Automatic Programming of Behavior-based Robots using Reinforcement Learning

Sridhar Mahadevan and Jonathan Connell
IBM T.J. Watson Research Center, Box 704
Yorktown Heights, NY 10598
(sridhar@ibm.com and jhc@ibm.com)

Abstract

This paper describes a general approach for automatically programming a behavior-based robot. New behaviors are learned by trial and error using a performance feedback function as reinforcement. Two algorithms for behavior learning are described that combine techniques for propagating reinforcement values temporally across actions and spatially across states. A behavior-based robot called OBELIX (see Figure 1) is described that learns several component behaviors in an example task involving pushing boxes. An experimental study using the robot suggests two conclusions. One, the learning techniques are able to learn the individual behaviors, sometimes outperforming a hand-coded program. Two, using a behavior-based architecture is better than using a monolithic architecture for learning the box pushing task.

Introduction

Behavior-based robots using the *subsumption* architecture [1, 4] decompose an agent into a layered set of task-achieving modules. Each module implements one specific behavior, such as "avoid hitting anything" or "keep following the wall". Thus, each module has to solve only the part of the perception or planning problem that it requires. Furthermore, this approach naturally lends itself to incremental improvement, since new layers can be easily added on top of existing layers.

One problem with behavior-based robots is that the component modules have to be laboriously programmed by a human designer. If new behaviors could be learned, it would free the designer from needing a deep understanding of the interactions between a particular robot and its task environment.

The problem of acquiring new behaviors has been addressed by work in reinforcement learning [5, 11, 12]. This studies how an agent can choose an action based on its current and past sensor values such that it maximizes over time a reward function measuring the agent's performance. Teaching robots using reinforcement learning is attractive because specifying a reward function for a task is often much easier than explicitly programming the robot to carry out the task. However, previous work has been limited to situations where the task is learned *monolithically*, that is as a single behavior.

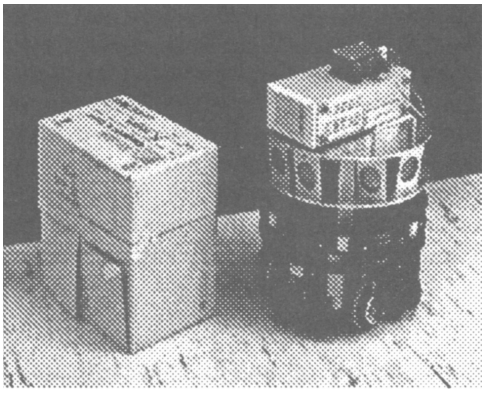

Figure 1: The OBELIX robot examining a box

This paper proposes instead using reinforcement learning to automate the programming of a behavior-based robot. Each behavior in such a robot is generally comprised of an applicability condition specifying when it is appropriate, and an action generation mechanism specifying the best action in any state. Our approach assumes the robot is initially given the applicability condition on each behavior, and a priority ordering to resolve conflicts among various behaviors. However, it does not depend on any particular set of sensors or actions. The key idea is to learn an action generation mechanism for each behavior that maximizes a fixed performance function over time.

Using a behavior-based architecture in reinforcement learning has several advantages. Separate (and simple) reward functions can be written for each module, allowing the robot to be rewarded more frequently. This simplifies the temporal credit assignment problem since rewards have to be propagated across fewer actions. The applicability function of each module provides a natural medium for encoding state history information. This also helps reduce the perceptual alias-

ing problem [13], since the same state may provoke different reactions from the different modules.

OBELIX: A Robot Vehicle

This section describes a robot that was used as a testbed for the learning experiments. Figure 1 shows a behavior-based robot called OBELIX that learns several constituent behaviors in an example task of pushing boxes. The robot uses a 9600 baud Arlan 130 radio link to continually send sensor data back to a workstation which in turn responds with motion commands.

The robot itself is built on a small, 12" diameter, 3-wheeled base from RWI. For our experiments, we limit the motion of the vehicle to either moving forward, or turning left or right in place by two different angles (22 degrees or 45 degrees).

OBELIX's primary sensory system is an array of 8 sonar units. Each sensor in the array has a field of view of roughly 20 degrees and can see out to about 35 feet. For the purposes of the experiments described here, we use only two range bins. One extends from 9" to 18" (NEAR) and another covers the distance between 18" and 30" (FAR). The individual sonar units are arranged in an orthogonal pattern. There are 4 sonars looking toward the front and 2 looking toward each side.

There are also two secondary sources of sensory information. There is an infra-red (IR) detector which faces straight forward and is tuned to a response distance of 4". This sensor provides a special bit called BUMP since it only comes on when something is right against the front of the robot. The robot also monitors the motor current being used for forward motion. If this quantity exceeds some fixed threshold, another special bit, STUCK, is turned on.

To summarize, 18 bits of information are extracted from the sensors on the robot. Of these, 16 bits of information come from the 8 sonar sensors (1 bit from the NEAR range and 1 bit from the FAR range). There is also 1 bit of BUMP information, and 1 bit of STUCK information. The 18 bits generate a total state space of about a quarter million states. It is the job of the learning algorithm to decide which of the 5 actions to take in each of these states.

The Box Pushing Task

This section describes an example task of having a robot push boxes across a room. One can view this as a simplified version of a task carried out by a warehouse robot, although clearly one would not use a *round* robot to push *square* boxes! Conceptually, the box pushing task involves three subtasks. First, the robot needs to find potential boxes and discriminate them from walls and other obstacles. Second, it needs to be able to push a box across a room. Finally, it needs to be able to recover from stalled situations where it has either pushed a box to a corner, or has attempted to push an immovable object like a wall. Our approach will be to learn each of these subtasks as a distinct reactive behavior. By reactive, we mean that we base the control decision on only the currently perceived sensory information.

Figure 2 illustrates the overall structure of a behavior-based robot for the box pushing task. It also depicts a priority network that imposes an ordering on the three subtasks of the box pushing task. The priority network is specified by "suppressor" nodes shown as circles containing the letter "S". The semantics of a suppressor node is that commands injected from the top of the node take precedence over those injected horizontally from the left side. Thus, Figure 2 shows that the unwedging behavior supersedes the pushing behavior, and that both of these in turn supersede the finding behavior.

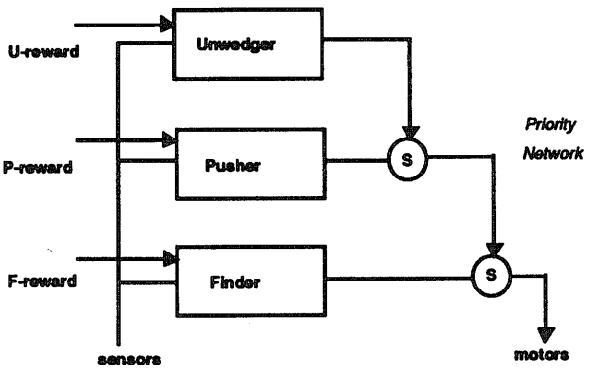

Figure 2: Modules in a Box Pushing Robot

Behavior 1: Finding a Box

In order to push a box, OBELIX has to first find one. At this point we need to define what constitutes a "box". The constraints on a box are that the robot should be able to physically push it, and be able to distinguish it from obstacles such as walls. In practice, we use empty rectangular paper cartons about a cubic foot in volume.

One way to encourage the robot to find boxes is to reward the robot whenever the NEAR sensor bits on the front sonars turn on. This encourages the box finder to go toward objects. We use a disjunction of the NEAR state bits of the central front facing sonars on the robot as a "matched filter" for recognizing boxes. If the robot went forward, and turned these bits on, the robot is "rewarded" by +3; if these bits are off, the robot is "punished" by -1; the default reward is 0. The box finder is always applicable; however, since its priority is the lowest, it controls the robot only when the other behaviors are inapplicable.

Behavior 2: Pushing a Box

Once OBELIX has found a box, it needs to push it until the box is wedged against an immovable obstacle

(like a wall). What makes this task difficult is that boxes tend to rotate if they are not pushed with a force directed through their center of drag. OBELIX has to learn to keep the box centered in its field of view, by turning whenever the box rotates to one side of the robot. The robot gets rewarded by +1 whenever it continues to be bumped and going forward. It gets punished by -3 whenever it loses contact with the box.

Intuitively, the box pushing behavior should be applicable whenever OBELIX is actively pushing a box, and should not be applicable otherwise. One problem with such a criterion is that the moment OBELIX loses contact with a box, the behavior is turned off, and OBELIX has no opportunity to correct its mistakes. A better scheme in practice is to allow a behavior to continue to be applicable for a fixed number of time steps after the applicability predicate (which first turned it on) ceases to be true. In particular, the box pushing behavior continues to be applicable 5 time units after OBELIX has lost contact with a box. This allows some time for it to try to recover and push the box once again.

Behavior 3: Getting Unwedged

Given that OBELIX is learning to find and push boxes in a cluttered laboratory environment, it is very likely that it will bump into walls and other immovable obstacles and become stalled or wedged. Pushing a box into a wall will also cause a stalled state. A separate behavior is dedicated in OBELIX to extricate it from such situations. The basic idea is for OBELIX to turn around sufficiently so that it can begin going forward again. Even though this task seems simple, it turns out to be quite hard. OBELIX can easily learn to turn once it gets into a stalled situation. It does not readily learn to turn in the right direction, and by the right amount.

The unwedging behavior is rewarded by +1 when the robot is no longer stalled, and is able to go forward once again. It is punished by -3 if the robot continues to be stalled. The unwedging behavior is deemed applicable any time the robot is stalled. As in the case of the box pushing behavior, the unwedging behavior continues to be applicable for 5 time steps after the robot is no longer stalled.

Learning Algorithms

This section briefly describes two learning algorithms that we have implemented on OBELIX. They combine a well known learning algorithm for temporal credit assignment, Q learning [12], with two different structural credit assignment techniques: weighted Hamming distance and statistical clustering. The goal of the learning is to acquire an action generation mechanism for each module that maximizes the reward obtained by the module over time. A much more detailed description of the two algorithms is given in [9].

Q Learning

Q learning [12] uses a single utility function $Q(x, a)$ across states (x) and actions (a) to evaluate both actions and states. By definition, $Q(x, a) = r + \gamma E(y)$, where r is immediate payoff or reward, and $E(y)$ is the utility of the state y resulting from the action. γ is a discount parameter between 0 and 1. In turn, $E(y) =$ maximum $Q(y, a)$ over all actions a. During learning, the stored utility values $Q(x, a)$ have not yet converged to their final value (i.e. to $r + \gamma E(y)$). Thus, the difference between the stored values and their final values gives the error in the current stored value. In particular, Q learning uses the following rule to update stored utility values.

$$Q(x, a) \leftarrow Q(x, a) + \beta(r + \gamma E(y) - Q(x, a))$$

Thus, the new Q value is the sum of the old one and the error term multiplied by a parameter β, between 0 and 1. The parameter β controls the rate at which the error in the current utility value is corrected.

Weighted Hamming Distance

To become better at box pushing, OBELIX needs to propagate rewards across states, so that "similar" states provoke the same response from it. The similarity metric used in the first algorithm is as follows. First, the state description is reduced from 18 to 9 bits by disjoining some neighboring sonar bits. Then, the reduced states are compared based on the Hamming distance between them. The Hamming distance between any two states is simply the number of state bits that are different between them. However, in our case, not all bits are equally important. Since it is important to never generalize across states in which BUMP or STUCK differ, these carry a higher weight than the other state bits. In particular, BUMP and STUCK carry a weight of 5, the near sonar bits carry a weight of 2, and the other bits carry a default weight of 1. With these weights, we define two states as being distinct if the weighted Hamming distance between them is greater than 2.

Statistical Clustering

The second algorithm uses statistical clustering to propagate reward values across states. Using this algorithm, the robot learns a set of clusters for each action that specify the utility of doing the action in particular classes of states. More formally, a cluster is a vector of probabilities $< p_1, ..., p_n >$, where each p_i is the probability of the ith state bit being a 1. Each cluster has associated with it a Q value indicating its worth. Clusters are extracted from instances of states, actions, and rewards that are generated by the robot exploring its task environment.

A state s is considered an instance of a cluster c if two conditions are satisfied. One, the probability $P(s \in c)$ – computed by multiplying the probabilities

p_i or $(1-p_i)$, depending on whether the ith bit of state s is a 1 or a 0 – should be greater than some threshold ϵ. Two, the absolute difference between the Q values of the state and the cluster should be less than some threshold δ. If a state matches a cluster, it is merged into the cluster by updating the cluster probabilities.

If a state does not match any of the existing clusters, a new cluster is created with the state being its only instance. Alternatively, two clusters can be merged into one "supercluster" if the Euclidean "distance" between the clusters (treating the clusters as points in n dimensional probability space) is less than some threshold ρ, and the absolute difference between their Q values is less than δ.

Action Generation

The best action a to perform in a given state x is the one that has the highest utility $Q(x,a)$. The first algorithm stores an array $Q(x,a)$ exhaustively specifying the utilities of doing any action a in any state x. In contrast, the second algorithm computes the utility as $Q(x,a) = \frac{\sum_{c \in a} Q_c P(x \in c)}{\sum_{c \in a} P(x \in c)}$. The numerator is the sum of the Q values of the clusters stored under an action, weighted by the probability of state x matching a cluster c. The denominator, which is a normalization factor, is the sum of the match probabilities of the state x over the clusters associated with action a.

Summary of Algorithms 1 and 2

Figure 3 combines the description of the two algorithms. Step 2b requires some explanation. In order to ensure the convergence of the Q values, it is important that every state be sampled periodically. This is ensured by taking a random action some of the time.

1. *Initialization:* (For algorithm 1, create an array $Q(x,a)$ whose initial entries are 0.) (For algorithm 2, initialize the clusters under each action a to NIL, and fix ϵ, δ, and ρ.)
2. *Do the following steps forever:*
a. Observe the current world state s.
b. 90% of the time, choose an action a that maximizes $Q(s,a)$. Else choose a random action.
c. Carry out a. Let the reward obtained be r.
d. Update $Q(s,a)$ via the Q learning update rule.
e. (For algorithm 1, also update $Q(s',a)$ for all s' s.t. $weighted\_hamming\_distance(s,s') \leq 2$.) (For algorithm 2, if $\exists$ a cluster c under a which matches s, merge s into c. Else create a new cluster c' whose only instance is s. Merge existing clusters under a if possible.)

Figure 3: The two learning algorithms

Experimental Results

This section describes a detailed experimental study evaluating the performance of the two learning algorithms described above. Mainly, we are interested in determining (i) how well the robot learns each individual behavior, and (ii) the effect of decomposing the overall task into a set of subsumption modules on learning.

The first question can be answered by measuring the improvement in performance of each individual behavior as a function of the learning. The second question can be answered by comparing the improvement in overall performance obtained by learning each behavior separately with that obtained by learning the box pushing task as a single monolithic control system.

Learning Each Behavior Separately

For the first set of experimental results, we focus on learning each behavior separately. Figure 4 presents data collected using the robot on learning to find boxes using four different algorithms: Q learning with weighted Hamming, Q learning with statistical clustering, a handcoded agent, and a random agent. The graph plots the average value of the reward obtained so far by the finder module at various points along the learning run. That is, the vertical axis represents the sum of all rewards received by the module divided by the number of steps that the module has been in control of the robot. The horizontal axis represents the percentage of the learning run of 2000 steps that has elapsed. Note that each module is active only for some fraction of these steps. The graph shows that the two learning algorithms improve steadily over the learning run, and do substantially better than the random agent but not as well as the handcoded agent. The handcoded and random agents show some performance variations over the learning run partially due to the fact that the robot takes a random action 10% of the time.

We have similarly analyzed the performance of the learning algorithms on the other two behaviors, pushing and unwedging. However, space does not permit showing these graphs (again, a more complete treatment is given in [9]). Instead, we summarize our results by extracting some quantitative information from the data. Table 1 compares the "ultimate performance" obtained using the learning algorithms, the handcoded agent, and the random agent. An example will help illustrate how these numbers were computed. At the end of the learning run, the average reward for the box finder behavior using Q learning with clustering was 0.16 (see Figure 4). The maximum and minimum reward values for the box finder are 3.0 and -1.0. Hence the percentage improvement for box finder from the lowest reward value is

$$\frac{0.16-(-1.0)}{3.0-(-1.0)} \times 100 = 29\%$$

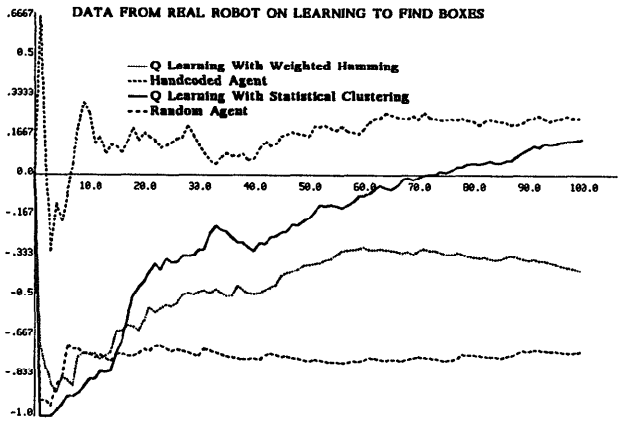

Figure 4: Data from robot on learning to find boxes

Technique	Finder	Pusher	Unwedger
Handcoded	31%	57%	73%
Clustering	29%	60%	72%
Hamming	15%	55%	74%
Random	8%	30%	68%

Table 1: Ultimate performance at end of learning run

The table indicates that the learning algorithms were fairly successful at learning to find and push boxes, and unwedge from stalled states – the ultimate performance is close to or better than the performance of the handcoded agent. The random agent does much worse in general, except at unwedging. Given some thought, this is not so surprising – if the robot is stuck against an obstacle, randomly thrashing around will very quickly unwedge it!

Learning Box Pushing Monolithically

Now we compare the subsumption approach with an agent who learns the box pushing task in its entirety without decomposing it – this was our initial unsuccessful approach to the problem. We created a monolithic learner by defining a single module that was active all the time. The single module was given a reward of 1 when it pushed a box – that is, it was bumped in two successive states while going forward, and was not stuck in the second state – and was given a reward of 0 otherwise. Table 2 compares the subsumption approach with the monolithic approach using as a metric the number of steps during which the robot was actually pushing a box. The table shows the number of box pushing steps for the two approaches over a learning run of 2000 steps.

Summarizing, analysis of the data shows that the learning algorithms were able to successfully learn the three separate behaviors in the box pushing task. Furthermore, the subsumption approach seems to be superior to the monolithic approach by at least a factor of two at learning the task.

Technique	Monolithic	Subsumption
Clustering	35	72
Hamming	27	65

Table 2: Number of steps a box was pushed over learning run

Limitations

Our work currently suffers from a number of limitations. The box pushing task is quite simple as it involves only 5 actions and a state representation of 18 bits. However, Lin [7] has recently shown that our approach can be extended to a more complex task by explicitly teaching the robot. Algorithm 1 scales badly since it requires explicitly storing all possible states. Although Algorithm 2 overcomes this problem, it requires fine tuning several parameters to ensure that the clusters under each action are semantically meaningful. Algorithm 2 is also limited in that clusters once formed are never split (see [2] for a splitting algorithm that seems to be a dual of Algorithm 2). The number of box pushing steps in Table 2 is admittedly low. This is partly because boxes are very difficult to detect using sonar. Finally, our experiments have been limited to comparing the subsumption approach versus a simple monolithic approach. It is conceivable that modular controller architectures other than subsumption may yield similar computational benefits [14].

Related Work

Our work draws extensively on previous work in machine learning. Q learning was developed by Watkins [12]. Sutton [11] showed how Q learning could be integrated into a planning system. Lin [6] presents a detailed study of different algorithms using Q learning. Our work differs from these in that we are studying the integration of spatial and temporal credit assignment on a real robot.

Our work also draws on earlier research on behavior-based robots using the subsumption architecture [1]. Our particular style of decomposition derives from Connell's thesis [4]. The main difference is that we are studying how to automatically program such robots by having them learn new behaviors.

Mitchell's task [10] of sliding a block using a finger contact is similar to the box pushing task. His approach involves using a partial qualitative physics domain theory to explain failures in sliding the block, which are then generalized into rules. Our approach instead uses an inductive trial and error method to improve the robot's performance at the task. Christiansen et. al. [3] describe a similar trial and error inductive approach for learning action models in a tile sliding task.

Kaelbling [5] describes some work on using reinforcement learning in the context of a mobile robot. The task was to move towards a bright light. Her task is much simpler than ours since a state consists of only 4 bits, as opposed to 18 bits in our case. Another difference is that we use the subsumption structure to speed up the learning.

Maes and Brooks [8] describe a technique for learning to coordinate existing behaviors in a behavior-based robot. Their work is complementary to our own. In our case new behaviors are learned assuming a priority ordering to coordinate the behaviors. In their case, the behaviors are known, and the priority ordering is learned. The reinforcement learning task in our case is more challenging since reward is a scalar variable, whereas in their case rewards are binary. Also, our techniques address the temporal credit assignment problem of propagating delayed rewards across actions. In their work, since rewards are available at every step in the learning, no temporal credit assignment is necessary.

Conclusions

This paper attempts to empirically substantiate two claims. One, reinforcement learning is a viable approach to learning individual modules in a behavior-based robot. Two, using a subsumption architecture is superior to using a one-part controller in reinforcement learning. We have provided experimental evidence that support these claims using a robot which successfully learns several component behaviors in a task involving pushing boxes.

An especially acute problem for real robots is the limited number of trials that can be carried out. Reinforcement learning, being a weak method, is often slow to converge in large search spaces. We have used a behavior-based architecture to speed up reinforcement learning by converting the problem of learning a complex non-reactive task into that of learning a simpler set of special-purpose reactive tasks.

We plan to extend our work in several directions. We would like to explore approaches intermediate between the monolithic and subsumption controllers as described in the paper. It would also be useful to learn both the priority ordering on behaviors and the action generation mechanism simultaneously. Finally, having the robot learn by "watching" a human do a task may enable our approach to scale to tasks more complex than box pushing [7].

References

[1] R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1), 1986.

[2] D. Chapman and L. Kaelbling. Learning from delayed reinforcement in a complex domain. Technical Report TR-90-11, Teleos Research, 1990.

[3] A. Christiansen, T. Mason, and T. Mitchell. Learning reliable manipulation strategies without initial physical models. In *Proceedings of the IEEE Conference on Robotics and Automation*, pages 1224–1230. Morgan Kaufmann, 1990.

[4] J. Connell. *Minimalist Mobile Robotics: A Colony-style Architecture for an Artificial Creature.* Academic Press, 1990. Also available as MIT AI TR 1151.

[5] L. Kaelbling. *Learning in Embedded Systems.* PhD thesis, Stanford University., 1990.

[6] L. Lin. Self-improving reactive agents: Case studies of reinforcement learning frameworks. Technical Report CMU-CS-90-109, Carnegie-Mellon University., 1990.

[7] L. Lin. Programming robots using reinforcement learning and teaching. In *Proceedings of the Ninth AAAI*, 1991. To appear.

[8] P. Maes and R. Brooks. Learning to coordinate behaviors. In *Proceedings of the Eighth AAAI*, pages 796–802. Morgan Kaufmann, 1990.

[9] S. Mahadevan and J. Connell. Automatic programming of behavior-based robots using reinforcement learning. Technical Report RC 16359, IBM, 1990.

[10] T. Mitchell. Towards a learning robot. Technical Report CMU-CS-89-106, Carnegie-Mellon University., 1989.

[11] R. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224. Morgan Kaufmann, 1990.

[12] C. Watkins. *Learning from Delayed Rewards.* PhD thesis, King's College, 1989.

[13] S. Whitehead and D. Ballard. Active perception and reinforcement learning. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 179–188. Morgan Kaufmann, 1990.

[14] L. Wixson and D. Ballard. Learning to find objects. Technical report, Univ. of Rochester, 1991. In preparation.