Artificial Intelligence 55 (1992) 311–365
Elsevier

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, USA

Received January 1991
Revised July 1991

Abstract

Mahadevan, S. and J. Connell, Automatic programming of behavior-based robots using reinforcement learning, Artificial Intelligence 55 (1992) 311–365.

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 Q learning, a well-known scheme for propagating reinforcement values temporally across actions, with statistical clustering and Hamming distance, two ways of propagating reinforcement values spatially across states. A real behavior-based robot called OBELIX is described that learns several component behaviors in an example task involving pushing boxes. A simulator for the box pushing task is also used to gather data on the learning techniques. A detailed experimental study using the real robot and the simulator suggests two conclusions.

(1) The learning techniques are able to learn the individual behaviors, sometimes outperforming a handcoded program.
(2) Using a behavior-based architecture speeds up reinforcement learning by converting the problem of learning a complex task into that of learning a simpler set of special-purpose reactive subtasks.

1. Introduction

Behavior-based robots, such as those using the subsumption architecture, are a promising and demonstrably successful approach to building intelligent

Correspondence to: S. Mahadevan, IBM T.J. Watson Research Center, Box 704, Yorktown Heights, NY 10598, USA. E-mail: sridhar@watson.ibm.com

autonomous agents [3,8]. They radically differ from the traditional approach of structuring an agent's architecture into *functional* modules—perception, planning, learning, etc.—by instead organizing an agent as a layered set of *task-achieving* modules. Each module implements one specific control strategy or behavior, such as "avoid hitting anything" or "keep following the wall". Thus, with this approach, each module has to solve only the part of the perception or planning problem that it requires. Furthermore, the subsumption approach naturally lends itself to incremental improvement, since new layers can be easily added on top of existing layers. Figure 1 contrasts the traditional approach with the subsumption approach to building intelligent agents.

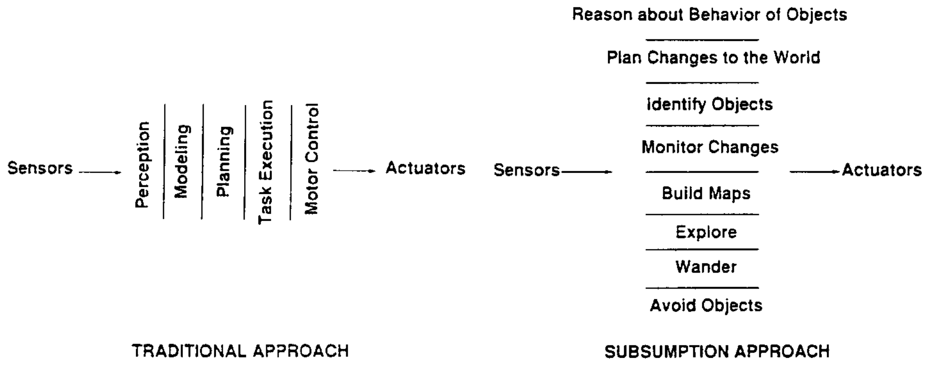

Fig. 1. Traditional and behavior-based architectures for an intelligent agent.

One of the problems with behavior-based robots is that the component modules have to be laboriously programmed by a human designer. Although some exciting progress has been made recently in learning to coordinate various behaviors [14], the task of programming in each individual behavior remains the burden of a human designer. If new behaviors could be learned, it would also free the designer from needing a deep understanding of the interactions between a particular robot and its application environment.

In this paper we are interested in developing learning algorithms that allow a real mobile robot to learn new behaviors in an initially unknown environment. This differs from approaches where the robot has considerable task knowledge to begin with [17]. Also, we want the learning algorithm to be sensitive to the task being learned. This contrasts with general algorithms for learning the structure of finite state environments [23]. Finally, we are interested in situations where a mobile robot can only perceive the portion of the task environment that is adjacent to it. This deviates from approaches using global sensing, such as an overhead camera [7].

Reinforcement learning [2,11,28,30] studies how an agent can optimally 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. This is a particularly appealing approach because it allows a robot to learn in unknown environments, adapt its learning to the particulars of a task, and choose actions based on the currently perceived state. Also, specifying a reward function for a task is often much easier than explicitly programming the robot to carry out the task.

However, one problem with reinforcement learning is that it is slow to converge in large search spaces, an especially acute problem in robotics. By using a behavior-based architecture, we are able to overcome this problem to some extent, since it decomposes the problem of learning a complex task into a set of simpler problems of learning each constituent behavior. This reduces the overall search space, and thereby accelerates the convergence of the learning algorithm—an important consideration for real robots. It also helps reduce the perceptual aliasing problem [32], which is caused by the many-to-one and one-to-many mapping between external world states and internal perceived states. In particular, state history information can be encoded as part of the module, which helps the robot disambiguate the current state. Thus, the same state may provoke different reactions from the different modules.

Each behavior in a behavior-based robot is generally comprised of an applicability condition specifying when it is appropriate, and a policy specifying the best action in any state. Our approach assumes the robot is initially supplied with an applicability condition on each behavior, as well as a priority ordering to resolve conflicts among the various behaviors. However, it does not depend on any particular set of sensors or actions. The key idea is to learn a policy for each behavior that over time maximizes a fixed performance function.

We describe an actual behavior-based robot called OBELIX (shown in Fig. 2) that learns several constituent behaviors in an example task of pushing boxes. In particular, OBELIX combines Q learning [30], a technique for propagating rewards temporally across sequences of actions, with two state generalization techniques—weighted Hamming distance and statistical clustering—which propagate rewards spatially across similar states.

We present detailed experimental evidence from learning runs on the real robot and a simulator showing how performance at the box pushing task improves with learning. In particular, we empirically compare the performance of two learning algorithms on the three behaviors involved in the box pushing task with a handcoded agent as well as a random agent. Finally, we contrast these results with those for learning the box pushing task using a monolithic architecture, that is without decomposing the task into simpler behaviors. The results indicate that the two algorithms are able to learn each behavior to a reasonable degree of performance, sometimes exceeding the handcoded agent's performance. Furthermore, the

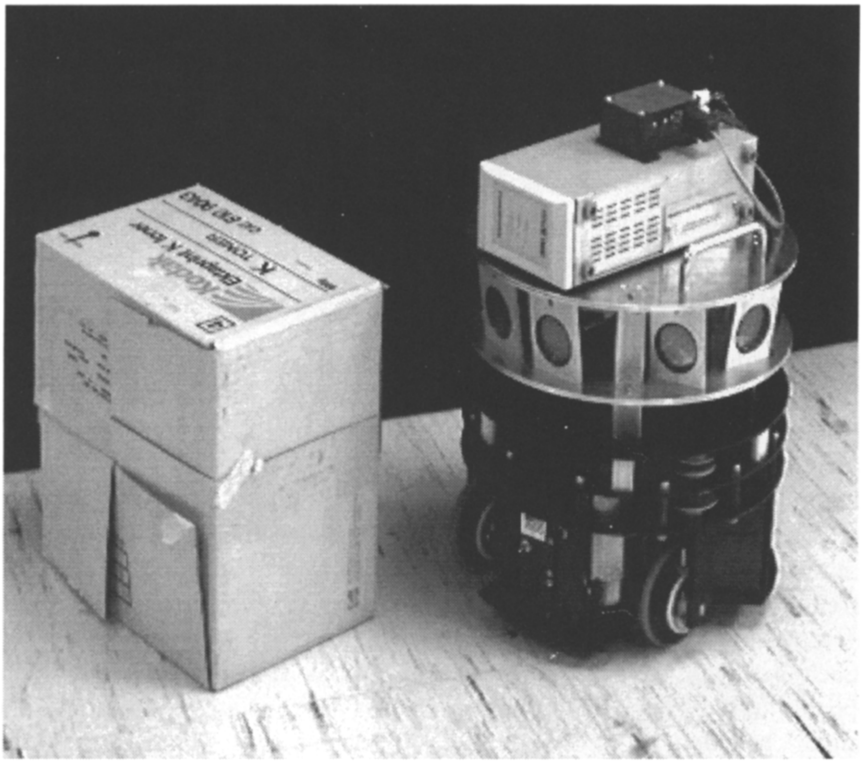

Fig. 2. The OBELIX robot examining a box.

results suggest that learning each behavior separately is superior to learning the entire task as one behavior.

The rest of the paper is organized as follows. Section 2 discusses some relevant characteristics of the robotics domain, and how they influence the design of any potential learning technique. Section 3 provides a brief overview of reinforcement learning. Section 4 presents a specific architecture for behavior-based robots. Section 5 describes a real and a simulated robot testbed for studying robot learning. Section 6 describes the example task of box pushing. Section 7 describes two reinforcement learning algorithms that we have implemented for the box pushing task. Section 8 contains experimental results comparing the two algorithms with handcoded and random agents. Section 10 discusses related work. Finally, Section 11 concludes, and suggests directions for further research.

2. Salient characteristics of the robot domain

There are a number of features of the robot domain that distinguish it from other domains. Robots are constrained to interact with the real world through sensors and effectors. One way this affects learning is that any proposed algorithm has to work with information that can be computed from the available sensors. For example, Sutton [28] describes a simulated robot that learns to navigate in a simple grid-based environment. However, his algorithm cannot be used on a sonar-based robot since it assumes the precise coordinates of the robot are known at all times. Most easy-to-build sensors are incredibly noisy. For example, Polaroid sonars are a popular sensor on many robots because they are readily available and cheap, but they are prone to many different kinds of errors. This means that the state representation constructed from sonar images is bound to have some incorrect elements, and any learning algorithm must be able to handle this problem.

Similarly, actions often do not have the desired effect because the robot cannot carry them out to the desired precision. For example, a robot for stacking blocks may not be able to align two blocks so that they are flush because of the limited resolution of the arm. Also, actions are almost never exactly invertible—turning left by a certain angle and then right by the same angle does not guarantee that the robot is back in the same state it was before doing the actions.

For these reasons, we believe it is important to experiment with real physical robots. There are many facets of the real world that are very difficult to simulate effectively—for example, how does a sonar reflect off a metal file cabinet? Often, certain difficulties with simulators disappear with real robots—for example, to get out of loops in a simulator requires doing random actions once in a while, whereas sufficient randomness is always present in the real world. Likewise, we have found algorithms that work effectively on a simulator sometimes do not work on a real robot—for example, because they converge too slowly or because real sensors contain more noise than expected. On the other hand, it is easier to test out new algorithms on a simulator since many more trials can be carried out in the same amount of time. Thus, we have used both real and simulated robot testbeds in our study.

2.1. Requirements for a robot learning algorithm

Robotics is particularly challenging as a domain for any learning algorithm. In particular, some requirements on a learning algorithm for it to be applicable to robots are given below:

- *Noise immunity*: The technique should be able to deal with noise. State descriptions may sometimes be wrong, and the use of probabilistic smoothing techniques is often required.
- *Fast convergence*: The technique should be able to converge in a reasonable number of trials since actions are required to obtain examples,

and it is difficult to carry out a million actions on a real robot.

- *Incrementality*: The learning algorithm should allow the robot to improve its performance while it is learning. Since the examples are generated by the robot itself, this allows the robot to explore its environment quicker and generate better examples.
- *Tractability*: The learning algorithm should be computationally tractable. That is, every iteration of the algorithm should be doable in real time.
- *Groundedness*: The technique should only depend on information that can actually be extracted from the sensors on a robot.

3. Reinforcement learning

Reinforcement learning studies the problem of inducing by trial and error a policy from states to actions that maximizes a fixed performance measure (or reward) [2,11,28]. It is similar to models used in some psychological studies of behavior learning in animals and humans. However, it differs in several important respects from the more traditional work on concept learning [16]. One, it is unsupervised learning meaning that examples are not carefully selected by a teacher. Instead, the distribution of examples is influenced by the learner's actions, since the states and rewards experienced by the learner depend on the actions it takes. Two, the learning has as a single goal choosing actions that maximize the cumulative reward over time. In contrast, other types of learners have different goals, such as minimizing description length or reformulating a concept definition. Finally, the learner is faced with a difficult credit assignment problem of evaluating the goodness of states and actions from a scalar reinforcement signal, not a very helpful explanation!

Figure 3 illustrates the two credit assignment problems, *temporal* and *structural*, in reinforcement learning [26]. The figure shows several possible action sequences leading to the current state ($t = 0$). The one actually carried out is drawn with darker lines. The temporal credit assignment problem is how to propagate the reward backwards in time from $t = 0$ to $t = -3$. The structural credit assignment problem, on the other hand, is how to propagate the reward spatially across states so that similar states cause the agent to take similar actions. The latter seems to be the harder of the two problems for real robots.

Following Kaelbling [11], we can view most reinforcement learning algorithms as instantiations of the general scheme shown in Fig. 4. Any reinforcement learner has the following components: an internal state S, an update function U that modifies S, and an evaluation function V for

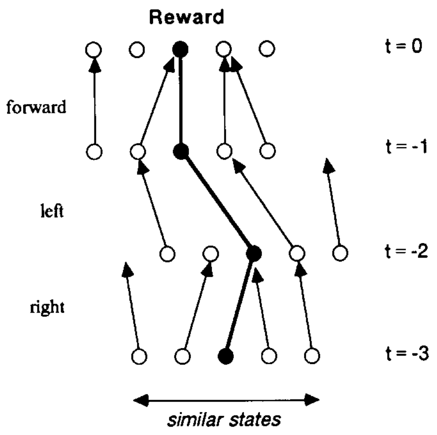

Fig. 3. Temporal versus structural credit assignment.

```
1. Initialize the learner's internal state $S$ to $S_0$.
2. Do Forever:
    (a) Observe the current world state $I$.
    (b) Choose an action $a = V(I,S)$ using the evaluation
        function $V$.
    (c) Carry out action $a$ in the world.
    (d) Let the immediate reward for executing $a$ in world
        state $I$ be $r$.
    (e) Update the learner's internal state $S_{\text{new}} = U(S_{\text{old}}, I, a, r)$
        using the update function $U$.
```

Fig. 4. General algorithm for reinforcement learning.

choosing actions. Different reinforcement algorithms vary in one or more of the parameters S, V, and U.

A classic example of reinforcement learning is the BOXES algorithm for the pole balancing task [15]. Many refinements have since been made to this algorithm [2,24], but it serves nicely to illustrate the key ideas underlying reinforcement learning. Briefly, the problem is to balance a pole using a cart that runs on a rail (with one degree of freedom). There are two possible actions—the cart can be moved left or right exerting equal but opposite forces on the pole. The world state I is composed of four variables: the displacement of the cart from some mean initial position, the angular displacement of the pole from the vertical position, and the time derivatives of both these displacements. The reward function punishes the learner if the pole tips over, and is otherwise zero. The internal state S consists of a table representing a policy function which maps states to possible actions. Rows in the table are formed by discretizing each of the four variables in a world state I into a number of contiguous regions. For every row in the table, the learner keeps a count of the number of steps the pole remained balanced when it chose to go left or right in a state corresponding to the row. Given the current position of the pole and the cart, the evaluation function V looks up the corresponding row in the table, and chooses the action which has been most successful in the past. The update function U increments the row count of the action chosen in the previous step provided there is no negative reward in the current step.

Reinforcement learning has several nice properties. It does not require supplying the robot with a theory of its domain, as is required by explanation-based learning [9,18], which would be a substantial undertaking in any real world robotics task. Second, reinforcement learning is incremental, as opposed to some types of inductive learning, thus the robot is continually improving its performance as it learns. Third, supplying the robot with suitable reward functions turns out to be relatively straightforward, especially for the kinds of tasks that subsumption architectures have been used for.

On the other hand, reinforcement learning suffers from a number of limitations. One, it is slow to converge, requiring several thousand instances. Two, it is difficult to "prime" the learner with domain knowledge that happens to be available, except by engineering the representation of states.

4. Behavior-based robots

In this section we discuss one specific implementation of behavior-based robots, namely the subsumption architecture [3,8]. As shown in Fig. 5, a subsumption-style architecture operates by breaking the overall control system into a number of smaller, concurrent parallel processes. Each of these branches, or "behaviors" as they are often called, uses a subset of the available sensory data to independently compute a set of values for the output control parameters. A hard-wired arbitration network or priority ordering then combines the individual results into a unified command for the actuators.

The primary advantage of such a decomposition is that each of the parallel control paths only needs to solve those perception, modeling, and planning problems relevant to the particular task for which the path is designed. Typically, such special-purpose subsystems are much easier to build than their all-encompassing monolithic counterparts. A secondary advantage of the architecture is that it can be developed incrementally. As new capabilities are needed, new paths can be added to the control system with little or no modification of the original paths.

In our version of the subsumption architecture each parallel control path consists of a "module". Essentially, each module generates a separate be-

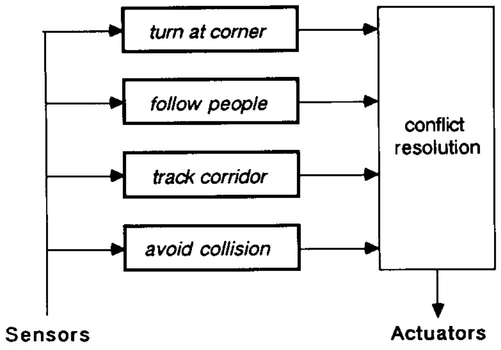

Fig. 5. Parallel paths and arbitration net in a subsumption architecture.

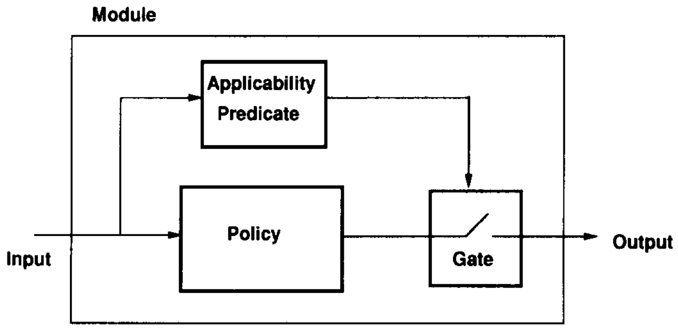

Fig. 6. A module in a subsumption architecture.

havior of the robot. A module has two internal components, as shown in Fig. 6. The policy says *what* to do given the current sensory information, whereas the applicability predicate says *when* to make such a recommendation instead of deferring to some other module. The applicability predicate essentially gates the result of the policy to the arbitration network when appropriate. Thus, there is no need for the policy to be complete and capable of making a reasonable action decision for every possible sensory configuration. It is the job of the applicability predicate to silence the policy when its advice is questionable or meaningless.

Although the mediation between different control paths could be arbitrarily complex, in practice we have found that a fixed priority scheme is sufficient. This is typically represented using "suppressor" nodes as shown in Fig. 7. The semantics of these devices is that a signal injected into the side of the node takes precedent over and replaces the signal that normally passes through the node. For instance, if module A decides that it is not applicable, any commands generated by module B will pass straight through and go directly to the robot's effectors. However, in those cases where A feels it has something to say, its commands will automatically supercede those of B. In this way A effectively grabs control of the particular actuator

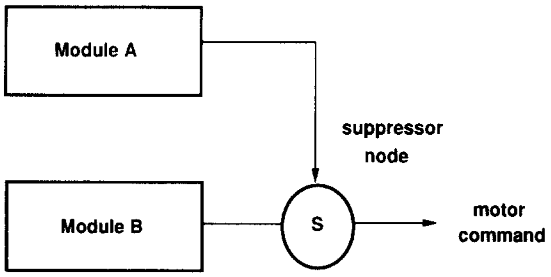

Fig. 7. A suppressor node.

resource in question.

In general, to build a subsumption-style control system we must first break the overall task into a number of subtasks. Then, for each subtask, we must devise suitable action-generation policies and applicability conditions. Finally, we must specify a priority ordering on the behaviors to allow the system to resolve any conflicts that may arise.

5. Real and simulated robot testbeds

We describe two robot testbeds, one real and one simulated, for studying the robot learning problem.

5.1. OBELIX: a real robot vehicle

OBELIX is a real mobile robot, shown in Fig. 2, which has a variety of functional sensory systems. The robot uses a 9600-baud Arlan 130 radio link to continually send the data it collects back to a workstation which in turn responds with motion commands. While there is a small amount of processing onboard, most of the learning work is done by an external Symbolics 3650 computer running Lisp. We chose this arrangement, rather than coding everything in assembly language, to avail ourselves of the rapid prototyping environment and debugging tools available on the Lisp machine.

The physical robot itself is built on a small, 12 inch diameter, 3-wheeled base from RWI, Inc. This platform is able to turn in place as well as go forward and backward. The sensors are mounted on top and always face in the direction of travel. For our experiments, we limit the motion of the vehicle to one of five different possibilities. The robot either moves forward, or turns left or right in place by two different angles (22 degrees or 45 degrees). It never stops or backs up. Although the base is capable of performing such moves (as well as much more complicated ones) we chose to simplify the learning problem by ignoring these features.

OBELIX's primary sensory suite includes both sonar and infra-red devices. In one set of experiments we use an array of eight multi-echo sonar units. These use standard Polaroid transducers and the Texas Instruments single-frequency driver boards. Each sensor in the array has a field of view of roughly 20 degrees and can see out to about 35 feet. The robot computes the range to obstacles by measuring the time between the emission of a pulse and the receipt of the return echo. Our system is capable of measuring a single echo to within about 1/4 inch or, alternatively, classifying each echo into one of a series of contiguous 1-inch range bins. This latter ability lets us extract potentially useful information from later echoes as well as the initial one.

For the purposes of the experiments described here, however, we use just two range bins. One extends from 9 to 18 inches and another covers the distance between 18 and 30 inches. Thus, any object in the "near zone" shown in Fig. 8 will activate the NEAR bit for that sensor, whereas objects in the "far zone" will cause the FAR bit to come on. Sometimes, if the nearer of two objects only partially blocks the sonar beam, both bits will come on. This is because each of the objects generates a detectable echo. Also, although the near zone stops 9 inches from the edge of the vehicle, very close objects are usually still sensed. This is because the obstacle and the robot's body form two ends of a resonant cavity. For instance, an object at 6 inches will also generate a secondary echo which looks like it came from an object located 12 inches away. This is within the near zone, so the NEAR bit will activated for the object. Higher-order echoes are possible, especially when the robot is almost touching the reflecting surface. Thus, an extremely close object may actually activate both the NEAR and the FAR bits.

Unfortunately, there are several situations in which the sonar will fail to see an obstacle in front of the robot. The most troublesome case is one in which the object has a smooth, hard surface. Such objects act like sonar mirrors and can only be detected when their surface normals are within

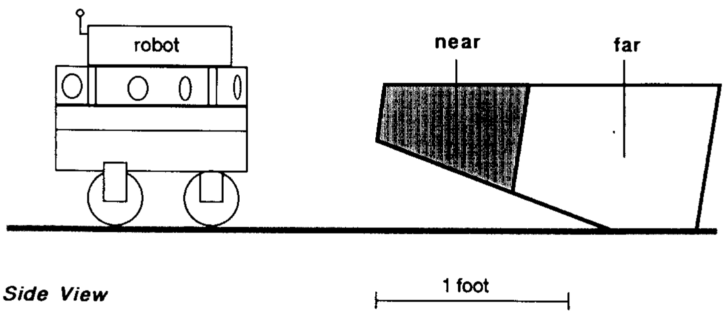

Fig. 8. Downward sonar deflection and range limits.

± 10 degrees from the centerline of the sensors. For instance, a box with a very crisp corner pointed toward the robot may not be sensed because each of the faces is at too steep an angle relative to the sonar beams. A different problem occurs for very short objects, such as the wheels of office chairs. As shown in Fig. 8, the sonar beams are angled downward about 10 degrees to enhance the detection of objects on the floor. However, it is still possible for a low object to go unnoticed if it is close enough to the robot to get under the sonar beam. Of course, there is a similar problem in detecting overhanging obstacles such as table tops or the seats of chairs.

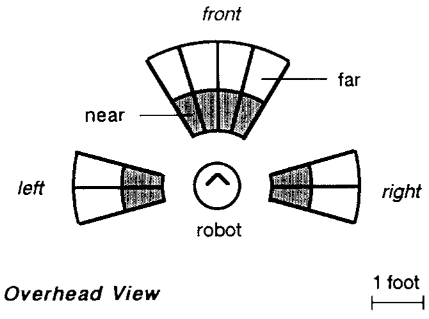

Fig. 9. Top view of sonar showing detection zones.

The individual sonar units are arranged in an orthogonal pattern as shown in Fig. 9. There are four sonars looking toward the front and two looking toward each side. The centerlines of adjacent sonars are offset by 15 degrees to allow some overlap of the detection zones. This configuration also improves the probability of detecting large flat surfaces in the robot's direction of travel. Since each of the front sonars will respond to a different range of surface normals, the robot can detect walls which are angled up to ± 30 degrees from its direction of travel (an important survival skill). Unfortunately, the robot is prone to grazing collisions since there is a big blind spot at about 45 degrees to each side of the front. Any objects in this area will escape detection. This can become an important consideration with regard to developing the concept of object permanence. If the robot sees something in front of it, turning a small amount may cause the object to totally disappear from view!

Finally, there are two secondary sources of sensory information that have higher semantic content than the general-purpose sonar array. There is an extra infra-red (IR) detector which faces straight forward and is tuned to a response distance of 4 inches. This sensor supplies the BUMP bit used by the learning algorithms since it only comes on when something is right against the front of the robot. The other source of auxiliary information available is the motor current being used for forward motion. If this quantity exceeds some fixed threshold, the STUCK bit is turned on. To prevent damage to the robot, an onboard emergency routine automatically stops the robot and backs it up slightly when this happens. Note, however, that even if the robot's current path is blocked and the robot is wedged up against an obstacle, the STUCK bit will not come on unless the robot is commanded to move forward.

Thus, to summarize, eighteen bits of information are extracted from the sensors on the robot. Sixteen bits of information come from the eight sonar sensors (one bit from the NEAR range and one bit from the FAR range). There is also one bit of BUMP information, and one bit of STUCK information. The eighteen 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 five actions to take in each case.

5.2. A simulated robot testbed

Carrying out exhaustive trials on real robots is often time-consuming. While the robot can usually take any number of steps unattended, a human is required each time the environment is reset to its initial state. For this purpose a simulator is more useful. While a successful run on a simulator does not necessarily imply that an algorithm will be successful on a real robot, nevertheless a simulator offers a fairly rapid prototyping environment.

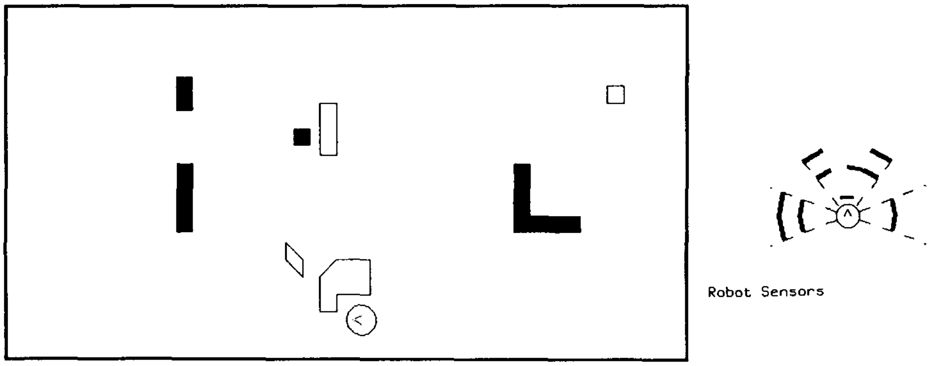

Fig. 10. A simulated robot environment.

Figure 10 illustrates a simulated robot environment that we have used in developing and testing our learning algorithms. The simulator depicts a room environment modeled after the actual environment in which OBELIX moves about. The robot is represented by the circular figure; the "nose" indicates the orientation of the robot. Dark shaded objects represent obstacles. Outline figures represent boxes.

The robot uses simulated "sonar" sensors that are patterned after the real sonar sensors on OBELIX. These indicate the presence or absence of objects at near and far distances from the robot. Figure 10 shows the sonar pattern seen by the robot at the location depicted in the simulator. The BUMP bit is indicated by a short horizontal bar placed directly over the circle. The robot can move about the simulator environment by taking actions such as going forward or turning left and right by varying degrees.

The simulator is a simplification of the real world situation in several important respects.

(1) Boxes in the simulator can only move translationally. Thus a box will move without rotation even if a robot pushes the box with a force which is not aligned with the axis of symmetry.
(2) The sonars on OBELIX are prone to various types of noise, whereas the sensors on a simulator robot are "clean".
(3) Actions are deterministic (in the sense that taking the same action twice from the same state will produce identical results) on the simulator, unlike on the real robot.

Even though the simulator is a fairly inaccurate model of reality, it has still proved to be a useful tool in our investigations.

One interesting scheme that could speed up learning is to use the learned results from the simulator on the real robot. Since our simulator is a fairly distorted model of reality, we have obtained disappointing results using this approach. The real robot simply unlearns all the simulator-learned material. However, Lin [13] appears to have obtained better results, perhaps because he used a more sophisticated simulator. Writing a good simulator is quite a difficult task; in the limit, consider using computer graphics techniques to synthesize video inputs. We believe that for many tasks it is more expeditious to use a robot operating in the real world, since the real world is its own best model.

6. The box pushing task

To explore ways in which robots can learn, we have to pick a sample task that is rich enough that it may potentially lead to practical use, yet simple enough to formulate and study. One such task is 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 moving packing cartons around from one location to another.[1] Mitchell [17] describes a similar task involving sliding a tile across a table using a finger contact, although he advocates a more knowledge-intensive approach to learning the task than the one pursued in this paper.

[1]Except that no one would ever use a *round* robot to push *square* boxes!

Conceptually, the box pushing task involves three subtasks.

(1) The robot needs to find potential boxes and discriminate them from walls and other obstacles, which is quite difficult to do using sonar or infra-red detectors.
(2) The robot needs to be able to push a box across a room, which is tricky because boxes tend to slide and rotate unpredictably.
(3) The robot 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 11 illustrates the overall structure of a behavior-based robot that follows this decomposition of the task, and also depicts a priority ordering on the three subtasks of the box pushing task. We discuss the need for this information next.

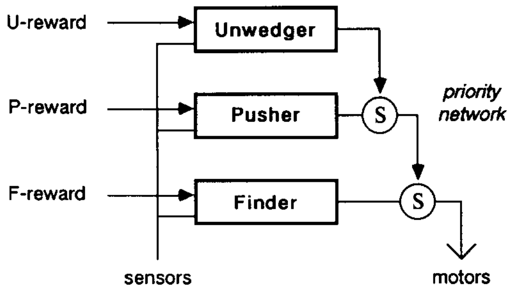

Fig. 11. Modules in a box pushing robot.

6.1. *Priority ordering*

Our intention is to have OBELIX learn the above three tasks in parallel, using three copies of the learning algorithm. In order to achieve this, OBELIX must be able to detect which of the three behaviors is active at any given time. The ordering on the three behaviors enables OBELIX to determine the dominant behavior in any situation.

The ordering condition used in OBELIX to coordinate the three behaviors—box finding, box pushing, and unwedging—is as follows. The unwedging behavior is highest in the priority ordering. If the robot is in a stalled situation, it cannot do anything else. Thus, the foremost task is to get the robot out of a stalled situation. The box pushing behavior is next in priority. If the robot has started to push a box, then clearly it should continue to do so as long as it can. Lastly, the box finding task is lowest in the priority ordering. If the robot is not stalled, and it is not pushing a box, then it should look for a box to push.

Another interesting way to obtain this priority ordering on the three behaviors is by examining the temporal sequence in which they will become applicable. First, the robot needs to find a box to push one, thus box finding is lowest in the priority ordering. Second, the robot needs to continue to push a box once it has found one, thus box pushing is second in the priority ordering. Finally, once the robot has pushed a box to a wall or against another obstacle, it needs to extricate itself from the stalled situation, thus unwedging is the third and highest in the priority ordering.

6.2. 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 on the real robot. For the simulator, boxes are convex or concave polygons of varying sizes.

Distinguishing boxes from obstacles using sonar or infra-red detectors is difficult since both generate remarkably similar patterns from a distance. One reasonable but far from perfect strategy is to cruise around the room looking for "narrow" objects (that is, objects that only turn on the front sonars on the robot), and try to push them. This strategy will often cause OBELIX to push various types of obstacles including walls, which in turn will cause the unwedging behavior to become active.

To encourage the robot to follow the above strategy, the box finding behavior should be rewarded whenever the sensor bits corresponding to the near ranges on the front sonars turn on. This happens whenever the robot is close to and facing a "potential" box. The reward encourages the box finder to go toward objects and try to push them. The exact reward function for the box finding task is described in Fig. 12. The reward function takes as arguments the previous state (*old_state*), the current state (*new_state*), and the action (*action*) that led from the former to the latter. The predicate FRONT_NEAR_SONAR_BITS is the disjunction of the state bits representing the near ranges of the central front facing sonars on the robot.

Occasionally, the robot may turn on the front near sonar bits by making a turn. In such cases, although it will receive no immediate reward, a further action of going forward will generate a positive reward, which is then propagated backwards to reinforce the turning action.

In our experiments we have used several types of reward functions. The ones described in the paper were selected because they gave the best results

```
Finder_reward(old_state, action, new_state):
begin
  IF action = FORWARD                        % went forward
     and FRONT_NEAR_SONAR_BITS(new_state)
                                             % facing object
  THEN return 3                              % reward robot
  ELSE IF ¬FRONT_NEAR_SONAR_BITS(new_state)
  THEN return −1                             % punish robot
  ELSE return 0                              % default is no reward
end
```

Fig. 12. Reward function for box finder.

overall. However, the actual numbers used in the various terms do not substantially alter the experimental results. What matters most is their qualitative value: whether they are positive or negative, and their rough relative magnitudes. Still, even with just two terms in the reward function, it is easy for the learner to get stuck in local minima. For example, if there is something close by and directly in front of the robot, it can turn slightly back and forth endlessly without incurring the -1 penalty. However, to discover that it can also get a $+3$ bonus, the robot must serendipitously move forward at some point. Zero rewards do not affect the behavior of the system.

Finally, we specify the applicability conditions for the box finding behavior. This turns out to be trivial since the box finding behavior is always applicable. However, it is the least preferable behavior among the three behaviors for the overall task. This ensures that it gets control only when the other two behaviors are not applicable.

6.3. 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 reward function supplied to OBELIX for the box pushing task is shown in Fig. 13. The robot gets rewarded whenever it continues to be bumped and going forward. It gets punished whenever it loses contact with the box.

Here the relative magnitudes of the reward values are important in preventing undesirable cyclic behavior. For instance, suppose the penalty term was -1 instead of -3. Now imagine the robot executing the following sequence of actions: push the box, turn to the left losing sight of the box, turn back to the right. The average reward over this whole cycle would be $(1 - 1 + 0)/3 = 0$ which is nonnegative and better than the state in which the robot is never pushing the box (-1). Thus, the learner would be predisposed to come up with this behavior. However, since reinforcement learning is so prone to local minima, we want to bias the learning more strongly toward the optimal solution. By setting the penalty term to -3 the average reward for the sequence described becomes $-2/3$. Since this cycle of actions generates an overall negative reward it will eventually be expunged.

```
Pusher_reward(old_state, action, new_state):
begin
  IF action = FORWARD           % went forward
     and BUMP(new_state)        % still bumped
  THEN return 1                 % reward robot
  ELSE IF ¬BUMP(new_state)      % lost the box
  THEN return −3                % punish robot
  ELSE return 0                 % default is no reward
end
```

Fig. 13. Reward function for box pusher.

The applicability condition used in OBELIX for the box pushing behavior is given in Fig. 14. 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 applicability conditions 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. This "potentiation" of a behavior is commonly used in robotic systems and is sometimes referred to as a *monostable* condition [4,8]. For example, the box pushing behavior continues to be applicable five time units after OBELIX has lost contact with a box. This value was chosen to allow the robot some time to try to recover and push the box once again, but does not let it flail indefinitely.

6.4. 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

```
Pusher_applicable(old_state, action, new_state):
begin
  IF BUMP(new_state)         % bumped now
  THEN
    begin
      *pusher_history* := 5  % set global variable
      return true
    end
  ELSE IF *pusher_history* > 0
  THEN
    begin
      *pusher_history* := *pusher_history* - 1
      return true            % one step less to recover
    end
  ELSE return false
end
```

Fig. 14. Applicability function for box pusher.

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 because typically no *single* action will unwedge the robot. Hence this task requires performing temporal credit assignment over multiple actions.

The reward function for the unwedging task is given in Fig. 15. The unwedging behavior is rewarded when the robot is no longer stalled, and is able to go forward once again. It is punished if the robot continues to be stalled.

Finally, the applicability conditions on the unwedging behavior is shown in Fig. 16. As in the case of the box pushing behavior, the unwedging behavior continues to be applicable for five time steps after the robot is no longer stalled.

6.5. Summary

We can visualize the control flow among modules in a box pushing robot as shown in Fig. 17. The robot always starts in the finder module (labeled **F** in the figure). If it is bumped, the pusher module (labeled **P**) turns on. The pusher module continues to have control of the robot as long as the robot continues to be bumped. If it is stuck, control transfers to the unwedger module (labeled **U**). The timeout refers to the number of steps (five in

```
Unwedger_reward(old_state, action, new_state):
begin
  IF STUCK(new_state)              % stalled now
  THEN return -3                   % punish robot
  ELSE IF action = FORWARD         % went forward
       and ¬STUCK(new_state)       % not stalled now
  THEN return 1                    % reward robot
  ELSE return 0                    % default is no reward
end
```

Fig. 15. Reward function for unwedger.

```
Unwedger_applicable(old_state, action, new_state):
begin
  IF STUCK(new_state)  % stuck now
  THEN
    begin
      *unwedger_history* := 5
                       % set global variable
      return true
    end
  ELSE IF *unwedger_history* > 0
  THEN
    begin
      *unwedger_history* := *unwedger_history* - 1
      return true      % one step less to recover
    end
  ELSE return false
end
```

Fig. 16. Applicability function for unwedger.

our case) that a module continues to be applicable after its initial triggering condition is no longer true.

Figure 17 intentionally bears a resemblance to a finite state machine. Our multi-part architecture provides a solution to a fundamental problem in reactive controllers known as *perceptual aliasing* [32]. This problem occurs when the same state vector calls for different actions depending on the context. For instance, if the BUMP bit is on, this indicates a good state for the pusher module to try going forward. However, this same situation will cause the unwedger module to try to turn instead. By having separate reactive controllers, the robot will not be constantly vacillating between

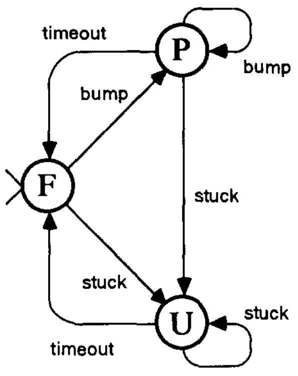

Fig. 17. Representation of control flow among modules.

these two actions. The applicability conditions essentially remember enough contextual information to differentiate the two scenarios.

To summarize, we provide the robot with a "sketch" of a plan for pushing a box, and leave it to flesh out the details. Lastly, Fig. 18 details the information supplied, and the objective of the learning system.

- **Given:**
  - A state representation composed of sonar occupancy information, an infra-red "bump" sensor, and a "stuck" sensor.
  - Five possible actions—Go forward, turn left, turn hard left, turn right, turn hard right.
  - Three task modules, one for finding boxes, one for pushing them, and one for unwedging the robot from stalled states.
  - Applicability conditions and a reward function on each module.
  - A priority ordering specifying that the unwedging module has priority over the box pushing module, which in turn dominates the box finding module.
- **Learn:**

  For each module, a policy which maximizes the cumulative expected reinforcement from the module reward function.

Fig. 18. The problem of learning component behaviors for pushing boxes.

7. Learning algorithms

This section describes two learning algorithms that we have implemented on OBELIX as well as on the simulator. The two algorithms use the same technique for temporal credit assignment, but differ in their approach to the structural credit assignment problem.

7.1. Algorithm 1: Q learning with weighted Hamming

The first algorithm combines a well-known learning algorithm for temporal credit assignment, Q learning [30], with a simple structural credit assignment technique based on Hamming distance. We describe each of these techniques below.

7.1.1. Q learning

The key idea behind Q learning [30] is the use of a single data structure—a utility function $Q(x,a)$ across states (x) and actions (a)—for evaluating actions and states. The utility of doing an action a in a state x is defined as the expected value of the sum of the immediate payoff or reward r plus the utility of the state resulting from the action *discounted* by a parameter γ between 0 and 1 [28]. That is,

$$Q(x,a) = E(r + \gamma e(y) \mid x,a),$$

where E denotes an expected value conditionalized upon being in state x and performing action a. The utility of a state, in turn, is defined as the maximum of the utilities of doing different actions in that state. That is,

$$e(x) = \text{maximum } Q(x,a) \text{ over all actions } a.$$

During learning, the above equality is not true because the stored utility values have not yet converged to their final value. Thus, the difference between the left-hand side and right-hand side of the above equality 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. Setting β to 1 results in *one-shot* learning.

Q learning can be performed over sequences of actions by storing the last k ⟨*state, action, reward*⟩ triples. After updating the Q value of a state using the above update rule, the Q value of the state before it can be updated, and so on in reverse chronological order. Multi-step Q learning does not change what can be learned, nor does it affect the steady-state Q values. All it does is speed up the back-propagation of reward to temporally remote steps. This form of multi-step Q learning is different from others described in the literature [13,30] in that there is no recency parameter λ as in the $TD(\lambda)$ methods [27].

For simplicity, we describe our algorithms as using single-step Q learning. However, the experimental data presented in Section 8 was collected using 5-step Q learning. Using a depth anywhere between 3 and 10 steps seems to give similar results.

7.1.2. *Weighted Hamming distance*

The robot starts with 18 bits of sensory information. This yields a space with a quarter of a million states. Since we want the robot to learn something useful in a few thousand steps, obviously we have no hope of experiencing all states. Thus we must make some sort of generalization that says for a number of "similar" states, taking a particular action yields a "similar" result. This notion of similarity can either be handcoded, as in Algorithm 1, or derived from statistical properties of the data, as in Algorithm 2.

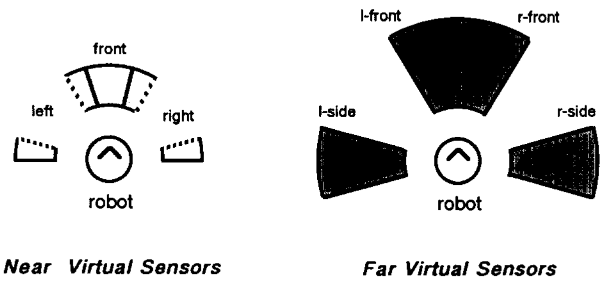

Fig. 19. Virtual sonar derived from actual sonar.

One way to solve this structural credit assignment problem is to change representations so that a small change in the syntax of a state vector causes only a small change in the associated semantics. For this purpose, we have defined a number of more meaningful "virtual" sonar sensors whose values are computed from combinations of the actual readings. Choosing an appropriate reduced representation (as well as the original 18 bits) requires care and at least a partial understanding of how the robot interacts with the environment while carrying out its task. Figure 19 illustrates the virtual sonar representation. There are 3 binary sensors covering space near the robot's body, and 4 other binary sensors which see farther out. These are shown in the figure. Each new sensor is just the logical OR of 2 to 4 of the original sensor bits. We use the resulting 7 bits, plus BUMP and STUCK, to define a new 9 bit input vector to the learning algorithm. Note that the virtual sonar is used only in Algorithm 1 below (which uses the weighted Hamming metric). Algorithm 2, which employs statistical clustering, uses the complete 18-bit state description.

Now to decide if two states are "similar", we see what the Hamming distance between them is. That is, we count how many of the bits are different. Yet not all bits are equally important. BUMP and STUCK are very useful so they are given a weight of 5. The nearby virtual sonars are fairly useful so they are given a weight of 2, whereas the far virtual sonars are less so and thus only carry a weight of 1. To compute the weighted Hamming distance between two reduced signatures, we simply add up all the weights of the bits positions in which they differ. Empirically, setting the threshold at 1 or 2 produces good results, but increasing it to 3 or 4 significantly degrades performance. In the experiments described later we use a maximum distance of 2 or less. This means two of the far virtual sensors can be different, or one of the near virtual sensors can be different. We never generalize between states in which BUMP or STUCK differ.

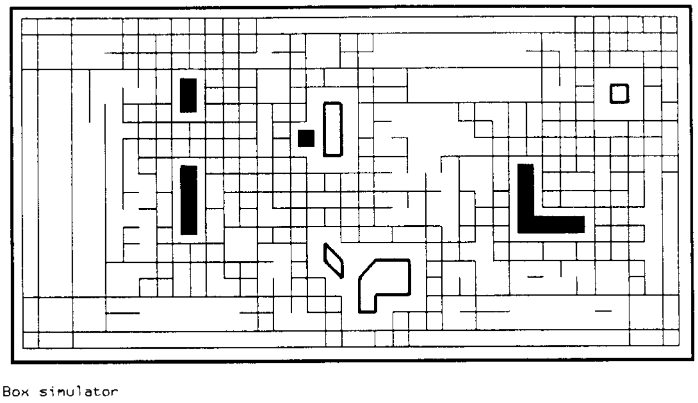

Fig. 20. Signature neighborhoods on the simulator.

To get a feel for the types of areas considered the same, examine the picture shown in Fig. 20. This is taken from our robot simulator. The dark objects are immovable walls, while the heavily outlined objects are boxes. Over this we have plotted boundaries in signature space. If the robot crosses one of the marked boundaries, then for some orientation of the robot the signature changes by a distance of at least 2. Notice that the boundaries are not necessarily closed. This is because our weighted Hamming distance does not generate a total ordering on all possible signatures. If A is 2 away from C but only 1 away from B, it may be that B is only 1 away from C. Thus it would be possible to get from A to C without crossing any borders if one went through B as an intermediate step. The main importance of the picture, however, is to show that the robot makes many discriminations near obstacles, but far fewer in large open regions. This in turn is important because our system is set up to learn "reactive" control programs. This means we associate one and only one correct action with each distinct state.

7.1.3. Summary of Algorithm 1

The complete description of Algorithm 1 is given in Fig. 21. It can be viewed as one possible instantiation of the generic reinforcement algorithm of Fig. 4.

The description of Algorithm 1 in Fig. 21 (as well as Algorithm 2 in Fig. 23) is simplified in one respect. The action selection step (Step 2(b)) does not always choose the action that results in the maximum utility value; such a policy will never explore new actions. Instead, some percentage of the time, a random action is chosen to ensure that all states in the state space will eventually be explored. This is a necessary condition for Q learning to converge [30]. We have experimentally found 10% to be a good compromise between exploratory and goal-directed activity. Usually, once the robot has learned a good control strategy, it can easily compensate for occasional deviations from its strategy caused by random actions.

```
1. Initialize the stored utility values $Q(x,a)$ for all $x$ and $a$ to 0.
2. Do Forever:
    (a) Observe the current world state $s$.
    (b) Choose an action $a$ that maximizes $Q(s,a)$.
    (c) Carry out action $a$ in the world. Let the new state be $t$
    (d) Let the immediate reward for executing $a$ in world
        state $s$ be $r$.
    (e) Update the stored utility values using the update rule:
        $Q(s',a) \leftarrow Q(s',a) + \beta(r + \gamma e(t) - Q(s',a))$
        (where $e(t)$ is the maximum $Q(t,a)$ over all actions $a$)
        over all states $s'$ such that the weighted Hamming distance
        between $s$ and $s'$ is $\leqslant k$.
```

Fig. 21. Algorithm 1: Q learning with weighted Hamming.

7.2. Algorithm 2: Q learning with statistical clustering

One problem with the weighted Hamming distance algorithm described above is that all possible states have to be stored. While this is reasonable for 9-bit state representations, it becomes more expensive for the full 18-bit state representation. We now describe a second algorithm that overcomes this problem by not storing states explicitly in a table, but instead storing clusters of states. The second algorithm still uses Q learning for temporal credit assignment, but is distinguished from the first by its use of statistical clustering for structural credit assignment.

Another distinction between the two algorithms is that the similarity metric in Algorithm 1 is handcoded, whereas the one used in Algorithm 2 is learned. Since our intention in developing Algorithm 2 was to see whether an adequate similarity metric could be learned, we did not want to implement it with the reduced 9-bit virtual sonar representation, but instead use it with the complete 18-bit state description. Thus, Algorithm 1 can be viewed as a control experiment involving only the temporal credit assignment problem.

In Algorithm 2, the robot learns a set of clusters for every action. Each cluster has associated with it a Q value. For example, the action "forward" may have a cluster with a positive Q value, and a cluster with a negative Q value. The positive Q value cluster represents a class of states where going forward should generate a positive reward. Similarly, the negative Q value cluster represents a class of states where going forward should generate a negative reward.

Clusters are extracted from the instances of states, actions, and rewards that are generated by the robot as it explores the task environment. Each time a new instance is encountered, it is matched against the existing clusters for the particular action. If it considered similar to one of them, it is incorporated into that particular cluster. Otherwise, a new cluster is created based on just this particular instance.

To select the best action to perform from a given state, the following procedure is carried out. First, the state is matched against all the clusters associated with each action, and the utility of doing the action from the state is computed as a function of the match probabilities and the cluster Q values. Then, the action that promises the best eventual reward is selected as the one to perform from the state.

7.2.1. Clusters

A cluster represents an aggregate of a set of states that are considered "similar" (the similarity metric is described below). More formally, a cluster c is an $(n+2)$-tuple

$$\langle (z_1, o_1), \ldots, (z_n, o_n), Q_c, m_c \rangle,$$

where z_i and o_i are decayed counts[2] of the number of times the ith bit of a state s matching cluster c was a 0 or 1, respectively; n is the number of bits in a state; Q_c represents the Q value of the cluster; and m_c represents the number of instances that have matched the cluster so far.

[2]When collecting statistics, it is useful to have the counts decayed so that past experiences have less impact than recent experiences. We describe the particular decaying scheme in Section 7.2.3.

Given the above representation, we can define the conditional probability of the ith state bit of a state s being a 1 given that s matches cluster c, denoted by $\hat{p}(s_i = 1 \mid s \in c)$, as

$$\hat{p}(s_i = 1 \mid s \in c) = \frac{o_i}{o_i + z_i}.$$

We use the $\hat{p}$ to emphasize that these are not true probabilities (since our counts are decayed counts, not true counts), but *probability estimates.* Note that clusters can represent "don't care" values as prob = 0.5. In fact, they can actually make much finer discriminations than simply 0, 1, or "don't care" since they use a real number between 0 and 1.

A cluster is always associated with a particular action; thus, the action "forward" has a different set of clusters than the action "left". The Q value associated with a cluster represents the predicted utility of performing the action from a state that matches the cluster.

An example of a cluster is given in Fig. 22. This cluster was learned by the box finder module, and is associated with the action "forward". It has a Q value of 2.17, and 25 instances have so far matched the cluster. The figure represents the probability information in a cluster graphically—darker regions correspond to higher probabilities and lighter regions correspond to lower probabilities. (The probability of a bit is shown at the corresponding sensor location on the robot.) Intuitively, the cluster can be interpreted as saying that if the probability of the BUMP bit is high (the dark horizontal bar just above the circle), and the probability of the front near sonar bits is high (the dark circular arc above and to the right of the BUMP bar), then going forward will generate a high reward. This makes sense because the box finder is rewarded for going forward when its front near sonar bits are turned on.

```
FINDER    FORWARD    2.1714284   25
```

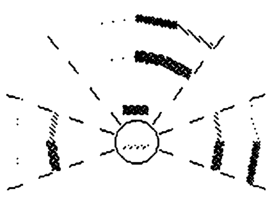

Fig. 22. Cluster associated with FINDER module and action "forward".

Algorithm 2 generates a qualitatively different type of similarity than Algorithm 1. While it can generate descriptions similar to those given by the weighted Hamming distance, it has a harder time creating the disjunctions used in the 9-bit reduced signatures (shown in Fig. 19). However, Algorithm 2 does not generalize solely on the basis of syntactic similarity. It also takes into account the utility of various actions with respect to the given task.

7.2.2. Matching states against clusters

Deciding if a state s is an instance of a cluster c requires computing the conditional probability of s being an instance of c given the bitwise description of s. Formally, we can denote this quantity as

$$\hat{p}(s \in c \mid s_1 = v_1, s_2 = v_2, \ldots, s_n = v_n),$$

where s_i denotes the ith bit of s, and v_i is either a 0 or a 1. Using Bayes' rule, we obtain the following equation

$$\hat{p}(s \in c \mid s_1 = v_1, \ldots, s_n = v_n) = \frac{\hat{p}(s_1 = v_1, \ldots, s_n = v_n \mid s \in c)\hat{p}(s \in c)}{\hat{p}(s_1 = v_1, \ldots, s_n = v_n)}.$$

Now, assuming the bits comprising a state are independent (we discuss this assumption below), we can rewrite the denominator of the above equation as

$$\hat{p}(s_1 = v_1, \ldots, s_n = v_n) = \prod_{i=1}^{n} \hat{p}(s_i = v_i).$$

Using the following identity from probability theory,

$$p(A, B \mid S) = p(A \mid S)p(B \mid S) \Leftarrow p(A, B) = p(A)p(B),$$

and the bitwise independence assumption, we can rewrite the first quantity in the numerator of the above equation as

$$\hat{p}(s_1 = v_1, \ldots, s_n = v_n \mid s \in c) = \prod_{i=1}^{n} \hat{p}(s_i = v_i \mid s \in c).$$

Now, we can compute the conditional probability of a state bit being a 1 or a 0 given the state matches a cluster directly from the probability values stored in a cluster. That is, $\hat{p}(s_i = 1 \mid s \in c)$ is directly available from each cluster, and

$$\hat{p}(s_i = 0 \mid s \in c) = 1 - \hat{p}(s_i = 1 \mid s \in c).$$

The other two probabilities—$\hat{p}(s_i = v_i)$, the probability of the ith state bit having a value v_i, and $\hat{p}(s \in c)$, the probability of any state s matching a cluster c—are estimated by collecting statistics on the sensors and keeping a count of the number of instances that have matched cluster c, respectively.

Given $\hat{p}(s \in c \mid s_1 = v_1, \ldots, s_n = v_n)$, two criteria are used to decide whether s is an instance of c. The first condition is that

$$\hat{p}(s \in c \mid s_1 = v_1, \ldots, s_n = v_n) > \varepsilon,$$

where ε is a fixed parameter input to the clustering algorithm. The value of the parameter ε directly influences the types of clusters formed by the algorithm. If it is high, then the clusters that will be formed will be similar to one another since states can be nearly identical yet belong to different clusters. A value close to 0 will cause very dissimilar clusters since most of the time states will tend to match existing clusters.

A second condition for a state to match a cluster is based on the Q values of the state and the cluster.[3] Intuitively, it would seem that we want to distinguish clusters at least by the sign of the Q value. Otherwise, we might merge states where doing an action gives a high reward with states where it does not. In practice, we have found that making finer distinctions between different Q values generates better performance. More precisely, if the Q values of a cluster and a state are both positive or both negative, and their absolute difference is less than a fixed parameter δ, then the state is considered an instance of the cluster.

Finally, the assumption that the bits comprising a state are statistically independent is not true for real sonar data. The estimates it generates tend to be lower than the actual probabilities would be. Typically $\hat{p}(s_i = 1 \mid s_j = 1) \geqslant \hat{p}(s_i = 1)$ for any two sensor bits s_i and s_j because objects in the world are usually extended in space.

7.2.3. Merging states with clusters

In this section we describe how a state that matches an existing cluster (using the above state cluster similarity metric) is used to update the cluster. Let

$$c = \langle (z_1, o_1), \ldots, (z_n, o_n), Q_c, m_c \rangle$$

be a cluster, and let s be a state matching cluster c. Denoting the updated cluster by

$$c' = \langle (z'_1, o'_1), \ldots, (z'_n, o'_n), Q'_c, m'_c \rangle,$$

we have for each state bit $s_i = 1$:

$$z'_i \leftarrow \mu z_i, \qquad o'_i \leftarrow 1 + \mu o_i,$$

[3]Although Algorithm 2 does not explicitly store Q values for states, they can be computed from the Q values of the clusters as we show in Section 7.2.6.

while for each state bit $s_i = 0$ we have:

$$z'_i \leftarrow 1 + \mu z_i, \qquad o'_i \leftarrow \mu o_i.$$

Here μ, a real number between 0 and 1, is used to implement the decayed count scheme we alluded to earlier. The value for μ is determined by how much we want to devalue past experiences over recent experiences. If we set μ to 1, then past experiences are not decayed, and the above rules will keep exact counts. Typically, we set $\mu = k/(k+1)$, where k is a fixed integer (such as 10). This particular decay scheme is based on the one described by Maes and Brooks [14].

The Q value associated with the updated cluster c' is the weighted sum of the Q values associated with cluster c, Q_c, and with state s and last action performed a, $Q(s,a)$. The number of instances associated with the state (assumed to be 1) and the cluster (denoted by m_c) is used to weight the final Q value.

$$Q_{c'} \leftarrow Q_c \left(\frac{m_c}{m_c + 1}\right) + Q(s,a) \left(\frac{1}{m_c + 1}\right).$$

Finally, the number of instances of c' is simply

$$m_{c'} \leftarrow m_c + 1.$$

7.2.4. Forming new clusters

If a state s does not match any existing clusters, then a new cluster c_s is formed as follows. First an empty cluster

$$c = \langle (z_1, o_1), \ldots, (z_n, o_n), Q_c, m_c \rangle$$

is created such that

$$z_i = o_i = \tfrac{1}{2}(k+1),$$
$$Q_c = 0,$$
$$m_c = 0.$$

Here k is the fixed integer used in defining the decay parameter μ (see above). The cluster c_s is now formed by simply merging state s with the empty cluster c as described in Section 7.2.3. Initializing a cluster's state bit probabilities in this manner has the effect of relaxing the state description, while taking account of the particular bit values of the state.

7.2.5. *Merging existing clusters*

Often two existing clusters are considered "close" enough that they can be merged to form a new cluster. This requires a "distance" metric on clusters.

$$distance(c1,c2) = \sqrt{\sum_{i=1}^{n}(\hat{p}(s_i = 1 \mid s \in c1) - \hat{p}(s_i = 1 \mid s \in c2))^2}.$$

The metric computes the Euclidean distance between two n-dimensional vectors represented by clusters $c1$ and $c2$, where the elements of each vector are real numbers between 0 and 1 representing the conditional probabilities of each state bit being a 1 given a state matching the cluster. Two clusters $c1$ and $c2$ are merged into a new cluster if and only if $distance(c1,c2) < \rho$, where ρ is a parameter to the clustering algorithm, and if the Q values of the two clusters agree in sign and their absolute difference is less than δ.

Given two clusters

$$\begin{aligned} a &= \langle (z_{a_1}, o_{a_1}), \ldots, (z_{a_n}, o_{a_n}), Q_a, m_a \rangle, \\ b &= \langle (z_{b_1}, o_{b_1}), \ldots, (z_{b_n}, o_{b_n}), Q_b, m_b \rangle, \end{aligned}$$

if the two clusters are close enough to be merged into a new cluster

$$c = \langle (z_1, o_1), \ldots, (z_n, o_n), Q_c, m_c \rangle,$$

the elements of c are formed as follows. For each state bit i we have

$$\begin{aligned} z_i &\leftarrow z_{a_i}\left(\frac{m_a}{m_a + m_b}\right) + z_{b_i}\left(\frac{m_b}{m_a + m_b}\right), \\ o_i &\leftarrow o_{a_i}\left(\frac{m_a}{m_a + m_b}\right) + o_{b_i}\left(\frac{m_a}{m_a + m_b}\right). \end{aligned}$$

The number of instances that make up each cluster is used to weight the counts. This weighting scheme tilts the final counts towards the original counts associated with the larger of the two clusters over those associated with the smaller cluster.

Similarly, the Q value associated with the merged cluster c is the weighted sum of the Q values associated with clusters a and b.

$$Q_c \leftarrow Q_a\left(\frac{m_a}{m_a + m_b}\right) + Q_b\left(\frac{m_b}{m_a + m_b}\right).$$

Finally, the number of instances of cluster c is the sum of that for clusters a and b.

$$m_c \leftarrow m_a + m_b.$$

7.2.6. *Action selection using clusters*

Now we turn to the problem of choosing an action from the stored state clusters. There are several possible action selection mechanisms. For example, the robot can decide to pick a cluster that most closely matches the current situation, and carry out the stored action for that cluster. The problem with such a scheme is that the closest matching cluster may not be the one with the highest Q value. Alternatively, the robot can pick the action which contains the cluster with the highest Q value. However, the highest Q value cluster may match the current state very poorly.

Instead we pick the action that maximizes some combination of utility and the degree of match. More precisely, the action selection strategy used by the robot is to pick the action a that maximizes the quantity

$$Q(s,a) = \frac{\sum_{c\in C_a} Q_c \, \hat{p}(s \in c \mid s_1 = v_1,\ldots,s_n = v_n)}{\sum_{c\in C_a} \hat{p}(s \in c \mid s_1 = v_1,\ldots,s_n = v_n)}.$$

The numerator in the above expression is the sum of the Q values of the clusters stored under an action (denoted by C_a), weighted by the probability of match between the state s and each cluster. The denominator, which is a normalization factor, is the sum of the match probabilities of the state s over the clusters associated with action a.

As an aside, note that Algorithm 2 computes the $Q(s,a)$ values of doing an action a in a state s from the Q values of clusters and the match probabilities. In contrast, Algorithm 1 looks up the utility values from an array containing the Q values for all possible states and actions. Thus, Algorithm 2 scales much better than Algorithm 1 with the number of sensor bits and actions. In particular, for the case when the number of bits $n = 18$, there are far fewer than $2^{18} \times 5$ clusters (typically less than 100).

7.2.7. *Summary of Algorithm 2*

Figure 23 highlights the main steps of Algorithm 2. Once again, Step 2(b) actually takes a random action 10% of the time as a tradeoff between exploration and policy-directed behavior. The algorithm uses Q learning to propagate rewards temporally across actions. It uses clustering to propagate rewards across states. By setting the parameters ε, δ, and ρ to different values, the algorithm can be made to cluster solely on states, solely on Q values, or on some combination of both. The exact values of these parameters that were used in our experiments were $\varepsilon = 10^{-6}$, $\delta = 0.45$, and $\rho = 2.0$. These values were obtained essentially by trying out a variety of different values, and selecting the ones that gave the best performance. However, the overall performance of the clustering algorithm does not seem to be sensitive to small percentage changes (such as 10%) in the values of the parameters δ and ρ, and ε can be increased to 10^{-3} with similar results.

```
1. Initialize clusters under each action $a$ to NIL. Fix the clustering
   parameters $\varepsilon$, $\delta$, and $\rho$.
2. Do Forever:
    (a) Observe the current world state $s$.
    (b) Choose an action $a$ that maximizes $Q(s,a)$.
    (c) Carry out action $a$ in the world. Let the new state be $t$.
    (d) Let the immediate reward for executing $a$ in world state $s$
        be $r$.
    (e) Update the Q value of state action pair $s,a$ using the rule:
        $Q(s,a) \leftarrow Q(s,a) + \beta(r + \gamma e(t) - Q(s,a))$
        where $e(t)$ is the maximum $Q(t,a)$ over all actions $a$.
    (f) If there exists a cluster $c$ which matches state $s$ to within $\varepsilon$,
        their Q values agree in sign, and their absolute Q value
        difference is $< \delta$, then merge $s$ into $c$. Otherwise create a
        new cluster $c'$ from $s$.
    (g) Merge any existing clusters $c1$ and $c2$ under action $a$ if
        $distance(c1,c2) < \rho$, their Q values agree in sign, and their
        absolute Q value difference is $< \delta$.
```

Fig. 23. Algorithm 2: Q learning with clustering.

8. Experimental results

This section describes a detailed experimental study evaluating the performance of the two learning algorithms described above. Mainly, we are interested in answers to the following two questions.

- *Competence*: How well does the robot learn each individual behavior in the box pushing task?
- *Decomposition*: To what extent does breaking the overall task into a set of subsumption modules help the learning?

The competence question can be answered by measuring the improvement in performance of each individual behavior as a function of the learning. The decomposition 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.

Measuring performance improvement

There are obviously many ways of monitoring the improvement in performance of the robot, ranging from some overall measure of performance—such as the number of boxes successfully pushed to a corner—to local measures such as the reward obtained during each trial. The box pushing task involves a variety of capabilities, ranging from finding boxes to pushing them to getting out of stuck situations. Therefore, we choose as a performance measure the cumulative value of the reward obtained thus far divided by the number of steps taken thus far. This measure is a running estimate of the average reward value over the experiment. Clearly, several alternative performance metrics could be used, such as the speed at which the policy function is learned or the convergence rate of the task reward values. Since our primary focus is on comparing the learning algorithms with a handcoded program, not on comparing different learning algorithms with themselves, we chose the average reward metric.

Real data and simulator data
Data is presented from both the simulator and the real physical robot OBELIX. The simulator is a simplified environment, in that the sonar values are repeatable, actions are invertible, and there are no rotational effects. Comparing the data from the real robot with that from the simulator allows us to assess the effect of removing these simplifications on the learning algorithms.

Data from both algorithms
We present data on the effectiveness of both Algorithm 1 and 2 in learning the box pushing task. Some caution must be observed in comparing the performance of these algorithms, because they do not operate on the same input representations. In particular, Algorithm 1 uses an abstracted sonar state description whereas Algorithm 2 uses the full 18 bits.

Using handcoded and random agents for box pushing
On each graph below we also present data on the performance that was obtained by handcoded and random agents. The handcoded agent is a simple reactive program that describes the best action to take in any given state for each of the three separate behaviors. The random agent chooses randomly between the actions with equal probability regardless of the state. Comparing the performance of the handcoded and random agents with that of the learned programs gives us a good basis for judging the effectiveness of the learning.

Experimental methodology
We used the following procedure to collect the experimental data. The learning runs were organized into 20 trials, each trial lasting for a 100 steps. Thus, the robot takes totally 2000 steps in each learning run. On our real robot, a learning run takes about 2 hours. At the beginning of each trial, the

world is restored to its initial state. This includes resetting the boxes and the robot to their initial locations. The robot starts with a randomly chosen orientation at the beginning of each trial.

We settled on a learning run of 2000 steps for several reasons. We wanted to obtain significant learning in a relatively short period of time (hours) instead of long periods of time (days). Also, we found that on the average the learning algorithms had converged in this number of steps. In a few cases we let the learning algorithms run for periods longer than 2000 steps, but this did not lead to a significant improvement in their performance.

The simulator world was initialized as shown earlier in Fig. 10. For data on the real robot OBELIX, a special "playpen" was created, as shown in Fig. 24. The playpen was modeled after the simulator world, except there were no obstacles in the middle of the room. The playpen was a portion of the robotics laboratory at IBM Hawthorne. Large packing cartons formed three walls of the playpen and a sheet rock wall formed the fourth. Experimental conditions were tightly controlled by using pieces of tape on the floor to demarcate the initial locations of the robot and the boxes at the beginning of each trial. Two boxes were placed in the playpen for OBELIX to push around. In contrast, the simulator used four boxes. The boxes used on the simulator are irregular polygons (concave and convex). On the real robot, however, we used only rectangular boxes.

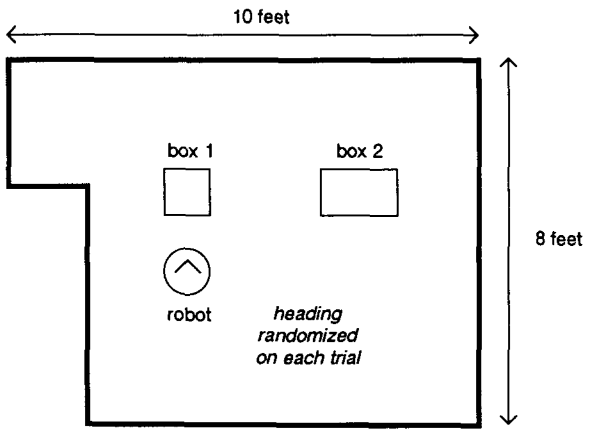

Fig. 24. Playpen for the real robot.

Parameter values

We carried out several experiments comparing the effect of different parameter values on Algorithms 1 and 2. These include the Hamming distance threshold, the number of steps to use in Q learning, and different values of ε, δ, and ρ in Algorithm 2 for clustering. The data we present below uses parameter values that were deemed to be the best from these experiments. In particular, the Hamming distance threshold was set at 2; Q learning was carried out each time over five steps; ε, δ, and ρ were set at 10^{-6}, 0.45, and 2.0, respectively; the learning rate β, and the discount factor γ used in the Q learning update rule were set at 0.5 and 0.9, respectively.

Plotting the experimental results

In the graphs below, the vertical axis plots the average reward value so far. This is computed by dividing the cumulative reward obtained so far by the behavior being learned by the total number of steps over which the behavior has been active so far. The vertical axis is scaled by different amounts in each graph in order to make the differences between the various techniques clear. The horizontal axis represents the percentage of the learning run that has been completed so far. In all cases, the robot was run as a unit with all three behaviors participating (except for the experiments involving the monolithic learner). Since each behavior is not active all the time, the total number of steps is usually less than 2000. For example, the total number of steps for the box finder is about 600 since it is active approximately 30% of the time. Thus, at the horizontal axis label "20", the robot has taken 400 steps, only some of which were generated by the box finder.

8.1. *Learning each behavior separately*

For the first set of experimental results, we focus on learning each behavior separately as a subsumption module. We look at how well Algorithms 1 and 2 were able to learn each of the three behaviors in the box pushing task on the real robot and on the simulator, and compare these results with the performance obtained by the handcoded and random agents.

8.1.1. *Learning to find boxes*

Figure 25 presents data collected using the simulator on learning to find boxes using four different algorithms: Q learning with weighted Hamming (Algorithm 1), Q learning with statistical clustering (Algorithm 2), a handcoded agent, and finally a random agent. The y-axis plots the average value of the reward function for finder, which was described earlier in Fig. 12, at various points along the learning run. The box finder is rewarded for getting something directly in front of the robot.

Figure 25 illustrates that both Algorithms 1 and 2 show steady improvement in performance over the learning run. As expected, both learning agents do substantially better than the random agent. The handcoded agent is clearly superior to Algorithm 1, and is better than Algorithm 2 till late in the learning run. For this particular behavior, Algorithm 2 (clustering) is much better than Algorithm 1 (weighted Hamming), and in fact approaches then exceeds the performance obtained by the handcoded agent. This suggests that Algorithm 2 learns a better similarity metric than the handcoded metric used in Algorithm 1.

In the graph shown in Fig. 25, and in the subsequent graphs, it is important to note that we are plotting the average value of the reward over *the complete run so far*. Thus, since the initial values for the learning algorithms are fairly poor, the fact that they approach the handcoded program's performance at the end of the run indicates that the learning algorithms are actually doing somewhat better than the handcoded program at that point! For example, in Fig. 26, the clustering algorithm (Algorithm 2) is actually 20% better at getting rewards for finding boxes if we just look at the reward values over the last quarter of the trial (that is, the most recent 500 steps).

Although they are fixed algorithms, both the handcoded and random agents show some performance variations over the learning run. Early values are also subject to instability due to the small number of data points so far. This is especially apparent for the random agent. Some of the variation is also due to the fact that the robot takes a random action 10% of the time. We left this feature active during the tests to demonstrate that the learned policy functions were robust and able to recover from errors.

Since the learning agents take random actions 10% of the time, it seemed fair to similarly handicap the handcoded agent. Thus, any performance difference between the learning and handcoded agents cannot be attributed to the fact that random actions need to be taken once in a while to explore the environment.

Figure 26 presents the same data for the real robot OBELIX. Once again, both algorithms show steady improvement over the learning run. Both algorithms are substantially better than the random agent. However, the handcoded agent is better than both algorithms over the complete run. Algorithm 2 (clustering) is superior to Algorithm 1 (weighted Hamming) at finding boxes, and approaches the performance of the handcoded agent. The performance attained by the various algorithms is clearly inferior to their performance on the simulator. One reason for this difference is that there are several obstacles in the interior of the simulator environment whereas there are none in the playpen used by OBELIX. Thus, many more potential "boxes" exist in the simulator.

In the graphs in Figs. 25 and 26 (and elsewhere), the ordering of the two learning algorithms should not be taken too seriously. One, the data represents only one learning run, and there are likely to be small performance variations over several learning runs. Two, in all the graphs the two learning algorithms show substantial improvement over random behavior, sometimes surpassing the performance of the handcoded agent. Finally, Algorithm 2 (clustering) never performs appreciably worse than Algorithm 1 (weighted Hamming) suggesting that there is no penalty for simultaneously learning a

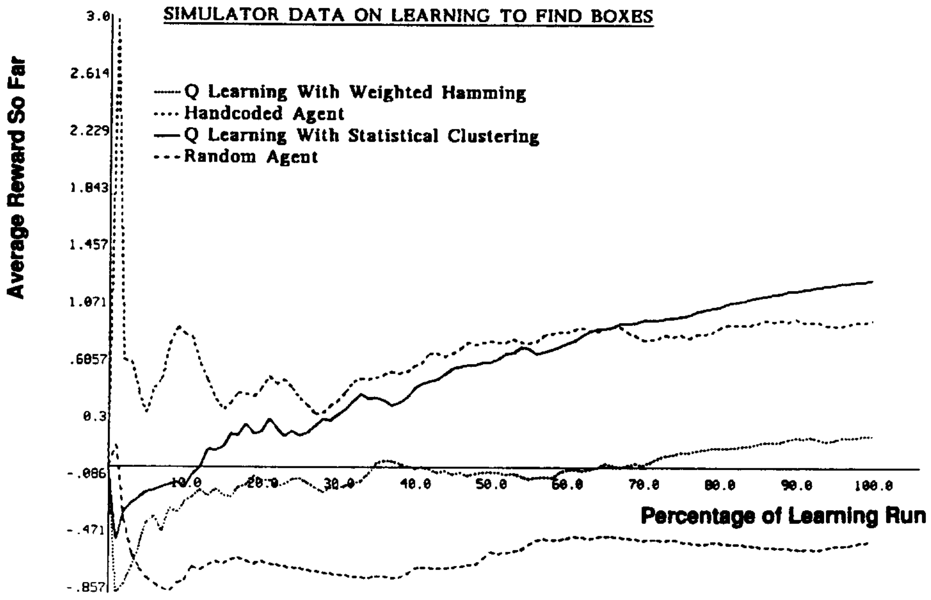

Fig. 25. Simulator data on learning to find boxes.

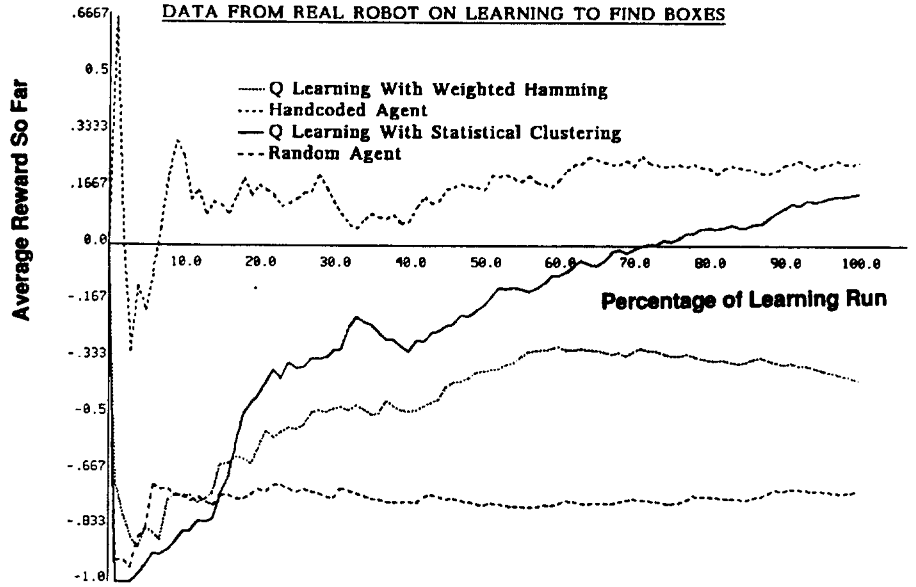

Fig. 26. Real data on learning to find boxes.

similarity metric across states and a policy function.

8.1.2. *Learning to push boxes*

Figure 27 compares the performance of the four techniques at pushing boxes in the simulator. The vertical axis plots the average value of the reward function for the pusher module, which was described earlier in Fig. 13, at various points during the learning run. The pusher module gets rewarded for staying in contact with an object. The data shows that the learning algorithms improve noticeably over the run, and are considerably superior to the random agent. The handcoded agent is better than all the other techniques. Algorithm 2 (clustering) is initially worse than Algorithm 1 (weighted Hamming), but improves dramatically to approach the performance attained by the handcoded agent.

Figure 28 compares the performance of the four techniques at pushing boxes in the real world. Both learning algorithms improve steadily over the run, and are clearly better than the random agent. Algorithm 2 starts off initially doing worse than the handcoded agent, but finally surpasses it. Algorithm 2 (clustering) outperforms Algorithm 1 (weighted Hamming) after recovering from a bad start. Once again, note that the handcoded and learning algorithms do not attain the level of performance on the simulator. This is hardly surprising, given that boxes do not rotate or flex on the simulator.

8.1.3. *Learning to unwedge from stalled states*

Figure 29 compares the performance of the four techniques at unwedging from a stalled state in the simulator. The vertical axis plots the average value of the reward function for the unwedger module, which was described earlier in Fig. 15, over various points along the learning run. The unwedger module gets rewarded for moving the robot forward without stalling. In this case, interestingly the random agent does quite well achieving a result only slightly worse than the other techniques. Given some thought, this is not so surprising—if the robot is stuck against an obstacle, randomly thrashing around will very quickly unwedge it! Once again, the handcoded agent is clearly superior to the others. In this case, however, Algorithm 1 (weighted Hamming) outperforms Algorithm 2 (clustering). The latter once again improves rather dramatically from a very poor start to achieve a level of performance only slightly worse than that attained by Algorithm 1.

Figure 30 compares the performance of the four techniques at unwedging the real robot OBELIX from a stalled state. The handcoded agent performs at about the same level as Algorithm 1 and Algorithm 2. The random agent performs worse than either Algorithm 1 or 2, but once again the differences between it and the others are small for the unwedging behavior. In this case,

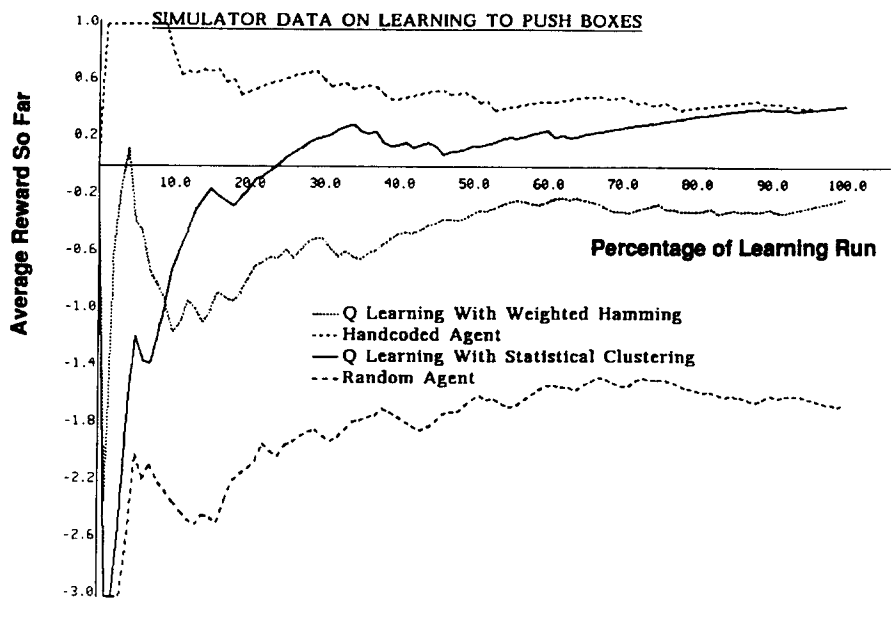

Fig. 27. Simulator data on learning to push boxes.

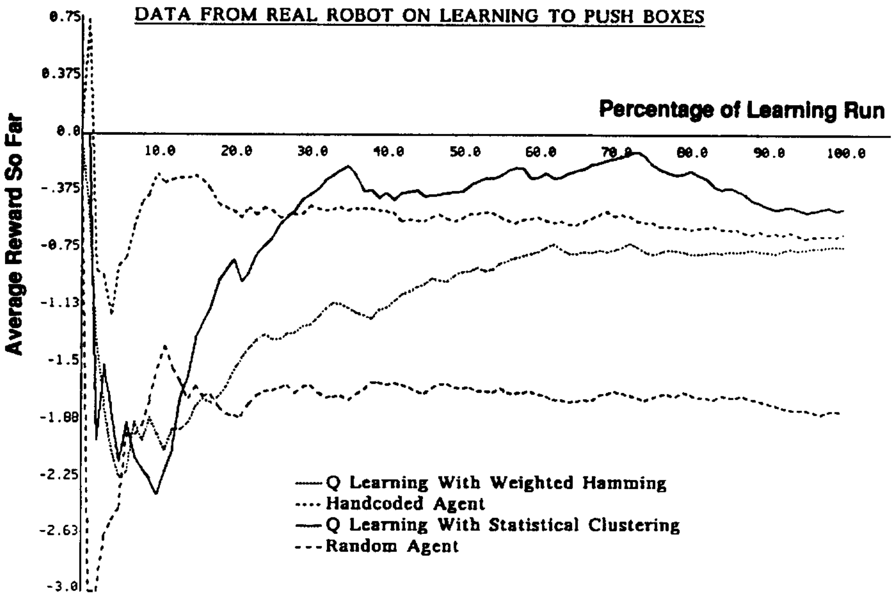

Fig. 28. Real data on learning to push boxes.

Algorithm 1 (weighted Hamming) is better than Algorithm 2 (clustering), as was also observed for the simulator data.

8.2. Analysis of data on learning individual behaviors

As stated before, one of the central questions is: "Can our approach learn the individual behaviors, and if so, with what degree of success?" We answered this question above by directly appealing to the graphs. Here we answer this question numerically by extracting some quantitative information from the data.

Tables 1 and 2 compare the performance obtained using the two learning algorithms, the handcoded agent, and the random agent on the simulator and the real robot, respectively. The percentage figures were derived using the above graphs and the reward functions used in the various modules. An example will help illustrate how the numbers were computed. The final average reward for the box finder behavior using Q learning with clustering on the real robot was 0.16 (see Fig. 26). The maximum and minimum reward values for the box finder are 3.0 and −1.0 (from the reward function for box finder in Fig. 12). Hence the percentage improvement for box finder from the lowest reward value is

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

This is a quantitative measure of the average performance achieved over a complete learning run. It will be somewhat lower than the "ultimate" performance level since it includes all the early steps during which the robot was not particularly competent yet. The same experiment was run a second time using the weighted Hamming algorithm on the real robot with similar results (within 3% of the values listed here).

Table 1
Average performance at end of learning run for simulator data.

Technique	Finder	Pusher	Unwedger
Handcoded agent	48%	88%	85%
Q learning with clustering	55%	88%	75%
Q learning with weighted Hamming	30%	70%	79%
Random agent	13%	33%	69%

The tables indicate that Algorithm 1 and Algorithm 2 were very successful at learning to push boxes and unwedge from stalled states on the simulator—the performance is very close to or better than the performance of the handcoded agent. The algorithms were understandably less successful on the real robot, indicating that the real world environment presents a greater

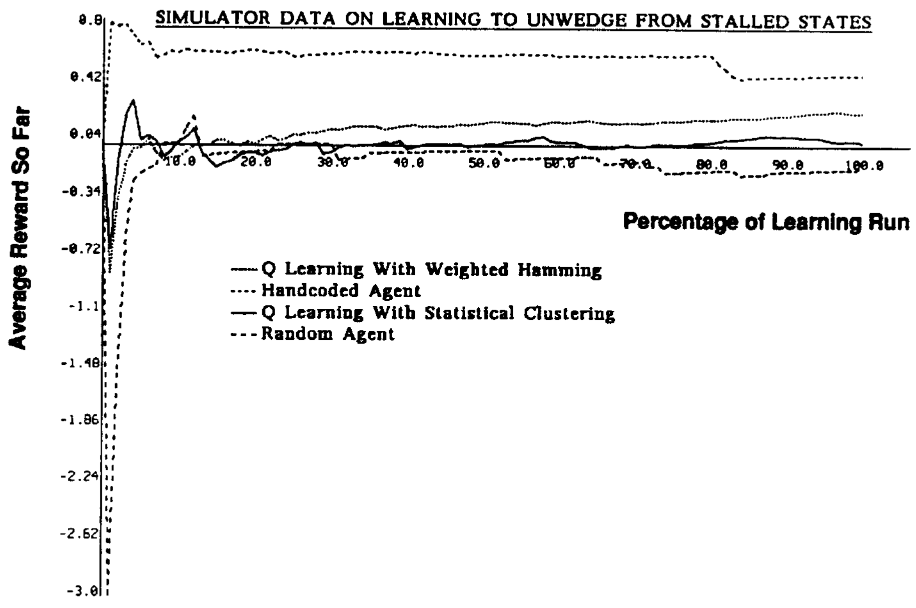

Fig. 29. Simulator data on learning to unwedge.

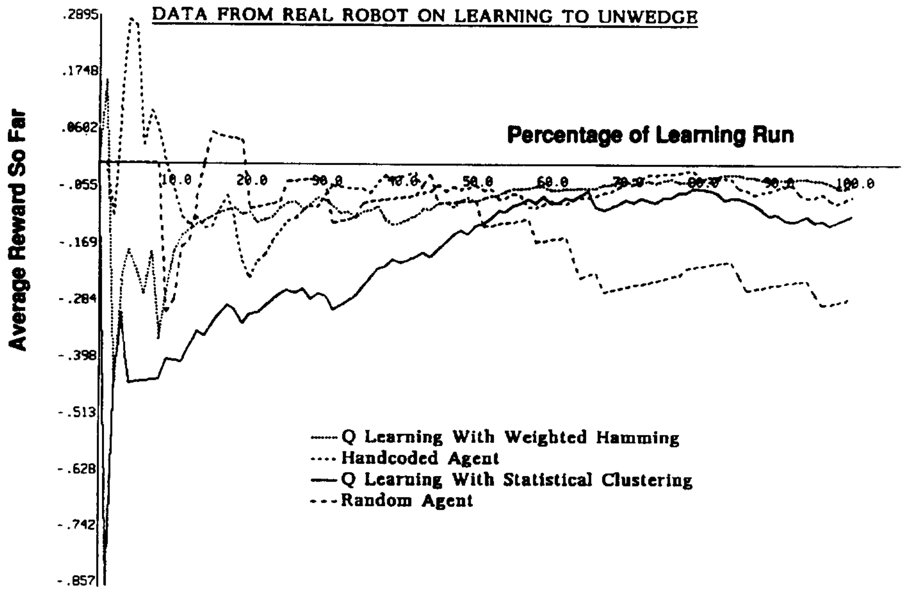

Fig. 30. Real data on learning to unwedge.

Table 2
Average performance at end of learning run for real data.

Technique	Finder	Pusher	Unwedger
Handcoded agent	31%	57%	73%
Q learning with clustering	29%	60%	72%
Q learning with weighted Hamming	15%	55%	74%
Random agent	8%	30%	68%

challenge to the learning algorithms. But even on the real robot, performance comes close to or exceeds that of the handcoded agent.

The algorithms are clearly less successful at learning to find boxes, especially on the real robot. Again, this is what we expected, given the difficulty of distinguishing boxes from obstacles using sonar. However, there is a noticeable difference between the performance of the handcoded agent and the learning algorithms at finding boxes. This suggests that there is considerable room for improvement at this task by developing better learning techniques.

Finally, note that the simulator correctly predicts the convergence rate of the algorithms (see earlier graphs), and the relative effectiveness of each. However, it tends to substantially overestimate the actual performance.

8.3. Learning box pushing as a monolithic task

Now we turn to analyzing the performance of an agent who learns the box pushing task in its entirety without decomposing it into the constituent behaviors of finding, pushing, and unwedging. We created a monolithic learner by defining a single module—box pusher—that was active all the time. The reward function for the monolithic learner is given in Fig. 31. 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. Thus, the monolithic reward function can be used to measure the percentage of time the robot was actually pushing a box.

8.3.1. Evaluating the subsumption approach using the monolithic reward function

Comparing the subsumption approach with the monolithic one is difficult unless the same reward function is used. However, the subsumption modules use reward functions different from the one used by the monolithic learner. To overcome this problem, while collecting data on the performance of the subsumption learner, we also measured its performance using the monolithic reward function. This gives us a common metric for comparing the two approaches.

```
Monolithic_reward(old_state, action, new_state):
begin
  IF action = FORWARD              % went forward
     and BUMP(old_state)           % bumped before
     and BUMP(new_state)           % bumped now
     and ¬STUCK(new_state)         % not stuck now
  THEN return 1                    % reward robot
  ELSE return 0                    % default is no reward
end
```

Fig. 31. Reward function for the monolithic learner.

Figure 32 shows the performance of the subsumption learner using the reward function given to the monolithic learner. In this case, Algorithm 1 (weighted Hamming) outperforms all the other techniques, including even the handcoded agent. Algorithm 2 (clustering) does slightly worse than the handcoded agent. Finally, the random agent does miserably—even on such a fine resolution plot, it can hardly be made out above the horizontal axis.

Figure 33 shows the performance of the subsumption learner on the real robot using the monolithic reward function. Algorithm 1 does better than any of the others, although Algorithm 2 starts performing better in the end. The random agent does much worse, as expected. The handcoded agent does slightly better than the others at the end.

For the individual behaviors, Algorithm 2 seemed slightly better than Algorithm 1. In particular, Algorithm 2 was better at finding and pushing than Algorithm 1, and roughly comparable at unwedging. For the overall task, however, Algorithm 1 seems better than Algorithm 2. This suggests that improved performance at the individual behaviors does not necessarily translate into improved overall task performance. In particular, the latter requires good coordination between the behaviors. For example, although Algorithm 2 finds a lot more objects than Algorithm 1, not many of these turn out to be boxes. Hence, its performance as evaluated by the monolithic reward function may not be as good as that of Algorithm 1, which could be more discriminating in its choice of "potential" boxes.

8.3.2. Evaluating the monolithic learner

Next we evaluate the performance of the monolithic learner at the box pushing task. One way to visualize the monolithic learner is as a degenerate instance of a subsumption approach. Namely, it comprises of a single module that is always applicable, and that attempts to learn all three behaviors—finding, pushing, and unwedging.

In some sense, the monolithic learner is a straw man because of its

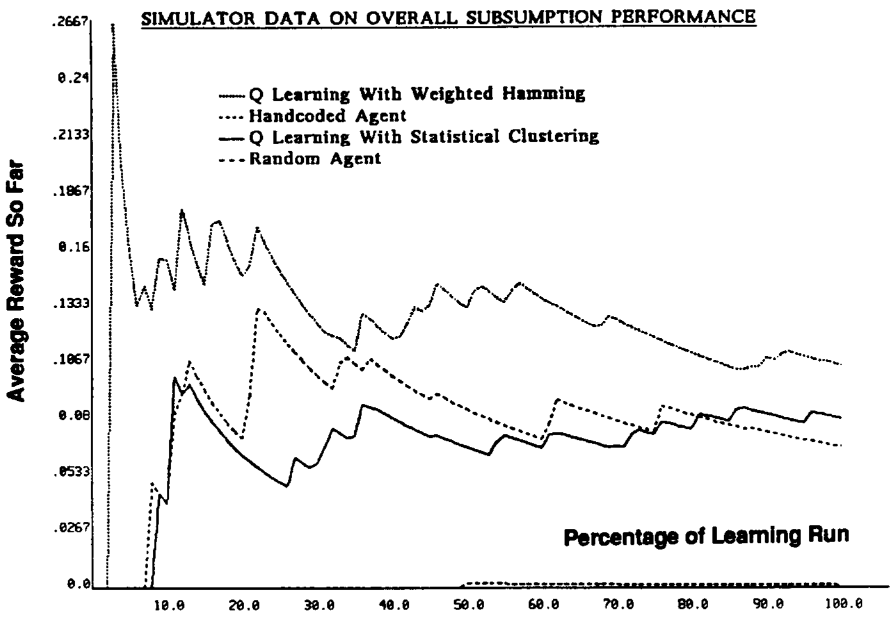

Fig. 32. Simulator data evaluating the subsumption approach using the monolithic reward function.

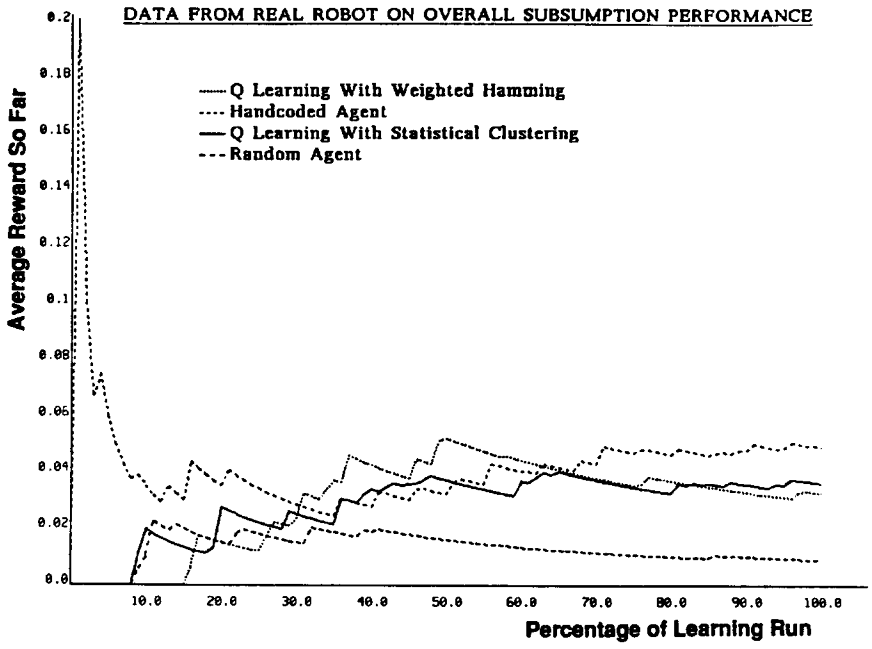

Fig. 33. Real data evaluating the subsumption approach using the monolithic reward function.

reward function. The three-part subsumption learner is given a lot more information on which states are useful since it has three separate reward functions. This also means that many of the steps taken by the subsumption learner generate a reward for one of the modules. In the monolithic case, rewards are few and far between. Still, it is important to contrast the subsumption learner with the monolithic learner, since the latter represents a first pass at a solution to the learning problem on which we spent a few frustrating months.

Furthermore, it is not the case that giving the monolithic learner a more complex reward function will necessarily improve its performance. In fact, we have found that using reward functions with many conditions in them tends to degrade task performance ultimately because the learner gets stuck in local minima. In the early stages of our work, we found the monolithic learner would converge to one of two strategies—aggressively pushing every object it found, or conservatively avoiding every object that it came near to—depending on whether it got rewarded more often for pushing boxes or punished more often for getting wedged.

One could look at this as an argument for using the subsumption approach. In a real robot, it is difficult to carry out a large number of trials. Therefore, we want the robot to receive some kind of reward on a large percentage of these steps. To do this with local sensing a complicated reward function with multiple terms is generally required. However, such functions are particularly susceptible to getting stuck in local minima. Thus, we need to divide the control function up into a number of pieces each of which has a fairly simple reward function.

The architectural change from subsumption to monolithic does not affect either the handcoded or the random agent—the programs remain the same for both cases. Thus, in the graphs below we directly compare the performance of Algorithms 1 and 2 for the subsumption case with that for the monolithic case.

Figure 34 compares the performance of Algorithms 1 and 2 for the monolithic case with that for the subsumption case on the simulator. The performance of the two algorithms for the monolithic case is clearly inferior to their performance for the subsumption case.

Figure 35 compares the performance of Algorithms 1 and 2 for the monolithic case with that for the subsumption case on the real robot. After the first half of the learning run, the performance of the two algorithms for the subsumption case is markedly better than their performance for the monolithic case.

To summarize, both the simulator data and the data from the real robot show that the performance of Algorithms 1 and 2 are noticeably better for the subsumption case than for the monolithic case.

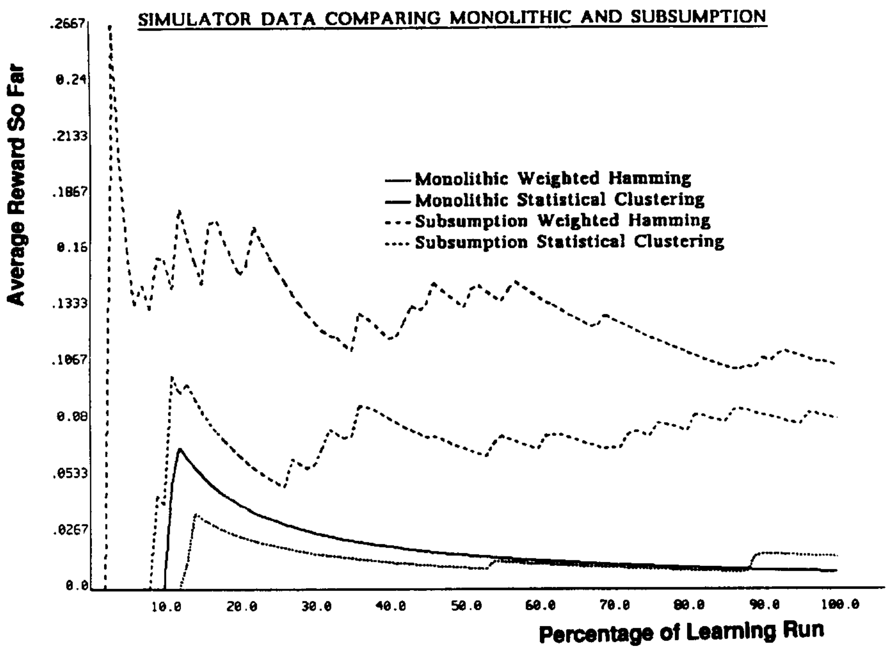

Fig. 34. Simulator data comparing the monolithic and subsumption learners.

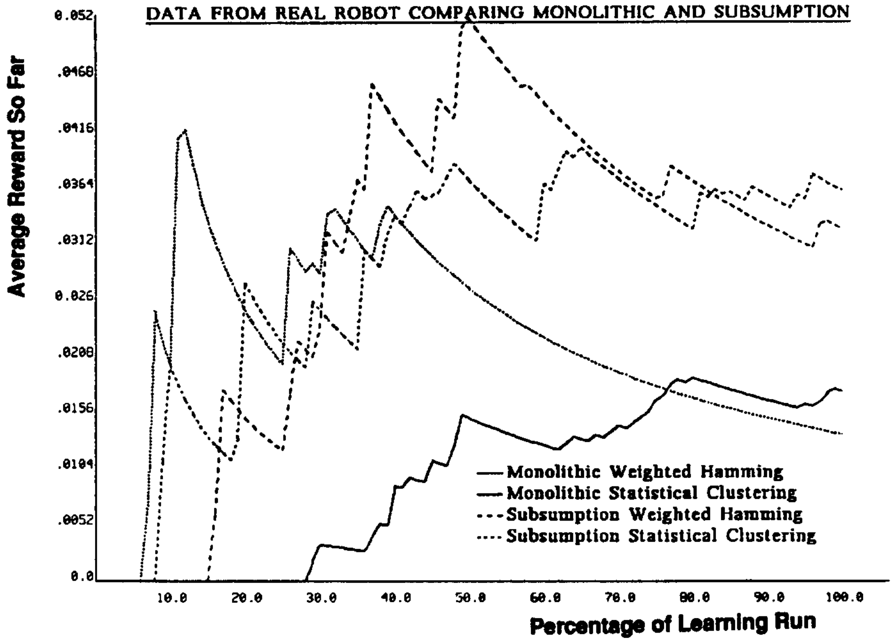

Fig. 35. Data from the real robot comparing the monolithic and subsumption learners.

8.3.3. *Subsumption versus monolithic architectures*

Let us now compare the subsumption approach with the monolithic approach using as a metric the number of steps during which the robot was actually pushing a box. This measure is somewhat noisy and should be viewed only in general terms. For instance, the same experiment was run a second time on the real robot using the weighted Hamming algorithm and produced a result about 50% better than reported here.

Table 3 shows the number of box pushing steps for the two learning algorithms on the simulator over a learning run of 2000 steps. The table shows that the two algorithms are at least seven times better at pushing boxes for the subsumption case than they are for the monolithic case.

Table 3
Number of steps a box was pushed over a learning run on the simulator.

Technique	Monolithic	Subsumption
Q learning with clustering	16	159
Q learning with weighted Hamming	30	209

Table 4 shows the number of box pushing steps for the two learning algorithms on the real robot over a learning run of 2000 steps. Once again, the table shows that the two algorithms are at least twice as good for the subsumption case than they are for the monolithic case. As expected, the data shows that the algorithms are less successful on the real robot than they are on the simulator.

Table 4
Number of steps a box was pushed over a learning run on the real robot.

Technique	Monolithic	Subsumption
Q learning with clustering	35	72
Q learning with weighted Hamming	27	65

The improvement of the subsumption learner over the monolithic learner at the box pushing task is not as dramatic as one might expect. This is partly because it is very difficult to find boxes using real sonar (see the low numbers in Table 2). Note that on the simulator, which has much cleaner sonar, the subsumption approach is markedly better than the monolithic learner.

Subjectively, however, even on the real robot the subsumption learner appears much more purposive over a learning run than the monolithic learner.

The subsumption learner converges to a clear strategy while the monolithic learner seems to wander around aimlessly. To corroborate this subjective observation, we evaluated the monolithic learner using the subsumption reward functions. This makes sense because the subsumption learner's reward functions actually embody a more detailed description of the task. The results of the evaluation are shown in Table 5 only for the Hamming technique (with the results for the random agent repeated for easy reference).

Table 5
Evaluating the monolithic learner using subsumption rewards.

Technique	Finder	Pusher	Unwedger
Hamming	7%	46%	67%
Random	8%	30%	68%

Comparing the rows in Table 5 reveals that the monolithic learner really only learned to push a box. In terms of finding boxes and unwedging from stalled states, the monolithic learner is only as good as the random agent. Thus, the reason that the monolithic learner performs the overall task only half as well as the subsumption learner is mainly because it never learned to find boxes. Note that the monolithic learner receives rewards only when it pushes a box, a fairly rare event during the learning run. In fact, with such a reward the early stages of the learning are essentially a random walk [31]. The monolithic system's poor performance at finding boxes suggests that, over the limited length of a learning run, the learning techniques were unable to resolve the long-term temporal credit assignment problem involved in approaching potential boxes.

Summarizing, analysis of the data shows that the two learning algorithms are able to successfully learn the three separate behaviors in the box pushing task. Furthermore, the subsumption approach seems clearly superior to the monolithic approach at learning the task.

9. Limitations of our work

Our work currently suffers from a number of limitations. 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 broken up.

Our approach assumes that the learner is given a partially specified subsumption structure for the task. In particular, this implies coming up with suitable applicability conditions and reward functions for each module. In our case, the applicability conditions are very straightforward—a single predicate like BUMP or STUCK that lingers for a few time instants. These might conceivably be inferred as "synthetic items" by Drescher's system [10]. Keeping a history of the last k states visited does not work well; it only complicates the structural learning problem. On the other hand, it took us several iterations to write reward functions that generated good performance figures. Ideally, we would like to just guide the robot through the task—pointing out certain desirable states (such as pushing a box) and some undesirable states (such as pushing a wall)—and have it automatically learn the rest.

The performance of the robot at the end of a learning run is somewhat disappointing. In particular, the number of box pushing steps in Table 4 is admittedly low. This is partly because boxes are very difficult to detect using sonar. We are currently investigating more complete representations to help overcome this difficulty.

The box pushing task is quite simple as it involves only five actions and a state representation of 18 bits. Pierce [20,21] has shown some interesting results with larger input representations. We are currently experimenting with a much richer sensory input to the learner (see Section 11). Also, Lin [13] has recently shown that our approach can be extended to a more complex task by explicitly teaching the robot. Similarly, Whitehead [31] describes some theoretical results showing that in certain domains the performance of Q learning can be greatly improved with the addition of teaching.

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. Some work along this direction has recently been reported [25,33].

10. Related work

Our work is based extensively on previous work in reinforcement learning. The two techniques described in our paper both use Q learning to deal with temporal credit assignment. Q learning was developed by Watkins [30]. Sutton [28] showed how Q learning could be integrated into a reactive planning system. Lin [12] presents a detailed study of different reinforcement algorithms using Q learning.

The clustering technique used in Algorithm 2 is similar in spirit to other clustering techniques studied in machine learning [5], and to decision tree techniques such as ID3 [22]. Tan and Schlimmer [29] show how ID3 can be applied to the robotics domain to learn sensing and grasping strategies for picking up objects. Our work differs from these in that we are applying clustering in the context of reinforcement learning.

Chapman and Kaelbling [6] describe a splitting technique that seems to be an interesting dual to our clustering technique. The basic idea is to start with completely indistinguishable state descriptions, and then start differentiating between them by splitting on the values of certain state variables.

The weighted Hamming technique is similar to techniques like CMAC [1] that generalize across state vectors. The pole balancing example mentioned in Section 3 has been extensively studied in reinforcement learning. Barto et al. [2] extend the earlier BOXES algorithm [15] in their ACE/ASE system using a temporal differencing technique.

Our work also draws on earlier research on behavior-based robots using the subsumption architecture [3]. The particular definition of modules we used in Section 4 derives from Connell's thesis [8]. The main difference is that we are studying how to automatically program such robots by having them learn new behaviors.

Mitchell's task [17] of sliding a block using a finger contact served as the inspiration for 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. [7] describe a similar trial and error inductive approach for learning action models in a tile sliding task.

Rivest and Schapire [23] describe a technique for learning a task environment in the form of a finite state machine. Their approach could be applied to our problem by learning the physics of the box pushing task (that is, how the world changes as the robot moves around in it). One problem with their approach is that it is not sensitive to the particular task being performed by the robot. Since there are many features of the robot's environment that are unrelated to its task of pushing boxes, their learning technique may spend lots of time learning parts of the environment not related to the task. Their technique also throws out most of the learned world structure every time it discovers a discrepancy, and thus requires on the order of a million steps to learn relatively simple environments.

Kaelbling [11] 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 the input space is only 4 bits, as opposed to 18 bits in our case. Another difference is that we use the subsumption structure to speed up the convergence of the learning. Finally, much of her experimental data seems to be derived from a simulator.

Maes and Brooks [14] 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.

Finally, Singh [25] describes an interesting approach to automatically learning the temporal decomposition of sequential tasks, such as "go to place A and then go to place B". His system automatically strings together simple, already-learned subtasks in order to achieve some more complex goal. The particular task domain is a simple simulated 8×8 static grid environment; it remains to be seen whether his approach can be scaled up to a real robot task.

11. Conclusions and future work

This paper attempts to empirically substantiate two claims.

(1) Reinforcement learning is a viable approach to learning individual modules in a behavior based robot.
(2) Using a subsumption architecture is superior to using a monolithic architecture in reinforcement learning.

We have provided detailed experimental evidence that support these claims using real and simulated robots which learn several component behaviors in a task involving pushing boxes.

Based on our experience with using reinforcement learning on real robots, we think the hard subproblems in reinforcement learning have to do with dealing with large sensory state spaces, long action sequences, and initial task specification.

Sensors on a real robot can potentially generate a lot more than the handful of bits we used. For example, the sonar system on OBELIX normally generates more than 1000 bits every tenth of a second. We manually compress this information (using range bins) down to a mere 18 bits every five seconds (time to take an action). Good preprocessing techniques are still somewhat of an art, and do not currently exist for many sensory systems. It would be more convenient if the reinforcement learning techniques could actually do some of the interpretation of the sensory information. However, this obviously exacerbates the structural credit assignment problem in reinforcement learning.

Action spaces on a real robot are usually discretized much more finely than the small set used by OBELIX. For example, OBELIX can carry out fairly complicated maneuvers, such as turning through an arbitrary angle while moving forwards or backwards at an arbitrary speed. Once again we discretized the action space on OBELIX down to a mere five actions. However, the temporal credit assignment problem becomes much more difficult as the branching factor of the action space increases.

Specifying a reinforcement learning task to a robot requires partitioning the task into simpler subtasks, and writing reward functions for each subtask. Obviously, there may be several ways of decomposing a given task, and coming up with a good decomposition is a nontrivial problem. In particular, it is difficult to choose reward functions for the subtasks that when individually optimized translate into optimal performance on the overall task.

There are examples of each of the above problems in the box pushing task. The robot never really learned any method for distinguishing boxes from walls, such as using the width of the object in front of the robot. Also, it never was able to push a box to a wall, and then go around to the side and push the box to a corner. Finally, although the clustering algorithm does as well or better than the weighted Hamming algorithm on each of the subtasks of the box pushing task, it does not push as many boxes overall.

We plan to extend our work in several directions to address some of the above problems with reinforcement learning. We are currently experimenting with a scrolling occupancy grid sonar sensor [19]. This generates a 12×12 array of numerical probability values representing which areas around the robot are likely to be solid objects. We are investigating the effectiveness of the clustering method in this new state space, as well as comparing it with several alternative schemes such as neural nets. As for the temporal credit assignment problem, one approach to combating this is to use a hybrid architecture which adds a symbolic layer above subsumption. This new layer would deal with larger temporal chunks, and propagate rewards across events instead of actions. As for the initial task specification issue, we are studying ways of inferring appropriate reward functions from instances of good and bad situations.

In conclusion, although there are some difficult problems with using reinforcement learning to program real robots, it does appear to be a promising approach. It does not require intimate knowledge of the structure of the robot or of the environment. The robot does not need constant supervision, which is good for inaccessible environments such as underwater or outer space. Furthermore, one does not require any operational knowledge about how to do the task, but only whether one is getting better or worse over some finite time interval.

Acknowledgement

We thank Sanjaya Addanki for encouraging us to write the paper. Norm Haas helped debug our software, and took OBELIX's photograph. Alka Indurkhya assisted us in collecting and analyzing data from the simulator and the real robot. Rick Kjeldsen wrote the software for displaying the graphs. Pandurang Nayak helped analyze the clustering algorithm. Many of our colleagues provided valuable feedback on an earlier version of our paper, including Richard Sutton, Nils Nilsson, Steve Whitehead, Leslie Kaelbling, Long-Ji Lin, Ming Tan, Maja Mataric, Armand Prieditis, and Prasad Tadepalli. Finally, we thank the reviewers for their comments on the paper.

References

[1] J. Albus, *Brains, Behaviors, and Robotics* (BYTE Books, 1981).

[2] A. Barto, R. Sutton and C. Anderson, Neuronlike adaptive elements that can solve difficult learning control problems, *IEEE Trans. Syst. Man Cybern.* **13**(5) (1983) 834–846.

[3] R.A. Brooks, A robust layered control system for a mobile robot, *IEEE J. Rob. Autom.* **2**(1) (1986) 14–23.

[4] R.A. Brooks, The behavior language: user's guide, Tech. Rept., AI Memo 1227, MIT, Cambridge, MA (1990).

[5] K. Chan and A. Wong, Performance analysis of a probabilistic inductive learning system, in: *Proceedings Seventh International Conference on Machine Learning*, Austin, TX (1990) 16–23.

[6] D. Chapman and L. Kaelbling, Learning from delayed reinforcement in a complex domain, in: *Proceedings IJCAI-91*, Sydney, NSW (1991).

[7] A. Christiansen, T. Mason and T. Mitchell, Learning reliable manipulation strategies without initial physical models, in: *Proceedings IEEE Conference on Robotics and Automation* (1990) 1224–1230.

[8] J.H. Connell, *Minimalist Mobile Robotics: A Colony-Style Architecture for an Artificial Creature* (Academic Press, New York, 1990); also: AI TR 1151, MIT, Cambridge, MA.

[9] G.F. Dejong and R. Mooney, Explanation-based learning: an alternative view, *Mach. Learn.* **1**(2) (1986) 145–176.

[10] G. Drescher, Made-up minds: a constructivist approach to artificial intelligence, Ph.D. Thesis, MIT, Cambridge, MA (1990).

[11] L. Kaelbling, Learning in embedded systems, Ph.D. Thesis, Stanford University, Stanford, CA (1990).

[12] L. Lin, Self-improving reactive agents: case studies of reinforcement learning frameworks, Tech. Rept. CMU-CS-90-109, Carnegie-Mellon University, Pittsbugh, PA (1990).

[13] L. Lin, Programming robots using reinforcement learning and teaching, in: *Proceedings AAAI-91*, Anaheim, CA (1991).

[14] P. Maes and R.A. Brooks, Learning to coordinate behaviors, in: *Proceedings AAAI-90*, Boston, MA (1990) 796–802.

[15] D. Michie and R. Chambers, Boxes: an experiment in adaptive control, in: E. Dale and D. Michie, eds., *Machine Intelligence* **2** (Oliver and Boyd, Edinburgh, Scotland, 1968).

[16] T.M. Mitchell, Generalization as search, *Artif. Intell.* **18**(2) (1982) 203–226.

[17] T.M. Mitchell, Towards a learning robot, Tech. Rept. CMU-CS-89-106, Carnegie-Mellon University, Pittsburgh, PA (1989).

[18] T.M. Mitchell, R. Keller and S. Kedar-Cabelli, Explanation-based generalization: a unifying view, *Mach. Learn.* **1**(1) (1986).

[19] H. Moravec and A. Elfes, High resolution maps from wide angle sonar, in: *Proceedings IEEE International Conference on Robotics and Automation*, St. Louis, MO (1985) 116–121.

[20] D. Pierce, Learning hill-climbing functions as a strategy for generating behaviors in a mobile robot, in: *Proceedings First International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, Paris (1990) 327–336.

[21] D. Pierce, Learning a set of primitive actions with an uninterpreted sensorimotor apparatus, in: *Proceedings Eighth International Workshop on Machine Learning* (1991) 338–342.

[22] J.R. Quinlan, Induction of decision trees, *Mach. Learn.* **1**(1) (1986) 81–106.

[23] R. Rivest and R. Schapire, Inference of finite automata using homing sequences, in: *Proceedings 21st Annual ACM Symposium on Theory of Computing* (1989) 364–375.

[24] C. Sammut and J. Cribb, Is learning rate a good performance criterion for learning, in: *Proceedings Seventh International Conference on Machine Learning*, Austin, TX (1990).

[25] S. Singh, Transfer of learning across compositions of sequential tasks, in: *Proceedings Eighth International Workshop on Machine Learning* (1991) 348–352.

[26] R. Sutton, Temporal credit assignment in reinforcement learning, Ph.D. Thesis, University of Massachusetts, Amherst, MA (1984).

[27] R. Sutton, Learning to predict by the method of temporal differences, *Mach. Learn.* **3** (1988) 9–44.

[28] R. Sutton, Integrated architectures for learning, planning, and reacting based on approximating dynamic programming, in: *Proceedings Seventh International Conference on Machine Learning*, Austin, TX (1990) 216–224.

[29] M. Tan and J. Schlimmer, Cost-sensitive concept learning of sensor use in approach and recognition, in: *Proceedings Sixth International Conference on Machine Learning*, Ithaca, NY (1989).

[30] C. Watkins, Learning from delayed rewards, Ph.D. Thesis, King's College (1989).

[31] S. Whitehead, A complexity analysis of cooperative mechanisms in reinforcement learning, in: *Proceedings AAAI-91*, Anaheim, CA (1991).

[32] S. Whitehead and D. Ballard, Active perception and reinforcement learning, in: *Proceedings Seventh International Conference on Machine Learning* Austin, TX (1990) 179–188.

[33] L. Wixson, Scaling reinforcement learning techniques via modularity, in: *Proceedings Eighth International Workshop on Machine Learning* (1991) 368–372.