we would like T-R programs to have the regression property, these states ought to be treated differently since the actions taken in them have different effects.

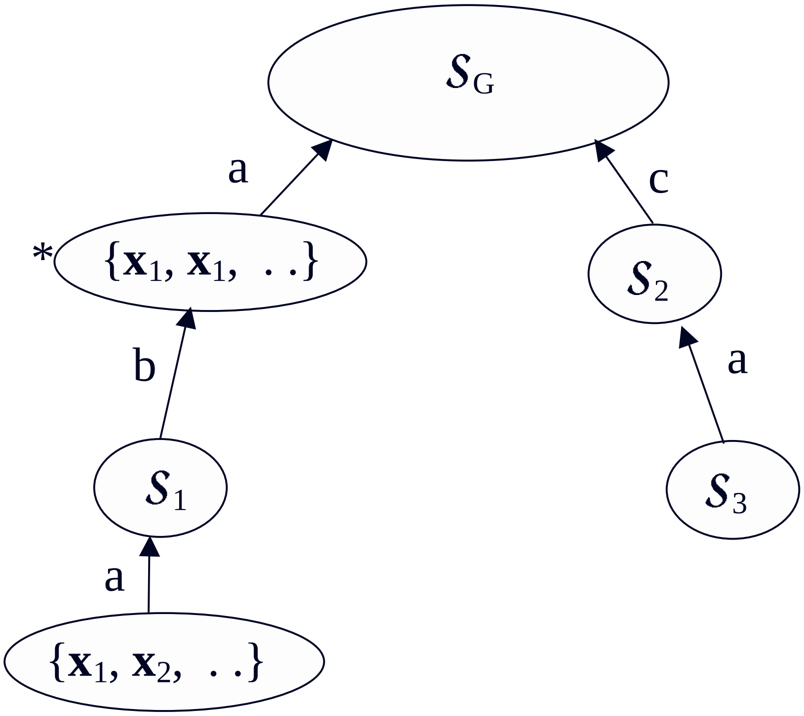
Corner-finding T-R programs are not difficult to write. Sonar noise can create some problems, however. In Willeke's words: "There comes a point in the turn when, due to sonars bouncing at steep angles off the wall, the perceptual input is exactly similar to that of the robot following along a wall. (See case 5 mentioned above.) So the T-R program changes state and the robot tends to leap forward (thinking that it has already completed the turn), slamming itself into the wall." Willeke's solution to this problem: ". . . we simply hand wrote in code so that the turns became ballistic actions. So, after determining that a turn was in order, the robot simply turned 90 degrees."

As mentioned previously, another way of dealing with sonar (or any) noise is to do some averaging. As Willeke suggests: "One simple solution to the turning bug would be to simply require that three sonar readings in a row be different before switching states. This, of course, makes the robot less reactive to sudden changes in the environment. And, of course, if the robot still ends up in the wrong state it will be even slower to recover. We did implement this idea and it clearly helped. The robot made turns that were much closer to the 90 degrees we wanted. It no longer slammed into the wall part way through the turn. But it still had an exit condition problem. Because the end of a turn and normal forward motion look very much the same, the robot didn't usually make it exactly 90 degrees around before restarting forward motion. This caused it to drift into the wall later, or drift off away from the wall if it over-turned."

2. Learning experiments

The training data was collected by driving the robot around with a joystick and recording the sensor values and joystick commands. Different runs were performed with different starting positions and configurations of obstacles but always ending in a corner. Willeke used the Squish algorithm to collapse the data, and in one set of experiments he trained a simple threshold logic unit (TLU) at each node of the T-R tree to generalize the input vectors at that node.

In training the TLU, Willeke used a modification of the procedure in which the inputs labeling a node of the tree are identified as positive instances and the inputs at all those nodes not ancestral to that node are labeled as negative instances. The requirement for the modification results from an instance of the case 4 situation mentioned above. The robot sometimes makes more than one right turn in a single data run, and the situation just before each turn produces identical (or very similar) input vectors. In these runs, positive input vectors associated with one node in the tree will be identical (or very similar) to negative input vectors at a lower node in the tree. The diagram below illustrates this problem and our modification:


S_G
a
c
*
$\{\mathbf{x}_1, \mathbf{x}_1, \ . . \}$
S_2
b
a
S_1
S_3
a
$\{\mathbf{x}_1, \mathbf{x}_2, \ . . \}$

In learning a condition to use at the node marked *, we take the vectors in $\{\mathbf{x}_1, \mathbf{x}_1, . . .\}$ as the positive instances and those in S_1 and S_2 as the only negative instances. In particular, we exclude the vectors in $\{\mathbf{x}_1, \mathbf{x}_2, . . .\}$ and those in S_3 from the set of negative instances because the action, a, at those nodes is identical to the action at the node marked *.

In general then, the modification involves throwing out all input vectors at the non-ancestral nodes that are associated with the same action as that at the node for which we are learning a condition. (Willeke points out some possible difficulties that this modification entails if it is used in conjunction with procedures that use memorized state information in addition to sensory inputs, but these need not concern us here.)

In several experiments of this sort for learning a corner-finding T-R program, Willeke states that the (modified) algorithm worked “surprising well.”

In another set of experiments, Willeke used a nearest-neighbor generalization scheme. Since this method does not involve separating inputs into positive and negative classes, no special consideration need be given to case 4 repeated action situations. These experiments were as successful as those in which TLUs were trained at each node. The robot was able to learn corner-finding behaviors from complex training runs that repeated perceptual states. The method does require remembering all of the inputs accumulated during training and extensive computation at run time, but recent work of A. J. Moore (Moore, 2000) appears to enable nearest-neighbor methods to scale well to large problems.

The “interval-box” method for generalizing the nodal conditions was not tried, but, just as in the nearest-neighbor method, it would not have required separating inputs into positive and negative classes.

The success of these preliminary experiments suggests that it would be reasonable to try these methods on more complex tasks performed by more complex robots.

VIII. Proposals for Reinforcement Learning of T-R Programs

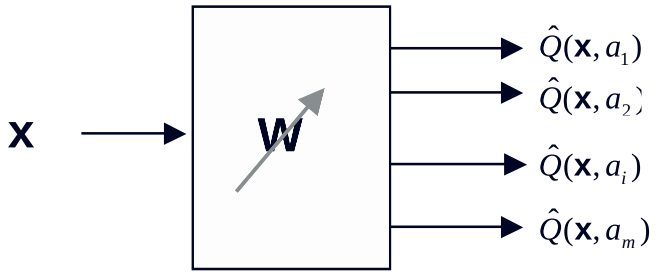
A. Neural Network Q-Learning

In reinforcement learning, “reward” amounts are given by a teacher (or by the environment itself) when a robot takes actions and thereby enters certain states. The rewards can be positive or negative. The rewards are used to change the “action policy” of the robot. One seeks a training method that results in an action policy that maximizes some function of the future expected reward. (Sutton & Barto, 1998) is a text on reinforcement learning.)

In a version of reinforcement learning called *Q-learning*, first proposed by Watkins (Watkins, 1989), the action policy is based on a function over perceived states and actions, $Q(\mathbf{x},a)$, where $\mathbf{x}$ is the perceived state (say an input vector), and a is a robot action. For any state $\mathbf{x}$, the robot takes that action a which maximizes $Q(\mathbf{x},a)$ over all possible actions.

In reinforcement learning, one seeks to learn an optimal policy by making adjustments to a trial policy in response to rewards for actions taken. Learning a policy can be accomplished by learning an estimate, $\hat{Q}$, of the optimal Q function.

Neural network methods have been proposed for implementing a policy and for learning a Q function (Lin, 1992, Tesauro 1995). Consider the neural network shown below:


x
W
$\hat{Q}(\mathbf{x},a_1)$
$\hat{Q}(\mathbf{x},a_2)$
$\hat{Q}(\mathbf{x},a_i)$
$\hat{Q}(\mathbf{x},a_m)$

$$a = \arg\max_i \hat{Q}(\mathbf{x},a_i)$$

The network computes estimates of the Q function for each possible action and evokes that action, a, corresponding to the largest of these estimates. Reinforcement learning of

Q-values uses a temporal difference method, such as TD(0) (Sutton, 1988), for changing the estimates. Suppose action *a* is taken (by a partially trained network) in response to input vector **x**, and that the corresponding *Q*-value estimate is $\hat{Q}(\mathbf{x}, a)$. (That is, $\hat{Q}(\mathbf{x}, a)$ = $\max_i \hat{Q}(\mathbf{x}, a_i)$.) Suppose that the durative execution of this action ultimately leads to an input vector **y** which evokes some other action, *b*. TD learning assumes that the quantity $r + \gamma\hat{Q}(\mathbf{y}, b)$ is a more accurate estimate of $Q(\mathbf{x}, a)$ than is $\hat{Q}(\mathbf{x}, a)$ and updates $\hat{Q}(\mathbf{x}, a)$ to make it more closely equal $r + \gamma\hat{Q}(\mathbf{y}, \mathrm{b})$.

0< γ <1 is the "temporal discount factor," and *r* is the immediate reward for executing *a* in that circumstance. Just that single $\hat{Q}(\mathbf{x}, a_i)$ which is the largest of the estimates is updated, and all others are left unchanged. The updating formula for that $\hat{Q}(\mathbf{x}, a_i)$ is:

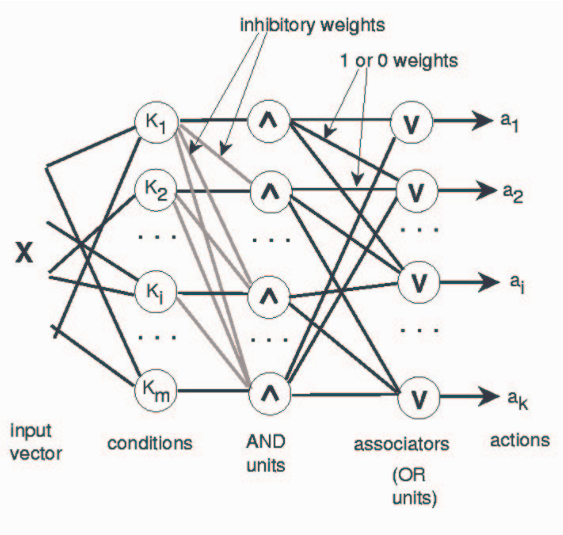
$$\hat{Q}(\mathbf{x}, a_i) \leftarrow \hat{Q}(\mathbf{x}, a) + \beta([r + \gamma\hat{Q}(\mathbf{y}, \mathrm{b})] - \hat{Q}(\mathbf{x}, a))$$

(That is, we move "β of the way" from $\hat{Q}(\mathbf{x}, a)$ to $r + \gamma\hat{Q}(\mathbf{y}, \mathrm{b})$, where $0 < \beta < 1$.)

Several researchers have used the standard backpropagation algorithm (Rumelhart *et al*., 1986) to effect these changes in the $\hat{Q}$ function implemented by neural networks.

B. Representing a T-R Program by a Neural Network

In order to use neural network *Q*-learning methods for learning T-R programs, we first have to be able to represent a T-R program by a neural network. There are several ways in which this might be done. We illustrate one method by the network below:



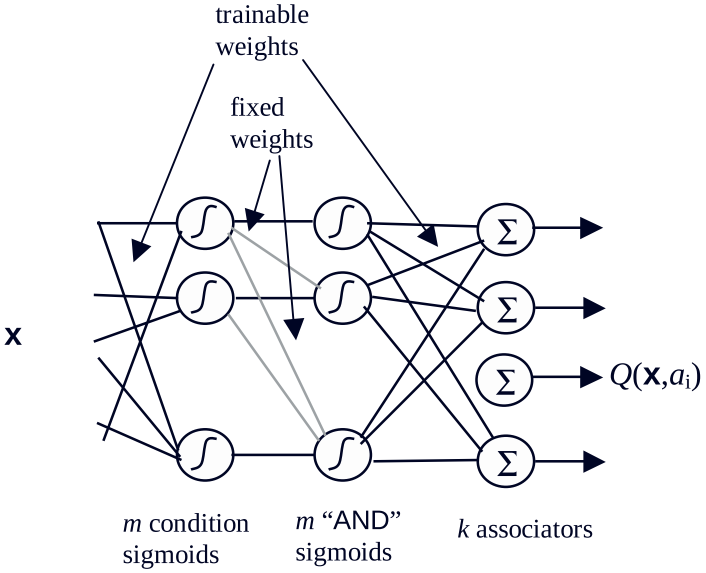
This network implements a T-R program whose *m* conditions (in order) are K_1, K_2, . . ., K_i, . . ., K_m. Assuming that the conditions are conjunctions of components of the input vector, **x**, these can all be implemented by the first layer of TLUs, as shown. Corresponding to each condition unit in the first layer is an AND unit in the second layer. These can also be implemented by TLUs. Each AND unit is wired up with appropriate inhibitory connections from the condition units so that the AND unit responds if and only if its associated condition unit (in the first layer) is the "highest" condition unit responding. That is, one and only one AND unit responds, and the one that does respond corresponds to the highest true condition among the K_i.

Now, we have only to associate the appropriate action with the highest true condition. This is done by a layer of OR associator units implemented, again, by TLUs. We have one such unit for each of the (let us say) *k* actions, a_1, a_2, . . ., a_i, . . ., a_k. An AND unit is wired up to an associator unit if and only if that associator unit corresponds to an action that is called for by the AND unit. Thus, an AND unit can be wired up to only one associator unit, but each associator unit might be wired up to more than one AND unit.

Thus, we have shown that a specific three-layer, feed-forward neural network can implement any T-R program. I will call such a neural network a T-R net.

C. Training a T-R Net by Q-learning

In order to use back-propagation to train a T-R net, we replace the TLUs by differentiable functions as usual. We replace the condition units and the AND units by sigmoids, and we replace the associator units by a simple summing device. We show the resulting network below:



The outputs of the summation unit associators are taken to be the Q values, $Q(\mathbf{x},a_i)$. In the case in which TLUs are used instead of sigmoids for the condition and AND units, one and only one of these Q values would have value 1; the others would have value 0. Selecting the action corresponding to the highest Q value would give us the appropriate action. We can regard the new network as implementing a "softened" version of this decision process---one that is amenable to backpropagation training.

In training the network only the weights in the first layer and the third layer are modified during the process of updating Q values. The weights in the second layer are left fixed. (These are the weights that implement the rule that the highest true condition evokes an action.) The training process thus creates the appropriate conditions and associates them with appropriate actions. Again, training can be accomplished by a standard backpropagation rule, constrained to leave the second-layer weights fixed. The conditions used by the resulting T-R program will now be linearly separable functions of the input components rather than simple conjunctions.

To date, no experiments have been conducted to test this technique. We note that even if training such a network succeeds, we are not guaranteed that the corresponding T-R program will satisfy the regression property. Perhaps some modification of the training process could be found that would achieve that restriction.

D. Learning a T-R Program by Reinforcement Learning of Prototype Points

I conducted some preliminary experiments at the Santa Fe Institute in 1991 on learning T-R programs using a method that combined cluster seeking with reinforcement learning. The experiments were with a simulated robot in "botworld," and the learning program was written in Lisp. (I still have the Lisp program, dated February 22, 1991.) The experiments were modestly successful and are described here with the hope that the method can serve as a starting point for further work.

The method learns regions and associated actions in the space of input vectors, **x**. Before discussing how the regions and actions are learned, I describe how they can be interpreted as T-R programs.

1. Region graphs

We define a set of regions in the space of input vectors by a set of prototype points $\mathcal{P} = \{\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_i, \ldots, \mathbf{P}_m\}$. The prototype points can be imagined as being at the centers of clusters of input vectors. The prototype points induce a set of m regions $\mathcal{R}$; Each region, R_i, contains all those input vectors that are closer to $\mathbf{P}_i$ than to any other $\mathbf{P}_j$, $j \neq i$.

We suppose that the $\mathbf{P}_i$ can be chosen in such a way that:

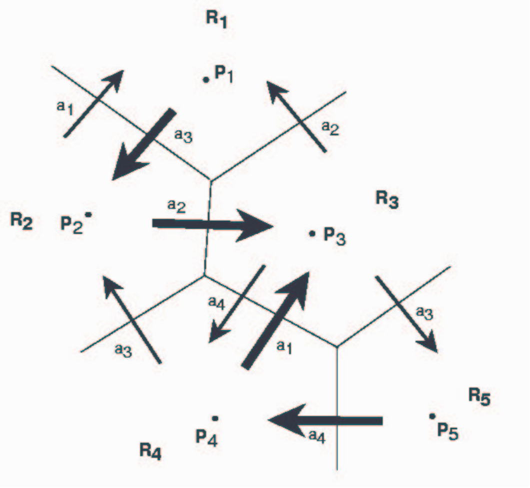
a) if an action can be executed at one point in a region, it can be executed at any point in that region, and

b) for every pair R_i and R_j, if continued execution of an action, a, at one point in R_i results in the input vector next moving to R_j , then for all points in R_i, continued execution of a results in the input vector next moving to R_j.

Thus, certain pairs of regions, R_i and R_j, are linked by actions, and we can define a directed graph whose nodes correspond to the regions (or, equivalently, to the prototype points defining the regions) and whose arcs correspond to actions. We shall call such a graph a *region graph*. Typically, only regions that are adjacent in the input space will have an action arc connecting them, although adjacency is neither necessary nor sufficient for linkage in the graph. Note also that the same action may label several different arcs in the graph and several arcs may emanate and converge on a single region node.

We use the region graph to describe the nominal preconditions and effects of the actions of the robot. An action, a_i, can be executed by the robot if and only if it labels an arc whose tail emanates from a region containing the current input vector. The result of continued execution of such an action is that the input vector traverses the region at the tail of the arc until it enters the region at the head of the arc. Thence, the control system of the robot must select an action corresponding to one of the outgoing arcs from that new region.

In the figure below, we show a two-dimensional example to illustrate these ideas.



There are five prototype points and corresponding regions. We illustrate the actions by arrows directed from one region to another. For example, action a_1 can be used either to go from R_4 to R_3 or from R_2 to R_1. Suppose the robot's task is to have the input vector in R_3. The heavy action arrows constitute a spanning tree of the region graph for this problem; regardless of the region of the initial input vector, executing an action corresponding to an outgoing heavy arrow will result ultimately in the input vector landing in R_3.

In terms of the region graph then, a robot's control strategy to achieve a given goal can be implemented by selecting that prototype point that is closest to the input vector and executing the action that labels the corresponding outgoing arc of the spanning tree for that goal. The spanning tree can be thought of as a T-R program: the conditions are membership in the regions, and the actions are the arcs of the spanning tree.

An alternative way of implementing the same strategy is to assign each region a value corresponding to its distance (counting action arcs) from the goal region. Then in any region, that action is selected that will next take the input vector to the accessible region having the smallest value. If the values of the regions are negated (so that larger values are preferred instead of smaller ones), then this strategy implements a version of a “policy function” for achieving maximum reward, suggesting that Q-learning techniques might be capable of learning the prototype points.

2. Reinforcement learning of prototype points and actions

Our learning algorithm starts with a set of prototype points, $\mathscr{P} = \{\mathbf{P}_1, \mathbf{P}_2, \ldots , \mathbf{P}_i, \ldots ,\mathbf{P}_m\}$, that are initially set to random values. Each prototype point is arbitrarily assigned one of the actions in the set of possible actions. To account for the possibility that the same action might be executed in different circumstances, multiple prototype points might be assigned the same action. As the robot executes actions during learning trials to achieve a particular goal, these points (and their corresponding regions) individually migrate through the input space (according to rules we shall describe) until they stabilize at positions defining an acceptable spanning tree of the region graph. Except when actions are selected by a teacher, soon to be discussed, that action is executed that is associated with the prototype point that is closest to the input vector.

Each prototype point is also assigned a scalar *mass* and a scalar *rank*, each initially set to zero, that are changed during learning and are used by the procedure that migrates the prototype points. The mass is supposed to represent how many times the action associated with its prototype point has been “successful,” and the rank is supposed to represent something similar to the “Q-value” of its corresponding region-action pair in terms of its proximity to the goal. In accord with the temporal-difference learning literature, the larger the rank or value of a region the closer it is to the goal. In the version of the algorithm described here, we assume we know a particular region of the input space that corresponds to achieving the goal. This region is assigned a large and unchangeable rank, say 100.

A learning run terminates whenever the robot enters the goal region. Learning can then continue by re-positioning the robot and beginning another run---using the prototypes, ranks, and masses learned in previous runs. Whenever the robot is not in the goal region and the closest prototype point has zero rank, as it will initially, the robot attempts to execute one of its actions, chosen randomly, for a random amount of time. Otherwise, the action corresponding to the prototype point that is closest to the input vector is executed. Whenever the rank of the prototype point of an executing action is above zero, the rank decays to zero at some constant rate during the execution of that action.

Learning takes place as follows: Suppose for some input, $\mathbf{x}$, not in the goal region, the closest prototype point and its corresponding action is denoted by the pair ($\mathbf{P}$,a). Suppose that continued execution of a results in a continuous stream of inputs for which $\mathbf{P}$ is still the closest prototype point, but that ultimately for some $\mathbf{x}'$ either the closest prototype point becomes $\mathbf{P}'$ or $\mathbf{x}'$ is in the goal region.

At that time, if $\mathbf{P}'$ is of higher rank than $\mathbf{P}$ (or if $\mathbf{x}'$ is in the goal region):

1. The rank of $\mathbf{P}$ is increased to an amount between its old rank (at that time) and the rank of $\mathbf{P}'$ (or of the goal region if $\mathbf{x}'$ is in the goal region), say to the average of the before and after ranks.

2. The mass of $\mathbf{P}$ is increased by 1.

3. $\mathbf{P}$ is changed to:

$$\mathbf{P} \leftarrow (m\,\mathbf{P} + \mathbf{x}_{\mathrm{avg}})/(m+1)$$

where m is the (old) mass of $\mathbf{P}$, and $\mathbf{x}_{\mathrm{avg}}$ is the average value of the input vector during the time a was being executed until the input vector became closer to $\mathbf{P}'$ than to $\mathbf{P}$ (or entered the goal region).

If $\mathbf{P}'$ is of lower rank than $\mathbf{P}$:

1. The rank of $\mathbf{P}$ is decreased to an amount between its old rank (at that time) and the rank of $\mathbf{P}'$, say to the average of the two.

2. The mass of $\mathbf{P}$ is left unchanged.

3. $\mathbf{P}$ is changed to:

$$\mathbf{P} \leftarrow (m\,\mathbf{P} - \mathbf{x}_{\mathrm{avg}})/(m+1)$$

In all cases, whenever the rank of a prototype point falls to zero (or, alternatively, below some low threshold value), the mass of that region is reset to zero.

The algorithm is a temporal-difference procedure (Sutton, 1988) because the rank of a prototype point is changed to make it closer to the rank of a temporally next region. It is a delayed reinforcement learning procedure because reward comes only when the input vector enters the goal region. We could make the algorithm more closely resemble classical reinforcement learning procedures by selecting actions according to a probability distribution that favors that action associated with the closest prototype point.

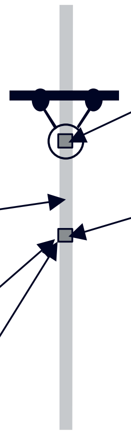
The algorithm is also a cluster-seeking procedure because the prototype points will tend to end up in the centers of clusters of frequently occurring input vectors for which the same action was successful in moving the input vector toward the goal. It is this latter property which gives the resulting action selection policy a capability to generalize over similar inputs (for which the same action ought to be selected). Experiments with a similar cluster-seeking, reinforcement procedure were conducted by (Mahadevan & Connell, 1992).

3. Preliminary experiments

Some preliminary experiments with this algorithm were conducted in 1991 in the botworld domain. The goal was to have a bot grab a bar. All reinforcement learning procedures work best when the task to be learned is learned “backwards” from large rewards. For our migrating prototype-point algorithm, backwards learning involves placing the robot near the goal or near prototype points already having been assigned high rank (such that random actions rather quickly get it close to the goal or to those prototype points). The high values of the prototype points previously learned are thus propagated backward to points associated with actions that move the robot toward these points learned earlier. With these facts in mind, I used a “teacher” to force backwards learning of a solution and then checked to see whether or not continued learning destroyed the solution. Solutions seemed to be stable; once achieved, further learning did not destroy them. It remains to be seen under what circumstances the algorithm can be used in a less guided fashion to obtain a solution in the first place.

There are some parameters, such as the rate of decay of a prototype point's rank, the increments by which the ranks and masses are changed, and the amount by which a prototype point is moved, whose adjustments would undoubtedly affect the performance of the algorithm.

The botworld domain in which the learning experiments were performed and the components of the input vector, **x**, are illustrated and defined in the figure below. The reader will note that I chose the components of the input vector, $\mathbf{x} = (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9)$, to be those that I knew were particularly relevant to performing the task. A more severe learning task would be one in which the input vector consisted of more primitive components.



$x_1 = 1$ iff bot is grabbing bar

$x_2 = 1$ iff bot's center is within this box (same as At-bar defined earlier)

$x_3 = 1$ iff bot's heading is normal to bar, within some tolerance (same as Facing-bar defined earlier)

$x_4 = 1$ iff bot's center is within this strip (same as On-midline defined earlier)

$x_5 = 1$ iff bot's center is within this box

ccwg = 1 iff bot should turn ccw to face this box

cwg = 1 iff bot should turn cw to face this box

ccwb = 1 iff bot should turn ccw to face in same direction as the normal to the bar

cwb = 1 iff bot should turn cw to face in same direction as the normal to the bar

$x_6 = \neg x_4 \wedge \mathrm{ccwg}$

$x_7 = \neg x_4 \wedge \mathrm{cwg}$

$x_8 = x_4 \wedge \mathrm{ccwb}$

$x_9 = x_4 \wedge \mathrm{cwb}$

The goal region consists of all of those inputs for which $x_1 = 1$. The usual domain "physics" applies; the robot can grab the bar only if $x_2 = 1$ and $x_3 = 1$.

I used seven prototype points. These and their associated actions were:

B, move
C, turn ccw
D, turn cw
BB, move
CC, turn ccw
DD, turn cw
E, grab

The initial value of each point was the vector (1/2, 1/2, 1/2, 1/2, 1/2, 1/2, 1/2, 1/2, 1/2). The initial ranks and masses were all 0.

A teacher can insert himself into the learning process by selecting a prototype point and its action (instead of letting the robot select it by the usual nearest-distance calculation). After a teacher-selected action, the learning process already described is then allowed to take place as usual. I used this teaching adjustment to the learning process in the following way:

1. The robot was first positioned so that x_2 and x_3 were both equal to 1, and I (the teacher) selected point **E**. The robot performed the associated grab action (which, of course succeeded), and the position, mass, and rank of **E** were changed accordingly. I did this initial "priming" several times.

2. Then, the robot was positioned so that x_3 and x_5 were both equal to 1, and I selected **B**. The robot performed the associated move action until the robot got closest to **E**, and the position, mass, and rank of **B** were changed accordingly. Again, this step was performed several times.

3. The robot was positioned so that x_5 was equal to 1, and I selected either **C** or **D**, as appropriate, etc.

4. And so on.

After priming in this way, I ceased teaching and allowed the robot to continue on its own for several more learning trials. The main result of this experiment was not that the robot could learn to grab the bar on its own without teaching, but that once it had been primed, continued learning was "stable" in the sense that the skill was not unlearned.

After priming and after several subsequent learning trials without the teacher, the prototype points, their masses and ranks had the following values:

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	mass	rank
B	0	0.006	0.85	0.84	0.29	0	0.17	0.08	0.03	33	92
C	0	0	0.01	0.94	0.33	0	0.05	0.88	0.06	18	63
D	0	0	0.02	0.83	0.28	0	0.16	0	0.8	18	66
BB	0	0	0	0	0.125	0	0	0	0	2	47
CC	0	0	0.02	0	0	1	0	0	0	3	23
DD	0	0	0.02	0	0	0	1	0	0	3	34
E	0	0.97	0.82	0.71	0	0	0.29	0.03	0	35	99

By way of comparison, we might note that the following values of the prototype points would yield ideal bar-grabbing behavior:

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
B	0	0	1.0	1.0	0.5	0	0	0	0
C	0	0	0	1.0	0.5	0	0	1.0	0
D	0	0	0	1.0	0.5	0	0	0	1.0
BB	0	0	0	0	0	0	0	0	0
CC	0	0	0	0	0	1	0	0	0
DD	0	0	0	0	0	0	1	0	0
E	0	1.0	1.0	1.0	0	0	0	0	0

These initial experiments suggest that the migrating prototype algorithm has potential for the reinforcement learning of T-R programs. Further development and experimentation thus seems justified.

IX. Conclusions

Versatile robots will need to be programmed, of course. But beyond explicit programming by a programmer, they will need to be able to plan how to perform new tasks and how to perform old tasks under new circumstances. They will also need to be able to learn. In addition to learning about their environments by making maps, they will need to learn under what conditions various actions have what effects. [See, for example, (Benson, 1996).]

In this article, I concentrated on two other types of learning, namely supervised learning and reinforcement learning of robot control programs. I argued also that it would be useful for all of these programs, those explicitly programmed, those planned, and those learned, to be expressed in a common language. I proposed what I think is a good candidate for such a language, namely the formalism of teleo-reactive (T-R) programs. Most of the article dealt with the matter of learning T-R programs. I asserted that such programs are PAC learnable and then described some techniques for learning them and the results of some preliminary learning experiments. The work on learning T-R programs is in a very early stage, but I think enough has been started to warrant further development and experimentation. For that reason I make this article available on the web, but I caution readers about the tentative nature of this work. I solicit comments and suggestions at: nilsson@cs.stanford.edu.

REFERENCES

Benson 1993

Benson, S., "Botworld Homepage," Robotics Laboratory, Department of Computer Science, Stanford University, 1993. http://robotics.stanford.edu/~sbenson/botworld.html

Benson 1996

Benson, S., *Learning Action Models for Reactive Autonomous Agents*, PhD Thesis, Department of Computer Science, Stanford University, 1996.

Connell 1992

Connell, J., "SSS: A Hybrid Architecture Applied to Robot Navigation," in *Proc 1992 IEEE International Conf. on Robotics and Automation*, pp. 2719-2724, 1992.

John 1994

John, G., "SQUISH: A Preprocessing Method for Supervised Learning of T-R Trees From Solution Paths," Draft Memo, Stanford Computer Science Dept., November 22, 1994.

Lin 1992

Lin, L. J., "Self-Improving Reactive Agents Based On Reinforcement Learning, Planning and Teaching," Machine Learning 8:293-321, 1992.

Mahadevan & Connell 1992

Mahadevan, S., and Connell, J., "Automatic Programming of Behavior-Based Robots Using Reinforcement Learning," *Proceedings ML92*, pp. 290-299, 1992.

Michie, *et al.* 1990

Michie, D., Bain, M., and Hayes-Michie, J. E., "Cognitive Models from Subcognitive Skills," in M. Grimble, S. McGhee, and P. Mowforth (eds.), *Knowledge-base Systems in Industrial Control*. Peter Peregrinus, 1990.

Moore 2000

Moore, A. W., "The Anchors Hierarchy: Using the Triangle Inequality to Survive High Dimensional Data," Proc. Twelfth Conference on Uncertainty in Artificial Intelligence, Menlo Park, CA: AAAI Press, 2000.

Mahadevan & Connell 1992

Mahadevan, S., and Connell, J., "Automatic Programming of Behavior-Based Robots Using Reinforcement Learning," *Proceedings ML92*, pp. 290-299, 1992.

Nilsson 1984

Nilsson, N. J., *Shakey the Robot*, Technical Note 325, SRI International, Menlo Park, CA, 1984.

Nilsson 1992

Nilsson, N. J., *Toward Agent Programs with Circuit Semantics*, Technical Report STAN-CS-92-1412, Stanford University Computer Science Department, 1992.

Nilsson 1994

Nilsson, N. J., "Teleo-Reactive Programs for Agent Control," *Journal of Artificial Intelligence Research*, 1, pp. 139-158, January 1994.

Pomerleau 1993

Pomerleau, D., *Neural Network Perception for Mobile Robot Guidance*, Boston: Kluwer Academic Publishers, 1993.

Rivest 1987

Rivest, R., "Learning Decision Lists," *Machine Learning*, 2:229-246, 1987.

Rumelhart *et al.* 1986

Rumelhart, D., Hinton, G., and Willliams, R., "Learning Internal Representations by Error Propagation," in Rumelhart, D., and McClelland, J., (eds.), *Parallel Distributed Processing*, Vol 1, pp. 318-362, Cambridge, MA: The MIT Press, 1986.

Sammut, *et al*. 1992

Sammut, C., Hurst, S., Kedzier, D., and Michie, D., “Learning to Fly,” in D. Sleeman and P. Edwards (eds.), *Proceedings of the Ninth International Conference on Machine Learning*, Aberdeen: Morgan Kaufmann, 1992.

Singh & Bertsekas 1997

Singh S, and Bertsekas D., *Reinforcement Learning for Dynamic Channel Allocation in Cellular Telephone Systems* . *Proceedings of NIPS*, 1997.

Sutton 1988

Sutton, R., “Learning to Predict by the Methods of Temporal Differences,” *Machine Learning*, 3:9-44, 1988.

Sutton & Barto 1998

Sutton, R., and Barto, A., *Reinforcement Learning: An Introduction*, Cambridge, MA: MIT Press, 1998.

Tesauro 1995

Tesauro, G., “Temporal-Difference Learning and TD-Gammon,” *Comm. ACM*, 38(3):58-68, March, 1995.

Urbancic & Bratko 1994

Urbancic, T., and Bratko, I., “Reconstructing Human Skill with Machine Learning,” in A. Cohn (ed.), *Proceedings of the 11th European Conference on Artificial Intelligence*, John Wiley & Sons, 1994.

Watkins 1989

Watkins, C. J. C. H., *Learning from Delayed Rewards*, PhD thesis, Cambridge University, Cambridge, England, 1989.

Willeke 1998

Willeke, T., “Learning Robot Behaviors with TR Trees,” unpublished memo, Robotics Laboratory, Department of Computer Science, Stanford University, May 19, 1998.