Efficient Learning and Planning Within the Dyna Framework

Jing Peng*
Northeastern University

Ronald J. Williams†
Northeastern University

Sutton's Dyna framework provides a novel and computationally appealing way to integrate learning, planning, and reacting in autonomous agents. Examined here is a class of strategies designed to enhance the learning and planning power of Dyna systems by increasing their computational efficiency. The benefit of using these strategies is demonstrated on some simple abstract learning tasks.

Key Words: reinforcement learning; dynamic programming; sequential decision problems

Introduction

Many problems faced by an autonomous agent in an unknown environment can be cast in the form of *reinforcement learning* tasks. Recent work in this area has led to a clearer understanding of the relationship between algorithms found useful for such tasks and asynchronous approaches to dynamic programming (Bertsekas & Tsitsiklis, 1989), and this understanding has led, in turn, both to new results relevant to the theory of dynamic programming (Barto, Bradtke, & Singh, 1991; Watkins & Dayan, 1992; Williams & Baird, 1990) and to the creation of new reinforcement learning algorithms, such as Q-learning (Watkins, 1989) and Dyna (Sutton, 1990, 1991). Dyna was proposed as a simple but principled way to achieve more efficient reinforcement learning in autonomous agents. This article proposes enhancements designed to improve this learning efficiency still further. The Dyna architecture is interesting also because it provides a way to endow an agent with cognitive capabilities beyond simple trial-and-error learning while retaining the simplicity of the reinforcement learning approach. The enhancements studied here can be considered to help strengthen these cognitive capabilities. After first providing a concrete illustration of the kind of task any of these reinforcement learning methods are designed

* College of Computer Science, Northeastern University, Boston, MA 02115; jp@ccs.northeastern.edu.
† College of Computer Science, Northeastern University, Boston, MA 02115; rjw@ccs.northeastern.edu.

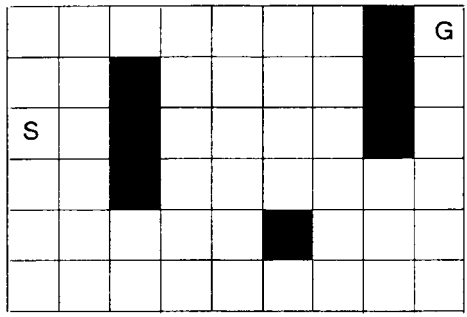


Figure 1
A two-dimensional maze navigation task.
(S = start; G = goal; *shaded cells* = barriers.)

to handle, we then outline the formal framework and existing techniques that serve as a backdrop for the new methods examined here.

2 An Illustrative Task

Assume that a learning agent is placed in the discretized two-dimensional maze shown in Figure 1. Shaded cells in this maze represent barriers, and the agent can occupy any other cell within this maze and can move about by choosing one of four actions at each discrete time tick. Each of these actions has the effect of moving the agent to an adjacent cell in one of the four compass directions—north, east, south, or west—except that any action that would ostensibly move the agent into a barrier cell or outside the maze has the actual effect of keeping the agent at its current location. The agent initially has no knowledge of the effect of its actions on what state (i.e., cell) it will occupy next, although it always knows its current state.

We also assume that this environment provides rewards to the agent and that this reward structure is initially unknown to the agent. For example, the agent may get a high reward any time it occupies the state marked G (for *goal*) in the maze. Loosely stated, the objective is for the learning agent to discover a policy, or assignment of choice of action to each state, that enables it to obtain the maximum rate of reward received over time. More specifically, every time the agent occupies the goal state we may place it back at S (for *start*). In this case, the agent's objective is simply to discover a shortest path from the start state to the goal state.

We use variations on this basic task to provide concrete illustrations of the different methods described here, and these also will serve as test problems on which to demonstrate the improvement obtained from use of the specific strategies we propose. We stress that all these methods apply more generally to a much wider range of learning control tasks, in which case the two-dimensional maze is simply a conceptual stand-in for the appropriate abstract state space for the actual problem, and the compass direction moves used here represent the various control actions available.

Task Formalization: Markov Decision Problem

This maze task can be viewed as a special case of a *Markov decision problem*, which has the following general form: At each discrete time step, the agent observes its current state x, uses this information to select action a, receives an immediate reward r, and then observes the resulting next state y, which becomes the current state at the next time step. In general, r and y may be random, but their probability distributions are assumed to depend only on x and a. For the maze task, we can give the agent an immediate reward of 1 for any state transition into the goal state and 0 for all other state transitions. Both the immediate reward and state transition functions are deterministic for this maze task, but later we will consider stochastic variants.

Informally, an agent can be considered to perform well in such a task if it can choose actions that tend to give it high rewards over the long run. A little more formally, at each time step k we would like the agent to select action $a(k)$ so that the *expected total discounted reward*

$$E\left\{\sum_{j=0}^{\infty}\gamma^j r(k+j)\right\}$$

is maximized, where $r(l)$ represents the reward received at time step l and γ is a fixed discount factor between 0 and 1. In the above maze task, it is not hard to see that optimal performance according to this criterion amounts to being able always to get to the goal state in as few moves as possible since a reward of 1 received j time steps in the future is only worth γ^j at the current time. A function that assigns to each state an action maximizing the expected total discounted reward is called an *optimal policy*.

Q-Learning and Dynamic Programming

One set of methods for determining an optimal policy is given by the theory of *dynamic programming* (Bertsekas, 1987). These methods entail first determining the *optimal state-value function*, V, which assigns to each state the expected total discounted reward obtained when an optimal policy is followed starting in that state. Following Watkins (1989), we can define a closely related function that assigns to each state-action pair a value measuring the expected total discounted reward obtained when the given action is taken in the given state and the optimal policy is followed thereafter.

That is, using the notation given above, with x the current state, a the current action, r the resulting immediate reward, and y the resulting next state,

$$\begin{aligned} Q(x,a) &= E\{r + \gamma V(y) \mid x, a\} \\ &= R(x,a) + \gamma \sum_y P_{xy}(a) V(y) \end{aligned} \tag{1}$$

where $R(x,a) = E\{r \mid x, a\}$, $V(x) = \max_a Q(x,a)$, and $P_{xy}(a)$ is the probability of making a state transition from x to y as a result of applying action a. Note that once we have this Q-function it is straightforward to determine the optimal policy. For any state x, the optimal action is simply $\arg\max_a Q(x,a)$.

Watkins's Q-learning algorithm is based on maintaining an estimate, $\hat{Q}$, of the Q-function and updating it so that equation 1, with estimated values substituted for the unknown actual values, comes to be more nearly satisfied for each state-action pair encountered. More precisely, the algorithm is as follows: At each transition from one time step to the next, the learning system observes the current state x, takes action a, receives immediate reward r, and observes the next state y. Assuming a tabular representation of these estimates, $\hat{Q}(x,a)$ is left unchanged for all state-action pairs not equal to (x,a) and

$$\hat{Q}(x,a) \leftarrow \hat{Q}(x,a) + \alpha \left[r + \gamma \hat{V}(y) - \hat{Q}(x,a)\right] \tag{2}$$

where $\alpha \in (0,1)$ is a learning rate parameter and $\hat{V}(y) = \max_b \hat{Q}(y,b)$. An estimate of the optimal action at any state x is obtained in the obvious way as $\arg\max_a \hat{Q}(x,a)$. This algorithm is an example of what Sutton (1988) has called a *temporal difference* method because the quantity $r + \gamma \hat{V}(y) - \hat{Q}(x,a)$ can be interpreted as the difference between two successive predictions of an appropriate expected total discounted reward. The general effect of such algorithms is to correct earlier predictions so that they more closely match later ones. Sutton has pointed out that the learning checkers-playing program of Samuel (1959) and Holland's (1986) bucket brigade algorithm are also of this general type. The relationship of this framework to the use of static evaluation functions in game-playing programs is particularly direct, and Tesauro (1992) has recently applied a version of these learning techniques in conjunction with neural network methods to obtain a highly successful and entirely self-taught backgammon-playing program.

Throughout this article, we will generally use the term *backup* to refer to a single application of equation 2. Each backup leads to updating of the Q-estimate for a single state-action pair. There are other related reinforcement learning algorithms in which a corresponding set of estimates of state values or state-action values are maintained and updated in a similar fashion, and it is common to apply the term *backup* in these cases as well. Here we use the more self-explanatory term *value*

function estimate update when referring generically to the corresponding step of any such algorithm.

Dyna

A key feature of the Q-learning algorithm is that when combined with sufficient exploration it can be guaranteed eventually to converge to an optimal policy without ever having to learn and use an internal model of the environment (Watkins, 1989; Watkins & Dayan, 1992). In adaptive control theoretic parlance, this qualifies it as a *direct* rather than an *indirect* method (Sutton, Barto, & Williams, 1992). From an AI point of view, what is interesting is that the eventual behavior of the system is as good as might be obtained if the system had carried out explicit *planning*, which can be thought of as simulation of possible future agent-environment interactions in an internal model to determine long-range consequences.

However, the cost of not using an internal model is that convergence to the optimal policy can be very slow. In a large state space, many backups may be required before the necessary information is propagated to a point at which it is important. In our maze example, it is clear that the optimal state value function is highest near the goal and declines exponentially as the number of steps required to reach the goal increases. Thus correct values must propagate from the goal state outward. Since Q-learning performs backups only for transitions as they are experienced, it is clear that many steps must be taken in the real world before the values are even close to correct near the start state. Even if the same path were taken on every trip from the start state to the goal state, the number of such trips required before Q-learning brings information about the goal state back to the start state is equal to the length of this path. In the maze example, this problem means that an agent using Q-learning will not perform well even after having arrived at the goal many hundreds of times.

To correct this weakness, Sutton (1990, 1991) has introduced the Dyna class of reinforcement learning architectures, in which a form of planning is performed in addition to learning. This means that such an architecture includes an internal world model along with mechanisms for learning it. However, the novel aspect of this approach is that planning is treated as being virtually identical to reinforcement learning except that while learning updates the appropriate value function estimates according to experience as it actually occurs, planning differs in that it updates these same value function estimates for simulated transitions chosen from the world model. It is assumed that there is computation time for several such updates during each actual step taken in the world, and the algorithm involves performing some fixed number of total updates during each actual time step. Figure 2 depicts the organization of a Dyna system.

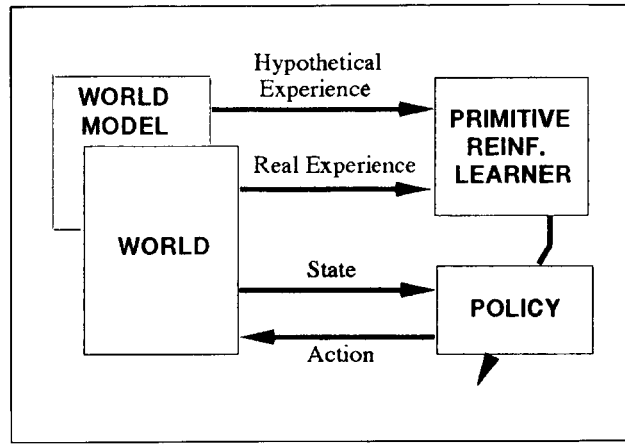


Figure 2
Overview of the Dyna architecture. The primitive reinforcement learner represents an algorithm like Q-learning. Not shown is the data path allowing the world model to learn to mimic the world.

In this article, we focus on a version of Dyna (called *Dyna-Q* by Sutton 1990, 1991) in which the underlying reinforcement learning algorithm is Q-learning. By performing several backups at each time step and avoiding the restriction that these backups occur only at current state transitions, such a system can perform much more effectively than simple Q-learning on tasks such as the maze example. In Sutton's work, the learned world model was simply a suitably indexed record (state, action, next-state, immediate-reward) of 4-tuples actually encountered in the past, and simulated experiences were obtained for planning purposes by selecting uniformly randomly from this record. His simulations demonstrated that such a system improves its performance much faster than a simple Q-learning system.

6 Queue-Dyna

Although Sutton's (1990, 1991) demonstrations of Dyna showed that use of several randomly chosen past experiences, along with the current experience for value function estimate updates, outperforms the use of current experience alone, it is natural to ask whether a more focused use of simulated experiences could lead to even better performance. The main contribution of this article is to introduce and study *queue-Dyna*, a version of Dyna in which value function estimate updates are prioritized, and only those having the highest priority are performed at each time step. Figure 3 depicts the organization of a queue-Dyna system.

Here we examine two specific versions of this strategy, employing slightly different criteria for prioritizing the potential updates. One of these applies to both deterministic and stochastic tasks, although the details of the respective algorithms differ somewhat, and the other currently applies only to deterministic tasks.

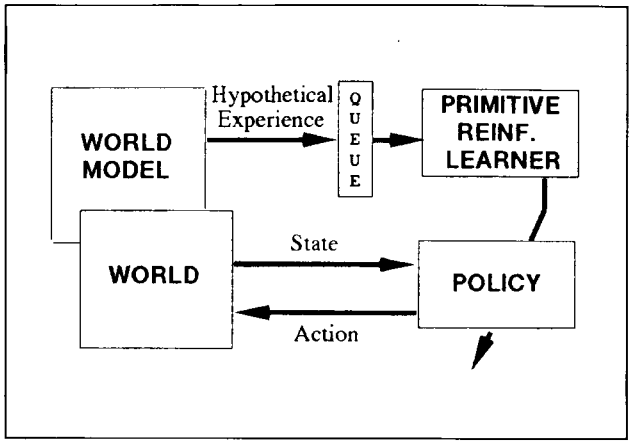


Figure 3
Overview of the queue-Dyna architecture. The primitive reinforcement learner represents an algorithm like Q-learning. Not shown is the data path allowing the world model to learn to mimic the world.

In all cases, an important aspect of the algorithm is the identification of places where the value function estimates may require updating, as will be more fully explained later. We call these places *update candidates*. In the Q-learning version we use here, these are state-action pairs for which a resulting next state and immediate reward prediction can be made. As in the work of Sutton, we take these to be simply state-action pairs that have been experienced at least once in the world.

6.1 Deterministic Environment

For any state-action pair (x, a) under consideration, define the *prediction difference* to be

$$r + \gamma \hat{V}(y) - \hat{Q}(x, a)$$

where r is the immediate reward and y the next state known from the model to result from state-action pair (x, a). Each update candidate is first checked to see if there is any significant prediction difference (based on comparison with an appropriate threshold). If there is, then its priority is determined and it is placed on the queue for eventual updating. The top several update candidates are then removed from the queue on each time step and the backup

$$\hat{Q}(x, a) \leftarrow r + \gamma \hat{V}(y) \tag{3}$$

is actually performed for each. Note that this is the same as equation 2 with $\alpha = 1$.

New update candidates are obtained from two main sources on each time step. For any backup actually performed, if the state-value estimate for the corresponding predecessor changes, then all transitions into that state become update candidates. The current transition also is made an update candidate, and this represents a further

1. Initialize $\hat{Q}(x,a)$ to 0 for all x and a and the priority queue $PQueue$ to empty.
2. Do Forever:
   (a) $x \leftarrow$ the current state.
   (b) Choose an action a that maximizes $\hat{Q}(x,a)$ over all a.
   (c) Carry out action a in the world. Let the next state be y and immediate reward be r.
   (d) Update world model from x,a,y, and r.
   (e) Compute $e = \left|r + \gamma\hat{V}(y) - \hat{Q}(x,a)\right|$. If $e \geq \delta$, insert (x,a,y,r) into $PQueue$ with key e.
   (f) If $PQueue$ is not empty, do planning:
      i. $(x',a',y',r') \leftarrow \mathbf{first}(PQueue)$.
      ii. Update:

$$\hat{Q}(x',a') = \hat{Q}(x',a') + \alpha(r' + \gamma\hat{V}(y') - \hat{Q}(x',a'))$$

      iii. For each predecessor x'' of x' do:
      - Compute $e = \left|r_{x''x'} + \gamma\hat{V}(x') - \hat{Q}(x'',a_{x''x'})\right|$. If $e \geq \delta$, insert $(x'', a_{x''x'}, x', r_{x''x'})$ into $PQueue$ with key e.

Figure 4
Main steps of a queue-Dyna algorithm using prediction difference magnitude as priority.

subtle difference between this approach and that investigated by Sutton. Whereas one of the several backups performed during a single time step was always reserved for the current state-action pair in Sutton's system, in our approach it is given no special priority.

One other interesting source of update candidates that can be used effectively in this approach is externally provided information about changes in the environmental reward or transition structure, as we examine later.

6.1.1 Priority Based on Prediction Difference Magnitude One straightforward way to assign priority to an update candidate, also recently explored extensively in independent work by Moore and Atkeson (1992), is simply to use the magnitude of the prediction difference. The larger the difference, the higher the priority. Figure 4 outlines the main steps of the queue-Dyna algorithm using this method for determining update priority.

If, as is typical, all initial Q-estimates are taken to equal 0, then in the maze

example the following behavior is observed when this algorithm is used. First, the queue is empty for every time step until the goal is discovered,[1] so no backups will occur during this initial exploration. Then, once the reward of 1 is obtained at the goal state, a prediction difference will occur at the Q-value for the state-action pair leading to the goal state, which will cause this transition to be placed on the queue as its only item. When this backup is performed, this predecessor state will have its state value changed from 0 to 1, which will cause any state-action pairs known to lead to it to be placed on the queue, and so forth. Because of the discount factor, it is clear that prediction differences will be smaller as distance from the goal increases, so the effect of this strategy is a breadth-first spread of value estimate updates outward from the recently discovered high-reward goal state.

6.1.2 Priority Based on Effect on Start-State Value Another interesting strategy that is appropriate for problems having a well-defined start state, S, is to try to estimate what the effect of any update would be on the estimated long-term reward at this start state. Our efforts at developing a general approach for this are still preliminary so we sketch the details here only for one useful subclass of problem where we have identified a sensible algorithm for this: deterministic environments having a single terminal positive reward, with all other rewards being 0. The maze task given earlier is an example. If we also assume that all initial value estimates are overly pessimistic, it then makes sense to use as the priority for updating $\hat{Q}(x, a)$,

$$\gamma^{d(x)}[r + \gamma \hat{V}(y)]$$

where $d(x)$ is an estimate of the minimum number of time steps required to go from S to x. There are various choices for $d(x)$ that one might consider using. For example, if the state space is equipped with a metric, one might choose $d(x)$ to be proportional to the metric distance between S and x. An even more interesting choice is to try to maintain an estimate of the length of the shortest path from S to x in a manner entirely analogous to the manner in which discounted total rewards are estimated. In particular, we can use another priority queue for the necessary updates and thereby very efficiently maintain fairly accurate estimates of the actual proximity of states visited to the start state.

The overall effect of a priority scheme of this type is that backups are directed in a more focused way. For example, in the maze task, once the goal is discovered, backups essentially proceed directly back to the start state before being done anywhere else. In fact, with Q-values initialized to 0, on discovery of the high-reward goal state G the algorithm performs value estimate updates in the same order as an A^* search

1 It is assumed that some form of exploratory behavior generally is available (especially when there is no clear-cut superior action, as when all Q-estimates are equal) so that the high-reward goal state eventually is discovered.

(Nilsson, 1980) would proceed from G backward toward S, with the d function serving as the heuristic estimate of remaining path length (backward) to S.

6.2 Stochastic Environment

Starting from equation (1) we can write:

$$\begin{aligned} Q(x,a) &= E\{r+\gamma V(y)|x,a\} \\ &= \sum_y P_{xy}(a)E\{r+\gamma V(y)|x,a,y\} \\ &= \sum_y P_{xy}(a)\left[E\{r|x,a,y\}+\gamma V(y)\right] \end{aligned} \tag{4}$$

For an environment that may be stochastic, it then makes sense to form a learned model by storing all past experiences together with appropriate counters and using this to compute estimates $[\hat{P}_{xy}(a)]$ of the relevant transition probabilities as well as estimates $[\hat{R}_y(x,a)]$ of $E\{r \mid x,a,y\}$, the expected immediate rewards conditioned on next state. We can then maintain estimates $[\hat{Q}_y(x,a)]$ of the bracketed expression on the right-hand side of equation 4 and update them using

$$\hat{Q}_y(x,a) \leftarrow \hat{R}_y(x,a)+\gamma\hat{V}(y) \tag{5}$$

with Q-value estimates computed using the following:

$$\hat{Q}(x,a) = \sum_y \hat{P}_{xy}(a)\hat{Q}_y(x,a)$$

In this case, a natural choice of prediction difference is

$$\hat{P}_{xy}(a)\left[\hat{R}_y(x,a)+\gamma\hat{V}(y)-\hat{Q}_y(x,a)\right]$$

and it is appropriate to count a single application of equation 5 as one backup.

Just as in the deterministic case, one useful way to assign priority to any update candidate (x,a) is to set it equal to the magnitude of this prediction difference, and this is used in one of the experiments described later. However, it is not straightforward to generalize the more focused method described earlier for tasks with a single starting state to stochastic environments, since both the probability of landing at a state and the number of actions taken must be considered in general.

7 Experimental Demonstrations

We conducted several experiments designed to test the performance of the methods proposed. In particular, the following three algorithms were used in these experiments: (1) Dyna with updating along randomly chosen transitions, together with the

current transition, as in Sutton's (1990, 1991) work; (2) queue-Dyna with priority determined by prediction difference magnitude; and (3) queue-Dyna with priority determined by estimated value of the start state. For this last algorithm, estimates of minimum distance from the starting state were maintained using an additional queue, as outlined previously. In the interest of brevity, henceforth we refer to these algorithms as *random-update Dyna*, *largest-first Dyna*, and *focused Dyna*, respectively.

All tasks studied involved variants on the maze task of Figure 1, as described earlier, with the agent having four possible actions at each state. In each case, there was a well-defined start state and a single goal state, with all rewards equal to 0 except on arrival at the goal state, when a reward of 100 was delivered. Arrival at the goal state was always followed by placing the agent back at the start state.

In every experiment, the agent's efficiency at negotiating the maze from the start state to a goal state at various points in the learning process was measured by interspersing test trials with the normal activities of the agent. Each such test trial consisted of placing the agent at the start state and letting it execute its currently best action for each state visited until a goal state was reached (or until an upper limit on number of moves allowed was reached). At the end of such a test trial, the agent resumed its usual activities from the state in which it had been before the test trial began. These test trials were performed solely to obtain this data, and no learning took place during them. Also, the normal activity of the agent included some random overriding of its current policy in order to foster exploration, and this was shut off during testing.

In all cases, each system was allowed five updates for every action taken in the world (excluding actions taken during test trials). The discount parameter used throughout was $\gamma = 0.95$, and the queue-Dyna threshold parameter δ was set to 0.0001. The learning rate parameter for Dyna was set at $\alpha = 0.5$, which was found experimentally to optimize its performance across tasks. All Q-values were initialized to 0 at the start of each experiment.

One set of experiments was performed on a series of related tasks, each using essentially the same deterministic maze environment shown in Figure 1 and described earlier except for the coarseness of representation of the states and actions. The number of states ranged from 47 for the coarsest partitioning to 6,016 for the finest. Two of these state spaces are shown in Figures 5 and 6.

Figure 7 shows the solution time (measured in number of backups) as a function of problem size for the three systems. These numbers represent averages over ten runs. Both forms of queue-Dyna clearly show slower growth with respect to state space size than does random-update Dyna. Figure 8 shows learning curves for the three systems, each an average over 100 runs, for the 752-state maze shown in Figure 6. Whereas Figure 7 does not show any clear difference between focused Dyna and

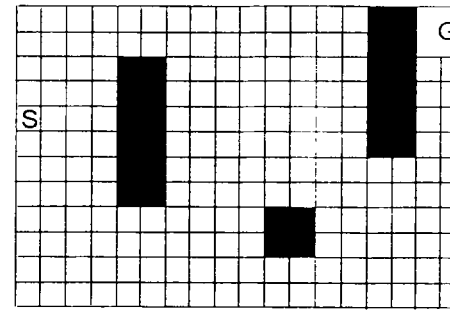


Figure 5
A more finely partitioned version of the maze in Figure 1. (S = start; G = goal; *shaded cells* = barriers.)

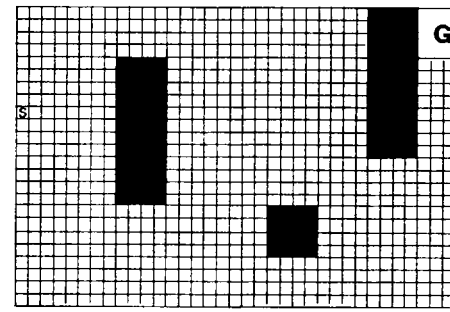


Figure 6
An even more finely partitioned version of the maze in Figure 1. (S = start; G = goal; *shaded cells* = barriers.)

largest-first Dyna in their ability to discover an optimal path, Figure 8 suggests that focused Dyna generally provides much more rapid improvement in the early stages of learning. The nature of the exploration strategy used may play a role in the specific results obtained, but this is an issue we have not addressed systematically in this work.

It is also possible in any Dyna system to make use of externally provided information about changes in the environmental reward or transition structure. In one interesting experiment along these lines, we first let an agent learn to find the shortest path to the goal state in the maze of Figure 9, with the cross-hatched cell occupied by a barrier. Then we opened up this cell and "told" the agent about it. More precisely, we updated its model to reflect correctly the effects of the three actions that could now move the agent into this cell and the four actions that could be taken from inside this cell. For queue-Dyna, we also placed these on the priority queue as if these actions had just been taken by the agent itself, and for the version that focuses backups toward the start state, we also placed these on the queue for updating of the minimum distance from the start state. Performance results for the three algorithms are shown in Figure 10.

In addition, we performed an experiment with a stochastic environment based on the maze of Figure 5 but with actions that have a somewhat random effect. In particular, the direction of actual movement matched that of the action selected

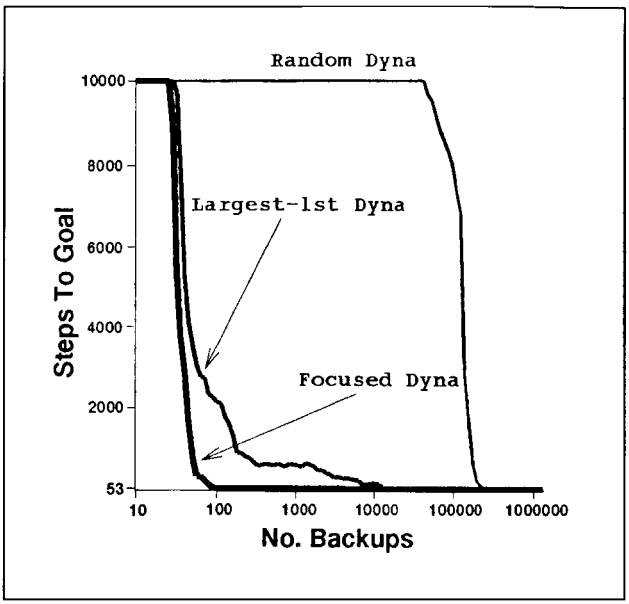


Figure 7
Growth of number of backups required until optimal performance with state space size.

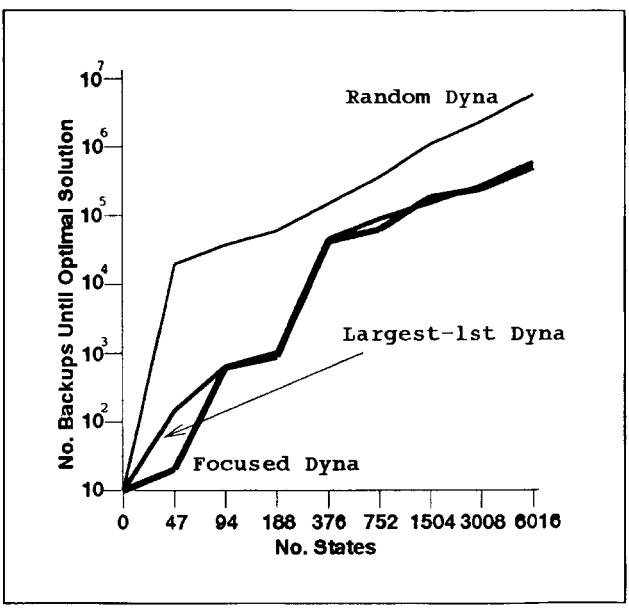


Figure 8
Performance on the maze in Figure 6.

with a two-thirds probability, but there was a one-third probability that the actual motion would be in one of the two adjacent compass directions instead, with the

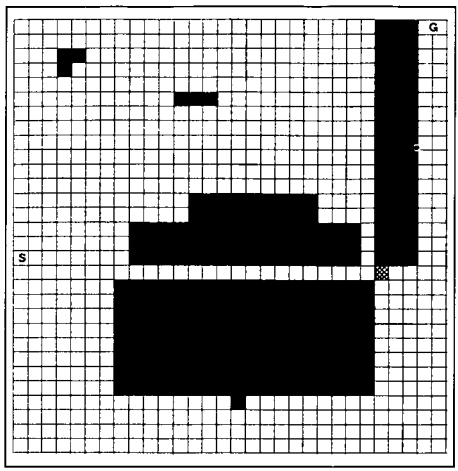


Figure 9
Maze used in one experiment. The agent first learned the shortest path to the goal with the cross-hatched block in place. This path, of length 66, goes along the bottom. Then this block was removed and the agent "told" about it. (S = start; G = goal; *shaded cells* = barriers.)

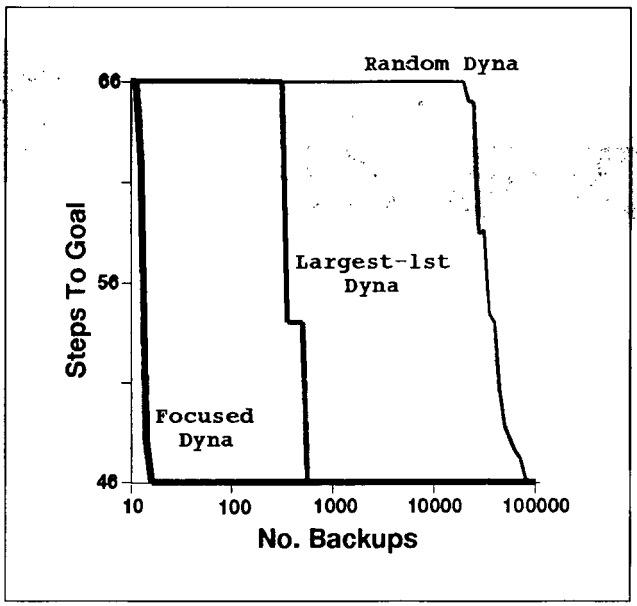


Figure 10
Performance on the task in Figure 9 after the agent was made aware of the removal of the cross-hatched block.

two "noisy" results being equally likely. For example, if the agent picked the action corresponding to the northward direction, its actual movement would be northward with a probability of two-thirds, westward with a probability of one-sixth, and eastward with a probability of one-sixth (where these represent "virtual" directions in the sense that they result in no movement at all whenever such movement is illegal).

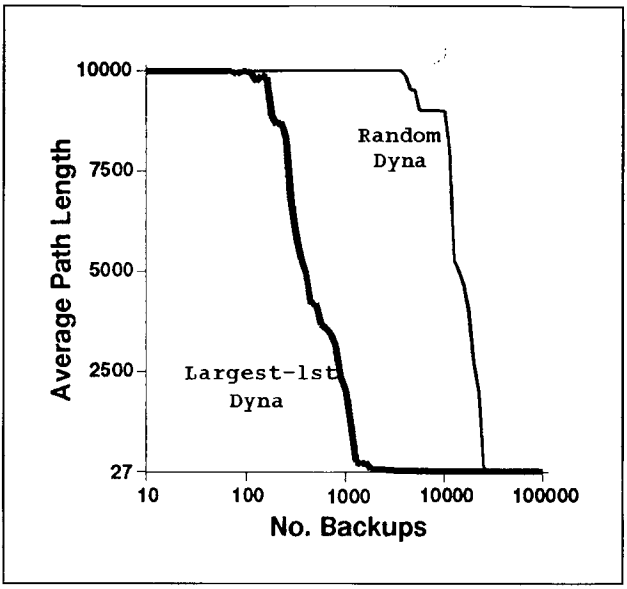


Figure 11
Performance on the stochastic maze task.

Figure 11 shows the results obtained when the largest-first and random-update Dyna systems were used on this task. The numbers obtained were averages over ten runs.

Issues and Further Directions

There are a number of relevant issues that we have not addressed here. Of substantial importance in realistic environments is the need for generalization in the world model. This presents some interesting challenges for the techniques investigated here, at least in their current form. One of the reasons that the queue-Dyna approach works well with the form of model used here is that the environmental transition and reward structure are assumed to be revealed one state-action pair at a time, and the model we have used is thus affected in a correspondingly simple and circumscribed way. If each state transition in the world could lead to model changes for many state-action pairs, it would generally be inappropriate to identify them all specifically. Instead, there needs to be a way to identify more concisely what direct consequences this may have on the estimated value function and to compute value function updates accordingly. A similar difficulty occurs if we imagine providing external information that is relevant in many parts of the state or state-action space.[2]

2 For example, suppose we tell the agent traversing the maze of Figure 6 at some point that any step in a southward direction taken from a state in the western half of the maze will henceforth incur a reward of −5.

An additional aspect considered in some depth by Moore and Atkeson (1992) in their work with this approach is the issue of efficient exploration. While we have highlighted the efficiency of queue-Dyna in terms of number of backups required and have emphasized its relationship to backward search from the goal, it is worth noting that a somewhat different behavior emerges when the initial value function estimates are overly optimistic compared with the actual reward structure. For example, consider the variant of the maze task where Q-values are once again initialized to 0 but the reward is 0 for any transition into the goal state and -1 for any other transition. Using priority based on prediction difference magnitude, one finds that more backups are required but exploration occurs automatically as a result of following the current policy. As Moore and Atkeson (1992) have emphasized, the efficiency with which queue-Dyna propagates value function updates leads to much more purposeful exploration in such cases, and the agent can thus discover high-reward states much more rapidly.

9 Conclusion

The combination of dynamic programming–based planning strategies with learned world models represents a promising approach to effective learning control (Moore, 1991), and the computational expense of more conventional planning makes incremental approaches such as Dyna (Sutton, 1990, 1991) especially appealing in this context. Here we have proposed that a prioritizing scheme be used to order the value function estimate updates in Dyna in order to improve its efficiency, and we have suggested two specific natural ways of setting this priority. In one, the largest known changes are made first, with predecessors of the corresponding states becoming the next candidates for consideration. In the other, the hierarchy of changes is based on what is effectively an estimate of the likelihood that the update is relevant for determining actions to be taken when starting from a given start state. We have demonstrated on simple tasks that use of such prioritizing schemes does indeed lead to drastic reductions in computational effort and corresponding dramatic improvements in performance of the learning agent. Thus we argue that not only does the Dyna framework represent a useful conceptual basis for integrated learning, planning, and reacting systems, but we are also optimistic that simple improvements such as those examined here can help give it sufficient power to allow it to serve as the basis for creating artificial autonomous agents that can adapt quickly to more complex and realistic environments.

Acknowledgments

We wish to thank Rich Sutton for his many valuable suggestions and continuing

encouragement. This work was supported by grant IRI-8921275 from the National Science Foundation.

References

Barto, A. G., Bradtke, S. J., & Singh, S. P. (1991). *Real-time learning and control using asynchronous dynamic programming* (COINS Technical Report No. 91-57). Amherst, MA: Department of Computer Science, University of Massachusetts.

Bertsekas, D. P. (1987). *Dynamic programming: Deterministic and stochastic models.* Englewood Cliffs, NJ: Prentice Hall.

Bertsekas, D. P., & Tsitsiklis, J. N. (1989). *Parallel and distributed computation: Numerical methods.* Englewood Cliffs, NJ: Prentice Hall.

Holland, J. H. (1986). Escaping brittleness: The possibility of general-purpose learning algorithms applied to rule-based systems. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol. 2). Los Altos, CA: Morgan Kaufmann.

Moore, A. W. (1991). Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued state-spaces. *Proceedings of the Eighth International Machine Learning Workshop.* San Mateo, CA: Morgan Kaufmann.

Moore, A. W., & Atkeson, C. G. (1993). Memory-based reinforcement learning: Efficient computation with prioritized sweeping. In S. J. Hanson, J. D. Cowan, & C. L. Criles (Eds.), *Advances in Neural Information Processing 5.* San Mateo, CA: Morgan Kaufmann.

Nilsson, N. J. (1980). *Principles of artificial intelligence.* San Mateo, CA: Morgan Kaufmann.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development, 3,* 210–229. (Reprinted in E. A. Feigenbaum & J. Feldman [Eds.] [1963], *Computers and thought.* New York: McGraw-Hill.)

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning, 3,* 9–44.

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. *Proceedings of the Seventh International Conference on Machine Learning.* San Mateo, CA: Morgan Kaufmann.

Sutton, R. S. (1991). Planning by incremental dynamic programming. *Proceedings of the Eighth International Machine Learning Workshop.* San Mateo, CA: Morgan Kaufmann.

Sutton, R. S., Barto, A. G., & Williams, R. J. (1992). Reinforcement learning is direct adaptive optimal control. *IEEE Control Systems Magazine, 12,* 19–22.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Advances in Neural Information Processing Systems, 4,* 259–266.

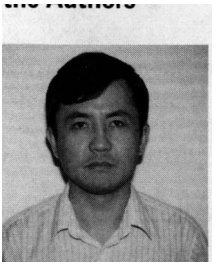
Watkins, C. J. C. H. (1989). *Learning from delayed rewards.* Unpublished doctoral dissertation, Cambridge University, Cambridge, England.

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning, 8,* 279–292.

Williams, R. J., & Baird, L. C., III (1990). A mathematical analysis of actor-critic architectures for learning optimal controls through incremental dynamic program-

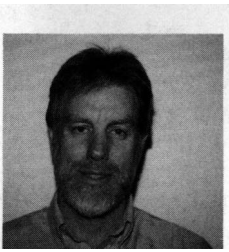
ming. *Proceedings of the Sixth Yale Workshop on Adaptive and Learning Systems.* New Haven, CT: Yale University Center for Systems Science.

About the Authors



Jing Peng

Jing Peng received the B.S. degree from Beijing Institute of Aeronautics and Astronautics, PRC in 1982, the M.A. degree from Brandeis University in 1987. Since 1988 he has been a Ph.D. student at Northeastern University working in the area of machine learning in general, connectionist learning and reinforcement learning in particular, and their applications to prediction and control under the supervision of professor Ronald J. Williams.



Ronald J. Williams

Ronald J. Williams is currently an associate professor of computer science at Northeastern University, a position he has held since 1986. He earned a B.S. in mathematics from the California Institute of Technology in 1966 and an M.A. and Ph.D. in mathematics from the University of California at San Diego in 1972 and 1975, respectively. From 1983 to 1986 he was a member of the Parallel Distributed Processing Research Group at UCSD's Institute for Cognitive Science, where he studied learning algorithms for artificial neural networks. His current research efforts are directed toward developing computationally efficient machine learning strategies for a variety of numerically based representations.