Measuring the Effectiveness of Reinforcement Learning for Behavior-Based Robots

John Shackleton* and Maria Gini
Department of Computer Science, University of Minnesota
200 Union Street SE, Room 4-192, Minneapolis, MN 55455
gini@cs.umn.edu

Abstract

We explore the use of behavior-based architectures within the context of reinforcement learning and examine the effects of using different behavior-based architectures on the ability to learn the task at hand correctly and efficiently.

In particular, we study the task of learning to push boxes in a simulated 2D environment originally proposed by Mahadevan and Connell [Mahadevan and Connell, 1992]. We examine issues such as effectiveness of learning, flexibility of the learning method to adapt to new environments, effect of the behavior architecture on the ability to learn, and we report results obtained on a large number of simulation runs.

Keywords: Reinforcement learning, behavior-based architectures, robot learning.

1 Introduction

Behavior-based architectures [Brooks, 1986] are extremely popular for robotics. In this paper we examine the use of behavior-based architectures within the context of reinforcement learning and examine the effects of using different behavior-based architectures on the ability of robots to learn the task at hand correctly and efficiently.

In particular, we study the task of learning to push boxes in a simulated 2D environment originally proposed by Mahadevan and Connell [Mahadevan and Connell, 1992]. The original paper described a box-pushing robot, called OBELIX, with a simple subsumption architecture with three behaviors, FINDER, PUSHER, and UNWEDGER. FINDER is responsible for locating the boxes, PUSHER will push the boxes once they are found, and UNWEDGER will unwedge the robot after it gets stuck against some obstacle. The behaviors act independently, and each has an *Applicability Function* assigned to it, which decides if the behavior is applicable for a given state of the sensors. When more than one behavior is applicable at a time, priority is given to UNWEDGER, followed by PUSHER, and then by FINDER).

*Currently at Honeywell Technology Center, Minneapolis, MN.

Each behavior also has a *Learning Table*, which is the main mechanism for learning. A Learning Table is a numeric table whereby the columns represent the actions available to the robot, and the rows represent the robot's sensor states. Each entry in the Learning Table represents the "quality" of a certain action during a certain sensor state, for a specific behavior. Each behavior has associated with it a *Reward Function*, which determines how successful a previous action was. If an action is deemed "bad" the Reward Function will decrease the appropriate Learning Table entry. If the action is considered "good" then the Learning Table entry is increased. In theory, as the robot continually updates its Learning Tables, the values in the Learning Tables will converge, and the robot will learn the best actions to take under various circumstances.

A large body of work exists in the area of reinforcement learning, as reported in the excellent survey [Kaelbling *et al.*, 1996], and many applications to robotics have been reported. Methods such as progress estimators have been proposed [Mataric, 1994] and applied to real robots, decomposition of problems into subproblems has been used [Connell and Mahadevan, 1993, Whitehead *et al.*, 1993] in addition to monolithic learning [Whitehead, 1992], methods for learning individual subgoals and then learning how to achieve them sequentially have been proposed [Singh, 1992], and methods of shaping have been developed for learning the architecture in addition to individual behaviors [Dorigo and Colombetti, 1994, Dorigo, 1995].

Mahadevan and Connell in [Mahadevan and Connell, 1992] pioneered the combination of the subsumption architecture with reinforcement learning. More recent work by Mahadevan [Mahadevan, 1994, Mahadevan, 1996] compares the Q-learning method with R-learning [Schwartz, 1993] for various domains (including the box-pushing domain) using different exploration strategies. The box-pushing domain is a rich domain for experimental work in learning because of its complexity in terms of number of states and actions, and because it requires learning sequences of actions. It is also interesting in robotics because it could easily be implemented on a real robot. Others have studied the box-pushing problem using different techniques. For instance, Koza [Koza and Rice, 1992] reports results on using a genetic programming approach to generate a program for the robot to push boxes. Only limited experimental results are reported in [Koza and Rice, 1992], which makes it impossible to compare the results of the genetic programming approach with the results reported in [Mahadevan and Connell, 1992] and with our results.

In the work we present here we are mainly interested in assessing the ability of the robot to learn the given task in different environments, and in measuring how well the robot learns.

2 Experimental Results

The experiments we describe have been designed to answer the following fundamental questions:

Measurements: How should the learning be measured? measuring the percentage of correct responses after a number of learning trials does not always capture how well the specific task at hand has been learned. We introduce a number of domain specific measures for the box-pushing task that capture more closely the performance of the learning method at learning the box-pushing task, and we use them in our experiments. We

also introduce some domain independent measures that measure the amount of learning in terms of the number of sensor states reached, the number of entries reached in the Learning Tables, and the percentage of non-zero value rewards assigned. Details on the specific measures we use are given in Section 3.

Behavior-based architecture: How does the behavior-based architecture affect learning? As reported in [Mahadevan and Connell, 1992], the decomposition in behaviors simplifies learning. The question we want to address is how different decompositions affect learning in terms of speed of learning and effectiveness at learning the task at hand. We show with our experiments that an architecture with more behaviors performs at least as well as a simpler architecture, and performs better in more cluttered environments. We explore two different behavior-based architectures, the original one and a new one with more behaviors. For our experiments we assume the architecture is given and not learned.

Adaptability to new environments: Is the learning sufficiently independent on the environment in which it was achieved? Ideally, we would like the robot to be able to learn a task in one environment and then be able to perform the same task in a different environment with only minimal learning. We report results on experiments done in three types of environments, one similar to the environment originally used in [Mahadevan and Connell, 1992], one much more cluttered, and one quite sparse but with many boxes. We show that a robot that learns in a more difficult environment is more capable to adapt to a different environment. This seems to contradict results obtained using the technique of shaping [Hilgard and Bower, 1975], in which a simple problem is presented first, and more complex problems are presented later. However, we have to keep in mind that in our case the architecture of the behaviors is already known and fixed, as opposed to cases in which the architecture has to be learned as well [Dorigo and Colombetti, 1994].

3 Measuring Robot Effectiveness

Mahadevan and Connell in [Mahadevan and Connell, 1992] argued that a robot with a Learning Table for each behavior will learn faster, and perform better than a robot with one, monolithic Learning Table that spans all behaviors. As a result, the primary characteristics they use to judge the robot's performance centers around how quickly the entries in the Learning Tables converge.

To address other issues such as effectiveness of learning and performance at the task learned, we need to introduce other metrics. This in turn will also help nail down what aspects of the original behavior architecture could be changed to improve performance.

When the original robot runs within complex environments, it quickly becomes evident that its major weakness is that it does not cover open ground very well. When the environment is confined, as it is with the testbench environment used in the original paper, this weakness is not as obvious as with a more spacious environment. Thus, a characteristic we want to measure, and an improvement we want to make, is the total distance the robot

travels, and the total area the robot covers. This is important because we want the robot to find as many boxes as possible and by traveling more the chance of finding boxes increases.

Similarly, we want the robot to perform its task well, namely to push as many boxes as far as possible in the shortest amount of time. Thus, we need to measure the number of boxes touched by the robot in a given period of time and the distance that the boxes are pushed. Although this metric alone is not very reliable, since the boxes and the environment could be arranged to favor one implementation over another, it does give an idea of how well the robot performs over an extended period in doing the task it is expected to do.

We should also not forget the learning aspect of the robot. We not only want the robot to learn fast (quality) but also to learn a lot (quantity). This will be accomplished mostly by improving the distance that the robot covers, thereby exposing the robot to new sensor states. To ensure that this is indeed accomplished we will need to measure the number of sensor states reached, the number of entries in the Learning Tables that are modified from the initial state, and the number of non-zero rewards assigned.

When we go about trying to improve the robot's performance, certain characteristics of the original robot have to remain constant, to ensure that any perceived improvements are for real. In the new architecture we present, the physical characteristics of the robot are the same: the new robot also has eight sensors positioned as before, and the same five actions. The Learning Tables thus have the same size, although the number of Learning Tables is different. We also continue to use the same method of reinforcement learning with percolation of reward values back in time (five levels of recursion), and Hamming distance grouping of sensor states. This makes performance comparisons possible.

With the physical characteristics and method of learning constant, we are left to experiment with a well defined set of robot conditions. We can change the underlying architecture, the corresponding Applicability and Reward Functions, and the actual implementation of the Learning Tables and examine what impact those changes have on the ability of the robot to learn the task of pushing boxes.

4 Validation of the simulation system

Before jumping into experimenting with a new architecture, it was necessary to do some sort of rudimentary validation of the simulation program we created. The best way to do this was to attempt to duplicate the results of the Mahadevan and Connell robot. This was done with surprising success. Figure 1 shows the floor map that the original physical robot used. A set of 20 learning trials were set up for the robot, each trial lasting 100 moves. The robot starts in the same initial position for each trial, although the starting direction is random. This is the same number of trials and steps as in the original system.

The performance of each robot behavior is computed, as in [Mahadevan and Connell, 1992] by the following equation:

$$Performance_rating = \frac{Average_reward - Minimum_reward}{Maximum_reward - Minimum_reward}$$

The $Performance_rating$ represents the quality of the decisions made by the robot after the trials. If this percentage is high, then the behavior is making decisions that receive high

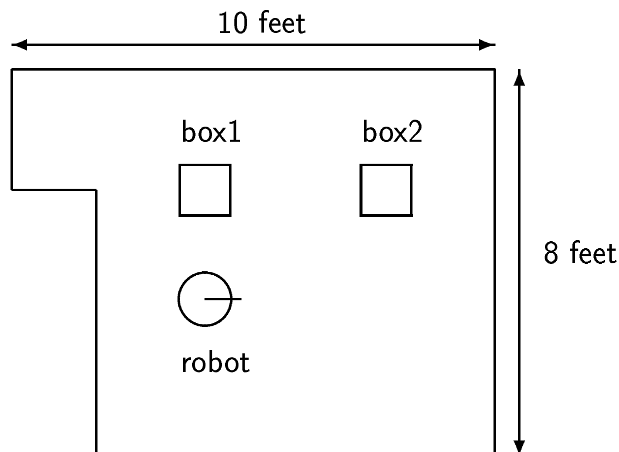

Figure 1: Original floor map for the learning trials.

rewards. Conversely, if the percentage is low, then the robot is getting a lot of negative punishment from the behavior. The reason for the $Performance_rating$ statistic is to show the convergence of the robot's Learning Tables as the simulation trials progressed.

The statistics for our simulation program and the original Mahadevan and Connell simulation are given below in Table 1. The disparity in the numbers can be attributed to slight differences in the simulation floor plans, although other factors probably also contribute, such as different lengths of the forward move, different sizes of the simulated boxes, and subtle differences in the way that the sensors are simulated. All in all, the new simulation program comes very close to mimicking the Mahadevan and Connell results.

Measure	Original implementation	Our implementation
Average reward for Finder	0.20	0.294
Average reward for Pusher	-0.20	-0.325
Average reward for Unwedger	0.16	-0.020
Performance rating for Finder	30%	32%
Performance rating for Pusher	70%	67%
Performance rating for Unwedger	79%	74%

Table 1: Learning measures reported for the original implementation and results with our implementation. The results obtained with our simulation program are for the original robot on the original floor map shown in Figure 1.

We have added a number of features to our simulator, including using B+ Trees to implement the Learning Tables (as detailed later in Section 6), a flexible user interface, and a method for measuring the area that the robot covered during simulation. We utilize a kind of "pixel map", a two-dimensional array that represents the entire area of the simulation's floor

map. The elements of the array are all initialized to zero (or false), and as the robot image on the screen overlaps screen pixels, the corresponding elements in the two-dimensional array are set to one (or true). At the end of the simulation, we can compute the area covered by the robot by simply counting all the elements in the array with a value of one. This method is not perfect, since the pixel coverage on the screen is just an approximation for a simulated robot, but it is a good approximation nonetheless.

5 The original 3-behavior architecture

The robot described in [Mahadevan and Connell, 1992] uses eight sonar sensors located on its front and sides to detect objects. Each sensor has three possible states: no object detected, a near object detected (from nine to 18 inches away), and a far object detected (18 to 30 inches away). We will use FrontNearSonarBitsOn and FrontFarSonarBitsOn to indicate the corresponding sensor values, and FrontSonarBitsOn to indicate that an object has been detected in front of the robot when we do not care if it is near or far. In addition, the robot has two special sensor readings. The Bump bit indicates whether or not the robot has bumped up against an object, and the Stuck bit indicates whether or not the robot is stuck against an object after trying to move forward.

The robot has eight sonar sensors, a Bump bit, and a Stuck bit, so in total there are $3^8 \times 2 \times 2 = 26,244$ number of rows in each Learning Table. The robot can move in five different ways: Forward, TurnLeft 22.5°, TurnRight 22.5°, TurnLeft 45°, and TurnRight 45°. Each Learning Table has $5 \times 26,244 = 131,220$ entries.

Populating the Learning Tables one entry at a time can reduce the effectiveness of the robot's learning. To improve the time in which the Learning Tables are filled, techniques can be used to group together similar sensor states, allowing several entries in the Learning Tables to be modified within a single move. Mahadevan and Connell describe two such grouping techniques: grouping by Hamming distances (of the sensor state bit vectors) and grouping by statistical clustering. The method we use is grouping by Hamming distance, which can be shortly explained as follows. When a sensor state entry is updated in a Learning Table, we also update all the other sensor states in the same table whose bit vector representation differs by k bits or less. For our simulation, k equals one. As a rule, higher Hamming distances do not increase the rate of "useful" learning information propagated through the table, because greater differences in sensor states will naturally require different actions in the long run.

The original architecture, shown in Figure 2 has three behaviors, Finder, Pusher, and Unwedger. Finder is responsible for locating the boxes, Pusher will push the boxes once they are found, and Unwedger will unwedge the robot after it gets stuck against some obstacle.

The behaviors act independently, and each has an Applicability Function assigned to it, which decides if the behavior is applicable for a given state of the sensors. The Finder behavior is always applicable, Pusher is applicable either if the Bump bit is on or has been on in the current history of the behavior, Unwedger is applicable if the Stuck bit is on or has been on in the current history of the behavior. Priority is given to Unwedger, then

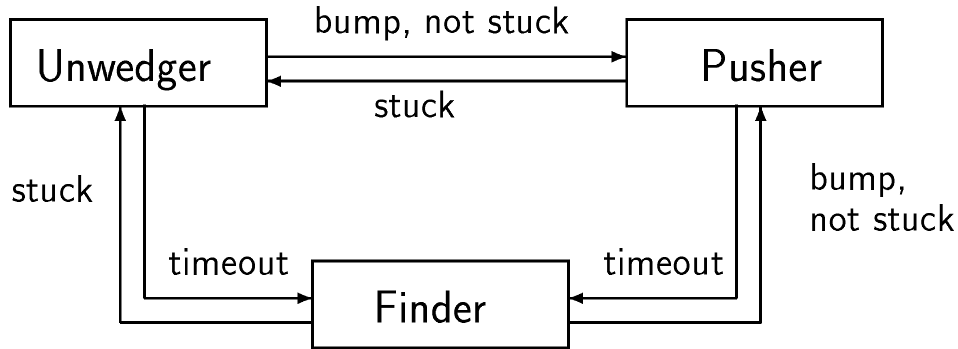

Figure 2: Original architecture for the 3-behavior robot.

PUSHER (and then FINDER) when more than one behavior is applicable at a time. Figure 3 shows the Reward Functions of the original robot.

$$Finder_reward = \begin{cases} 3 & \text{if action=FORWARD and FRONTNEARSONARBITSON} \\ -1 & \text{if not FRONTNEARSONARBITSON} \\ 0 & \text{otherwise} \end{cases}$$

$$Pusher_reward = \begin{cases} 1 & \text{if action=FORWARD and BUMP} \\ -3 & \text{if not BUMP} \\ 0 & \text{otherwise} \end{cases}$$

$$Unwedger_reward = \begin{cases} -3 & \text{if STUCK} \\ 1 & \text{if action=FORWARD and not STUCK} \\ 0 & \text{otherwise} \end{cases}$$

Figure 3: Original reward functions for the 3-behavior robot.

The concept of a behavior's current history should be explained. A behavior's history indicates how many moves the robot is allowed to continue under the same behavior, even though the original conditions that made the behavior applicable are no longer true. For example, when the unwedger_history is set to 5, then the robot may continue in the UNWEDGER behavior for five more moves, even though the robot's STUCK bit is no longer set. This has the effect of allowing the robot to learn a series of actions to take when certain conditions are encountered.

The learning method we use, the same as the one reported in [Mahadevan and Connell, 1992], is based on Q-learning [Watkins, 1989, Watkins and Dayan, 1992]. The method is as follows:

1. Initialize the Learning Table.
2. Do forever:
   (a) Observe the current sensor state s.

(b) Given the sensor state s and the most recently active behavior determine the new active behavior using the Applicability Functions, and select the corresponding Learning Table.

(c) For the given sensor state s and the Learning Table, choose the action with the largest value a. If all actions have a zero value (which means that the sensor state has never been encountered before) then select the FORWARD action.

(d) Carry out action a.

(e) Let the new sensor state be t, and the immediate reward for executing a in sensor state s be r. Update the last used entry in the Learning Table according to the following rule:
$Q(s,a) := Q(s,a) + \beta(r + \gamma * e(t) - Q(s,a))$
where $e(t)$ is the maximum $Q(t,a)$ over all actions a performed in state t, β and γ are values between 0 and 1. β controls the rate at which the error of the current Q value is corrected, γ is a discount factor.

6 The new 6-behavior architecture

A new behavior-based architecture, developed as an alternative to the original, is shown in Figure 4.

The PUSH and UNWEDGE behaviors are almost identical to the ones used in the original robot, except that UNWEDGE simply turns away from the obstacle and uses WANDER to move away. The FIND behavior is significantly different than the original FINDER and is limited to finding obstacles, leaving to PURSUE the job of approaching them. The WANDER, PURSUE, and ALIGN behaviors are new.

The relationship between the behaviors goes as follows: the robot will normally start out in the FIND behavior, scanning its immediate environment for an object without moving forward (although this method does not guarantee that the robot will not move forward during the FIND behavior). If an object is found, the robot will enact the PURSUE behavior, moving forward towards the object. If no object is found, and the robot receives a time-out from the FIND behavior, the robot will enact the WANDER behavior and attempt to travel a significant distance away from its current position. The robot will wander until the WANDER behavior has a time-out (at which time the robot returns to the FIND behavior) or the robot bumps into something. Whenever the robot hits an object, it will enact the PUSH behavior, and try to push the object in hopes that the object is a box. If the robot should lose contact with the box while in the PUSH behavior, the ALIGN behavior will start up, and attempt to realign the robot with its lost box. Whenever the robot becomes stuck (whether it is pushing a box or not) it will enact the UNWEDGE behavior, and try to free itself from the stuck position.

A major concern in designing the new architecture was that increasing the number of Learning Tables would make the memory load unbearable, especially if this new robot was physically implemented. If the number of sensors were increased (usually a desired step, since increasing the number of sensors will improve almost every navigational robot approach), then the memory load would be worsened even more.

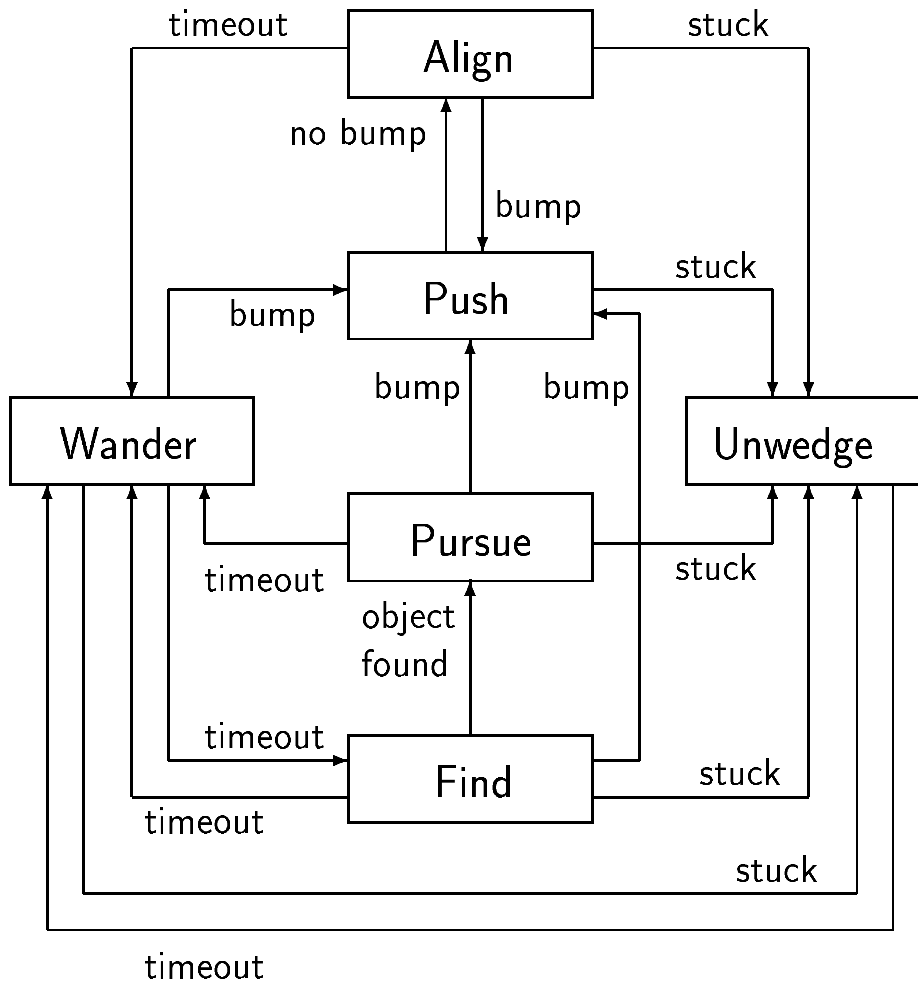

Figure 4: New architecture for the 6-behavior robot.

To alleviated this potential problem, we have implemented the Learning Tables as B+ Trees, a common data structure used in database implementations [Comer, 1979]. Each leaf node of the B+ Tree represents a single row in the Learning Table, indexed by a sample of the robot's sensor states. To access the nodes, a particular state of the sensors is encoded into a key. Each key corresponds to one state of the sensors, and viceversa, to maintain consistency. Initially, the B+ tree is empty. As the robot updates new sensor states, the B+ tree creates new nodes. If the robot tries to read the values for a sensor state that has not been encountered yet, (which also means that there is no node in the tree for that sensor state), then a value of zero is returned.

Since the relationships between the new behaviors are more complex than the simple priority scheme of the original robot, the Applicability Functions we defined for them depend on the current active behavior. For ease of implementation, the determination of the behavior's applicability has been combined into one large function, that simulates the action of a Finite State Machine, where the behaviors are the states. Details of the Applicability

function are shown in Figure 5.

The 6-behavior robot relies on behavior histories more so than the 3-behavior robot, because the new robot has more behaviors, and thus more behavior interaction. The initial behavior history values for the 6-behavior architecture, assigned when a behavior is first enacted, are shown later in Table 2. These values were not decided upon through a rigorous analytical approach, but through trial-and-error. More discussion on the behavior histories and their importance to avoid local minima is in the next Section.

7 Rewards and the Problem of Local Minima

The Reward Functions must be diverse enough to encourage the expected interaction of the robot's behaviors, while at the same time remaining straight-forward and useful under a variety of different environments. The new Reward Functions, shown in Figure 6, look quite similar to the original ones, although there are clear differences.

In the original robot, the UNWEDGER behavior was rewarded for successfully moving forward without getting stuck. In the new robot, the UNWEDGE behavior is only responsible for unwedging itself within its current position, and does not need to move forward at all. The WANDER behavior will make sure that the robot gets away from the stuck position. Thus the new Reward Function rewards the robot for not moving forward to get unstuck.

In the PUSH behavior, it is assumed that the robot is currently BUMPed. The original case where the robot uses the pusher_history to become realigned with a lost box is no longer needed (because of the ALIGNER behavior, as explained earlier). Therefore, the Reward Function for PUSH will reward the robot when it moves forward, and punish the robot for all other moves.

We want ALIGN to cause the robot to turn in place (hopefully in one or two moves), until the robot bumps back up against the wayward box. Thus, the robot will be rewarded for BUMPing against an object without moving forward, otherwise it will be punished.

The PURSUE behavior is responsible for moving the robot towards an object once it has been detected in the FIND behavior. Thus, we want to punish the robot whenever it loses sight of the detected object (or when its front sensors no longer detect the object). Similarly, we want to reward the robot whenever it stays in-line with the detected object and moves forward.

We want the FIND behavior to turn the robot around in its current position to locate a nearby object, without moving forward. As a result, the robot will be punished for moving forward, and will be rewarded for locating an object with its front sensors (without moving forward). In theory, if the robot finds an object while in the FIND behavior, all non-forward moves in that FIND sequence will be rewarded through percolation of reward values back in time.

Finally, the WANDER behavior is successful any time it makes the robot move forward. However, the WANDER function will be used often, and in dissimilar circumstances, so we do not want to punish the robot whenever it does not move forward. The only time the WANDER behavior is actually unsuccessful is when the robot becomes stuck. Thus, the Reward Function for WANDER will reward the robot when it moves forward, and hand down a punishment whenever the robot becomes stuck.

```
Select case by current_behavior:
case UNWEDGE:
    if STUCK
      then initialize unwedge_history and return UNWEDGE
      else if unwedger_history > 0
            then decrement unwedge_history and return UNWEDGE
            else initialize wander_history and return WANDER
case PUSH:
    if STUCK
      then initialize unwedge_history and return UNWEDGE
      else if BUMP
            then return PUSH
            else initialize align_history and return ALIGN
case ALIGN:
    if STUCK
      then initialize unwedge_history and return UNWEDGE
      else if BUMP
            then return PUSH
            else if align_history > 0
                  then decrement align_history and return ALIGN
                  else initialize wander_history and return WANDER
case PURSUE:
    if STUCK
      then initialize unwedge_history and return UNWEDGE
      else if BUMP
            then return PUSH
            else if pursue_history > 0
                  then decrement pursue_history and return PURSUE
                  else initialize wander_history and return WANDER
case FIND:
    if STUCK
      then initialize unwedge_history and return UNWEDGE
      else if BUMP
            then return PUSH
            else if FRONTSONARBITSON
                  then initialize pursue_history and return PURSUE
                  else if find_history > 0
                        then decrement find_history and return FIND
                        else initialize wander_history and return WANDER;
case WANDER:
    if STUCK
      then initialize unwedge_history and return UNWEDGE
      else if BUMP
            then return PUSH
            else if wander_history > 0
                  then decrement wander_history and return WANDER
                  else initialize find_history and return FIND
```

Figure 5: Applicability function for the 6-behavior robot.

$$Wander_reward = \begin{cases} 1 & \text{if action=}\textsc{Forward}\text{ and not }\textsc{Stuck} \\ -3 & \text{if }\textsc{Stuck} \\ 0 & \text{otherwise} \end{cases}$$

$$Find_reward = \begin{cases} 3 & \text{if action }\neq\textsc{Forward}\text{ and }\textsc{FrontSonarBitsOn} \\ -1 & \text{if action=}\textsc{Forward} \\ 0 & \text{otherwise} \end{cases}$$

$$Pursue_reward = \begin{cases} 1 & \text{if action=}\textsc{Forward}\text{ and }\textsc{FrontSonarBitsOn} \\ -3 & \text{if not }\textsc{FrontSonarBitsOn} \\ 0 & \text{otherwise} \end{cases}$$

$$Align_reward = \begin{cases} 3 & \text{if action }\neq\textsc{Forward}\text{ and }\textsc{Bump} \\ -1 & \text{otherwise} \end{cases}$$

$$Push_reward = \begin{cases} 1 & \text{if action=}\textsc{Forward} \\ -3 & \text{otherwise} \end{cases}$$

$$Unwedge_reward = \begin{cases} -3 & \text{if }\textsc{Stuck} \\ 1 & \text{if action }\neq\textsc{ Forward}\text{ and not }\textsc{Stuck} \\ 0 & \text{otherwise} \end{cases}$$

Figure 6: Reward functions for the 6-behavior robot. Tha values have been obtained by trial-and-error.

Notice that there is no history kept for the PUSH behavior. In the original robot, the push_history was kept so that the robot could try to re-align after it lost contact with a box. In the new architecture, this requirement is taken over by the ALIGN behavior, so there is no need for the PUSH behavior to time-out. If the robot is pushing a box, we want it to continue as long as it can, until the box is stuck against an obstacle.

Also, the value for the WANDER history is rather high, 30 moves. This provides the new robot with enough opportunity to explore new areas. The PURSUE history is 10, because that is the number of moves it takes the robot to travel further than the length of its sensor range.

A feature we added is the following additional update to the behavior histories, that is applied every time a reward or punishment is computed:

if not STUCK and action= FORWARD
  then initialize steps_without_forward to 0
  else increment steps_without_forward

The global variable steps_without_forward represents the current number of moves the robot has enacted without moving in a forward direction. If the robot reaches the point where it has not moved forward for a number of moves equal to the maximum allowed for the current active behavior (shown in Table 2), then the robot will automatically enact a random move (of constant distribution). This is so that the robot has a chance to rescue itself if its state of rewards and punishments for a given behavior reaches a local minimum.

A local minimum will be encountered in the following scenario: the robot gets stuck against a wall or in a corner. The robot performs a series of turning actions, but when the UNWEDGE behavior does a time-out, the robot finds itself in the same position as it was when it first got stuck (which would happen if the entries in the UNWEDGE's Learning Table for the current sensor state held the right values). Through percolation of reward values back in time, the past actions are summarily punished, and the robot ends up still unable to move forward. The UNWEDGE behavior repeats the same series of actions (because each action received a proportional amount of punishment), which will result in the same failure, over and over, indicating that a local minima has been reached.

Mahadevan and Connell claimed that their approach avoided local minima, and to a certain extent this is true because the robot will not reach a local minima when operating in open space. However, experiments with more complex environments have shown that under the right conditions, the 3-behavior robot will get mired in local minima when attempting to navigate a cluttered environment. The random operation thus acts as a last resort, intended to kick the robot back into gear after becoming permanently stuck against an obstacle. In general, the chances of the robot reaching a local minima decrease as the robot experiences more and more new sensor states.

The maximum number of moves the robot can enact without moving forward, given for each behavior, are listed in Table 2. The values have been determined by trail-and-error.

	Initial value for history	Max No. steps w/o forward
UNWEDGE	5	20
PUSH	0	5
ALIGN	5	20
PURSUE	10	20
FIND	5	20
WANDER	30	5

Table 2: Values used to initialize the history of behaviors, and maximum number of steps allowed in a history without a forward movement for the 6-behavior robot.

8 Comparison of performance of the 3-behavior with the 6-behavior robot

In Table 3 we show results obtained when running the 3-behavior robot on the original map. For this set of experiments, we performed 20 learning trials of 100 steps each, as in the original implementation. We have found this number of steps to be sufficient to fully train the robot.

Note that the number of entries per move in the Learning Table can be greater than one, because the Hamming distance propagation algorithm allows multiple entries to be updated

	Finder	Pusher	Unwedger
Average reward	0.294	-0.325	-0.020
Performance rating	32%	67%	74%
% time behavior active	17	18	65
% zero-value rewards assigned	12%	24%	20%
No. sensor states reached	141	156	185
No. Learning Table entries reached	377	305	474
No. Learning Table entries per move	1.11	0.85	0.36

Table 3: Performance of the 3-behavior robot using the original floor map.

	Wander	Find	Pursue	Align	Push	Unwedge
Average reward	0.099	-0.157	0.279	-0.208	-0.006	0.131
Performance rating	77%	21%	82%	20%	75%	78%
% time behavior active	36	1	1	5	8	50
% zero-value rewards assigned	8%	16%	2%	31%	10%	6%
No. sensor states reached	232	7	10	60	54	220
No. Learning Table entries reached	667	29	25	195	174	603
No. Learning Table entries per move	0.92	4.14	1.39	1.82	1.13	0.61

Table 4: Performance of the 6-behavior robot using the original floor map.

within the same move.

The next logical step is to run the 6-behavior robot on the same environment for the same number of trials, and compare the results. Table 4 describes the results from the series of learning trials, which were executed in the same manner as before, i.e. 20 trials with 100 steps per trial. The performance rating for the new robot's behaviors are comparable to the behaviors of the original 3-behavior robot.

Table 4 also includes the percent that each behavior was active, the percentage of rewards assigned that have a value of zero, the number of unique sensor states reached for each behavior, the number of unique Learning Table entries reached by each behavior, and the number of unique Learning Table entries reached per active move of the behavior.

At this point, we begin to see the improvement of the 6-behavior robot over the 3-behavior robot. During the simulation trials, the 3-behavior robot spent two-thirds of the time unwedging itself from a stuck position, while the 6-behavior robot spends roughly half of the time unwedging. Also, the ratio of Learning Table entries to number of active moves is generally higher for each behavior of the 6-behavior robot. The ratio for the Find and Pursue behaviors of the 6-behavior robot are probably not representative of the actual performance of the behaviors, since these behaviors are not active for very long. The assertion does hold for the other behaviors.

What about the robot overall? Remember that our original motivation was to improve

	3-behavior	6-behavior
Total distance traveled	859 ft.	1335 ft.
Average distance per move	5.1 in.	8 in.
No. Learning Table entries per move	0.574	0.729
No. box touches	17	26
Total distance boxes pushed	131 ft.	200 ft.

Table 5: Comparison of the 3-behavior and 6-behavior robots using the original floor map.

the robot's performance and increase the amount of information that the robot learns.

Consider the statistics given in Table 5. The new 6-behavior robot consistently pushes more boxes and travels further distances in the same amount of time as the original 3-behavior robot, primarily because the 6-behavior robot, through the implementation of the WANDER behavior, is prevented from gravitating towards walls and obstacles already explored. In the process, the Learning Tables of the 6-behavior robot are exposed to about 24% more sensor information. More boxes are pushed, and by larger distances. The total distance that the boxes are pushed is measured in Manhattan distance (not the linear distance between starting and ending positions) because in the simulation boxes are only allowed to move translationally.

We cannot claim that the 6-behavior robot is learning faster than the 3-behavior robot, but the 6-behavior robot is learning more, and in the end performs its task more effectively. Both Robots appear to be able to learn the task in the same amount of trials. Adding more trials did not significantly changed the results.

9 Other Environments

To support our claim that the 6-behavior robot learns to do the task more effectively than the 3-behavior robot we need more experiments. In particular, the next question we will investigate is: how well does the 6-behavior robot operate in different environments? To answer this question we have created tests for two other scenarios: a cluttered environment with many obstacles, and a sparse environment with very few obstacles and a lot of open space. Both new environments are much larger than the original one, so the robot has to learn to cover much greater distances. The cluttered environment has many fixed obstacles, so the robot has to learn not to spend too much time trying to push unmovable objects. Because of the clutter, it is also easier for the robot to get stuck.

9.1 Cluttered Environment

First we examine the cluttered environment, whose floor map is given in Figure 7. The large rectangular objects are fixed obstacles, the square objects are the boxes. The robot is shown as a circle with a line pointing to its front. Both the 3-behavior robot and the 6-behavior robot were put through 10 trials within the cluttered environment, each trial starting in

the same position and lasting 1,000 moves. A trial of 1,000 moves gives the robot plenty of opportunity to explore a large portion of the environment.

The results obtained indicate that the performances of the 3-behavior and the 6-behavior robots are roughly the same. The performance rating for each behavior of the two robots is almost identical to the corresponding results of the first learning trials, and both robots spend approximately one-third of the time unwedging.

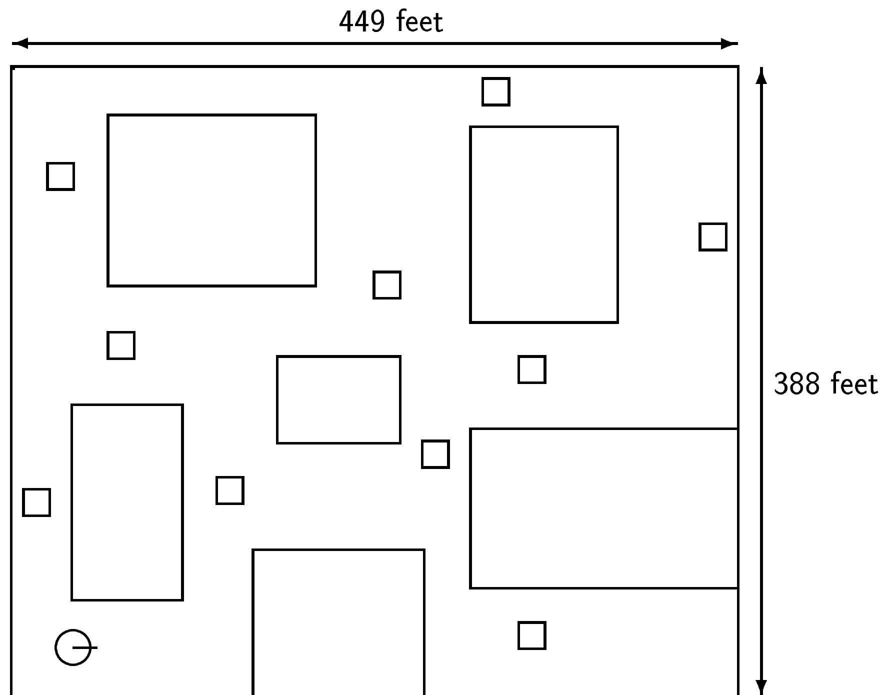

Figure 7: Floor map of a cluttered environment. The map is not drawn to scale. The robot appears much bigger than its size really is.

The 3-behavior robot tends to cling to the outer walls of the environment without venturing into new, open space, so it will repeatedly try to push obstacles that have already been pushed. Usually this redundant action by the PUSHER behavior is rewarded, because there is no built in mechanism to punish the robot for repeating actions under identical circumstances (the robot also is unable to distinguish the explored areas of the environment from the unexplored areas).

In reality, the 3-behavior robot rarely escapes the cluttered corner of the starting position, while the 6-behavior robot covers much more ground. As Table 6 shows, the 6-behavior robot travels over seven times further in linear distance (absolute distance, not factored for relative starting and ending positions), reaches four times more new Learning Table entries, pushes five times more boxes during the trials (and pushes those boxes much further), and perhaps most importantly, the 6-behavior robot covers four times more area than the 3-behavior robot.

	3-behavior	6-behavior
Total distance traveled	537 ft.	4014 ft.
Average distance per move	0.6 in.	4.8 in.
Total area covered in 10 trials	605 sq.ft.	2641 sq.ft.
No. Learning Table entries per move	0.060	0.250
No. box touches	2	10
Total distance boxes pushed	2 ft.	86 ft.

Table 6: Comparison of the 3-behavior and 6-behavior robots in the cluttered environment.

	3-behavior	6-behavior
Total distance traveled	1428 ft.	6789 ft.
Average distance per move	1.7 in.	8.5 in.
Total area covered in 10 trials	2822 sq.ft.	9848 sq.ft.
No. Learning Table entries per move	0.074	0.190
No. box touches	16	41
Total distance boxes pushed	95 ft.	955 ft.

Table 7: Comparison of the 3-behavior and 6-behavior robots in the sparse environment.

9.2 Sparse Environment

The experiments on the sparse environment shown in Figure 8 had similar results. Ten trials of 1000 moves were executed, each trial starting at the same initial position with a random heading. Like the cluttered environment trials, the sparse map trials produced almost identical performance ratings for both the 3-behavior and 6-behavior robots for each behavior, and both robots spent nearly the same percentage of time, 25% and 19% respectively, in the UNWEDGE behavior. Again, the original 3-behavior robot spent more time in the PUSHER behavior.

Table 7 gives clearer indication of which robot performed better. The 6-behavior robot traveled almost five times further in absolute distance, touched twice as many boxes, filled 61% more Learning Table entries, and covered ten times more area. As with the experiments on the cluttered environment, these statistics show that the 6-behavior robot performs the box pushing task more effectively, and would likely maintain that effectiveness in similar, sparse environments.

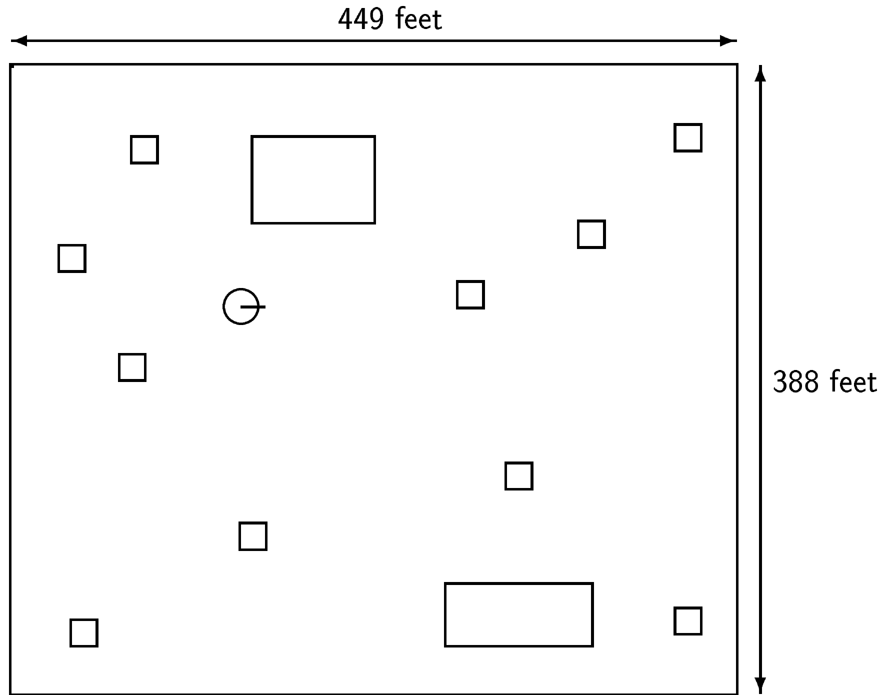

Figure 8: Floor map of a sparse environment. The map is not drawn to scale. The robot appears much bigger than its size really is.

9.3 More comparisons

There are other factors that might account for the improved performance of the 6-behavior robot architecture. For instance, it may be that the performance increase is really related to the fact that the reward functions shown in Figure 6 and used in the 6-behavior robot are active more of the time and hence provide more information about how to perform the task.

Tables 8 and 9 list the percentages of zero-value rewards assigned to each behavior during the simulations in the different environments. We can see from the tables that decomposing the behaviors, as we did going from the 3-behavior to the 6-behavior robot, leads to substantially fewer zero-value rewards. By specializing behaviors, the reward functions become more fine-tuned and leave less ambiguity in assigning rewards. Zero-value rewards are not desirable. The robot can make better decisions when fewer zero-value rewards are assigned, and therefore the 6-behavior robot performs better.

10 Adapting to a Different Environment

In theory, a robot controlled by a behavior-based architecture should be more capable of adapting from one environment to another than a robot controlled with a traditional architecture. Ultimately, it is desirable to create a robot that can take the information it has learned from one set of surroundings, and apply that information to an entirely new envi-

	FINDER	PUSHER	UNWEDGER	ALL BEHAVIORS
Original environment	12%	24%	20%	20%
Cluttered environment	4%	39%	36%	33%
Sparse environment	81%	26%	48%	67%

Table 8: Percentage of zero value rewards assigned for the 3-behavior robot.

	WANDER	FIND	PURSUE	ALIGN	PUSH	UNWEDGE	ALL BEHAVIORS
Original environment	8%	16%	2%	31%	10%	6%	8%
Cluttered environment	14%	12%	7%	8%	3%	3%	8%
Sparse environment	16%	13%	2%	6%	2%	3%	9%

Table 9: Percentage of zero value rewards assigned for the 6-behavior robot.

ronment. This is what we have done with the 6-behavior robot, to see how well this machine learning method is applicable across different environments.

The prior experiments were repeated on both the cluttered and sparse floor maps, except this time, the robot begins with the Learning Table data it had gained after completing the very first set of learning trials (the experiments run on the floor map given in Figure 1). The robot was loaded with the Learning Tables from the original learning, initialized to the same starting position with a random heading, and run for 10 trials each of 1000 moves. The statistics gathered from these experiments (number of sensor states reached, number of Learning Table entries reached, etc.) do not include the data loaded into the robot beforehand, and solely reflect the robot's performance in the new environments. The statistics reflect the robot's performance on a per trial basis, and not accumulated over 10 trials, since the Learning Tables were re-initialized to the same original table before each trial.

The cluttered environment results are almost identical to those obtained by running the experiments directly in the cluttered environment (without any prior learning). The key difference is in the WANDER behavior. The number of new sensor states reached and the number of new Learning Table entries reached for the WANDER behavior are half of what the were for the WANDER behavior in the cluttered environment trials. This shows that the robot did not wander into unfamiliar territory like it should do, but instead wandered over and over in the same corner of the cluttered environment. Table 10 reinforces this suspicion. The robot adapting to the cluttered environment travels less distance than the robot starting its learning in the cluttered environment; it reaches 50% fewer Learning Table entries, and on average covers 17% less area after ten trials. This seems to indicate that learning in a smaller and simpler environment does not provide much useful information for a much larger and more complicated environment.

The experiments on the robot adapting to the sparse environment had better success. In fact, the robot adapting to the sparse environment performed better than the robot starting from scratch in every category. It seems as if the robot actually benefited from the prior

	Cluttered	Sparse
Total distance traveled	3,821 ft.	7,567 ft.
Average distance per move	4.6 in.	9.0 in.
Total area covered in 10 trials	2,179 sq.ft.	10,094 sq.ft.
No. learning table entries per move	0.170	0.130
No. box touches	5	49
Total distance boxes pushed	53 ft.	1293 ft.

Table 10: Comparison of performance of the 6-behavior robot, when adapting to the cluttered environment vs. adapting to the sparse environment.

knowledge, which follows conventional intuition, although it is not clear to what extent the prior knowledge improved the robot's performance. These experiments indicate that the initial learning helped the robot to adapt to a new much larger environment. Perhaps, in the smaller environment the robot had more chances of touching boxes and so it learned how to push them. When learning directly in a large and sparse environment, often the robot does not have much chance of touching many boxes and so the learning is limited.

The results of similar experiments we performed using the 3-behavior robot are less conclusive. As we have shown earlier, the 3-behavior robot is not very successful at finding and pushing boxes in the large environments, and so it does not do very well at the task even with prior learning.

To verify our results, we run another set of experiments, this time making the robots to learn first in the sparse environment (shown in Figure 8) and then in the cluttered environment (shown in Figure 7), and viceversa. The idea here is to verify if the size and complexity of the space where the initial learning is done have any noticeable impact on the subsequent learning. The 6-behavior robot did quite well when learning initially in the cluttered environment and then moving to the sparse environment, but was also successful (even though to a lesser extent) when doing the reverse. The 3-behavior robot was again not very successful. The major reason for the better success of the 6-behavior robot is due to its Wander behavior, that allows it to explore more of the space. In sparse environments the Wander behavior is active much more than the Unnwedge behavior, but even in cluttered environments Wander is the most active behavior.

We can make the claim, nevertheless, that prior knowledge potentially can improve the robot's performance when adapting to different environments. The disparity of the results between the cluttered and sparse environment experiments shows that the improvement is still very dependent on the robot's current environment. Environments in which the robot can perform the box pushing task well, i.e. sparse environment for the 6-behavior robot, will also be the environments that the robot can more easily adapt to.

This conclusion is not terribly revolutionary, and the experimental results from the robot adapting to the cluttered environment indicate that practical success of the robot applying knowledge from different environments is rather limited. In fact, as the experiment shows,

at times prior knowledge can be a detriment.

11 Conclusions

The first conclusion we must make is that the underlying behavior-based architecture is crucial to the robot's learning performance, perhaps more important than how the sensor data is grouped and how the reward functions are parameterized. The more detailed and explicit the architecture is, the better the robot will learn and perform its box-pushing task. Stated another way, the original Mahadevan and Connell approach to machine learning can be improved, and without significantly changing the machine learning method itself. The method does not replace or over-shadow the architecture, but instead enhances it. The best aspect of the Mahadevan and Connell approach is that it fully utilizes the advantages of the behavior-based architecture, i.e. its robustness, limited error propagation, and its ability to operate in real-time.

Similarly, we can conclude that the machine learning method examined in this paper is scalable, scalable in the sense that its learning can carry over into more complex environments. The method is also flexible, in the sense that the type of environment does not matter. The experiments show that the 6-behavior robot performs reasonably well in both the cluttered environment and the open, sparse environment, although the degree of improvement is still dependent on the environment. How much more complex the environment can be remains a subject for future investigation.

The last set of experiments shows that by using this machine learning approach, the robot's learned sensor information can be applied from one environment to a different environment with some success. Again, that success is dependent on the environment. Even though the performance improvements are limited and environment dependent, the results are encouraging. Any improvement in performance is better than having to learn from scratch every time the environment changes.

It is also evident through prolonged observations of the simulated robot, that further improvements can be made. The 6-behavior robot still spends too much effort exploring territory that it has already explored. A possible solution would be to implement a kind of localized map making in the robot, whereby the robot stores patterns of sensor states, and check to make sure that recent patterns are not repeated. Another problem concerns the rate at which the robot learns. There is a point during the robot's learning trials where the continued application of rewards and punishments are no longer helpful. This becomes evident in the simulation when the robot appears "jittery" and begins to spend more time turning in open space than moving forward. The propagation of reward values over and over causes the robot to look indecisive, and degrades effectiveness. It would be wise then to implement the rewards and punishments not as constant values, but as functions that decrease their value over time, as done, for instance, in [Hougen *et al.*, 1996]. Such a change would likely improve the robot's learning ability even more.

References

[Brooks, 1986] Rodney Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[Comer, 1979] D. Comer. The ubiquitous B tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[Connell and Mahadevan, 1993] Jonathan Connell and Sridhar Mahadevan. Rapid task learning for real robots. In J. H. Connell and S. Mahadevan, editors, *Robot Learning*. Kluwer Academic Publ., 1993.

[Dorigo and Colombetti, 1994] Marco Dorigo and Marco Colombetti. Robot shaping: developing autonomous agents through learning. *Artificial Intelligence*, 71(2):321–370, 1994.

[Dorigo, 1995] Marco Dorigo. Alecsys and the autonomouse: Learning to control a real robot by distributed classifier systems. *Machine Learning*, 19(3):209–240, 1995.

[Hilgard and Bower, 1975] E. R. Hilgard and G. H. Bower. *Theories of Learning*. Prentice-Hall, Englewood Cliffs, NJ, fourth edition, 1975.

[Hougen *et al.*, 1996] Dean F. Hougen, John Fischer, Maria Gini, and James Slagle. Fast connectionist learning for trailer backing using a real robot. In *IEEE Int'l Conf. on Robotics and Automation*, pages 1917–1922, 1996.

[Kaelbling *et al.*, 1996] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[Koza and Rice, 1992] John R. Koza and James P. Rice. Automatic programming of robots using genetic programming. In *Proc. Nat'l Conf. on Artificial Intelligence*, pages 194–201, 1992.

[Mahadevan and Connell, 1992] Sridhar Mahadevan and Jonathan Connell. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55:311–365, 1992.

[Mahadevan, 1994] Sridhar Mahadevan. To discount or not to discount in reinforcement learning: A case study comparing R-learning and Q-learning. In *Proceedings of the 11th International Conference on Machine Learning*, pages 164–172, New Brunswick, N.J., July 1994.

[Mahadevan, 1996] Sridhar Mahadevan. Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22:159–196, 1996.

[Mataric, 1994] Maja Mataric. Reward functions for accelerated learning. In *Proceedings of the 11th International Conference on Machine Learning*, New Brunswick, N.J., July 1994.

[Schwartz, 1993] A. Schwartz. A reinforcement learning method for maximizing undiscounted rewards. In *Proceedings of the 10th International Conference on Machine Learning*, pages 298–305, 1993.

[Singh, 1992] S. P. Singh. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8(3–4):323–339, 1992.

[Watkins and Dayan, 1992] C.J. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

[Watkins, 1989] C.J. Watkins. *Models of delayed reinforcement learning.* PhD thesis, Cambridge University, Cambridge, UK, 1989.

[Whitehead *et al.*, 1993] S. D. Whitehead, J. Karlsson, and J. Tenenberg. Learning multiple goal behavior via task decomposition and dynamic policy merging. In J. H. Connell and S. Mahadevan, editors, *Robot Learning*, pages 45–78. Kluwer Academic Publ., 1993.

[Whitehead, 1992] S. D. Whitehead. *Reinforcement learning for the adaptive control of perception and action.* PhD thesis, University of Rochester, 1992.