Generalized Model Learning for Reinforcement Learning in Factored Domains

Todd Hester
University of Texas at Austin
Department of Computer Sciences
Austin, TX 78751
todd@cs.utexas.edu

Peter Stone
University of Texas at Austin
Department of Computer Sciences
Austin, TX 78751
pstone@cs.utexas.edu

ABSTRACT

Improving the sample efficiency of reinforcement learning algorithms to scale up to larger and more realistic domains is a current research challenge in machine learning. Model-based methods use experiential data more efficiently than model-free approaches but often require exhaustive exploration to learn an accurate model of the domain. We present an algorithm, Reinforcement Learning with Decision Trees (RL-DT), that uses supervised learning techniques to learn the model by generalizing the relative effect of actions across states. Specifically, RL-DT uses decision trees to model the relative effects of actions in the domain. The agent explores the environment exhaustively in early episodes when its model is inaccurate. Once it believes it has developed an accurate model, it exploits its model, taking the optimal action at each step. The combination of the learning approach with the targeted exploration policy enables fast learning of the model. The sample efficiency of the algorithm is evaluated empirically in comparison to five other algorithms across three domains. RL-DT consistently accrues high cumulative rewards in comparison with the other algorithms tested.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms, Experimentation

Keywords

Reinforcement Learning, Supervised Learning

1. INTRODUCTION

Reinforcement learning (RL) studies the problem of finding effective solutions to sequential decision making problems [11]. For many agent-based applications, it is critical that an RL algorithm be very sample efficient: that it takes very few actions to learn an effective policy. We focus on sample efficiency as the key evaluation criterion for RL algorithms because in many agent-based applications, acquiring experiences can be very expensive and time-consuming. Two of the main approaches towards this goal are to incorporate generalization (function approximation) into model-free methods and to develop model-based algorithms. Model-based methods achieve high sample efficiency by learning a model of the domain and simulating experiences in their model, thus saving precious samples in the real world.

Cite as: Generalized Model Learning for Reinforcement Learning in Factored Domains, Todd Hester, Peter Stone, *Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, Decker, Sichman, Sierra and Castelfranchi (eds.), May, 10–15, 2009, Budapest, Hungary, pp. 717–724

Once a model-based reinforcement learning method has built an accurate model of the domain, it can quickly find an optimal policy by performing value iteration within its model. Thus the key to making model-based methods more sample efficient is to make their model learning more efficient.

Methods such as R-MAX [2] attempt to learn the model efficiently by driving the agent to explore parts of the domain where the model needs improvement. Still, R-MAX requires that the agent exhaustively explore every state m times in order to fully learn an accurate model of the domain. Particularly in large domains, this exploration can be impractical.

One way to learn the model of the domain quickly is to introduce generalization into the learning of the model using modern machine learning techniques. Learning the model is essentially a supervised learning problem where the input is the agent's current state and action, and the learning algorithm has to predict the agent's next state and reward. Many existing supervised learning algorithms are able to generalize their predictions to new or unseen parts of the state space. By applying these methods, the algorithm can learn a model of the domain with far fewer state visits than an algorithm like R-MAX.

Unlike many traditional supervised learning problems, in reinforcement learning, the algorithm has control of which training examples it receives. Learning the model efficiently requires a combination of a fast learning algorithm and a policy that acquires the necessary training examples quickly. The agent can target states for exploration that it expects will improve the model. These states could be where the agent has not visited frequently or where the model has low confidence in its predictions.

An interesting thought experiment is to consider how a human might behave if put into a gridworld environment with unknown actions and unknown goals. One might try each of the actions a few times to determine what they do, and then assume they will behave roughly the same across the remaining states. Then the person may explore all the states of the domain exhaustively searching for rewarding states. At some point, the human will decide that she has done enough exploration and exploit what she has learned.

This type of behavior is exactly what we strive to achieve with our algorithm.

In this paper, we present a novel reinforcement learning algorithm called Reinforcement Learning with Decision Trees (RL-DT). This algorithm uses decision trees to learn a model of the domain efficiently, incorporating generalization into the learning of the model. The algorithm explores early to learn an accurate model before switching to an exploitation mode. In the following sections, we describe the algorithm and how it relates to other work in this area. We then demonstrate the effectiveness of this algorithm empirically in three domains, and compare it to other sample efficient algorithms. RL-DT attained higher cumulative rewards than the other algorithms on the majority of the episodes of the experiments.

2. BACKGROUND

We adopted the standard Markov Decision Process (MDP) formalism for this work [11]. An MDP consists of a set of states S, a set of actions A, a reward function $R(s,a)$, and a transition function $P(s'|s,a)$. In each state $s \in S$, the agent takes an action $a \in A$. Upon taking this action, the agent receives a reward $R(s,a)$ and reaches a new state s'. The new state s' is determined from the probability distribution $P(s'|s,a)$. The domain can be modeled by approximating its transition and reward functions. In many domains, the discrete state s is represented by a vector of n discrete state variables $s = \langle x_1, x_2, ..., x_n \rangle$. The goal of the agent is to find the policy π mapping states to actions that maximizes the expected discounted total reward over the agent's lifetime.

The value $Q^*(s,a)$ of a given state-action pair (s,a) is determined by solving the Bellman equation:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q^*(s',a') \qquad (1)$$

where $0 < \gamma < 1$ is the discount factor. The optimal value function Q^* can be found through value iteration by iterating over the Bellman equations until convergence [11]. The optimal policy π is then as follows:

$$\pi(s) = \text{argmax}_a Q^*(s,a) \qquad (2)$$

3. ALGORITHM

The main contribution of this paper is a novel algorithm called Reinforcement Learning with Decision Trees (RL-DT). Our algorithm achieves high sample efficiency by learning a model of the MDP quickly. It takes advantage of the factored representations present in many domains to generalize transition effects when learning the model, similar to SLF-RMAX [10]. This approach allows it to learn the model much faster than an approach which models each state-action individually such as R-MAX [2]. Our algorithm starts out explicitly exploring the domain to learn an accurate model, and then exploits its model once it believes it is correct.

The RL-DT algorithm is a model-based reinforcement learning algorithm, shown in Algorithm 1. The algorithm maintains the set of all the states that it has seen in the set S_M and counts the number of visits to each state-action pair. From a given state s, it executes the action a as specified by its action-values and increments the visit count $visits(s,a)$. It obtains a reward r and a next state s'. It adds the state to its state set S_M if it is not already there. Then the algorithm updates its models with this new experience through

Algorithm 1 RL-DT($RMax, s$)

```
1: $A \leftarrow$ Set of Actions
2: $S_M \leftarrow \{s\}$
3: $\forall a \in A : visits(s,a) \leftarrow 0$
4: loop
5:    $a \leftarrow \text{argmax}_{a'} Q(s,a')$
6:    Execute $a$, obtain reward $r$, observe state $s'$
7:    Increment $visits(s,a)$
8:    if $s' \notin S_M$ then
9:       Add $s'$ to set $S_M$
10:      $\forall a \in A : visits(s',a) \leftarrow 0$
11:   end if
12:   $(P_M, R_M, \text{CH}) \leftarrow$ UPDATE-MODEL($s, a, r, s', S_M, A$)
13:   // Check if the model is ok
14:   $exp \leftarrow$ CHECK-MODEL($P_M, R_M$)
15:   if CH then
16:      COMPUTE-VALUES($RMax, P_M, R_M, S_M, A, exp$)
17:   end if
18:   $s \leftarrow s'$
19: end loop
```

the model learning approach described in section 3.1. The algorithm decides to explore or exploit based on whether it believes its model is accurate. This check is performed in the call to CHECK-MODEL, which is explained below. Next, the algorithm re-computes the action-values using value iteration if the model was changed (CH is set to true). It then continues executing actions until the end of the experiment or episode.

The algorithm starts out with a poor model of the domain and in some cases, it will learn an incorrect model of the domain. In both these cases, exploration is required to correct the model. The algorithm uses a heuristic to determine when it should explore in the call to CHECK-MODEL on line 14 of Algorithm 1. If the model predicts that the agent can only reach negative rewards (or only reach positive rewards) from a state, then the model is assumed to be incorrect. Many episodic tasks consist of negative rewards for all states other than the terminal state, where it receives a non-negative reward. If the algorithm determines that the model is incorrect, it goes into *exploration* mode, exploring the state-actions with the fewest visits. Otherwise, it remains in *exploitation* mode, where it takes what it believes is the optimal action at each step. This heuristic causes the agent to explore exhaustively early, when it has no knowledge of the domain. Once it has discovered a state with a non-negative reward, it will stop exploring and exploit its model to find an efficient path to the rewarding state.

3.1 Model Learning

A distinguishing characteristic of RL-DT is the way in which it learns models of the transition and reward functions. In particular, we would like to learn a model of the underlying MDP in as few samples as possible. Algorithms such as R-MAX [2] can achieve high sample efficiency by driving the agent to explore states that are unknown in its model. R-MAX counts the number of visits to each state and uses the counts to determine if the states are known. States with enough visits are considered known, while states without enough visits are unknown and the agent is encouraged to explore those states to improve its model. We can improve upon this approach by generalizing when learning the model. The transition and reward functions in many states may be similar, and if so, the model can be learned much

Algorithm 2 UPDATE-MODEL(s, a, r, s', S_M, A)

```
1: $n \leftarrow$ size of $s$
2: CH $\leftarrow$ FALSE
3: // Update the tree for each state feature
4: for $i = 1$ to $n$ do
5:     // Calculate the relative transition
6:     $x_i^r = s_i' - s_i$
7:     CH $\leftarrow$ADD-EXPERIENCE-TO-TREE($i, s, a, x_i^r$) or CH
8: end for
9: // Update the tree for reward
10: CH $\leftarrow$ADD-EXPERIENCE-TO-TREE($n+1, s, a, r$) or CH
11: // Combine results for each tree into model
12: for all $s_M \in S_M$ do
13:     for all $a_M \in A$ do
14:         $P_M(s_M, a_M) \leftarrow$COMBINE-RESULTS($s_M, a_M$)
15:         $R_M(s_M, a_M) \leftarrow$GET-PREDICTIONS($n+1, s_M, a_M$)
16:     end for
17: end for
18: Return ($P_M, R_M$, CH)
```

faster by generalizing these functions across similar states. This algorithm may be able to learn an accurate model of the MDP without visiting every state.

In many domains, the relative transition effects of actions are similar across many states, making it easier to generalize actions' relative effects than their absolute ones. For example, in many gridworld domains, there is an *east* action that usually increases the agent's x variable by 1. It is easier to generalize the relative effect of this action ($x \leftarrow x + 1$) than the absolute effects ($x \leftarrow 7$ or $x \leftarrow 8$). Leffler et al. [6] use this fact to learn relocatable action models when provided with some part of the model. RL-DT differs in that it attempts to learn the model without any prior knowledge. This idea is also used by Jong and Stone [5] to build instance-based models in continuous domains. RL-DT takes advantage of this idea by using supervised learning techniques to generalize the relative effects of actions across states when learning its model. This generalization allows it to make predictions about the effects of actions even in states that it has not visited often or at all.

The agent learns models of the transition and reward functions using decision trees, shown in Algorithm 2. Decision trees were used because they generalize well while still making accurate predictions. The decision trees are an implementation of Quinlan's C4.5 algorithm [8]. The state features used as inputs to the decision trees are treated both as numerical and categorical inputs, meaning both splits of the type $x = 3$ and $x > 3$ are allowed. The C4.5 algorithm chooses the optimal split at each node of the tree based on information gain. Our implementation includes a modification to make the algorithm incremental. Each tree is updated incrementally by checking at each node whether the new experience changes the optimal split in the tree. If it does, the tree is re-built from that node down. When re-building the tree, split points at each node are still determined based on information gain.

Similar to the approach of Degris et al. [3], a separate decision tree is built to predict the reward and each of the n state variables. The first n trees each make a prediction of the probabilities $P(x_i^r|s, a)$, while the last tree predicts the reward $R(s, a)$. The input to each tree is a vector containing the n state features and action a: $\langle a, s_1, s_2, ..., s_n\rangle$. For the first n decision trees, the desired output is the relative change in the state variable, which is calculated on line 6 of Algorithm 2. For the last tree, the desired output is the reward r. The algorithm updates each tree incrementally with the input vector and desired output in the call to ADD-EXPERIENCE-TO-TREE on lines 7 and 10 of Algorithm 2. This method also returns whether the model has changed in the CH variable.

After all the trees are updated, they can be used to predict the model of the domain. The transition function $P(s'|s, a)$ and reward function $R(s, a)$ are updated for every state $s_M \in S_M$ and action $a_M \in A$ in lines 12-17 of Algorithm 2. For a queried state vector s and action a, each tree makes predictions based on the leaf of the tree that matches the input. The first n trees output probabilities for the relative change, x_i^r, of their particular state features. This output $P(x_i^r|s, a)$ is the number of occurrences of x_i^r in the matching leaf of the tree divided by the total number of experiences in that leaf. The predictions $P(x_i^r|s, a)$ for the n state features are combined to create a prediction of probabilities of the relative change of the state $s^r = \langle x_1^r, x_2^r, ..., x_n^r\rangle$. Assuming that each of the state variables transition independently, the probability of the change in state $P(s^r|s, a)$ is the product of the probabilities of each of its n state features:

$$P(s^r|s, a) = \Pi_{i=0}^n P(x_i^r|s, a) \tag{3}$$

The relative change in the state, s^r, is added to the current state s_m to get the next state s'. COMBINE-RESULTS on line 14 of the algorithm uses these steps to calculate $P(s'|s, a)$. The last tree predicts reward $R(s, a)$ by outputting the average reward in the matching leaf of the tree in the call to GET-PREDICTIONS on line 15 of Algorithm 2. The combination of the model of the transition function and reward function make up a complete model of the underlying MDP.

Figure 1 shows an example of a decision tree classifying the relative change in the x variable in the gridworld shown in the figure. The tree first splits on the action (if the action was LEFT) and then splits on the x and y variables. In some cases, it can ignore large parts of the state space. For example, when the action is not left or right, the tree predicts a change of 0 in the x variable. In other cases, it makes a prediction that is specific to a single state, such as when it predicts a change of 0 for the action right when x is not 1 and y equals 1.

3.2 Value Iteration

Once the model has been updated, value iteration is performed on the model to find a policy. Algorithm 3 shows the COMPUTE-VALUES function, which computes exact value iteration on the approximate model for all states, S_M, that the algorithm determines may be reachable based on its model. The action-values are updated using the models R_M and P_M.

Before performing value iteration, the algorithm decides whether to go into exploration or exploitation mode. When the algorithm believes its model is incorrect, it sets *exp* to true and goes into exploration mode.. In this mode, the agent follows a policy similar to R-MAX, with the algorithm giving the least visited states a bonus of $RMax$ to drive the agent to explore them (lines 15-16 of Algorithm 3). When the algorithm is in exploitation mode, it takes what it believes to be the optimal action from each state.

In many cases, it may not be possible to know exactly what states are in the statespace S of the domain. Therefore, our algorithm is not initially given the state space of the domain. Since the algorithm does not have any prior in-

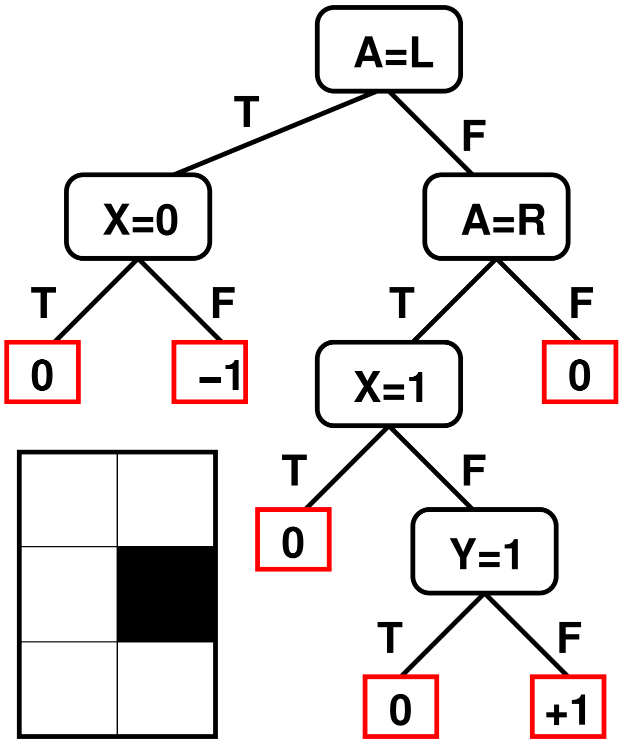

Figure 1: Example tree predicting the change in the x variable in a hypothetical gridworld.

Algorithm 3 COMPUTE-VALUES($RMax, P_M, R_M, S_M, A, exp$)

```
// Initialize all state's step counts
for all $s \in S_M$ do
   if $\exists a \in A : visits(s, a) > 0$ then
      $K(s) \leftarrow 0$
   else
      $K(s) \leftarrow \infty$
   end if
end for
$minvisits \leftarrow \min_{s \in S_M} visits(s)$
// Perform value iteration on the model
while not converged do
   for all $s \in S_M$ do
      for all $a \in A$ do
         if $exp$ and $visits(s) = minvisits$ then
            // Unknown states are given exploration bonus
            $Q(s, a) \leftarrow RMax$
         else if $K(s) >$ MAXSTEPS then
            // States out of reach
            $Q(s, a) \leftarrow RMax$
         else
            // Update remaining state's action-values
            $Q(s, a) \leftarrow R_M(s, a)$
            for all $s' \in P_M(s'|s, a)$ do
               if $s' \notin S_M$ then
                  Add $s'$ to set $S_M$
                  for all $a \in A$ do
                     $visits(s', a) = 0$
                  end for
               end if
               // Update steps to this state
               if $K(s) + 1 < K(s')$ then
                  $K(s') \leftarrow K(s) + 1$
               end if
               // Update action-values using Eqn 1
               $Q(s, a)$ += $\gamma P_M(s'|s, a) \max_{a'} Q(s', a')$
            end for
         end if
      end for
   end for
end while
```

formation about the state space, it must predict transition and reward dynamics for unseen and possibly non-existent states. It is possible that the model might predict that the world is infinite (for example, if it predicts that every action increments one of the state variables). To avoid calculating action-values for an infinite number of states, the algorithm records how many steps each state is from the nearest visited state ($K(s)$). Any state that is more than MAXSTEPS away from a visited state is given a value of $RMax$.

4. RELATED WORK

In Section 5, we compare RL-DT with several related approaches. In this section we summarize those approaches and explain why they are appropriate points of comparison.

In this paper, we focus on model-based reinforcement learning because of its sample efficiency in comparison to model-free methods. Nonetheless, research on generalization for model-free learning is relevant to our approach. There are a number of model-free reinforcement learning methods such as Q-LEARNING [12] and SARSA [9]. Q-LEARNING is a representative example of this type of algorithm that maintains action-values for every state-action pair. In its most basic form, it has a table with values for each state-action pair. After each action, the algorithm uses the reward r it receives to update its action-values with the Bellman equations. Q-LEARNING provably converges to the optimal policy when visiting every state and action infinitely often [13].

There are many ways to incorporate generalization into model-free algorithms. The most common way is to use a function approximator, such as neural networks or CMACs [1], to approximate the action-values for each state. The function approximator allows the algorithm to generalize similar action-values to similar states. There is no guarantee that these approaches will converge to the exact optimal policy. While these approaches are often more efficient than table-based approaches, the algorithm still only updates its value function when taking an action, requiring a large number of actions to learn an accurate value function.

Model-based reinforcement learning algorithms such as Prioritized Sweeping [7] and R-MAX [2] learn a model of the transition and reward functions. Once this model is learned, the algorithm performs value iteration on the model to determine an optimal policy. R-MAX is a representative model-based method that explicitly explores unknown states to learn its model quickly. It records the number of visits to each state-action in the domain. In R-MAX, the agent is driven to explore state-actions that have not been visited enough to be considered "known" by assuming they have the maximum reward of $RMax$. Once the model is completely learned, the algorithm can determine the optimal policy through value iteration. Unlike our approach, this method must learn the transition and reward models for each state-action pair separately. Learning the models separately requires the agent to visit each state-action pair m times, which in large domains can be impractical. Our approach does use an R-MAX-like exploration policy when it knows that its model is incorrect.

Several methods attempt to learn a model of the domain more efficiently by incorporating generalization into the model learning in factored domains. SLF-RMAX [10] learns which state features are relevant for predictions of the transition and reward functions. The algorithm enumerates all possible combinations of input features as elements and then creates counters of visits and outcomes for all pairs of these elements. It determines if an element is the only relevant factor for making a prediction by comparing the values of all the counters which contain that element. If all of these counters have m visits and their predictions are all within ϵ of each other, then this element is the relevant factor. The algorithm makes predictions when a relevant factor 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 $RMax$ to unknown states in value iteration to encourage the agent to explore them.

SLF-RMAX is very similar to R-MAX, in that it keeps counts of visits and outcomes for sets of states. Since SLF-RMAX can ignore some state features for its counters, it requires fewer total visits than R-MAX. Our approach uses a more powerful generalization technique than SLF-RMAX. While SLF-RMAX can only determine whether certain state features are irrelevant for a particular set of input features, our algorithm can do this for entire ranges of state features at once. In addition, RL-DT's model can be more precise than SLF-RMAX, only ignoring parts of the state space instead of deciding that entire state features are irrelevant. SLF-RMAX is also computationally expensive; enumerating every possible combination of input features and maintaining counters for all pairs of these combinations can take a prohibitively long amount of time.

Degris et al. [3] use decision trees to learn a model of the MDP. A separate decision tree is built to predict each next state feature as well as the reward. Their algorithm attempts to predict the absolute transition function, while our approach is to predict the relative effects of transitions, which may be easier in many domains. Degris et al.'s algorithm calculates an ϵ-greedy policy through value iteration based on the model provided by the decision trees. Our approach differs from theirs by explicitly choosing between an exploration mode and an exploitation mode. In exploration mode, our agent follows an R-MAX-like policy, which provides a more guided exploration approach than an ϵ-greedy policy.

5. EXPERIMENTS

We evaluated the sample complexity of RL-DT empirically in comparison to five other algorithms in three domains. Each experiment was run over a trial period of 500 episodes. The results are averaged over 30 independent runs. The experiments will demonstrate that RL-DT learns a reasonable policy in fewer samples than other related algorithms by learning its model more efficiently.

RL-DT was designed with the goal of leveraging the sample complexity advantages of model-based methods combined with the advantages of generalization. Therefore we selected comparison algorithms to evaluate the benefits of generalization and model-based methods in these domains. We compared RL-DT with the following algorithms:

- Table-based Q-LEARNING [12], a baseline comparison as it is a simple algorithm that is theoretically proven to converge.
- Q-LEARNING using CMACs [1] as a function approximator, an example of an algorithm with generalization, although it has generalization in the value function rather than in the model.
- R-MAX [2], a typical example of a model-based reinforcement learning algorithm.
- SLF-RMAX [10], another algorithm that uses generalization in learning the model of the domain, and should learn a model faster than R-MAX.
- An algorithm based on Degris et al.'s algorithm [3]. This algorithm also uses decision trees, with our implementation of it differing from RL-DT only in its modeling of absolute transitions instead of relative effects and its ϵ-greedy policy instead of explicit exploration and exploitation modes.

We chose to compare with SLF-Rmax and Degris et al.'s algorithms instead of other tree-based methods because they were the most related to our algorithm.

RL-DT was run with the following parameters. $RMax$ was set to the maximum one-step reward in each domain. The parameter MAXSTEPS was set to 5. Based on informal experiments, the algorithm is not particularly sensitive to these parameters.

Q-LEARNING was run with a learning rate of 0.3, ϵ-greedy exploration with $\epsilon = 0.1$, and action-values initialized to 0. The CMACs were created using 10 tilings with a width of four tiles across each state variable. R-MAX and SLF-RMAX were run with $m = 10$. SLF-RMAX was run with $\epsilon = 0.2$. Finally, the algorithm of Degris et al. was run using ϵ-greedy exploration with $\epsilon = 0.1$.

5.1 Domains

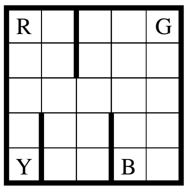

Figure 2: Taxi

The first domain used in the experiments is the classic Taxi domain introduced by Dietterich [4]. The domain, shown in Figure 2, 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 (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.

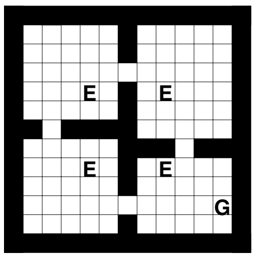

Figure 3: Energy Rooms

We created the second domain, Energy Rooms, as an example of a domain where actions could be easily generalized. It is a simple $11x11$ gridworld split into four rooms. In addition to the normal x and y state features, the agent also has an *energy* level, which ranges from 0 to 10. The agent has four actions, *north*, *south*, *east*, and *west*, which behave the same as in the Taxi domain. On each step, there is a probability of 0.8 that the agent's *energy* is reduced by 1 (while *energy* > 0). The agent starts at a random location in the upper left room and its goal is to reach the terminal state in the lower right room while maintaining *energy* > 0. There are four squares in the gridworld, marked E in the figure, that reset the agent's *energy* to 10. The agent receives a reward of -1 on each step if *energy* > 0, otherwise it receives a reward of -2. Upon reaching the terminal state, the agent receives a reward of 0.

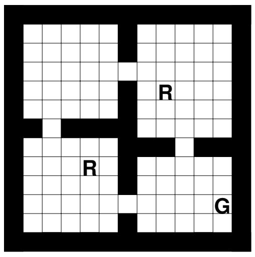

Figure 4: Red Herring

Finally, Figure 4 shows the Red Herring domain, which we created to explore a possible limitation of RL-DT due to the fact that it does not necessarily explore the whole state space. This domain is made up of a $11x11$ gridworld which is split into four rooms. The agent has four possible actions: *north*, *south*, *east*, and *west*. Each of these actions works identically to the other gridworld domains. The agent starts in a random location in the upper left room and gets a reward of -1 for every action. The lower right room has a terminal state which provides a reward of +50. There are two "red herring" terminal states in the other two rooms that provide a reward of 0. The optimal policy in this domain is to go to the high-valued terminal state in the lower right room, but without exploring fully, the agent is likely to find a policy leading to one of the "red herring" terminal states.

6. RESULTS

Figure 5 shows the average rewards of each algorithm over the first 50 episodes of the Taxi domain. SLF-RMAX was not run on this domain because the number of counters needed explodes exponentially with the number of state features in the domain, making it impractical to run the algorithm on this domain in a reasonable amount of time. RL-DT performed better than any other algorithm on the initial episodes on the domain and converged to the optimal policy in fewer steps than the other algorithms. While Degris et al.'s method converged to a near-optimal policy in fewer episodes, it took many more steps in each of the early episodes to explore the domain. This difference is reflected in Figure 6, which shows the cumulative rewards of each algorithm over all 500 episodes. RL-DT converged to the optimal policy faster than any other algorithm and had higher cumulative rewards than the other algorithms after every episode of the experiment.

The average rewards of each algorithm over the first 50 episodes in the Energy Rooms domain are shown in Figure 7. Once again, Degris et al.'s algorithm and RL-DT were the fastest algorithms to converge to a policy, with Degris et al.'s algorithm converging in fewer episodes but taking more actions in each episode to do so. Figure 8 shows the cumulative rewards of each algorithm over the 500 episodes. Both RL-DT and Degris et al.'s algorithm converged to a near optimal policy quickly and had better cumulative rewards than the other four algorithms.

The cumulative rewards of each algorithm over the 500 episodes of the Red Herring domain are shown in Figure 9. This domain shows a failure case for RL-DT, which only found the high-valued goal in six of the 30 trial runs. Even though the algorithm did not always converge to the optimal policy,

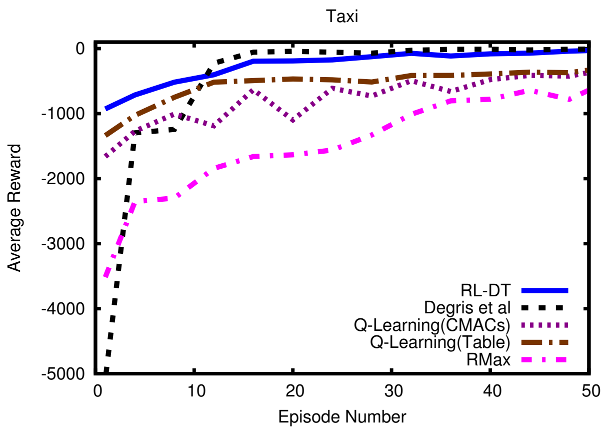

Figure 5: Average Rewards in Taxi

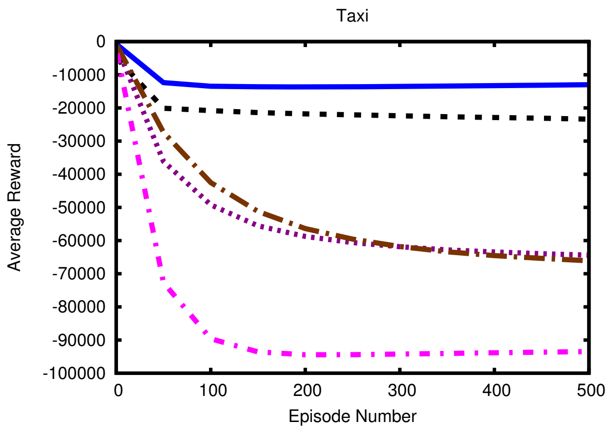

Figure 6: Cumulative Rewards in Taxi

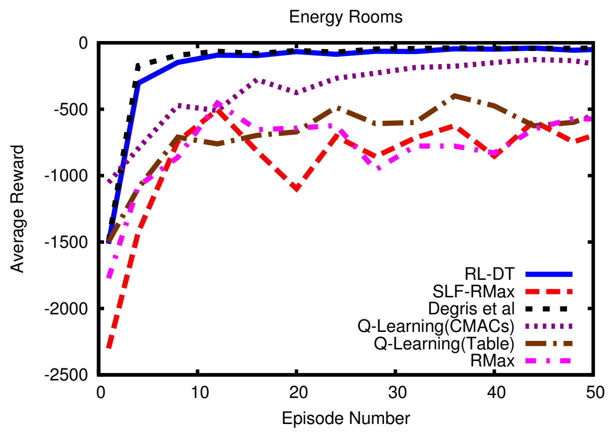

Figure 7: Average Rewards in Energy Rooms

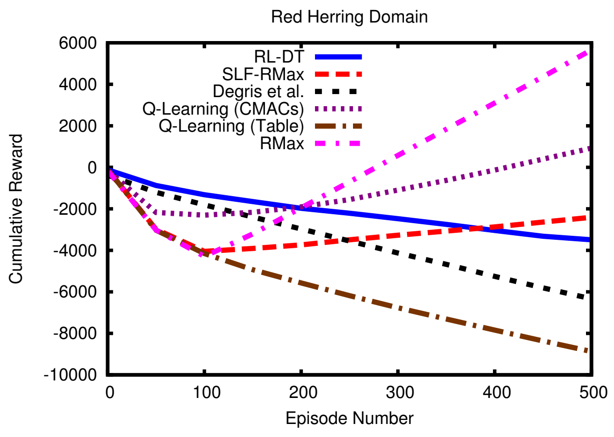

Figure 9: Cumulative Rewards in Red Herring

it did converge to a policy after fewer steps than any other algorithm, enabling it to obtain more cumulative rewards than the other algorithms in many of the episodes of the experiment. RL-DT had better cumulative rewards than every other algorithm until Q-LEARNING with function approximation passed it in episode 194. Degris et al.'s algorithm was the second fastest to converge to a policy, but took more than twice as many steps to converge as RL-DT. In addition, it only converged to a near-optimal policy in two of the 30 trials. Although in this example it would be possible to find the optimal policy through exhaustive exploration, in many larger domains it would be infeasible, thus necessitating decisions about which states to explore. In large domains, it is acceptable (even necessary) to risk missing the high-valued states to improve convergence time. In addition, in some domains the high penalties accrued in early episodes by algorithms that search exhaustively may be unacceptable.

In each of the three domains, RL-DT performed well in comparison to the other algorithms tested. It had greater accumulated rewards for every episode of the experiments run in the Taxi domain and for the first 194 episodes of the Red Herring domain. Both RL-DT and Degris et al.'s algorithm had the highest accumulated rewards on the Energy Rooms domain. RL-DT obtained higher rewards on the early episodes of most of the domains, enabling it to accumulate higher total rewards.

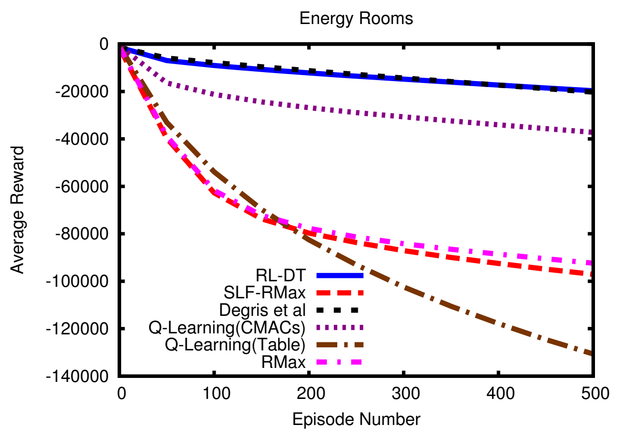

Figure 8: Cumulative Rewards in Energy Rooms

7. DISCUSSION AND CONCLUSION

We have developed an algorithm, RL-DT, which efficiently learns a model of the domain and uses it to compute a reasonable policy. It is able to learn the model quickly by using decision trees to generalize the relative effects of actions across similar states in the domain. In addition, it is able to perform well in early episodes by employing an explicit exploration mode where it explores the states with the fewest visits. These two components of the algorithm allow it to be more sample efficient and accrue more cumulative rewards than the other algorithms tested in the three domains.

Our algorithm starts by exhaustively exploring the domain until it finds a state with non-negative rewards. Then it switches into exploitation mode, where it takes the actions it believes to be optimal. This approach gives it an advantage over other methods such as Degris et al.'s algorithm that explore randomly. Early in the agent's experiences, when it has little knowledge of the domain, it is important for the algorithm to have a systematic way of exploring the domain and finding the states with high-valued rewards. Our algorithm's switch to exploitation mode after finding a rewarding state enables it to outperform algorithms that explore exhaustively such as R-MAX, by cutting off exploration earlier to exploit its model.

This approach of exploring exhaustively early and then switching to an exploitation mode works well in domains with a single goal state. In other domains with multiple goal states, such as the Red Herring domain, it does not perform as well. In these domains, it would be preferable to continue exploring after finding the first terminal state instead of cutting off exploration. In general, our algorithm will not work well in domains where it would be beneficial to have more extensive exploration. In addition, our heuristic of differentiating negative and positive rewards only works in cases where there is a special state with rewards different from the other states. If the goal state had the same reward as all the other states, this heuristic would not work.

In future work, we plan to develop an algorithm where the agent's decision to explore or exploit is handled more naturally. One idea is to have the decision trees report confidence in their predictions for individual states and to use this confidence estimate to drive exploration to specific states where the model needs improvement.

Since RL-DT generalizes when learning a model of the domain, there is a chance that it will not learn the optimal policy. When the agent is in exploitation mode, it is not required to visit every state in the domain and therefore the algorithm may not learn a fully accurate model of the domain. In most cases, however, the agent's exploration policy worked well. It consistently obtained higher rewards than the other algorithms in the first episode of the experiments, showing that its early exploration led it to find a good policy quickly. In some cases, it may be worthwhile to give up the guarantee of convergence to optimality in exchange for quickly converging to a reasonable policy and accumulating more rewards in early episodes.

One of the main advantages of our method is its use of decision trees to model the MDP. In every domain, the two algorithms using decision trees to model the domain converged to a policy the fastest. Decision trees perform well at this task because they have very powerful generalization capabilities. The splits in the trees nicely represent the splits in the state space where actions have different effects (i.e. near walls vs away from walls, or passenger in taxi vs passenger out of taxi). This approach works better than the approach in SLF-RMAX because decision trees are also capable of very precisely refining the state space, even down to individual states. Our algorithm performed better than the other tree-based method in most domains mainly because it used a targeted R-MAX-like exploration policy during its exploration mode instead of ϵ-greedy exploration.

RL-DT learns a model by predicting the relative transitions of states. This type of model makes the algorithm particularly well suited for domains where the actions incrementally change the state variables. The experiments in this paper were performed in gridworld domains, which we believe are a good starting point for improving sample efficiency. In future work, we hope to extend this algorithm to work in continuous domains with relative transitions such as robot locomotion or mountain car.

Our algorithm was compared empirically with five other algorithms across three domains. In every case, it obtained more rewards than the other algorithms on the first episode of the domain, and greater cumulative rewards than most of the other algorithms in every episode of every domain except the Red Herring domain. Even in that domain, RL-DT had greater cumulative rewards than every other algorithm for the first 194 episodes of the domain. The high sample efficiency of RL-DT makes it a good algorithm for large or agent-based domains where samples may be very expensive.

Acknowledgments

The authors would like to thank Nick Jong, Tobias Jung, Michael Quinlan, Doran Chakroborty, Katherine Bontrager and the anonymous reviewers for their comments and suggestions on this paper. 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 (CNS-0615104), DARPA (FA8750-05-2-0283 and FA8650-08-C-7812), the Federal Highway Administration (DTFH61-07-H-00030), and General Motors.

8. REFERENCES

[1] J. S. Albus. A new approach to manipulator control: The cerebellar model articulation controller. *Journal of Dynamic Systems, Measurement, and Control*, 97(3):220–227, 1975.

[2] R. I. Brafman and M. Tennenholtz. R-MAX - a general polynomial time algorithm for near-optimal reinforcement learning. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 953–958, 2001.

[3] T. Degris, O. Sigaud, and P.-H. Wuillemin. Learning the structure of factored markov decision processes in reinforcement learning problems. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 257–264, New York, NY, USA, 2006. ACM.

[4] T. G. Dietterich. The MAXQ method for hierarchical reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 118–126, 1998.

[5] N. K. Jong and P. Stone. Model-based function approximation for reinforcement learning. In *The Sixth International Joint Conference on Autonomous Agents and Multiagent Systems*, May 2007.

[6] B. R. Leffler, M. L. Littman, and T. Edmunds. Efficient reinforcement learning with relocatable action models. In *Proceedings of the Twenty-Second National Conference on Artificial Intelligence*, pages 572–577, 2007.

[7] A. W. Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13:103–130, 1993.

[8] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[9] G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Department, 1994.

[10] A. L. Strehl, C. Diuk, and M. L. Littman. Efficient structure learning in factored-state mdps. In *AAAI*, pages 645–650. AAAI Press, 2007.

[11] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

[12] C. Watkins. *Learning From Delayed Rewards.* PhD thesis, University of Cambridge, 1989.

[13] C. Watkins and P. Dayan. Technical note: Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.