In Marco Wiering and Martijn van Otterlo, editors, *Reinforcement Learning: State of the Art*, Springer Verlag, 2011.

Learning and Using Models

Todd Hester and Peter Stone

Abstract As opposed to model-free RL methods, which learn directly from experience in the domain, model-based methods learn a model of the transition and reward functions of the domain on-line and plan a policy using this model. Once the method has learned an accurate model, it can plan an optimal policy on this model without any further experience in the world. Therefore, when model-based methods are able to learn a good model quickly, they frequently have improved sample efficiency over model-free methods, which must continue taking actions in the world for values to propagate back to previous states. Another advantage of model-based methods is that they can use their models to plan multi-step exploration trajectories. In particular, many methods drive the agent to explore where there is uncertainty in the model, so as to learn the model as fast as possible. In this chapter, we survey some of the types of models used in model-based methods and ways of learning them, as well as methods for planning on these models. In addition, we examine the typical architectures for combining model learning and planning, which vary depending on whether the designer wants the algorithm to run on-line, in batch mode, or in real-time. One of the main performance criteria for these algorithms is sample complexity, or how many actions the algorithm must take to learn. We examine the sample efficiency of a few methods, which are highly dependent on having intelligent exploration mechanisms. We survey some approaches to solving the exploration problem, including Bayesian methods that maintain a belief distribution over possible models to explicitly measure uncertainty in the model. We show some empirical comparisons of various model-based and model-free methods on two example domains before concluding with a survey of current research on scaling these methods up to larger domains with improved sample and computational complexity.

Todd Hester
Department of Computer Science, The University of Texas at Austin, 1616 Guadalupe, Suite 2.408, Austin, TX 78701, e-mail: todd@cs.utexas.edu

Peter Stone
Department of Computer Science, The University of Texas at Austin, 1616 Guadalupe, Suite 2.408, Austin, TX 78701, e-mail: pstone@cs.utexas.edu

1 Introduction

The reinforcement learning (RL) methods described in the book thus far have been *model-free* methods, where the algorithm updates its value function directly from experience in the domain. *Model-based* methods (or *indirect* methods), however, perform their updates from a model of the domain, rather than from experience in the domain itself. Instead, the model is learned from experience in the domain, and then the value function is updated by planning over the learned model. This sequence is shown in Figure 1. This planning can take the form of simply running a model-free method on the model, or it can be a method such as *value iteration* or *Monte Carlo Tree Search*.

The models learned by these methods can vary widely. Models can be learned entirely from scratch, the structure of the model can be given so that only parameters need to be learned, or a nearly complete model can be provided. If the algorithm can learn an accurate model quickly enough, model-based reinforcement learning can be more sample efficient (take fewer actions to learn) than model-free methods. Once an accurate model is learned, an optimal policy can be planned without requiring any additional experiences in the world. For example, when an agent first discovers a goal state, the values of its policy can be updated at once through planning over its new model that represents that goal. Conversely, a model-free method would have to follow a trajectory to the goal many times for the values to propagate all the way back to the start state. This higher sample efficiency typically comes at the cost of more computation for learning the model and planning a policy and more space to represent the model.

Another advantage of models is that they provide an opportunity for the agent to perform *targeted* exploration. The agent can plan a policy using its model to drive the agent explore particular states; these states can be states it has not visited or is uncertain about. A key to learning a model quickly is acquiring the right experiences needed to learn the model (similar to *active learning*). Various methods for exploring

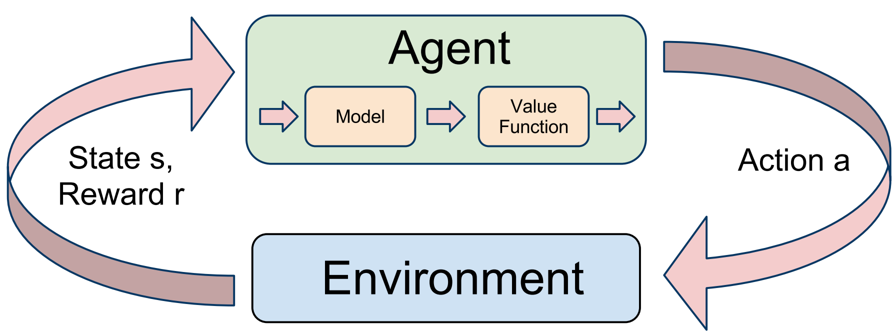


Fig. 1 Model-based RL agents use their experiences to first learn a model of the domain, and then use this model to compute their value function.

in this way exist, leading to fast learning of accurate models, and thus high sample efficiency.

The main components of a model-based RL method are the model, which is described in Section 2, and the planner which plans a policy on the model, described in Section 3. Section 4 discusses how to combine models and planners into a complete agent. The sample complexity of the algorithm (explained in Section 5) is one of the main performance criteria that these methods are evaluated on. We examine algorithms for factored domains in Section 6. Exploration is one of the main focuses for improving sample complexity and is explained in Section 7. We discuss extensions to continuous domains in Section 8, examine the performance of some algorithms empirically in Section 9, and look at work on scaling up to larger and more realistic problems in Section 10. Finally, we conclude the chapter in Section 11.

2 What is a model?

A model is the information an agent would need in order to simulate the outcome of taking an action in the Markov Decision Process (MDP). If the agent takes action a from state s, the model should predict the next state, s', and the reward, r. If the domain is stochastic, the model may provide a full probability distribution over next states, or it may be a *generative* model that simply provides a sample from the next state distribution. The model is updated as the agent interacts with its environment and is then used with a planning algorithm to calculate a policy.

A common approach to model-learning is to learn a tabular maximum likelihood model. In this case, the algorithm maintains a count, $C(s,a)$, of the times action a was taken from state s as well as a count, $C(s,a,s')$, of the number of times that each next state s' was reached from (s,a). The probability of outcome s' is then:

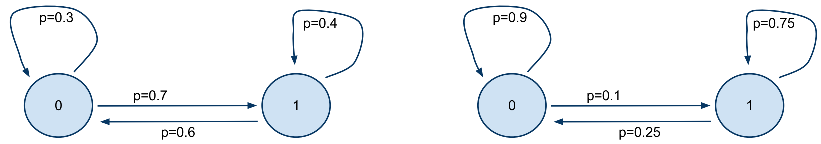
$$P(s'|s,a) = T(s,a,s') = C(s,a,s')/C(s,a)$$

The algorithm also maintains the sum of rewards it has received from each transition, $Rsum(s,a)$, and computes the expected reward for a particular state-action to be the mean reward received from that state-action:

$$R(s,a) = Rsum(s,a)/C(s,a)$$

This tabular model learning is very straightforward and is used in some common model-based algorithms such as R-MAX (Brafman and Tennenholtz, 2001).

Table 1 shows an example of the counts and predictions of a tabular model after 50 experiences in the domain shown in Figure 2. Based on the empirical counts of each experienced transition, the model can make predictions about the actual transition probabilities, $T(s,a,s')$, and reward, $R(s,a)$. While the model's predictions for action 0 from state 1 are exactly correct, the probabilities for the other state-action pairs are only approximately correct, but will improve with more samples.



(a) Action 0. (b) Action 1.

Fig. 2 This figure shows the transition probabilities for a simple two state MDP for both actions 0 and 1. We assume a reward of -1 for transitions leading to state 0, and a reward of +1 for transitions leading to state 1.

State	Action	C(s,a)	C(s,a,0)	C(s,a,1)	RSum(s,a)	T(s,a,0)	T(s,a,1)	R(s,a)
0	0	12	4	8	4	0.33	0.67	0.33
0	1	14	13	1	-12	0.93	0.07	-0.86
1	0	10	6	4	-2	0.6	0.4	-0.2
1	1	14	4	10	6	0.29	0.71	0.43

Table 1 This table shows an example of the model learned after 50 experiences in the domain shown in Figure 2.

Learning a tabular model of the domain can be viewed as learning a separate model for every state-action in the domain, which can require the agent to take many actions to learn a model. If we assume that the transition dynamics may be similar across state-action pairs, we can improve upon tabular models by incorporating *generalization* into the model learning. Model learning can be viewed as a supervised learning problem with (s,a) as the input and s' and r as the outputs the supervised learner is trying to predict. One of the benefits of incorporating generalization is that the supervised learner will make predictions about the model for unseen states based on the transitions it has been trained on.

A number of approaches from the field of adaptive dynamic programming use neural networks to learn a model of the domain (Prokhorov and Wunsch, 1997; Venayagamoorthy et al, 2002). These approaches train the neural network to estimate the next state values based on previous state values. The neural network model is used within an an actor-critic framework, with the predictions of the model used as inputs for the critic neural network that estimates the value of a given state.

Learning a model with a supervised learning technique inherently incorporates generalization into the model learning. With this approach, the supervised learner will make predictions about the transitions from unseen state-action pairs based on the state-action pairs it has been trained on. In many domains, it is easier to generalize the *relative* effects of transitions rather than their absolute outcomes. The relative transition effect, s^r, is the difference between the next state and the current state:

$$s^r = s' - s$$

For example, in a robotic control task, it may be easier to generalize that a given action increases the angle of a joint, rather than trying to generalize the absolute value of the joint angle after the action. This approach is taken in a number of algorithms. Random forests are used to model the relative transition effects in (Hester and Stone, 2010). Jong and Stone (2007) make predictions for a given state-action pair based on the average of the relative effects of its nearest neighbors.

One of the earliest model-based RL methods used locally weighted regression (LWR) to learn models (Schaal and Atkeson, 1994; Atkeson et al, 1997). All experiences of the agent were saved in memory, and when a given state-action was queried, a locally weighted regression model was formed to provide a prediction of the average next state for the queried state-action pair. Combined with a unique exploration mechanism, this algorithm was able to learn to control a robot juggling.

3 Planning

Once the agent has learned an approximate model of the domain dynamics, the model can be used to learn an improved policy. Typically, the agent would replan a policy on the model each time it changes. Calculating a policy based on a model is called *planning*. These methods can also be used for planning a policy on a provided model, rather than planning while learning the model online. One option for planning on the model is to use the dynamic programming methods described in Chapter 1, such as Value Iteration or Policy Iteration. Another option is to use Monte Carlo methods, and particularly Monte Carlo Tree Search (MCTS) methods, described below. The main difference between the two classes of methods is that the dynamic programming methods compute the value function for the entire state space, while MCTS focuses its computation on the states that the agent is likely to encounter soon.

3.1 Monte Carlo Methods

Chapter 1 describes how to use Monte Carlo methods to compute a policy while interacting with the environment. These methods can be used in a similar way on experiences simulated using a learned model. The methods simulate a full trajectory of experience until the end of an episode or to a maximum search depth. Each simulated trajectory is called a *roll-out*. They then update the values of states along that trajectory towards the discounted sum of rewards received after visiting that state. These methods require only a generative model of the environment rather than a full distribution of next states. A variety of MCTS methods exist which vary in how they choose actions at each state in their search.

Monte Carlo Tree Search methods build a tree of visited state-action pairs out from the current state, shown in Figure 3. This tree enables the algorithm to reuse information at previously visited states. In vanilla MCTS, the algorithm takes greedy actions to a specified depth in the tree, and then takes random actions from there until the episode terminates. This tree search focuses value updates on the states that MCTS visits between the agent's current state and the end of the roll-out,

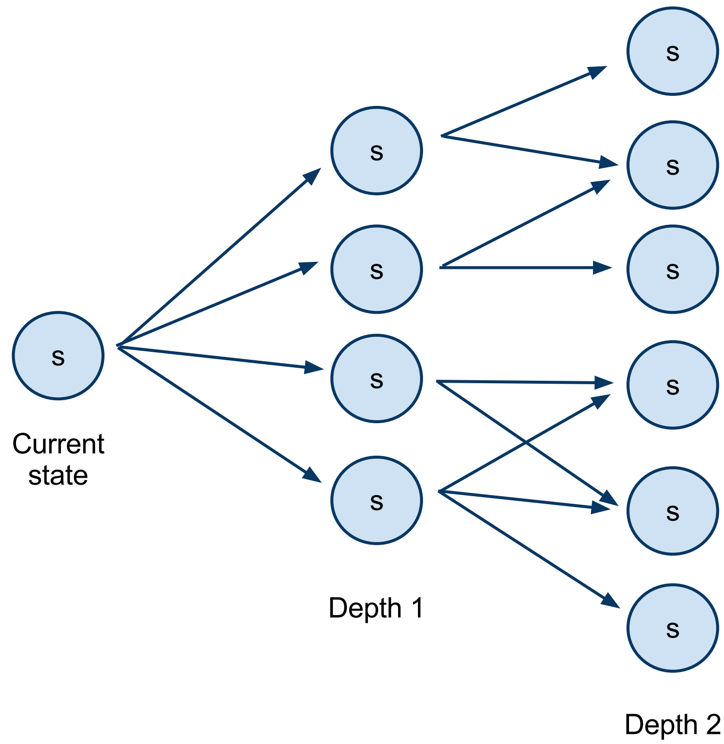


Fig. 3 This figure shows a Monte Carlo Tree Search roll-out from the agent's current state out to a depth of 2. The MCTS methods simulate a trajectory forward from the agent's current state. At each state, they select some action, leading them to a next state one level deeper in the tree. After rolling out to a terminal state or a maximum depth, the value of the actions selected are updated towards the rewards received following it on that trajectory.

which can be more efficient than planning over the entire state space as dynamic programming methods do. We will discuss a few variants of MCTS that vary in how they select actions at each state to efficiently find a good policy.

Sparse sampling (Kearns et al, 1999) was a pre-cursor to the MCTS planning methods discussed in this chapter. The authors determine the number, C, of samples of next states required to accurately estimate the value of a given state-action, (s,a), based on the maximum reward in the domain, R_{max}, and the discount factor, γ. They also determine the horizon h that must be searched to given the discount factor γ. The estimate of the value of a state-action pair at depth t is based on C samples of the next states at depth $t+1$. The value of each of these next states is based on C samples of each of the actions at that state, and so on up to horizon h. Instead of sampling trajectories one at a time like MCTS does, this method expands the tree one level deeper each step until reaching the calculated horizon. The algorithm's running time is $O((|A|C)^h)$. As a sampling-based planning method that is proven to converge to accurate values, sparse sampling provides an important theoretical basis for the MCTS methods that follow.

UCT (Kocsis and Szepesvári, 2006) is another MCTS variant that improves upon the running time of Sparse Sampling by focusing its samples on the most promising actions. UCT searches from the start state to the end, selecting actions based on upper confidence bounds using the UCB1 algorithm (Auer et al, 2002). The algorithm maintains a count, $C(s,d)$, of visits to each state at a given depth in the search, d, as well as a count, $C(s,a,d)$, of the number of times action a was taken from that state at that depth. These counts are used to calculate the upper confidence bound to select the action. The action selected at each step is calculated with the following equation (where C_p is an appropriate constant based on the range of rewards in the domain):

$$a = \text{argmax}_a Q^d(s,a) + 2C_p \sqrt{\frac{\log(C(s,d))}{C(s,a,d)}}$$

By selecting actions using the upper tail of the confidence interval, the algorithm mainly samples good actions, while still exploring when other actions have a higher upper confidence bound. Algorithm 1 shows pseudo-code for the UCT algorithm, which is run from the agent's current state, s, with a depth, d, of 0. The algorithm is provided with a learning rate, α, and the range of one-step rewards in the domain, r_{range}. Line 6 of the algorithm recursively calls the UCT method, to sample an action at the next state one level deeper in the search tree. Modified versions of UCT have had great success in the world of Go algorithms as a planner with the model of the game already provided (Wang and Gelly, 2007). UCT is also used as the planner inside several model-based reinforcement learning algorithms (Silver et al, 2008; Hester and Stone, 2010).

Algorithm 1 UCT (Inputs s, d, α, r_{range})

1: **if** TERMINAL or d == MAXDEPTH **then**
2:     **return** 0
3: **end if**
4: $a \leftarrow \text{argmax}_{a'}(Q^d(s,a') + 2 \cdot r_{range} \cdot \sqrt{\log(c(s,d))/c(s,a',d)})$
5: $(s',r) \leftarrow \text{SAMPLENEXTSTATE}(s,a)$
6: $retval \leftarrow r + \text{UCT}(s', d+1, \alpha)$
7: $c(s,d) \leftarrow c(s,d) + 1$
8: $c(s,a,d) \leftarrow c(s,a,d) + 1$
9: $Q^d(s,a') \leftarrow \alpha \cdot retval + (1-\alpha) \cdot Q(s,a',d)$
10: **return** *retval*

Having separately introduced the ideas of model learning (Section 2) and planning (Section 3), we now discuss the challenges that arise when combining these two together into a full model-based method, and how these challenges have been addressed.

4 Combining Models and Planning

There are a number of ways to combine model learning and planning. Typically, as the agent interacts with the environment, its model gets updated at every time step with the latest transition, $< s, a, r, s' >$. Each time the model is updated, the algorithm re-plans on it with its planner (as shown in Figure 4). This approach is taken by many algorithms (Brafman and Tennenholtz, 2001; Hester and Stone, 2009; Degris et al, 2006). However, due to the computational complexity of learning the model and planning on it, it is not always feasible.

Another approach is to do model updates and planning in batch mode, only processing them after every k actions, an approach taken in (Deisenroth and Rasmussen, 2011). However, this approach means that the agent takes long pauses between some actions while performing batch updates, which may not be acceptable in some problems.

DYNA (Sutton, 1990, 1991) is a reactive RL architecture. In it, the agent starts with either a real action in the world or a saved experience. Unlike value iteration, where Bellman updates are performed on all the states by iterating over the state space, here planning updates are performed on randomly selected state-action pairs. The algorithm updates the action-values for the randomly selected state-action using the Bellman equations, thus updating its policy. In this way, the real actions require only a single action-value update, while many model-based updates can take place in between the real actions. While the DYNA framework separates the model updates from the real action loop, it still requires many model-based updates for the policy to become optimal with respect to its model.

Prioritized Sweeping (Moore and Atkeson, 1993) (described in Chapter 1) improves upon the DYNA idea by selecting which state-action pairs to update based on priority, rather than selecting them randomly. It updates the state-action pairs in order, based on the expected change in their value. Instead of iterating over the entire state space, prioritized sweeping updates values propagating backwards across the state space from where the model changed.

Using MCTS to plan instead of dynamic programming techniques such as Value Iteration or Policy Iteration leads to a slightly different architecture. With the dy-

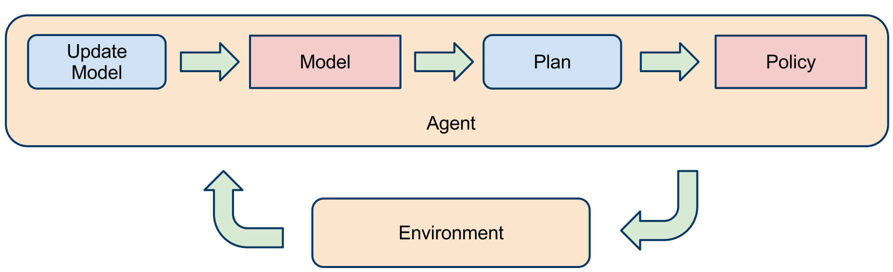


Fig. 4 Typically, model-based agent's interleave model learning and planning sequentially, first completing an update to the model, and then planning on the updated model to compute a policy.

namic programming techniques, a value function is computed for the entire state space after the model has changed. Once it is computed, dynamic programming does not need to be performed again unless the model changes. With sample-based MCTS methods that focus computation on the states the agent is likely to visit soon, planning roll-outs must occur every step to look ahead at likely future states. These methods are typically anytime algorithms, so each step can be limited to a given amount of computation if desired.

DYNA-2 is an architecture that extends the DYNA idea of updating its value function using both real and simulated experiences to using sample-based planning (Silver et al, 2008). DYNA-2 maintains separate linear function approximators to represent both *permanent* and *transient* memories of the value function. The permanent memory is updated through real experiences in the world. Between each action in the world, the transient memory is updated by running UCT on the agent's model of the world. The transient memory is focused on a narrower part of the state space (where UCT is sampling right now) and is used to augment the global value function. Actions are selected based on the combination of the permanent and transient value functions. This architecture combines a rough permanent global value function with a more refined transient local value function created by UCT planning each step.

For some problems, such as controlling robots or other physical devices, it may desirable to have a *real-time* architecture. In the previous architectures that we have described, it is possible for the model update or planning steps to take a significant amount of time. Hester et al (2011) developed a threaded architecture for real-time model-based RL that puts the model learning, planning, and acting in three parallel threads, shown in Figure 5. The threads communicate through four shared data structures: a list of experiences to be added to the model, a copy of the model that the planner uses, the current state for the planner to plan from, and the agent's policy. The model-learning thread runs in a loop, removing experiences from the list of new experiences, updating its model with these experiences, and then copying the model over to a version used by the planner. The planning thread runs a sample-based planning algorithm (such as UCT) on its version of the model, planning from the agent's current state, and updating the policy. The action thread adds the latest experience to the update list, sets the agent's current state for planning, and returns the best action for that state from its policy. The action thread is able to return an action immediately at whatever frequency is required. Depending on the action frequency and the length of time it takes to update the model, the model learning thread can be waiting for new experiences each time, or updating many new experiences into the model at a time. When the model is incorporating many experiences at a time, the algorithm's behavior is similar to batch learning methods, however the agent will continue taking actions while the batch updates takes place[1].

[1] Source code for this architecture is available at: `http://www.ros.org/wiki/rl_agent`

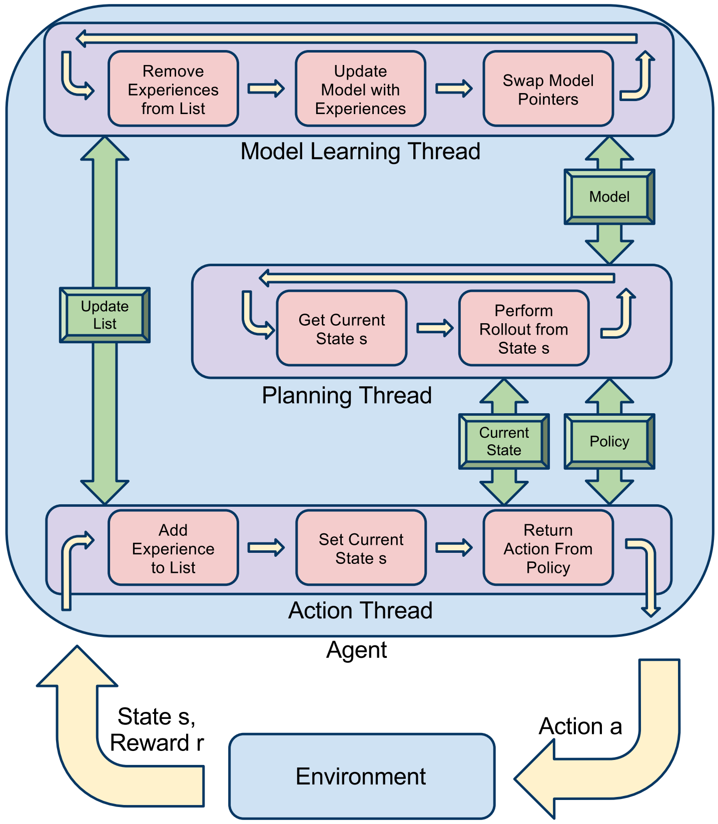


Fig. 5 The parallel architecture for real-time model-based RL proposed by Hester et al (2011). There are three separate parallel threads for model learning, planning, and acting. Separating model learning and planning from the action selection enables it to occur at the desired rate regardless of the time taken for model learning or planning.

5 Sample Complexity

A key benefit of model-based methods in comparison to model-free ones is that they can provide better *sample efficiency* than model-free methods. The *sample complexity* of an algorithm refers to the number of actions that an algorithm takes to learn an optimal policy. The sample complexity *of exploration* is the number of sub-optimal exploratory actions the agent must take. Kakade (2003) proves the lower bound for this sample complexity is $O(\frac{NA}{\varepsilon(1-\gamma)} log \frac{1}{\delta})$ for stochastic domains, where N is the number of states, A is the number of actions, γ is the discount factor, and the algorithm finds an ε-optimal policy (a policy whose value is within ε of optimal) with probability $1-\delta$.

Model-based methods can be more sample efficient than model-free approaches because they can update their value functions from their model rather than having to take actions in the real world. For this to be effective, however, the method must have a reasonable model of the world to plan on. Therefore, the main restriction on the sample efficiency of these methods is the number of actions it takes them to learn an accurate model of the domain. The methods presented in this section drive the agent to acquire the samples necessary to learn an accurate model quickly.

The Explicit Explore or Exploit (E^3) algorithm (Kearns and Singh, 1998) was the first algorithm to be proved to converge to a near-optimal policy in polynomial time[2]. The authors analyze their algorithm in terms of a mixing time, T, which is the horizon over which the value of the policy is calculated. For discounted domains,

$$T = \frac{1}{1-\gamma}$$

The algorithm maintains counts of visits to each state and considers states with fewer than m visits to be unknown. When the agent reaches an unknown state, it performs balanced wandering (taking the action it has selected the least from that state). If it reaches a known state, it attempts to plan an exploration policy that gets to an unknown state as quickly as possible. If the probability of reaching an unknown state is greater than $\varepsilon/(2R_{max})$, then this policy is followed. If that bound is not exceeded, then the authors have proven that planning an optimal policy *must* result in a policy that is within ε of optimal. Therefore, if the probability of reaching an unknown state does not reach that bound, the algorithm plans an approximate optimal policy and follows it. With probability no less than $1-\delta$, the E^3 algorithm will attain a return greater than the optimal value $-\varepsilon$, in a number of steps polynomial in N, T, R_{max}, $\frac{1}{\varepsilon}$, and $\frac{1}{\delta}$.

R-MAX (Brafman and Tennenholtz, 2001) is a similar model-based algorithm with proven bounds on its sample efficiency. It also learns a tabular maximum likelihood model and tracks known and unknown states. The key insight of the R-MAX algorithm is 'optimism in the face of uncertainty'. The algorithm replaces unknown transitions in its model with transitions to an *absorbing state*. An absorbing state is a state where all actions leave the agent in the absorbing state and provide the agent with the maximum reward in the domain, R_{max}. Doing so encourages the agent to explore all state-action pairs m times, allowing it to learn an accurate model and thus an optimal policy. This algorithm is simpler than the E^3 algorithm, and it employs an *implicit* explore or exploit approach rather than explicitly deciding between two policies as E^3 does. With probability no less than $1-\delta$, the R-MAX algorithm will attain an expected return with 2ε of optimal within a number of steps polynomial in N, A, T, $\frac{1}{\varepsilon}$, and $\frac{1}{\delta}$. Pseudo-code for the R-MAX algorithm is shown in Algorithm 2[3].

The principles used in R-MAX are formalized in a learning framework called Knows What It Knows (KWIK) (Li et al, 2008). For a model learning method to fit the KWIK framework, when queried about a particular state-action pair, it must always either make an accurate prediction, or reply "I don't know" and request a label for that example. KWIK algorithms can be used as the model learning methods in an

[2] Q-LEARNING, which was developed before E^3, was not proved to converge in polynomial time until after the development of E^3 (Even-dar and Mansour, 2001).

[3] Source code for R-MAX is available at: `http://www.ros.org/wiki/rl_agent`

Algorithm 2 R-Max (Inputs S, A, m, R_{max})

```
// Initialize s_r as absorbing state with reward R_max
for all a ∈ A do
    R(s_r, a) ← R_max
    T(s_r, a, s_r) ← 1
end for
Initialize s to a starting state in the MDP
loop
    Choose a = π(s)
    Take action a, observe r, s'
    // Update model
    Increment C(s, a, s')
    Increment C(s, a)
    RSUM(s, a) ← RSUM(s, a) + r
    if C(s, a) ≥ m then
        // Known state, update model using experience counts
        R(s, a) ← RSUM(s, a)/C(s, a)
        for all s' ∈ C(s, a, ·) do
            T(s, a, s') ← C(s, a, s')/C(s, a)
        end for
    else
        // Unknown state, set optimistic model transition to absorbing state
        R(s, a) ← R_max
        T(s, a, s_r) ← 1
    end if
    // Plan policy on updated model
    Call VALUE-ITERATION
    s ← s'
end loop
```

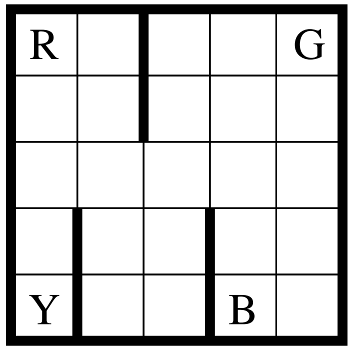
RL setting, as the agent can be driven to explore the states where the model replies "I don't know" to improve its model quickly. The drawback of KWIK algorithms is that they often require a large number of experiences to guarantee an accurate prediction when not saying "I don't know."

6 Factored Domains

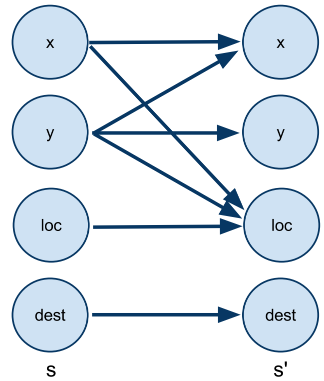
Many problems utilize a *factored* state representation where the state is represented by a vector of n state features:

$$s = < x_0, \ldots, x_n >$$

For example, an agent learning to control a robot could represent the state with a separate state feature for each joint. In many cases, the transition probabilities in factored domains are assumed to be determined by a Dynamic Bayesian Network (DBN). In the DBN model, each state feature of the next state may only be depen-



(a) Taxi Domain.



(b) DBN Transition Structure.

Fig. 6 6(a) shows the Taxi domain. 6(b) shows the DBN transition model for this domain. Here the x feature in state s' is only dependent on the x and y features in state s and the y feature is only dependent on the previous y. The passenger's *destination* is only dependent on its previous *destination*, and her current *location* is dependent on her location the step before as well as the taxi's x,y location the step before.

dent on some subset of features from the previous state and action. The features that a given state feature are dependent on are called its *parents*. The maximum number of parents that any of the state features has is called the *maximum in-degree* of the DBN. When using a DBN transition model, it is assumed that each feature transitions independently of the others. These separate transition probabilities can combined together into a prediction of the entire state transition with the following equation:

$$P(s'|s,a) = T(s,a,s') = \Pi_{i=0}^{n} P(x_i|s,a)$$

Learning the structure of this DBN transition model is known as the *structure learning problem*. Once the structure of the DBN is learned, the *conditional probabilities* for each edge must be learned. Typically these probabilities are stored in a conditional probability table, or CPT.

Figure 6 shows an example DBN for the Taxi domain (Dietterich, 1998). Here the agent's state is made up of four features: its x and y location, the passenger's *location*, and the passenger's *destination*. The agent's goal is to navigate the taxi to the passenger, pick up the passenger, navigate to her destination, and drop off the passenger. The y location of the taxi is only dependent on its previous y location, and not its x location or the current location or destination of the passenger. Because of the vertical walls in the domain, the x location of the taxi is dependent on both x and y. If this structure is known, it makes the model learning problem much easier, as the same model for the transition of the x and y variables can be used for any possible value of the passenger's location and destination.

Model-based RL methods for factored domains vary in the amount of information and assumptions given to the agent. Most assume a DBN transition model, and that the state features do transition independently. Some methods start with no knowledge of the model and must first learn the structure of the DBN and then learn the probabilities. Other methods are given the structure and must simply learn the probabilities associated with each edge in the DBN. We discuss a few of these variations below.

The DBN-E^3 algorithm (Kearns and Koller, 1999) extends the E^3 algorithm to factored domains where the structure of the DBN model is known. With the structure of the DBN already given, the algorithm must learn the probabilities associated with each edge in the DBN. The algorithm is able to learn a near-optimal policy in a number of actions polynomial in the number of parameters of the DBN-MDP, which can be exponentially smaller than the number of total states.

Similar to the extension of E^3 to DBN-E^3, R-MAX can be extended to FACTORED-R-MAX for factored domains where the structure of the DBN transition model is given (Guestrin et al, 2002). This method achieves the same sample complexity bounds as the DBN-E^3 algorithm, while also maintaining the implementation and simplicity advantages of R-MAX over E^3.

Structure Learning Factored R-MAX (SLF-R-MAX) (Strehl et al, 2007) applies an R-MAX type approach to factored domains where the structure of the DBN is not known. It learns the structure of the DBN as well as the conditional probabilities when given the maximum in-degree of the DBN. The algorithm enumerates all possible combinations of input features as elements and then creates counters to measure which elements are relevant. The algorithm makes predictions when a relevant element is found for a queried state; if none is found, the state is considered unknown. Similar to R-MAX, the algorithm gives a bonus of R_{max} to unknown states in value iteration to encourage the agent to explore them. The sample complexity of the algorithm is highly dependent on the maximum in-degree, D, of the DBN. With probability at least $1-\delta$, the SLF-R-MAX algorithm's policy is ε-optimal except for at most k time steps, where:

$$k = O\left(\frac{n^{3+2D} A D ln(\frac{nA}{\delta}) ln(\frac{1}{\delta}) ln(\frac{1}{\varepsilon(1-\gamma)})}{\varepsilon^3 (1-\gamma)^6}\right)$$

Here, n is the number of factors in the domain, D is the maximum in-degree in the DBN, and γ is the discount factor.

Diuk et al (2009) improve upon the sample complexity of SLF-R-MAX with the k-Meteorologists R-MAX (MET-R-MAX) algorithm by introducing a more efficient algorithm for determining which input features are relevant for its predictions. It achieves this improved efficiency by using the mean squared error of the predictors based on different DBNs. This improves the sample complexity for discovering the structure of the DBN from $O(n^{2D})$ to $O(n^D)$. The overall sample complexity bound for this algorithm is the best known bound for factored domains.

Chakraborty and Stone (2011) present a similar approach that does not require knowledge of the in-degree of the DBN called Learn Structure and Exploit with R-MAX (LSE-R-MAX). It takes an alternative route to solving the structure learning problem in comparison to MET-R-MAX by assuming knowledge of a planning horizon that satisfies certain conditions, rather than knowledge of the in-degree.

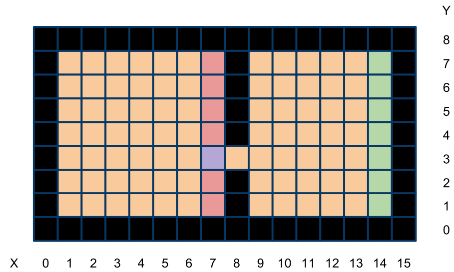
With this assumption, it solves the structure learning problem in sample complexity bounds which are competitive with MET-R-MAX, and it performs better empirically in two test domains.

Decision trees present another approach to the structure learning problem. They are able to naturally learn the structure of the problem by using information gain to determine which features are useful to split on to make predictions. In addition, they can generalize more than strict DBN models can. Even for state features that are parents of a given feature, the decision tree can decide that in portions of the state space, that feature is not relevant. For example, in taxi, the location of the passenger is only dependent on the location of the taxi when the *pickup* action is being performed. In all other cases, the location of the passenger can be predicted while ignoring the taxi's current location.

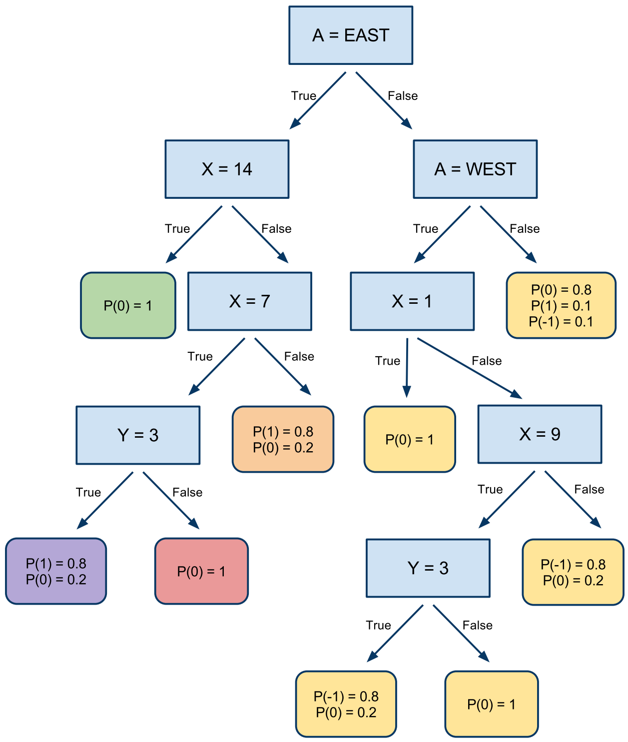
Decision trees are used to learn models in factored domains in the SPITI algorithm (Degris et al, 2006). The algorithm learns a decision tree to predict each state feature in the domain. It plans on this model using Structured Value Iteration (Boutilier et al, 2000) and uses ϵ-greedy exploration. The generalization in its model gives it better sample efficiency than many methods using tabular or DBN models in practice. However, there are no guarantees that the decision tree will fully learn the correct transition model, and therefore no theoretical bounds have been proven for its sample efficiency.

RL-DT is another approach using decision trees in its model that attempts to improve upon SPITI by modeling the relative transitions of states and using a different exploration policy (Hester and Stone, 2009). By predicting the relative change in each feature, rather than its absolute value, the tree models are able to make better predictions about the transition dynamics for unseen states. In addition, the algorithm uses a more directed exploration scheme, following R-MAX type exploration of driving the agent to states with few visits until the agent finds a state with reward near R_{max}, at which point it switches to exploiting the policy computed using its model. This algorithm has been shown to be effective on gridworld tasks such as Taxi, as well as on humanoid robots learning to score penalty kicks (Hester et al, 2010).

Figure 7 shows an example decision tree predicting the relative change in the x variable of the agent in the given gridworld domain. The decision tree is able to split on both the actions and the state of the agent, allowing it to split the state space up into regions where the transition dynamics are the same. Each leaf of the tree can make probabilistic predictions based on the ratio of experienced outcomes in that leaf. The grid is colored to match the leaves on the left side of the tree, making predictions for when the agent takes the *east* action. The tree is built on-line while the agent is acting in the MDP. At the start, the tree will be empty, and will slowly be refined over time. The tree will make predictions about broad parts of the state space at first, such as what the $EAST$ or $WEST$ actions do, and eventually refine itself to have leaves for individual states where the transition dynamics differ from the global dynamics.



(a) Two room gridworld domain.



(b) Decision tree model predicting the change in the x feature (Δx) based on the current state and action.

Fig. 7 This figure shows the decision tree model learned to predict the change in the x feature (or Δx). The two room gridworld is colored to match the corresponding leaves of the left side of the tree where the agent has taken the *east* action. Each rectangle represents a split in the tree and each rounded rectangle represents a leaf of the tree, showing the probabilities of a given value for Δx. For example, if the action is *east* and $x = 14$ we fall into the green leaf on the left, where the probability of Δx being 0 is 1.

7 Exploration

A key component of the E^3 and R-MAX algorithms is how and when the agent decides to take an exploratory (sub-optimal) action rather than exploit what it knows in its model. One of the advantages of model-based methods is that they allow the agent to perform *directed* exploration, planning out multi-step exploration policies rather than the simple ε-greedy or softmax exploration utilized by many model-free methods. Both the E^3 and R-MAX algorithms do so by tracking the number of visits to each state and driving the agent to explore all states with fewer than a given number of visits. However, if the agent can measure uncertainty in its model, it can drive exploration without depending on visit counts. The methods presented in this section follow this approach. They mainly vary in two dimensions: 1) how they measure uncertainty in their model and 2) exactly how they use the uncertainty to drive exploration.

Model-based Bayesian RL methods (Chapter 3) seek to solve the exploration problem by maintaining a posterior distribution over possible models. This approach is promising for solving the exploration problem because it provides a principled way to track the agent's uncertainty in different parts of the model. In addition, with this explicit uncertainty measure, Bayesian methods can plan to explore states that have the potential to provide future rewards, rather than simply exploring states to reduce uncertainty for its own sake.

Duff (2003) presents an 'optimal probe' that solves the exploration problem optimally, using an augmented statespace that includes both the agent's state in the world and its beliefs over its models (called a *belief state MDP*). The agent's model includes both how an action will affect its state in the world, and how it will affect the agent's beliefs over its models (and what model it will believe is most likely). By planning over this larger augmented state space, the agent can explore optimally. It knows which actions will change its model beliefs in significant and potentially useful ways, and can ignore actions that only affect parts of the model that will not be useful. While this method is quite sample efficient, planning over this augmented state space can be very computationally expensive. Wang et al (2005) make this method more computationally feasible by combining it with MCTS-like planning. This can be much more efficient than planning over the entire state space, as entire parts of the belief space can be ignored after a few actions. BEETLE (Poupart et al, 2006) takes a different approach to making this solution more computationally feasible by parametrizing the model and tying model parameters together to reduce the size of the model learning problem. However, this method is still impractical for any problem with more than a handful of states.

Other Bayesian methods use the model distribution to drive exploration without having to plan over a state space that is augmented with model beliefs. Both Bayesian DP (Strens, 2000) and Best of Sampled Set (BOSS) (Asmuth et al, 2009) approach the exploration problem by sampling from the distribution over world models and using these samples in different ways.

Bayesian DP samples a single model from the distribution, plans a policy using it, and follows that policy for a number of steps before sampling a new model. In between sampling new models, the agent will follow a policy consistent with the sampled model, which may be more exploratory or exploitative depending on the sampled model.

BOSS on the other hand, samples k models from the model posterior distribution whenever it collects sufficient data for some part of the model. It then merges the models into a single optimistic model with the same state space, but an augmented action space of kA actions. Essentially, there is an action modeled by each of the predictions of the k models for each of the A actions. Planning over this optimistic model allows the agent to select at each state an action from any of the k sampled models. The agent is driven to explore areas of the state space where the model is uncertain because due to the variance in the model distribution in that part of the state space, at least one of the sampled models is likely to be optimistic.

Model Based Bayesian Exploration (MBBE) is a similar approach taken by Dearden et al (1999). They maintain a distribution over model parameters and sample and solve k models to get a distribution over action-values. They use this distribution over action-values to calculate the value of perfect information (VPI) that is based on the uncertainty in the action-values along with the expected difference in reward that might be gained. They use the VPI value to give a bonus value to actions with high information gain.

These three methods (Bayesian DP, BOSS, and MBBE) provide three different approaches to sampling from a Bayesian distribution over models to solve the exploration problem. While these methods provide efficient exploration, they do require the agent to maintain Bayesian distributions over models and sample models from the distribution. They also require the user to create a well-defined model prior.

Kolter and Ng (2009) attempt to extend the polynomial convergence proofs of E^3 and R-MAX to Bayesian methods with an algorithm called Bayesian Exploration Bonus (BEB). BEB follows the Bayesian optimal policy after a polynomial number of steps, using exploration rewards similar to R-MAX to drive the agent towards the Bayesian optimal policy. One drawback of BEB is that it requires the transition functions to be drawn from a Dirichlet distribution.

Model Based Interval Estimation (MBIE) (Wiering and Schmidhuber, 1998; Strehl and Littman, 2005) is a similar approach that looks at the distribution over transition probabilities rather than action-value distributions. The algorithm maintains statistical confidence intervals over the transition probabilities where transitions that have been sampled more often have tighter distributions around the same mean. When selecting actions, the algorithm computes the value function according to the transitions probabilities that are both within the calculated confidence interval *and* result in the highest policy values. Effectively, MBIE solves for the maximum over likely transition probabilities in addition to the maximum over individual actions.

SLF-R-MAX, MET-R-MAX, and LSE-R-MAX perform directed exploration on factored domains (Strehl et al, 2007; Diuk et al, 2009; Chakraborty and Stone, 2011). They use an R-MAX type exploration bonus to explore to determine the structure of the DBN transition model and to determine the conditional probabilities. They can explore less than methods such as R-MAX since their DBN model should be able to determine that some features are not relevant for the predictions of certain features. With fewer relevant features, the space of states where the relevant features differ is much smaller than when assuming each state can have different transition dynamics.

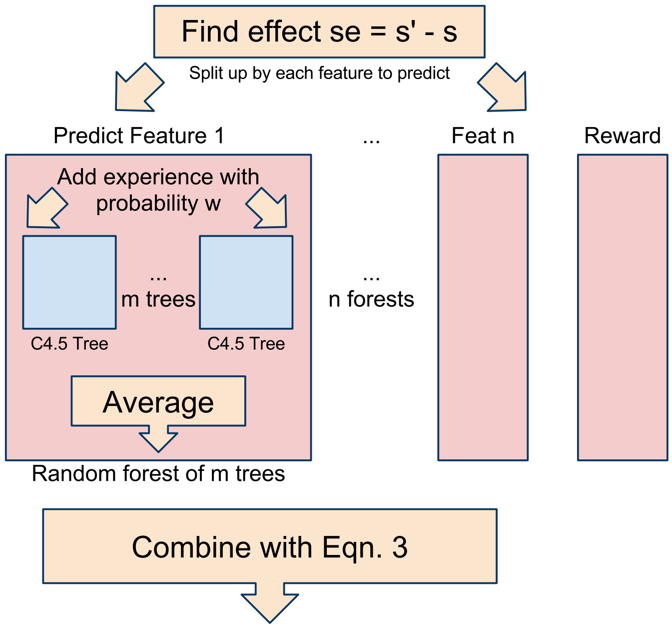


Fig. 8 *TEXPLORE's Model Learning.* This figure shows how the TEXPLORE algorithm (Hester and Stone, 2010) learns a model of the domain. The agent calculates the difference between s' and s as the transition effect. Then it splits up the state vector and learns a random forest to predict the change in each state feature. Each random forest is made up of stochastic decision trees, which are updated with each new experience with probability w. The random forest's predictions are made by averaging each tree's predictions, and then the predictions for each feature are combined into a complete model of the domain.

With tabular models, it is clear that the agent must explore each state-action in order to learn an accurate model for each one. This task can be accomplished through random exploration, but it is more effective to direct exploration to the unvisited states like R-MAX does. When dealing with models that generalize such as decision trees or DBNs, however, we do not want the agent exploring every state. One of the benefits of these models is that they are able to make reasonable predictions about unseen state-action pairs. Therefore, instead of exploring every state-action pair, it would be more efficient for the agent to use an approach similar to the Bayesian methods such as BOSS described above.

The TEXPLORE algorithm (Hester and Stone, 2010) is a model-based method for factored domains that attempts to accomplish this goal. It learns multiple possible decision tree models of the domain, in the form of a random forest (Breiman, 2001). Figure 8 shows the random forest model. Each random forest model is made up of m possible decision tree models of the domain and trained on a subset of the agent's experiences. The agent plans on the average of these m models and can include an exploration bonus based on the variance of the models' predictions.

The TEXPLORE agent builds k hypotheses of the transition model of the domain, similar to the multiple sampled models of BOSS or MBBE. TEXPLORE combines the models differently, however, creating a merged model that predicts next state distributions that are the average of the distributions predicted by each of the models. Planning over this average distribution allows the agent to explore promising transitions when their probability is high enough (when many of the models predict the promising outcome, or one of them predicts it with high probability). This is similar to the approach of BOSS, but takes into account the number of sampled models that predict the optimistic outcome. In TEXPLORE, the prediction of a given model has probability $\frac{1}{k}$, while the extra actions BOSS creates are always assumed to transition as that particular model predicts. This difference becomes clear with an example. If TEXPLORE's models disagree and the average model predicts there is a small chance of a particular negative outcome occurring, the agent will avoid it based on the chance that it may occur. The BOSS agent, however, will simply select an action from a different model and ignore the possibility of these negative rewards. On the other hand, if TEXPLORE's average model predicts a possibility of a high-valued outcome occurring, it may be worth exploring if the value of the outcome is high enough relative to its probability. TEXPLORE has been used in a gridworld with over $300,000$ states (Hester and Stone, 2010) and run in real-time on an autonomous vehicle (Hester et al, 2011)[4].

Schmidhuber (1991) tries to drive the agent to where the model has been improving the most, rather than trying to estimate where the model is poorest. The author takes a traditional model-based RL method, and adds a confidence module, which is trained to predict the absolute value of the error of the model. This module could be used to create intrinsic rewards encouraging the agent to explore high-error state-action pairs, but then the agent would be attracted to noisy states in addition to poorly-modeled ones. Instead the author adds another module that is trained to predict the changes in the confidence module outputs. Using this module, the agent is driven to explore the parts of the state space that most improve the model's prediction error.

Baranes and Oudeyer (2009) present an algorithm based on a similar idea called Robust Intelligent Adaptive Curiosity (R-IAC), a method for providing intrinsic reward to encourage a developing agent to explore. Their approach does not adopt the RL framework, but is similar in many respects. In it, they split the state space into regions and learn a model of the transition dynamics in each region. They maintain an estimate of the prediction error for each region and use the gradient of this error as the intrinsic reward for the agent, driving the agent to explore the areas where the prediction errors are improving the most. Since this approach is not using the RL framework, their algorithm selects actions only to maximize the immediate reward, rather than the discounted sum of future rewards. Their method has no way of incorporating external rewards, but it could be used to provide intrinsic rewards to an existing RL agent.

[4] Source code for the TEXPLORE algorithm is available at: `http://www.ros.org/wiki/rl_agent`

8 Continuous Domains

Most of the algorithms described to this point in the chapter assume that the agent operates in a discrete state space. However, many real-world problems such as robotic control involve continuously valued states and actions. These approaches can be extended to continuous problems by quantizing the state space, but very fine discretizations result in a very large number of states, and some information is lost in the discretization. Model-free approaches can be extended fairly easily to continuous domains through the use of function approximation. However, there are multiple challenges that must be addressed to extend model-based methods to continuous domains: 1) learning continuous models, 2) planning in a continuous state space, and 3) exploring a continuous state space. Continuous methods are described further in Chapter 8.

Learning a model of a continuous domain requires predictions about a continuous next-state and reward from a continuous state. Unlike the discrete case, one cannot simply learn a tabular model for some discrete set of states. Some form of function approximation must be used, as many real-valued states may never be visited. Common approaches are to use regression or instance-based techniques to learn a continuous model.

A more difficult problem is to plan over a continuous state space. There are an infinite number of states for which the agent needs to know an optimal action. Again, this can be done with some form of function approximation on the policy, or the statespace could be discretized for planning purposes (even if used as-is for learning the model). In addition, many of the model-free approaches for continuous state spaces discussed in Chapter 8, such as policy gradient methods (Sutton et al, 1999) or Q-LEARNING with function approximation, could be used for planning a policy on the model.

Fitted value iteration (FVI) (Gordon, 1995) adapts value iteration to continuous state spaces. It iterates, updating the values of a finite set of states sampled from the infinite state space and then fitting a function approximator to their values. If the function approximator fits some contraction criteria, then fitted value iteration is proven to converge.

One of the earliest model-based RL algorithms for continuous state-spaces is the PARTI-GAME algorithm (Moore and Atkeson, 1995). It does not work in the typical RL framework, having deterministic dynamics and a goal region rather than a reward function. The algorithm discretizes the state space for learning and planning, adaptively increasing its resolution in interesting parts of the state space. When the planner is unable to find a policy to the goal, cells on the border of ones that succeed and fail are split further to increase the resolution of the discretization. This approach allows the agent to have a more accurate model of dynamics when needed and a more general model elsewhere.

Unlike the partitioning approach of the PARTI-GAME algorithm, Ormnoneit and Sen (2002) use an instance-based model of the domain in their kernel-based RL algorithm. The algorithm saves all the transitions it has experienced. When making a prediction for a queried state-action, the model makes a prediction based on an average of nearby transitions, weighted using the kernel function. This model is combined with approximate dynamic programming to create a full model-based method.

Deisenroth and Rasmussen (2011) use Gaussian Process (GP) regression to learn a model of the domain in their algorithm called Probabilistic Inference for Learning Control (PILCO). The GP regression model generalizes to unseen states and provides confidence bounds for its predictions. The agent plans assuming the next state distribution matches the confidence bounds, encouraging the agent to explore when some states from the next state distribution are highly valued. The algorithm also uses GP regression to represent its policy and it computes the policy with policy iteration. It runs in batch mode, alternatively taking batches of actions in the world and then re-computing its model and policy. The algorithm learns to control a physical cart-pole device with few samples, but pauses for 10 minutes of computation after every 2.5 seconds of action.

Jong and Stone (2007) present an algorithm called FITTED-R-MAX that extends R-MAX to continuous domains using an instance-based model. When a state-action is queried, the algorithm uses the nearest instances to the queried state-action. They use the relative effects of the nearest instances to predict the relative change in state that will occur for the queried state-action. Their algorithm can provide a distribution over next states which is then used for planning with fitted value iteration. The agent is encouraged to explore parts of the state space that do not have enough instances with an R-MAX type exploration bonus.

Least-Squares Policy Iteration (LSPI) (Lagoudakis and Parr, 2003) is a popular method for planning with function approximation (described further in Chapter 4). It performs approximate policy iteration when using linear function approximation. It calculates the policy parameters that minimize the least-squares difference from the Bellman equation for a given set of experiences. These experiences could come from a generative model or from saved experiences of the agent. However, LSPI is usually used for batch learning with experiences gathered through random walks because of the expensive computation required. Li et al (2009) extend LSPI to perform online exploration by providing exploration bonuses similar to FITTED-R-MAX.

Nouri and Littman (2010) take a different approach with a focus on exploration in continuous domains. They develop an algorithm called Dimension Reduction in Exploration (DRE) that uses a method for dimensionality reduction in learning the transition function that automatically discovers the relevant state features for prediction. They predict each feature independently, and they use a 'knownness' criterion from their model to drive exploration. They combine this model with fitted value iteration to plan every few steps.

While RL typically focuses on discrete actions, there are many control problems that require a continuous control signal. Trying to find the best action in this case can be a difficult problem. Binary Action Search (Pazis and Lagoudakis, 2009) provides a possible solution to this problem by discretizing the action space and breaking down the continuous action selection problem into a series of binary action selections, each one deciding one bit of the value of the continuous action to be taken. Alternatively, one can represent the policy with a function approximator (for example, in an actor-critic method) and update the function approximator appropriately to output the best continuous action (Sutton et al, 1999; van Hasselt and Wiering, 2007). Weinstein et al (2010) develop a different approach called HOOT, using MCTS-type search to partition and search the continuous action space. The search progresses through the continuous action space, selecting smaller and smaller partitions of the continuous space until reaching a leaf in the search tree and selecting an action.

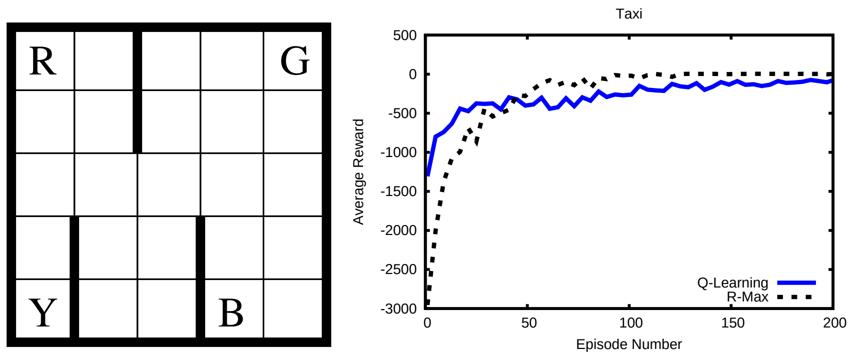
9 Empirical Comparisons

Having surveyed the model-based approaches along the dimensions of how they combine planning and model learning, how they explore, and how they deal with continuous spaces, we now present some brief representative experiments that illustrate their most important properties. Though model-based methods are most useful in large domains with limited action opportunities (see Section 10), we can illustrate their properties on simple toy domains[5].

We compared R-MAX (described in Section 5) with Q-LEARNING, a typical model-free method, on the Taxi domain. R-MAX was run with the number of visits required for a state to be considered known, M, set to 5. Q-LEARNING was run with a learning rate of 0.3 and ε-greedy exploration with $\varepsilon = 0.1$. Both methods were run with a discount factor of 0.99. The Taxi domain (Dietterich, 1998), shown in Figure 9(a), is a 5x5 gridworld with four landmarks that are labeled with one of the following colors: *red*, *green*, *blue* or *yellow*. The agent's state consists of its location in the gridworld in x, y coordinates, the location of the passenger (at a landmark or in the *taxi*), and the passenger's destination (a landmark). The agent's goal is to navigate to the passenger's location, pick the passenger up, navigate to the passenger's destination and drop the passenger off. The agent has six actions that it can take. The first four (*north*, *south*, *west*, *east*) move the agent to the square in that respective direction with probability 0.8 and in a perpendicular direction with probability 0.1. If the resulting direction is blocked by a wall, the agent stays where it is. The fifth action is the *pickup* action, which picks up the passenger if she is at the taxi's location. The sixth action is the *putdown* action, which attempts to drop off the passenger. Each of the actions incurs a reward of -1, except for unsuccessful *pickup* or *putdown* actions, which produce a reward of -10. The episode is terminated by a successful *putdown* action, which provides a reward of $+20$. Each episode starts with the passenger's location and destination selected randomly from the four landmarks and with the agent at a random location in the gridworld.

Figure 9(b) shows a comparison of the average reward accrued by Q-LEARNING and R-MAX on the Taxi domain, averaged over 30 trials. R-MAX receives large negative rewards early as it explores all of its 'unknown' states. This exploration, however, leads it to find the optimal policy faster than Q-LEARNING.

[5] Source code to re-create these experiments is available at: `http://www.ros.org/wiki/reinforcement_learning`

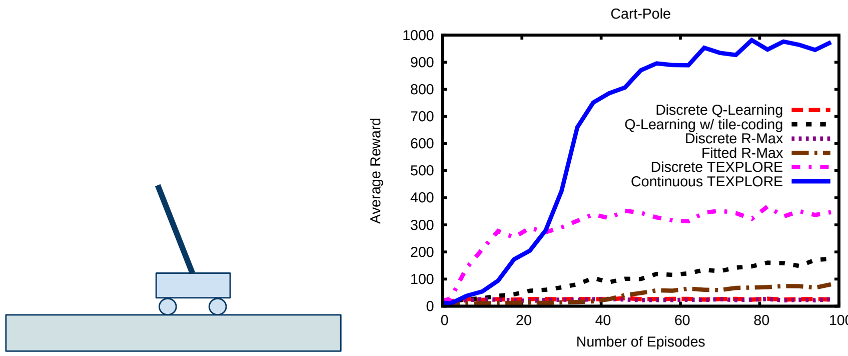


(a) Taxi Domain. (b) Average Rewards.

Fig. 9 9(a) shows the Taxi domain. 9(b) shows the average reward per episode for Q-LEARNING and R-MAX on the Taxi domain averaged over 30 trials.

Next, we show the performance of a few model-based methods and Q-LEARNING on the Cart-Pole Balancing task. Cart-Pole Balancing is a continuous task, shown in Figure 10(a), where the agent must keep the pole balanced while keeping the cart on the track. The agent has two actions, which apply a force of 10 N to the cart in either direction. Uniform noise between -5 and 5 N is added to this force. The state is made up of four features: the pole's position, the pole's velocity, the cart's position, and the cart's velocity. The agent receives a reward of $+1$ each time step until the episode ends. The episode ends if the pole falls, the cart goes off the track, or $1,000$ time steps have passed. The task was simulated at 50 Hz. For the discrete methods, we discretized each of the 4 dimensions into 10 values, for a total of $10,000$ states.

For model-free methods, we compared Q-LEARNING on the discretized domain with Q-LEARNING using tile-coding for function approximation on the continuous representation of the task. Both methods were run with a learning rate of 0.3 and ε-greedy exploration with $\varepsilon = 0.1$. Q-LEARNING with tile coding was run with 10 conjunctive tilings each with dimension split into 4 tiles. We also compared four model-based methods: two discrete methods and two continuous methods. The discrete methods were R-MAX (Brafman and Tennenholtz, 2001), which uses a tabular model, and TEXPLORE (Hester and Stone, 2010). TEXPLORE uses decision trees to model the relative effects of transitions and acts greedily with respect to a model that is the average of multiple possible models. Both of these methods were run on the discretized version of the domain. Here again, R-MAX was run with $M = 5$. TEXPLORE was run with $b = 0, w = 0.55$, and $f = 0.2$. We also evaluated FITTED-R-MAX (Jong and Stone, 2007), which is an extension of R-MAX for continuous domains, and CONTINUOUS TEXPLORE, an extension of TEXPLORE to continuous domains that uses regression trees to model the continuous state instead of discrete decision trees. CONTINUOUS TEXPLORE was run with the same parameters as TEXPLORE and FITTED-R-MAX was run with a model breadth of 0.05 and a resolution factor of 4.



(a) Cart-Pole. (b) Average Rewards.

Fig. 10 10(a) shows the Cart-Pole Balancing task. 10(b) shows the average reward per episode for the algorithms on Cart-Pole Balancing averaged over 30 trials.

The average rewards for the algorithms averaged over 30 trials are shown in Figure 10(b). Both versions of R-MAX take a long time exploring and do not accrue much reward. Q-LEARNING with tile coding out-performs discrete Q-LEARNING because it is able to generalize values across states to learn faster. Meanwhile, the generalization and exploration of the TEXPLORE methods give them superior performance, as they accrue significantly more reward per episode than the other methods after episode 6. For each method, the continuous version of the algorithm outperforms the discrete version.

These experiments illustrate some of the trade-offs between model-based and model-free methods and among different model-based methods. Model-based methods such as R-MAX spend a long time exploring the domain in order to guarantee convergence to an optimal policy, allowing other methods such as Q-LEARNING to perform better than R-MAX during this exploration period. On the other hand, methods such as TEXPLORE that attempt to learn a model without exploring as thoroughly as R-MAX can learn to act reasonably well quickly, but do not provide the same guarantees of optimality since their model may not be completely accurate. In addition, when working in continuous domains, algorithms that work directly with the continuous variables without requiring discretization have a distinct advantage.

10 Scaling Up

While model-based RL provides great promise in terms of sample efficiency when compared to model-free RL, this advantage comes at the expense of more computational and space complexity for learning a model and planning. For this reason, evaluation of these methods has often been limited to small toy domains such as Mountain Car, Taxi, or Puddle World. In addition, even efficient algorithms may take too many actions to be practical on a real task. All of these issues must be resolved for model-based RL to be scaled up to larger and more realistic problems.

With factored models and generalization, model-based RL has achieved the sample efficiency to work on larger problems. However, in these cases, it still needs to plan over the entire state space when its model changes, requiring significant computation. One way that this problem has been addressed in the literature is to combine these methods with sample-based planners such as UCT (Kocsis and Szepesvári, 2006). In (Walsh et al, 2010), the authors prove that their method's bounds on sample complexity are still valid when using a sample-based planner called Forward-Search Sparse Sampling (FSSS), which is a more conservative version of UCT. FSSS maintains statistical upper and lower bounds for the value of each node in the tree and only explores sub-trees that have a chance of being optimal.

In order to guarantee that they will find the optimal policy, any model-based algorithm must visit at least every state-action in the domain. Even this number of actions may be too large in some cases, whether it is because the domain is very big, or because the actions are very expensive or dangerous. These bounds can be improved in factored domains by assuming a DBN transition model. By assuming knowledge of the DBN structure ahead of time, the DBN-E^3 and FACTORED-R-MAX algorithms (Kearns and Koller, 1999; Guestrin et al, 2002) are able to learn a near-optimal policy in a number of actions polynomial in the number of parameters of the DBN-MDP, which can be exponentially smaller than the number of total states.

Another approach to this problem is to attempt to learn the structure of the DBN model using decision trees, as in the TEXPLORE algorithm (Hester and Stone, 2010). This approach gives up guarantees of optimality as the correct DBN structure may not be learned, but it can learn high-rewarding (if not optimal) policies in many fewer actions. The TEXPLORE algorithm models the MDP with random forests and explores where its models are uncertain, but does not explore every state-action in the domain.

Another approach to improving the sample efficiency of such algorithms is to incorporate some human knowledge into the agent. One approach for doing so is to provide trajectories of human generated experiences that the agent can use for building its model. For example, in (Ng et al, 2003), the authors learn a dynamics model of a remote control helicopter from data recorded from an expert user. Then they use a policy search RL method to learn a policy to fly the helicopter using their learned model.

One more issue with real-world decision making tasks is that they often involve partially-observable state, where the agent can not uniquely identify its state from its observations. The U-TREE algorithm (McCallum, 1996) is one model-based algorithm that addresses this issue. It learns a model of the domain using a tree that can incorporate previous states and actions into the splits in the tree, in addition to the current state and action. The historical states and actions can be used to accurately determine the agent's true current state. The algorithm then uses value iteration on this model to plan a policy.

Another way to scale up to larger and more complex domains is to use relational models (described further in Chapter 12). Here, the world is represented as a set of literals and the agent can learn models in the form of STRIPS-like planning operators (Fikes and Nilsson, 1971). Because these planning operators are relational, the agent can easily generalize the effects of its actions over different objects in the world, allowing it to scale up to worlds with many more objects. Pasula et al (2004) present a method for learning probabilistic relational models in the form of sets of rules. These rules describe how a particular action affects the state. They start with a set of rules based on the experiences the agent has seen so far, and perform a search over the rule set to optimize a score promoting simple and general rules. Object-oriented RL (Diuk et al, 2008) takes a similar approach to relational RL, defining the world in terms of objects.

11 Conclusion

Model-based methods learn a model of the MDP on-line while interacting with the environment, and then plan using their approximate model to calculate a policy. If the algorithm can learn an accurate model quickly enough, model-based methods can be more sample efficient than model-free methods. With an accurate learned model, an optimal policy can be planned without requiring any additional experiences in the world. In addition, these approaches can use their model to plan out multi-step exploration policies, enabling them to perform more directed exploration than model-free methods.

Table 2 shows a summary of the model-based algorithms described in this chapter. R-MAX (Brafman and Tennenholtz, 2001) is one of the most commonly used

Algorithm	Section	Model Type	Key Feature
DHP	2	Neural Network	Uses NN model within actor-critic framework
LWR RL	2	Locally Weighted Regression	One of the first methods to generalize model across states
DYNA	4	Tabular	Bellman updates on actual actions and saved experiences
Prioritized Sweeping	4	Tabular	Performs sweep of value updates backwards from changed state
DYNA-2	4	Tabular	Use UCT for simulated updates to transient memory
Real-Time Architecture	4	Any	Parallel architecture to allow real-time action
E^3	5	Tabular	Explicit decision to explore or exploit
R-MAX	5	Tabular	Reward bonus given to 'unknown' states
DBN-E^3	6	Learn probabilities for provided DBN model	Learns in number of actions polynomial in # of DBN parameters
FACTORED-R-MAX	6	Learn probabilities for provided DBN model	Learns in number of actions polynomial in # of DBN parameters
SLF-R-MAX	6	Learn models for all possible DBN structures	Explore when any model is uncertain or models disagree
MET-R-MAX	6	Learn DBN structure and probabilities efficiently	Explore when any model is uncertain or models disagree
LSE-R-MAX	6	Learn DBN efficiently without in-degree	Explore when any model is uncertain or models disagree
SPITI	6	Decision Trees	Model generalizes across states
RL-DT	6	Decision Trees with relative effects	Model generalizes across states, R-MAX-like exploration
Optimal Probe	7	Maintain distribution over models	Plan over augmented belief state space
BEETLE	7	Maintain distribution over parametrized models	Plan over augmented belief state space
Bayesian DP	7	Maintain distribution over models	Plan using model sampled from distribution
BOSS	7	Maintain distribution over models	Plan using merged model created from sampled models
MBBE	7	Maintain distribution over models	Use distribution over action-values to compute VPI
BEB	7	Maintain Dirichlet distribution over models	Provide reward bonus to follow Bayesian policy
MBIE	7	Maintain distribution over transition probabilities	Take max over transition probabilities as well as next state
TEXPLORE	7	Random Forest model for each feature	Plan approximate optimal policy on average model
Intrinsic Curiosity	7	Supervised Learning method	Provide intrinsic reward based on improvement in model
R-IAC	7	Supervised Learning method	Select actions based on improvement in model
PILCO	8	Gaussian Process Regression	Plan using uncertainty in next state predictions
PARTI-GAME	8	Split state-space based on transition dynamics	Splits state space non-uniformly
Kernel-based RL	8	Instance-based model with kernel distance	Use kernel to make predictions based on similar states
FITTED-R-MAX	8	Instance-based model	Predict based on applying the relative effect of similar transitions
DRE	8	Dimensionality Reduction techniques	Works in high-dimensional state spaces
U-TREE	10	Tree model that uses histories	Makes predictions in partially observable domains
Relational	10	Relational rule sets	Learns STRIPS-like relational operators

Table 2 This table presents a summary of the algorithms presented in this chapter.

model-based methods because of its theoretical guarantees and ease of use and implementation. However, MET-R-MAX (Diuk et al, 2009) and LSE-R-MAX (Chakraborty and Stone, 2011) are the current state of the art in terms of the bounds on sample complexity for factored domains. There are other approaches that perform as well or better without such theoretical guarantees, such as Gaussian Process RL (Deisenroth and Rasmussen, 2011; Rasmussen and Kuss, 2004) or TEXPLORE (Hester and Stone, 2010).

A major direction of current research in model-based reinforcement learning is on scaling methods up to domains with larger and continuous state and action spaces. Accomplishing this goal will require making algorithms more sample efficient, relying on more computationally efficient planners, and developing better methods for exploration.

Acknowledgements This work has taken place in the Learning Agents Research Group (LARG) at the Artificial Intelligence Laboratory, The University of Texas at Austin. LARG research is supported in part by grants from the National Science Foundation (IIS-0917122), ONR (N00014-09-1-0658), and the Federal Highway Administration (DTFH61-07-H-00030).

References

Asmuth J, Li L, Littman M, Nouri A, Wingate D (2009) A Bayesian sampling approach to exploration in reinforcement learning. In: Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence (UAI)

Atkeson C, Moore A, Schaal S (1997) Locally weighted learning for control. Artificial Intelligence Review 11:75–113

Auer P, Cesa-Bianchi N, Fischer P (2002) Finite-time analysis of the multiarmed bandit problem. Machine Learning 47(2):235–256

Baranes A, Oudeyer PY (2009) R-IAC: Robust Intrinsically Motivated Exploration and Active Learning. IEEE Transactions on Autonomous Mental Development 1(3):155–169

Boutilier C, Dearden R, Goldszmidt M (2000) Stochastic dynamic programming with factored representations. Artificial Intelligence 121:49–107

Brafman R, Tennenholtz M (2001) R-Max - a general polynomial time algorithm for near-optimal reinforcement learning. In: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI), pp 953–958

Breiman L (2001) Random forests. Machine Learning 45(1):5–32

Chakraborty D, Stone P (2011) Structure learning in ergodic factored MDPs without knowledge of the transition function's in-degree. In: Proceedings of the Twenty-Eighth International Conference on Machine Learning (ICML)

Dearden R, Friedman N, Andre D (1999) Model based Bayesian exploration. In: Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI), pp 150–159

Degris T, Sigaud O, Wuillemin PH (2006) Learning the structure of factored Markov Decision Processes in reinforcement learning problems. In: Proceedings of the Twenty-Third International Conference on Machine Learning (ICML), pp 257–264

Deisenroth M, Rasmussen C (2011) PILCO: A model-based and data-efficient approach to policy search. In: Proceedings of the Twenty-Eighth International Conference on Machine Learning (ICML)

Dietterich T (1998) The MAXQ method for hierarchical reinforcement learning. In: Proceedings of the Fifteenth International Conference on Machine Learning (ICML), pp 118–126

Diuk C, Cohen A, Littman M (2008) An object-oriented representation for efficient reinforcement learning. In: Proceedings of the Twenty-Fifth International Conference on Machine Learning (ICML), pp 240–247
Diuk C, Li L, Leffler B (2009) The adaptive-meteorologists problem and its application to structure learning and feature selection in reinforcement learning. In: Proceedings of the Twenty-Sixth International Conference on Machine Learning (ICML), p 32
Duff M (2003) Design for an optimal probe. In: Proceedings of the Twentieth International Conference on Machine Learning (ICML), pp 131–138
Even-dar E, Mansour Y (2001) Learning rates for q-learning. In: Journal of Machine Learning Research, pp 1–25
Fikes R, Nilsson N (1971) Strips: A new approach to the application of theorem proving to problem solving. Tech. Rep. 43r, AI Center, SRI International, 333 Ravenswood Ave, Menlo Park, CA 94025, sRI Project 8259
Gordon G (1995) Stable function approximation in dynamic programming. In: Proceedings of the Twelfth International Conference on Machine Learning (ICML)
Guestrin C, Patrascu R, Schuurmans D (2002) Algorithm-directed exploration for model-based reinforcement learning in factored MDPs. In: Proceedings of the Nineteenth International Conference on Machine Learning (ICML), pp 235–242
van Hasselt H, Wiering M (2007) Reinforcement learning in continuous action spaces. In: IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL), pp 272 –279
Hester T, Stone P (2009) Generalized model learning for reinforcement learning in factored domains. In: Proceedings of the Eight International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)
Hester T, Stone P (2010) Real time targeted exploration in large domains. In: Proceedings of the Ninth International Conference on Development and Learning (ICDL)
Hester T, Quinlan M, Stone P (2010) Generalized model learning for reinforcement learning on a humanoid robot. In: Proceedings of the 2010 IEEE International Conference on Robotics and Automation (ICRA)
Hester T, Quinlan M, Stone P (2011) A real-time model-based reinforcement learning architecture for robot control. ArXiv e-prints 11051749
Jong N, Stone P (2007) Model-based function approximation for reinforcement learning. In: Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)
Kakade S (2003) On the sample complexity of reinforcement learning. PhD thesis, University College London
Kearns M, Koller D (1999) Efficient reinforcement learning in factored MDPs. In: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI), pp 740–747
Kearns M, Singh S (1998) Near-optimal reinforcement learning in polynomial time. In: Proceedings of the Fifteenth International Conference on Machine Learning (ICML), pp 260–268
Kearns M, Mansour Y, Ng A (1999) A sparse sampling algorithm for near-optimal planning in large Markov Decision Processes. In: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI), pp 1324–1331
Kocsis L, Szepesvári C (2006) Bandit based Monte-Carlo planning. In: Proceedings of the Seventeenth European Conference on Machine Learning (ECML)
Kolter JZ, Ng A (2009) Near-Bayesian exploration in polynomial time. In: Proceedings of the Twenty-Sixth International Conference on Machine Learning (ICML), pp 513–520
Lagoudakis M, Parr R (2003) Least-squares policy iteration. Journal of Machine Learning Research 4:1107–1149
Li L, Littman M, Walsh T (2008) Knows what it knows: a framework for self-aware learning. In: Proceedings of the Twenty-Fifth International Conference on Machine Learning (ICML), pp 568–575

Li L, Littman M, Mansley C (2009) Online exploration in least-squares policy iteration. In: Proceedings of the Eight International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp 733–739

McCallum A (1996) Learning to use selective attention and short-term memory in sequential tasks. In: From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior

Moore A, Atkeson C (1993) Prioritized sweeping: Reinforcement learning with less data and less real time. Machine Learning 13:103–130

Moore A, Atkeson C (1995) The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. Machine Learning 21:199–233

Ng A, Kim HJ, Jordan M, Sastry S (2003) Autonomous helicopter flight via reinforcement learning. In: Advances in Neural Information Processing Systems (NIPS) 16

Nouri A, Littman M (2010) Dimension reduction and its application to model-based exploration in continuous spaces. Mach Learn 81(1):85–98

Ormnoneit D, Sen Ś (2002) Kernel-based reinforcement learning. Machine Learning 49(2):161–178

Pasula H, Zettlemoyer L, Kaelbling LP (2004) Learning probabilistic relational planning rules. In: Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS)

Pazis J, Lagoudakis M (2009) Binary action search for learning continuous-action control policies. In: Proceedings of the Twenty-Sixth International Conference on Machine Learning (ICML), p 100

Poupart P, Vlassis N, Hoey J, Regan K (2006) An analytic solution to discrete Bayesian reinforcement learning. In: Proceedings of the Twenty-Third International Conference on Machine Learning (ICML), pp 697–704

Prokhorov D, Wunsch D (1997) Adaptive critic designs. IEEE Transactions on Neural Networks 8:997–1007

Rasmussen C, Kuss M (2004) Gaussian processes in reinforcement learning. In: Advances in Neural Information Processing Systems (NIPS) 16

Schaal S, Atkeson C (1994) Robot juggling: implementation of memory-based learning. Control Systems Magazine, IEEE 14(1):57 –71

Schmidhuber J (1991) Curious model-building control systems. In: Proceedings of the International Joint Conference on Neural Networks, IEEE, pp 1458–1463

Silver D, Sutton R, Müller M (2008) Sample-based learning and search with permanent and transient memories. In: Proceedings of the Twenty-Fifth International Conference on Machine Learning (ICML), pp 968–975

Strehl A, Littman M (2005) A theoretical analysis of model-based interval estimation. In: Proceedings of the Twenty-Second International Conference on Machine Learning (ICML), pp 856–863

Strehl A, Diuk C, Littman M (2007) Efficient structure learning in factored-state MDPs. In: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, pp 645–650

Strens M (2000) A Bayesian framework for reinforcement learning. In: Proceedings of the Seventeenth International Conference on Machine Learning (ICML), pp 943–950

Sutton R (1990) Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In: Proceedings of the Seventh International Conference on Machine Learning (ICML), pp 216–224

Sutton R (1991) Dyna, an integrated architecture for learning, planning, and reacting. SIGART Bulletin 2(4):160–163

Sutton R, McAllester D, Singh S, Mansour Y (1999) Policy gradient methods for reinforcement learning with function approximation. In: Advances in Neural Information Processing Systems (NIPS) 12, pp 1057–1063

Venayagamoorthy G, Harley R, Wunsch D (2002) Comparison of heuristic dynamic programming and dual heuristic programming adaptive critics for neurocontrol of a turbogenerator. IEEE Transactions on Neural Networks 13(3):764 –773

Walsh T, Goschin S, Littman M (2010) Integrating sample-based planning and model-based reinforcement learning. In: Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence

Wang T, Lizotte D, Bowling M, Schuurmans D (2005) Bayesian sparse sampling for on-line reward optimization. In: Proceedings of the Twenty-Second International Conference on Machine Learning (ICML), pp 956–963

Wang Y, Gelly S (2007) Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In: IEEE Symposium on Computational Intelligence and Games

Weinstein A, Mansley C, Littman M (2010) Sample-based planning for continuous action Markov Decision Processes. In: ICML 2010 Workshop on Reinforcement Learning and Search in Very Large Spaces

Wiering M, Schmidhuber J (1998) Efficient model-based exploration. In: From Animals to Animats 5: Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior, MIT Press, Cambridge, MA, USA, pp 223–228