Integrating POMDP and Reinforcement Learning for a Two Layer Simulated Robot Architecture

Larry D. Pyeatt Adele E. Howe

Computer Science Department
Colorado State University
Fort Collins, CO 80523

{pyeatt,howe}@cs.colostate.edu
http://www.cs.colostate.edu/~{pyeatt,howe}

Abstract

Two layer control systems are common in robot architectures. The lower level is designed to provide fast, fine grained control while the higher level plans longer term sequences of actions to achieve some goal. Our approach uses reinforcement learning (RL) for the low level and Partially Observable Markov Decision Process (POMDP) planning for the high level. Because both levels can adapt their behavior within the scope of their tasks, the combination is expected to be robust to degradations in sensor and actuator failures and so to enhance overall system reliability. We implemented our architecture for use in the Khepera robot simulator. In a set of experiments, we show that good performance can be difficult to achieve with hand coded low level control and that performance of our RL/POMDP system degrades slowly with increasing sensor and actuator failure.

1 Introduction

Two layer control systems are common in robot architectures. Two layer control is a pragmatic solution to different time and control granularities of activity. The lower layer provides fast, short horizon decision about quickly executed actions; the upper layer adopts a more goal oriented view and plans over a longer scope. Thus, the lower layer is designed to keep the robot out of trouble (e.g., running into obstacles) and make minor modifications to an otherwise good course of action based on sensory feedback (e.g., course corrections). The upper layer ensures that the robot continually works toward its target task or goal (e.g., does not paint itself into a corner or go in circles).

The two layer control architecture offers reliability. For example, autonomous robots must be able to deal with failure of sensors and actuators in a robust way when they may be operating out of direct human control (e.g., space exploration, hazardous environments). In these situations, hardware failure can result in failure of the mission unless the control system is sufficiently adaptive. In less critical situations, robust operation is also necessary. Even an office robot should be capable of dealing with some sensor failures or miscalibrations and actuator failures.

We wish to develop a robust control system for an autonomous robot that can operate in an office-like environment. By robust, we mean that the robot should be able to deal with sensor and actuator malfunction caused by hardware failure or low battery conditions. Our current system has two levels of control. The lower level controls the actuators that move the robot around and provides a set of behaviors that can be used by the higher level control and planning system. The upper level plans a sequence of behaviors to run in order to move the robot from its current location to the goal.

In our architecture, the bottom level is accomplished by reinforcement learning (RL); the top level is a Partially Observable Markov Decision Process (POMDP) planner. It is this combination of RL Learning and POMDP that distinguishes our approach from previous work. We believe that RL is a good choice for implementing low level control because it is able to learn online and can adapt to changes in the environment. RL simultaneously learns the value function and the policy, reducing the intervention of the programmer and accommodating further learning as environment conditions change. POMDP planning is a good framework for the higher level control because it operates quickly once a policy is generated and can provide the reinforcement needed by the lower level behaviors.

We have been testing our architecture on the Khepera robot simulator [14]. Khepera is a very small robot with limited sensors and a well-defined environment. This makes simulation an attractive option for testing our ideas. The simulator can be run much faster than real-time and does not require human intervention for low battery conditions and sensor failures as a physical robot could.

The research presented in this paper shows that reinforcement learning at the low level and POMDP planning at the higher level can result in a system that is more robust than one that uses only pre-programmed behaviors.

1.1 Reinforcement Learning at the Bottom

Several methods are available for implementing low-level behaviors. The most direct is to program them as reactions [6]. Alternatively, they can be learned from examples. Sammut [17] uses an approach referred to as behavioral cloning to create rules for an airplane control system. Behavioral cloning extracts situation-action rules from logs of actions taken by a human subject. Homogeneous multi-layer architectures such as subsumption [2] have supported incremental development of behaviors and demonstrated robust performance.

In general, neural systems tend to be robust to noise and perturbations in the environment. For this reason, neural learning is a popular approach to low level robot control [19]. Neural control systems have been developed to solve inverse kinematics for robot arms and visual robot positioning in several domains. GeSAM is a neural network based robot hand control system based on human prehensile function [12]. It uses an adaptive neural network to learn the relationship between object primitives and a set of grasp modes for picking up cylindrical objects. However, neural networks often require long training periods and large amounts of data.

Some researchers have examined RL techniques for mobile robot navigation and fine manipulation for robot arms. RL modules can deal with relatively high sensor data rates and do not require much sensor pre-processing [18].

Reinforcement learning modules can learn continuously and provide adaptation to sensor drift and changes in actuators caused by changes in battery power. Also, in many extreme cases of sensor or actuator failure, the reinforcement learning modules can adapt enough to allow the robot to accomplish its mission. Another desirable property is that each behavior learns its own value function, independent of the others, and is thus self contained.

1.2 POMDP Planning at the Top

As with the lower level, a variety of approaches have been explored for higher level control. One common approach has been to integrate a deliberative, symbolic planner with reactive control levels (e.g., Cypress [20], GLAIR [8], RALPH-MEA [15]). In general, these planners produce more complex plans of longer duration than is appropriate for our application at present.

For our application, we assume limited sensor and actuator capabilities in an environment with a mostly known terrain (offices). Khepera has only pulse encoders on the wheels and no vision, so determining the exact state of the robot would be very difficult. Also, the effects of actuators may not be deterministic; worse still, the effects of behaviors as implemented in the RL modules are unlikely to be predictable, especially during training. The POMDP framework provides a way to deal with this uncertainty. The POMDP frameworkalso allows for the learning of state transition and observation probabilities on-line. Thus, the higher level can also adapt to changes in the environment or in the robot's actuators and sensors.

Goals in the POMDP framework are expressed as reward functions. Some rewards may be associated with the goal state, or with performing some action in a certain state. Most POMDP solution algorithms try to maximize the reward received. Thus, we can define complex, compound goals within this framework.

The major drawback with using POMDP planning is that current POMDP solution methods do not scale well with the size of the state space. Exact solutions are only feasible for very small POMDP planning problems. POMDP planning also requires that the robot be given a map of its environment, which is not always feasible. We use a small domain and provide the map to the robot.

1.3 What is Gained by Integration

By combining POMDP planning with RL, we can construct a system that is robust to changes in the environment and to failure of sensors and actuators. RL provides the mechanism for adapting to changes or malfunctions in actuators and sensors. The low level behaviors are continuously learning to maximize the reward provided by the POMDP planner.

Continuous learning allows reinforcement learning to compensate for gradual changes in sensors and actuators. Large changes in actuators or sensors can cause the reinforcement learning modules to fail completely. However, when enough negative rewards have accumulated, the RL modules will begin searching for a better solution and will learn how to use the damaged sensors and actuators, if possible. Since the only goal for the low level behaviors is to maximize reward, they are free to search the possible solution space. In many instances, the existing policy will require only slight modification after sensor or actuator failure.

Continuous learning may have some drawbacks, especially when using backpropagation neural networks that use activation functions with global support to approximate the value function. These drawbacks include over-training leading to cyclical performance and failure to find a stable solution while randomly exploring the state space. Our current implementation uses a lookup table for storing the value function, which does not suffer from these problems, but will not scale well to include more sensors.

The POMDP planner at the high level is given a map of the environment, but must learn the relationship between the behaviors and the positions in the state space. This means that it can also adapt to changes in the environment and to sensor and actuator failures by adjusting the transition probabilities that it uses in planning which behavior to use. Thus, reliability is increased in the overall system through adaption at *both* levels.

1.4 Our Application Domain: Khepera

Our research uses the Khepera robot simulator [14]. Khepera is a small (55mm diameter) desktop robot with limited sensors and actuators. The sensors consist of eight infrared transmitter/receiver pairs arranged as shown in Figure 1. It also has pulse encoders on the wheels, but we do not use them in this work. The robot has two

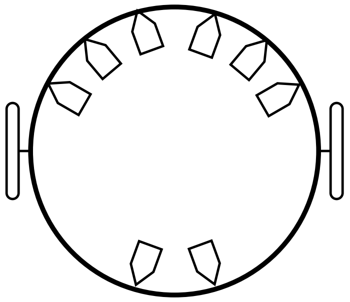

Figure 1: Arrangement of Khepera infrared sensors.

drive wheels, one on each side that can be controlled independently, providing the ability to control speed and direction of the robot.

The Khepera robot is simple enough to be adequately modeled in a simulator. Previous research has shown that simulation results can be successfully transferred to a real robot if the simulator is reasonably accurate [13, 5]. The Khepera simulator is a discrete time simulator, and can run in real-time or much faster than real time. The user-supplied robot controller is called by the simulator at each time tick. The sensor model includes stochastic modeling of noise and responds similarly to the real sensors. The simulation environment includes some stochastic modeling of wheel slippage and acceleration. Michel [14] reports good transfer from the Khepera simulator to the real Khepera robot.

The user writes a set of subroutines that conform to a simple API. The subroutines are used by the simulator to initialize, run and shutdown the user control code. The simulator provides routines that allow the controller to query sensors and control effectors.

The user can create, save and restore a simulated environment with obstacles. However, the simulator does not include any settings to simulate failures. We added hooks into the simulator to allow us to simulate sensor failures, and we simulated effector failures within our own code.

2 Reinforcement Learning Behaviors

In our RL/POMDP implementation for Khepera, we implemented three basic behaviors: move forward, turn right, turn left. The robot is *always* moving and performing one of the behaviors. When turning, the number of time steps that the turn behavior is invoked determines how far the robot turns. The low level behaviors are responsible for dealing with obstacles and dealing with other problems that might arise, such as adjusting for sensor or actuator malfunction. Thus, the "move forward" behavior must adjust its activity if its path is blocked.

The goal of the RL modules is to maximize the reward given them by the POMDP planner. The reward is a function of how long it took to make a desired state transition; thus, a state transition in the shortest amount of time maximizes the reward. We did not reward the robot for making efficient use of battery power although this is a possibility as well.

The low level behaviors are not trained to avoid obstacles, but learn to do so because it is necessary for maximizing the reward given by the POMDP planner. As an example of this, suppose the policy for state s is to perform action a and the expected next state is s'. Upon entering state s, the POMDP planner activates action a and waits for a change in the robot's state. If a drives the robot into a wall, then it will not receive a reward. The only way it can receive the reward is to cause the robot's state to change to s'. Action a will receive zero reward until the robot leaves state s. When that happens, it will receive a positive reward only if the new state is s'.

Each of the behaviors is implemented in its own reinforcement learning module. All RL modules receive all data for the sensors and actuators, as well as a reinforcement signal from a POMDP planner at the higher level. Only one of the RL modules can be active at any given time. When it becomes active, an RL module gains complete control of the actuators and has full access to sensor data. All of the RL modules are implemented using the same underlying code. They perform different behaviors due to the rewards they receive.

The RL modules may receive more complete information from the sensors than the POMDP planner does. We pre-process the sensory input given to the POMDP, reducing it to only a few bits, in order to make the POMDP solvable. The RL modules do not need such restrictions and can deal with more data.

We use Q-learning with table lookup for approximating the value function. Although able to handle larger problems, neural network reinforcement learning is not guaranteed to converge and often performs poorly even on relatively simple problems [1]. Neural networks also have a tendency to over-train on the portion of the state space that it visits often and forget the value function for portions of the state space that it has not visited recently. This leads to a cycle where it learns to perform well for a time and then begins to perform poorly. Eventually it does re-learn the value function and performance improves again for a period. As the number of separate behaviors/networks is increased, the probability of one or more of them being in the forget part of the learn/forget cycle increases, which can cause severe problems as the system tries to recover from multiple failures caused by malfunctioning behaviors. Table lookup avoids these problems, and so we have adopted that representation. Fortunately, the problem so far is small enough for table lookup to be feasible.

3 POMDP planning

Markov Decision Processes have previously been examined as an approach to planning in robotic systems [10, 9]. A Markov Decision Process Model consists of a finite set of states S, a finite set of actions A, a set of actions $A(s) \subseteq A$ for each state $s \in S$ that can be executed in that state, and transition probabilities $p(s'|s,a) \forall s, s' \in S$ and $a \in A(s)$. The transition probabilities just specify the probability that the state is s' given that the previous state was s and action a was taken. Finally, the MDP includes a set of observation probabilities $P_i(o|s) \forall s \in S$ and $o \in O(i)$ which specify the probability that sensor i detects observation o in state s. In a *completely observable* MDP, it is assumed that the current state can be determined solely from information immediately available from sensors.

Completely observable MDP methods do not work well in many real-world robot planning tasks because robots can rarely determine their state from immediate sensor observations. Such determination is also usually computationally expensive. One way to deal with this uncertainty is by using a generalization of MDP known as Partially Observable Markov Decision Processes. Since the robot is unable to determine the current state based solely on the current sensor observation, it is necessary to update the state probability distribution $P(s)$ over states $s \in S$ using information about transition and observation probabilities. So, the state is expressed by a probability density function over the possible states. The probability distribution at each time step is calculated using

$$P(s) = K \sum_{s' \in S | a \in A(s')} p(s|s', a) \times P(s') \tag{1}$$

where K is a normalization factor which ensures that the probabilities sum to one. This is necessary since every action may not be defined in some states. When a sensor report comes in from sensor i, the probability distribution is further updated by

$$P(s) = K \times p_i(o|s') \times p(s') \tag{2}$$

The Khepera robot has 8 sensors that report distance values between 0 and 1024. We must reduce the amount of data. To do so, we group the sensors in pairs to make 4 pseudo sensors and then threshold the output from the pseudo sensors. This results in only 16 possible observations and makes the POMDP planner relatively robust to single sensor failures.

3.1 Solving a POMDP

Planning can be performed by selecting a state to be the goal and finding a *policy* that can be used to reach that state. The policy is simply a mapping that specifies what action to take in each state. For completely observable MDPs the representation for the policy is straightforward. Each state is mapped to exactly one action. For a robot to follow its policy, it simply determines what state it is in, looks up that state in the policy, and performs the specified action. For POMDP, however, the robot can never tell exactly what state it is in. The state is represented as a probability distribution, which could also be interpreted as a unit length vector. The policy for a POMDP is a set of vectors in the state space. Each vector is associated with an action. When using the policy to select an action, the robot calculates the vector dot product between the current belief state and every vector in the policy. Whichever vector has the largest dot product wins, and the action associated with that vector is used. The set of vectors that make up the policy form the upper surface of a piecewise linear function. This surface is known as the *value function*.

Several algorithms can compute the optimal policy for a POMDP, but all of the current algorithms are very computationally expensive. The Witness algorithm [11], is probably the best known, although several other algorithms converge more quickly. Of them, we implemented incremental pruning [4] as the algorithm underlying our POMDP planner.

Incremental pruning, like the other exact POMDP solution methods, does not scale well with the size of the state space. Therefore, it is necessary to keep the state space representation as simple as possible. For this work, we used a small area divided into 16 discrete positions and discretized the robot's heading into the four compass directions. This resulted in 64 possible states for the robot. Sensor

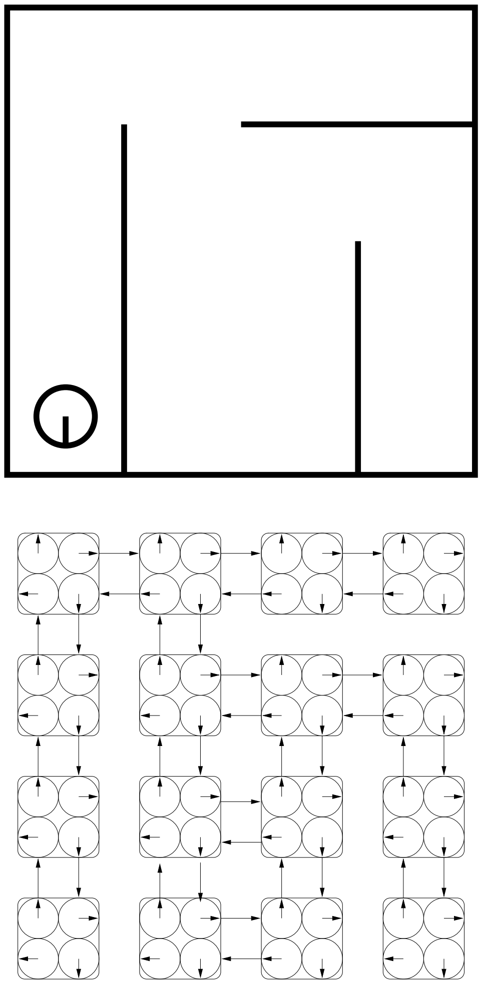

Figure 2: Environment and state space for our experiments.

information was reduced to 4 bits by combining the sensors in pairs and setting a threshold on the combined sensor values. The environment and state space is shown in Figure 2. Even with this simple state space, generating a policy required several days of computation on a Sun Ultra 2 workstation.

The major computational bottleneck is in the solution of a large number of linear programs. In order for incremental pruning to work, the linear program solver must have a very high degree of numerical stability. A great deal of effort was put into creating a linear program solver with sufficient numerical stability and speed. Our linear program solver is a sparse vector implementation based on the revised simplex method [3].

4 Interface between layers

When following a policy, the POMDP maintains a probability distribution over the possible states. This probability distribution is referred to as the *belief state*. The POMDP uses the current belief state to select a low level behavior to activate. Our implementation also tracks the state with the highest probability: the most likely current state. When the most likely current state changes, then we generate a non-zero reward for the behavior that is currently active. For all time steps where the most likely current state does not change, we generate a zero reward.

From the state transition probabilities, it is easy to find the expected next state when performing the current action in the most likely current state. If the most likely current state changes to the state that we want, then a reward of 1 is generated; otherwise a reward of -1 is generated. Note that there is a possibility for incorrect reward assignment. This is not a problem as reinforcement learning can still converge with some incorrect reward assignment.

5 Evaluation

The primary advantage of the RL/POMDP combination is its adaptability. Thus, we expect that our system should perform reliably even when the robot's hardware degrades. We hypothesize that the overall performance should degrade gracefully (i.e., roughly linearly with a shallow slope) as sensors and actuators gradually fail.

To test the hypothesis, we degraded sensor or actuator performance and measured the performance of two systems: one with the RL modules and the other with hand coded behaviors for low level control. The hand coded behaviors provide a control or baseline of performance to demonstrate task difficulty. As with the RL modules, we programmed three low level behaviors for the robot: turn left, turn right, and move forward. When the hand coded behaviors hit an obstacle, they turn to try to avoid it. We used the three hand coded behaviors to obtain the estimates of the state transition and observation probabilities needed by the POMDP planner. We placed the robot at a random position and orientation within each of the 64 possible states and recorded the effects of performing one of the actions from that starting position. We repeated this for ten trials in each of the 64 states. This data collection process provided an initial estimate of all state transition and observation probabilities to be used by the POMDP planner.

For simplicity and due to computational limitations, we chose state 13 (the robot's location in Figure 2) to be the goal for all of our experiments. We used the iterative pruning algorithm to find a policy that would direct the robot to state 13 from any initial state. This is a very large POMDP problem, so we used a high threshold on the value returned by the linear program solver. This probably resulted in a policy that was far from optimal, but was sufficient for our purposes. This policy was used for all of the experiments.

With the policy in place, we were ready to train the RL modules. We replaced the hand-coded behaviors with RL modules and trained them by placing the robot at a random position and orientation within each of the 64 states in turn, much as we did when generating the transition and observation probabilities. As before, we repeated this process ten times.

For each trial in an experiment, we degraded the simulated sensor or actuator performance by a set amount. For each system configuration (RL or hand coded), we started the simulated robot from every position and orientation in environment and recorded its performance. After a failure, the robot is re-started in the same position and orientation to try again. Each position and orientation is tried

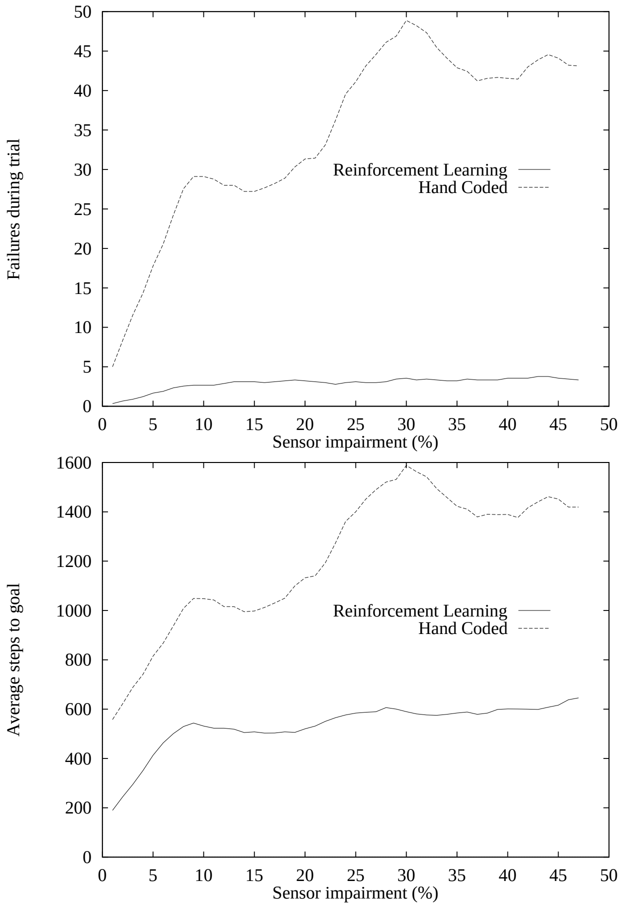

Figure 3: Response to gradual failure of all sensors.

until the robot successfully achieves the goal. Thus, for each simulator setting, we collect data from 64 instances of the robot achieving the goal.

Finally, in each case, we assessed performance on two metrics. *Failures during trial* counts the number of times that the robot was unable to reach the goal within 5000 time steps over all 64 configurations in a trial; thus, this is our estimator of reliability. *Average steps to goal* assesses efficiency: how long does it take to achieve the goal. Obviously, repeated failures in a trial causes this number to increase considerably.

5.1 Gradual Sensor Failure

Sensors can fail gradually as battery power is used up, when dust accumulates on the sensors, or sensors drift over time. To assess the effect of gradual sensor failure on system performance, we set a scaling factor on all of the sensors to restrict their range. If the range of the sensors was 90% of normal, then the sensors were said to be impaired by 10%. We started with all sensors fully operational (0% impairment). For each unit (1%) impairment from 0% to 50%, we directed the robot to go to the goal state from every possible starting location.

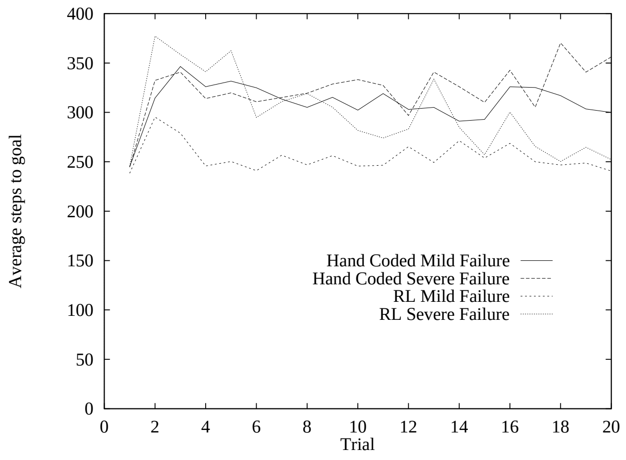

Figure 4: Response to intermittent failure of an actuator.

Figure 3 shows the results of this experiment. The curves have been mean smoothed with window of nine, which results in losing the end points. For the hand coded behaviors, as expected, the time steps to reach the goal is strongly correlated with the number of failures. In the hand coded behaviors, there is a bump reaction that is triggered when a threshold on the IR distance sensors is exceeded. Examination of the data showed that the most common failure was caused by the robot running into a corner and not triggering a bump reaction because the sensors could not generate a signal that was above threshold. Due to this threshold effect, the hand coded behaviors begin to perform poorly with even small amounts of sensor impairment.

The RL module shows excellent reliability; the number of failures during a trial increases gradually, staying under five, even with sensor impairment as high as 50%. The cost of the RL reliability is a sharp increase in cost (decrease in efficiency) at first, but the degradation rises more slowly after sensor impairment of 10%.

5.2 Intermittent Actuator Failure

Actuators may partially or intermittently fail due to loose contacts, shorted motor windings, or other causes. To assess the effect of intermittent partial actuator failure, we simulated loss of power in the right motor by adding a subroutine to reduce the value of the command signal by a fixed amount with a 50% probability. For example, although the robot control system issued a command for the motor to run at a speed of -5, the failure simulation routine may actually send a -4 command to the right motor. If the command signal was negative, then 1 was added; if the signal was positive, then -1 was added. We simulated mild failures with ± 1 and severe failures with ± 2 added to the right motor command signal.

We tested the response of both hand-coded and learned behaviors to mild and severe failures. For each test, we ran one trial with all sensors and actuators operational to establish a baseline and then ran 19 more trials while simulating an intermittent failure in the right motor.

Figure 4 shows the results of this experiment on the *average steps to goal* performance metric. The reinforcement learning behaviors did very well with mild failure; the performance initially degraded, but adapted quickly. For the severe failure, it took longer to recover, but after 20 trials, the performance was approaching the pre-failure level. As expected, the hand coded behaviors did not recover at all. Even with the severe failure, the robot was able to reach the goal on almost every trial with both hand coded and reinforcement learning behaviors; thus, the *failures during trial* data does not differentiate performance.

6 Conclusions

As we expected, the RL/POMDP combination exhibits robust behavior in the presence of sensor and actuator degradation. For our current test application, we were not inhibited by the known limitations of these methods, specifically amount of training required or problem size. However, our future work on this project is directed at scaling up the size of the problem and giving the architecture further autonomy.

The problem size is constrained in part by the limitations of the table lookup representation. Reinforcement learning is only guaranteed to converge to the value function when using a table lookup method. However, the table lookup method does not scale well with the size of the input space (sensors). To overcome the scaling problem, neural networks or some other approximation technique may be used to represent the value function. When used for value function approximation in reinforcement learning, the backpropagation neural network has one serious flaw: it makes non-local changes to the value function with each update. This property is very undesirable for reinforcement learning because over-training of the neural network in one area of the value function can cause the network to "forget" other areas that have already been learned. This can lead to poor performance overall and a learn/forget cycle where performance is good for some time and then degrades.

To increase the sensory input to the RL modules, we will be modifying the standard neural network based reinforcement learning to use a decision tree representation for the value function instead [16]. Like table lookup, decision trees subdivide the input space into intervals; unlike table lookup, the resolution of the intervals depends on the problem. The tree can be used to map an input vector to one of the leaf nodes, which corresponds to a region in the search space. Q-learning associates a value with each region. Our new approach avoids the cycling problem by providing stable and reliable convergence to the estimated value function even with large problems. The decision tree approach to function approximation is a state aggregation or *averaging* technique. This class of function approximation has been proven to converge [7].

To increase the size of the space for the POMDP, we will be investigating non-optimal solution algorithms. These algorithms operate much faster, but at the cost of approximate solutions.

To augment the autonomy, the architecture will be enhanced to add new behaviors as dictated by the demands of the environment. The combination of POMDP and reinforcement learning supports learning new low level behaviors as needed. The POMDP planner will recognize situations where the existing behavior set is not reliable for performing certain state transition and will instantiate a new RL module for use in that state transition. After some training, this specific behavior could be added to the set of available behaviors for use in other similar states.

7 Acknowledgments

This research was supported in part by National Science Foundation Career Award IRI-9624058. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation herein.

References

[1] Justin A. Boyan and Andrew W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D.S. Touretsky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems 7*, Cambridge, MA, 1995. MIT Press.

[2] Rodney A. Brooks. How to build complete creatures rather than isolated cognitive simulators. In Kurt VanLehn, editor, *Architectures for Intelligence: The Twenty-second Carnegie Mellon Symposium on Cognition*, chapter 8, pages 225–239. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1991.

[3] James Calvert and William Voxman. *Linear Programming*. Harcourt Brace Javanovich, 1989.

[4] Anthony Cassandra, Michael L. Littman, and Nevin L. Zhang. Incremental pruning: A simple, fast, exact algorithm for partially observable markov decision processes. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence*, 1997.

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

[6] R. James Firby, Roger E. Kahn, Peter N. Prokopowicz, and Michael J. Swain. An architecture for vision and action. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, volume 1, pages 72–79, 1995.

[7] Geoffrey J. Gordon. Stable function approximation in dynamic programming. Technical Report CMU-CS-95-103, Carnegie Mellon University, Computer Science Department, Pittsburgh, January 1995.

[8] Henry Hexmoor, Johan Lammens, Guido Caicedo, and Stuart C. Shapiro. Behavior based AI, cognitive processes, and emergent behaviors in autonomous agents. Technical Report 93-15, State University of New York at Buffalo, Department of Computer Science, 226 Bell Hall, Buffalo, New York 14260, April 1993.

[9] Sven Koenig, Richard Goodwin, and Reid G. Simmons. Robot navigation with Markov models: A framework for path planning and learning with limited computational resources. *Lecture Notes in Computer Science*, 1093:322, 1996.

[10] Sven Koenig and Reid G. Simmons. Risk-sensitive planning with probabilistic decision graphs. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*, pages 363–373, 1994.

[11] Michael L. Littman. The witness algorithm: Solving partially observable markov decision processes. Technical Report CS-94-40, Brown University, Department of Computer Science, Providence, RI, December 1994.

[12] Huan Liu, Thea Iberall, and George A. Beckey. Neural network architecture for robot hand control. In *Proceedings of IEEE International Conference on Neural Networks*. IEEE, July 1989.

[13] Sridhar Mahadevan and Jonathan Connell. Automatic programming of behaviour-based robots using reinforcement learning. *Artificial Intelligence*, 55(2/3):311–365, 1992.

[14] Olivier Michel. Khepera simulator. Available for download from `http://diwww.epfl.ch/-lami/team/michel/khep-sim/`, 1995.

[15] Gary H. Ogasawara. *RALPH-MEA: A Real-Time, Decision-Theoretic Agent Architecture*. PhD thesis, University of California, Berkeley, California 94720, 1993.

[16] Larry D. Pyeatt and Adele E. Howe. Decision tree function approximation in reinforcement learning. Tech Report TR CS-98-112, Colorado State University, Fort Collins, Colorado, October 1998.

[17] Claude Sammut. Automatic construction of reactive control systems using symbolic machine learning. *The Knowledge Engineering Review*, 11(1):27–42, 1996.

[18] Satinder Pal Singh. The efficient learning of multiple task sequences. In *Advances in Neural Information Processing Systems 3*, San Mateo, CA, 1991. Morgan Kaufman.

[19] Carme Torras. Neural learning for robot control. In A. G. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence*, pages 814–819, Chichester, August8–12 1994. ECAI, John Wiley and Sons.

[20] David E. Wilkins, Karen L. Myers, John D. Lowrance, and Leonard P. Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI*, 7(1):197–227, 1995.