In *Machine Learning Journal (MLJ)*,
DOI 10.1007/s10994-012-5322-7, 2012.

Noname manuscript No.
(will be inserted by the editor)

TEXPLORE: Real-Time Sample-Efficient Reinforcement Learning for Robots

Todd Hester · Peter Stone

Received: date / Accepted: date

Abstract The use of robots in society could be expanded by using reinforcement learning (RL) to allow robots to learn and adapt to new situations online. RL is a paradigm for learning sequential decision making tasks, usually formulated as a Markov Decision Process (MDP). For an RL algorithm to be practical for robotic control tasks, it must learn in very few samples, while continually taking actions in real-time. In addition, the algorithm must learn efficiently in the face of noise, sensor/actuator delays, and continuous state features. In this article, we present TEXPLORE, the first algorithm to address all of these challenges together. TEXPLORE is a model-based RL method that learns a random forest model of the domain which generalizes dynamics to unseen states. The agent explores states that are promising for the final policy, while ignoring states that do not appear promising. With sample-based planning and a novel parallel architecture, TEXPLORE can select actions continually in real-time whenever necessary. We empirically evaluate the importance of each component of TEXPLORE in isolation and then demonstrate the complete algorithm learning to control the velocity of an autonomous vehicle in real-time.

Keywords Reinforcement Learning · Robotics · MDP · Real-Time

Todd Hester
Department of Computer Science
The University of Texas at Austin
Austin, TX 78712
Tel.: 512-232-1874
E-mail: todd@cs.utexas.edu

Peter Stone
Department of Computer Science
The University of Texas at Austin
Austin, TX 78712
E-mail: pstone@cs.utexas.edu

1 Introduction

Robots have the potential to solve many problems in society, because of their ability to work in dangerous places doing necessary jobs that no one wants or is able to do. Robots could be used for space exploration, mining, underwater tasks, caring for the elderly, construction, and so on. One barrier to their widespread deployment is that they are mainly limited to tasks where it is possible to hand-program behaviors for every situation that may be encountered. For robots to meet their potential, they need methods that enable them to learn and adapt to novel situations that they were not programmed for.

Reinforcement learning (RL) (Sutton and Barto 1998) algorithms learn sequential decision making processes and could solve the problems of learning and adaptation on robots. An RL agent seeks to maximize long-term rewards through experience in its environment. The decision making tasks in these environments are usually formulated as Markov Decision Processes (MDPs).

Learning on robots poses many challenges for RL, because a successful method must learn quickly while running on the robot. In addition, the method must handle continuous state as well as noisy and/or delayed sensors and actuators. RL has been applied to a few carefully chosen robotic tasks that are achievable with limited training and infrequent action selections (e.g. (Kohl and Stone 2004)), or allow for an off-line learning phase (e.g. (Ng et al 2003)). However, to the best of our knowledge, none of these methods allow for continual learning on the robot running in its environment. In this article, we identify four properties of an RL algorithm that would make it generally applicable to a broad range of robot control tasks:

1. The algorithm must learn from very few samples (which may be expensive or time-consuming).
2. It must learn tasks with continuous state representations.
3. It must learn good policies even with unknown sensor or actuator delays (i.e. selecting an action may not affect the environment instantaneously).
4. It must be computationally efficient enough to take actions continually in real-time.

In addition to these four properties, it would be desirable for the algorithm to require minimal user input. Addressing these challenges not only makes RL applicable to more robotic control tasks, but also many other real-world tasks. We demonstrate the importance of each of these challenges in learning to control an autonomous vehicle.

While algorithms exist that address various subsets of these challenges, we are not aware of any that are easily adapted to address all four issues. A full comparison with prior work appears in Section 4, but as an example, PILCO (Deisenroth and Rasmussen 2011) uses a Gaussian Process regression model to achieve very high sample efficiency on continuous tasks. However, it is computationally intensive and requires 10 minutes of computation for every 2.5 seconds of interaction on a physical Cart-Pole device. It is also not trivial to accommodate delays in actuation or state observations into this method. Bayesian RL methods, such as BOSS (Asmuth et al 2009) and Bayesian DP (Strens 2000), maintain a distribution over likely MDP models and can utilize information from this distribution to explore efficiently and learn optimal policies. However, these methods are also computationally expensive, cannot easily handle delays, and require the user to provide a model parametrization that will be useful for generalization.

In contrast to these approaches, we present the TEXPLORE algorithm, the first algorithm to address all four challenges at once. To address challenge 1, an algorithm needs to limit its exploration to learn an accurate domain model quickly, such that it can exploit that model during its short lifetime. TEXPLORE does so by 1) utilizing the generalization properties of decision trees in building its model of the MDP, and 2) using random forests of those tree models to explore efficiently to learn a good policy quickly. Unlike methods such as R-MAX (Brafman and Tennenholtz 2001) that explore more thoroughly to guarantee an optimal policy, TEXPLORE explores in a limited way, focusing on promising state-actions to learn a good policy with fewer exploration steps. This approach enables TEXPLORE to learn in large domains where methods with strong convergence guarantees such as R-MAX would explore indefinitely, but it also means that TEXPLORE may not explore some unexpected but high-rewarding state-actions.

TEXPLORE addresses challenge 2 by using linear regression trees to model continuous domains. For delayed domains (challenge 3), TEXPLORE takes the k-Markov approach (Katsikopoulos and Engelbrecht 2003). It gives its models the previous k actions for training and takes advantage of the ability of decision trees to select the inputs with the correct delay for each task. In response to challenge 4, TEXPLORE utilizes a unique parallel architecture and Monte Carlo Tree Search (MCTS) planning, enabling the algorithm to provide actions continually in real-time at whatever frequency is required. In contrast to Bayesian methods, TEXPLORE does not need to maintain and update a full distribution over models (saving computation), and does not need a user-defined model parametrization, instead taking advantage of the generalization properties of decision trees.

We demonstrate that TEXPLORE's solution to each of these tasks performs better than state of the art alternatives empirically on the novel task of controlling the velocity of an autonomous vehicle. In addition, we show that solving each challenge is essential for robust and effective robot learning, as a learning agent that addresses all of the challenges accrues more reward than agents missing any one of the components.

There are four main contributions of this article:

1. The use of regression trees to model continuous domains.
2. The use of random forests to provide targeted, limited exploration for an agent to quickly learn good policies.
3. A novel multi-threaded architecture that is the first to parallelize model learning in addition to planning and acting.
4. The complete implemented TEXPLORE algorithm, which is the first to address all of the previously listed challenges in a single algorithm.

The TEXPLORE algorithm and architecture presented in this paper has been fully implemented, empirically tested, and released publicly as a ROS package at: `http://www.ros.org/wiki/rl-texplore-ros-pkg`. With the code released as a ROS package, TEXPLORE can be easily downloaded and applied to a learning task on any robot running ROS with minimal effort. The goal of this algorithm and code release is to encourage more researchers to perform learning on robots using state-of-the-art algorithms.

This paper includes material from two conference papers: (Hester and Stone 2010) and (Hester et al 2012). (Hester and Stone 2010) includes material on learning models using random forests and has similar Fuel World experiments. (Hester et al 2012) presents the parallel architecture for real-time actions and some similar real-time car experiments. All the other contributions are unique to this article, including TEXPLORE's approach for learning in continuous and delayed domains, and all the experiments.

The remainder of this article is organized as follows. We present some background on RL and MDPs in Section 2 before describing the TEXPLORE algorithm in Section 3. Section 4 presents work related to each aspect of the algorithm. In Section 5, we demonstrate the ability of the algorithm to address each of the above challenges on a task that requires all of the components: learning to control the velocity of an autonomous vehicle in real-time. Finally, we present conclusions in Section 6.

2 Background

We adopt the standard Markov Decision Process (MDP) formalism for this work (Sutton and Barto 1998). An MDP is defined by a tuple $\langle S, A, R, T \rangle$, which consists of a set of states S, a set of actions A, a reward function $R(s,a)$, and a transition function $T(s,a,s') = P(s'|s,a)$. 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', determined from the probability distribution $P(s'|s,a)$. Many domains utilize a factored state representation, where the state s is represented by a vector of n state variables: $s = \langle s_1, s_2, ..., s_n \rangle$. A policy π specifies for each state which action the agent will take.

The value $Q^\pi(s,a)$ of a given state-action pair (s,a) is an estimate of the expected future reward that can be obtained from (s,a) when following policy π. 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 optimal value function $Q^*(s,a)$ provides maximal values in all states and is determined by solving the Bellman equation:

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

where $0 < \gamma < 1$ is the discount factor. The optimal policy π is then as follows:

$$\pi(s) = \mathrm{argmax}_a Q^*(s,a). \tag{2}$$

RL methods fall into two general classes: model-based and model-free methods. Model-based RL methods learn a model of the domain by approximating $R(s,a)$ and $P(s'|s,a)$ for each state and action. The agent can then calculate a policy (i.e. plan) using this model through a method such as value iteration (Sutton and Barto 1998) or UCT (Kocsis and Szepesvári 2006), effectively updating the Bellman equations for each state using its model. Model-free methods work without a model, updating the values of actions only when taking them in the real task. Generally model-based methods are more sample efficient than model-free methods. Model-free methods must visit each state many times for the value function to converge; while the sample efficiency of model-based methods is only constrained by how many samples it takes to learn a good model of the domain.

The agent's model of the domain can be learned using a number of techniques. A common approach is to use a maximum-likelihood tabular model where the agent learns a model for each state-action separately based on the frequencies of the seen outcomes. Alternatively, the agent could learn the model using a supervised learning technique, such as decision trees or Gaussian Process regression.

3 TEXPLORE

In this article, we introduce TEXPLORE (Hester and Stone 2010), a sample-efficient model-based real-time RL algorithm. When learning on robots, the agent has very few samples to learn since the samples may be expensive, dangerous, or time-consuming. Therefore, learning algorithms for robots must be greedier than typical methods in order to exploit their knowledge in the limited time they are given. Since these algorithms must perform limited exploration, their exploration must be efficient and target state-actions that may be promising for receiving reward. TEXPLORE achieves high sample efficiency by 1) utilizing the generalization properties of decision trees in building its model of the MDP, and 2) using random forests of those tree models to limit exploration to states that are promising for learning a good (but not necessarily optimal) policy quickly, instead of exploring more exhaustively to guarantee optimality. These two components constitute the key insights of the algorithm, and are explained in Section 3.2 (Model Learning) and Section 3.4 (Exploration). Modifications to the basic decision tree model enable TEXPLORE to operate in domains with continuous state spaces as well as domains with action or observation delays.

The other key feature of the algorithm is that it can act in real-time, at the frequencies required by robots (typically 5 - 20 Hz). For example, an RL agent controlling an autonomous vehicle must provide control signals to the gas and brake pedals immediately when a car in front of it slams on its brakes; it cannot stop to "think" about what to do. An alternative approach for acting in real-time would be to learn off-line and then follow the learned policy in real-time after the fact. However, we want the agent to be capable of learning on-line in-situ for the lifetime of the robot, adapting to new states and situations. TEXPLORE combines a multi-threaded architecture with Monte Carlo Tree Search (MCTS) to provide actions in real-time, by performing the model learning and planning in background threads while actions are returned in real-time. Each aspect of TEXPLORE is presented separately in Sections 3.1 to Section 3.4 before putting together the complete algorithm in Section 3.5.

3.1 Real-Time Architecture

In this section, we introduce TEXPLORE's parallel architecture, enabling it to return actions in real-time. Most current model-based RL methods use a sequential architecture such as the one shown in Figure 1. Pseudo-code for the sequential architecture is shown in Algorithm 1. In this sequential architecture, the agent receives a new state and reward; updates its model with the new transition $\langle s, a, s', r \rangle$

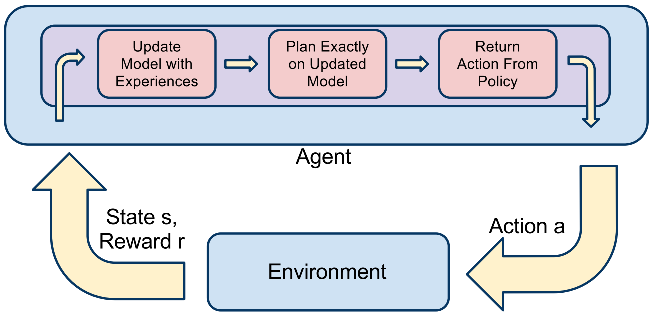


Fig. 1 A diagram of how model learning and planning are typically interleaved in a model-based agent.

(i.e. by updating a tabular model or adding a new training example to a supervised learner); plans exactly on the updated model (i.e. by computing the optimal policy with a method such as value iteration); and returns an action from its policy. Since both the model learning and planning can take significant time, this algorithm is not real-time. Alternatively, the agent may operate in batch mode (updating its model and planning on batches of experiences at a time), but this requires long pauses for the batch updates to be performed. Making the algorithm real-time requires two modifications to the standard sequential architecture: 1) utilizing sample-based approximate planning (presented in Section 3.1.1) and 2) developing a novel parallel architecture (presented in Section 3.1.2). We later evaluate this planning method and parallel architecture in comparison with other approaches in Section 5.4.

Algorithm 1 Sequential Model-Based Architecture

Input: S, A ▷ S: state space, A: action space
Initialize M to empty model
Initialize policy π randomly
Initialize s to a starting state in the MDP
loop
  Choose $a \leftarrow \pi(s)$
  Take action a, observe r, s'
  $M \Rightarrow$ UPDATE-MODEL($\langle s, a, s', r \rangle$) ▷ Update model M with new experience
  $\pi \leftarrow$ PLAN-POLICY(M) ▷ Exact planning on updated model
  $s \leftarrow s'$
end loop

3.1.1 Monte Carlo Tree Search (MCTS) Planning

The first component for providing actions in real-time is to use an anytime algorithm for approximate planning, rather than performing exact planning using a method such as value iteration. This section describes TEXPLORE's use of UCT for approximate planning as well as the modifications we have made to the algorithm. We have modified UCT to use λ-returns, generalize values across depths in the search tree, maintain value functions between selected actions, and work in continuous domains. All of these changes are described in detail below.

TEXPLORE follows the approach of Silver et al (2008) and Walsh et al (2010) (among others) in using a sample-based planning algorithm from the MCTS family (such as Sparse Sampling (Kearns et al 1999) or UCT (Kocsis and Szepesvári 2006)) to plan *approximately*. These sample-based planners use a generative model to sample ahead from the agent's current state, updating the values of the sampled actions. These methods can be more efficient than dynamic programming approaches such as value iteration or policy iteration in large domains because they focus their updates on states the agent is likely to visit soon rather than iterating over the entire state space.

The particular MCTS method that TEXPLORE uses is a variant of UCT (Kocsis and Szepesvári 2006), with pseudo-code shown in Algorithm 2. Our variation of UCT, called UCT(λ), uses λ-returns, similar to the TD-SEARCH algorithm (Silver et al 2012). UCT maintains visit counts for each state to calculate confidence bounds on the action-values. UCT differs from other MCTS methods by sampling actions more greedily by using the UCB1 algorithm (Auer et al 2002), shown on line 29. UCT selects the action with the highest upper confidence bound (with ties broken uniformly randomly). The upper confidence bound is calculated using the visit counts, c, to the state and each action, as well as the range of possible discounted returns in the domain, $\frac{r_{max} - r_{min}}{1-\gamma}$. Selecting actions this way drives the agent to concentrate its sampling on states with the best values, while still exploring enough to find the optimal policy.

UCT samples a possible trajectory from the agent's current state. On line 30 of Algorithm 2, the model is queried for a prediction of the next state and reward given the state and selected action (QUERY-MODEL is described in detail later in Section 3.2 and shown in Algorithm 4). UCT continues sampling forward from the given next state. This process continues until the sampling has reached a terminal state or the maximum search depth, $maxDepth$. Then the algorithm updates the values of all the state-actions encountered along the trajectory. In normal UCT the *return* of a sampled trajectory is the discounted sum of rewards received on that trajectory. The value of the initial state-action is updated towards this return, completing one *rollout*. The algorithm does many rollouts to obtain an accurate estimate of the values of the actions at the agent's current state. UCT is proven to converge to an optimal value function with respect to the model at a polynomial rate as the number of rollouts goes to infinity (Kocsis and Szepesvári 2006).

We have modified UCT to update the state-actions using λ-returns, which average rewards received on the simulated trajectory with updates towards the estimated values of the states that the trajectory reached (Sutton and Barto 1998). Informal experiments showed that using intermediate values of λ $(0 < \lambda < 1)$ provided better results than using the default UCT without λ-returns.

In addition to using λ-returns, we have also modified UCT to generalize values across depths in the tree, since the value of a state-action in an infinite horizon discounted MDP is the same no matter when in the search it is encountered (due to the Markov property). One possible concern with this approach is that states at the bottom of the search tree may have poor value estimates because the search does not continue for many steps after reaching them. However, these states are not severely affected, since the λ-returns update them towards the values of the next states.

Most importantly, UCT is an anytime method, and will return better policies when given more time. By replacing the PLAN-POLICY call on line 9 of Algorithm 1, which performs exact planning, with PLAN-POLICY from Algorithm 2, which performs approximate planning, the sequential architecture could be made faster. TEXPLORE's real-time architecture, which is presented later in Algorithm 3, also uses UCT(λ) for planning.

UCT(λ) maintains visit counts for each state and state-action to determine confidence bounds on its action-values. When the model that UCT(λ) is planning on changes, its value function is likely to be incorrect for the updated model. Rather than re-planning entirely from scratch, the value function UCT(λ) has already learned can be used to speed up the learning of the value function for the new model. TEXPLORE's approach to re-using the previously learned value function is similar to the way Gelly and Silver (2007) incorporate off-line knowledge of the value function by providing an estimate of the value function and a visit count that represents the confidence in this value function. When UCT(λ)'s model is updated, the visit counts for all states are reset to a lower value that encourages UCT(λ) to explore again, but still enables UCT(λ) to take advantage of the value function learned for the previous model. The UCT-RESET procedure does so by resetting the visit counts for all state-actions to $resetCount$, which will be a small non-zero value. If the exact effect the change of the model would have on the value function is known, $resetCount$ could be set based on this change, with higher values for smaller effects. However, TEXPLORE does not track the changes in the model, and even a small change in the model can have a drastic effect on the value function.

Some modifications must be made to use UCT(λ) on domains with continuous state spaces. One advantage of using UCT(λ) is that rather than planning ahead of time over a discretized state space, UCT(λ) can perform rollouts through the exact real-valued states the agent is visiting, and query the model for the real-valued state predictions. However, it cannot maintain a table of values for an infinite number of states. Instead, it discretizes the state on line 28 by discretizing each state feature into $nBins_i$ possible values. Since the algorithm is only using the discretization for the value function update, and not for the modeling or planning rollouts, it works well even on fine discretizations in high-dimensional domains. Then the algorithm updates the value and visit counts for the discretized state on lines 32 to 34.

3.1.2 Parallel Architecture

In addition to using MCTS for planning, we have developed a multi-threaded architecture, called the Real-Time Model Based Architecture (RTMBA), for the agent to learn while acting in real-time (Hester et al 2012). Since UPDATE-MODEL and PLAN-POLICY can take significant computation (and thus also wall-clock time), they are placed in parallel threads in the background, as shown in Figure 2. A third thread selects actions as quickly as dictated by the robot control loop, while still being based on the most recent models and plans available. Pseudo-code for all three threads is shown in Algorithm 3. This architecture is general, allowing for any type of model learning method, and only requiring any method from the MCTS family for planning. In addition to enabling real-time actions, this architecture enables the agent to take full advantage of multi-core processors by running each thread on a separate core. Similar approaches have been taken to parallelize MCTS

Algorithm 2 PLAN: UCT(λ)

1: **procedure** UCT-INIT($S, A, maxDepth, resetCount, rmax, nBins, minVals, maxVals$)
2: Initialize $Q(s,a)$ with zeros for all $s \in S, a \in A$
3: Initialize $c(s,a)$ with ones for all $s \in S, a \in A$ ▷ To avoid divide-by-zero
4: Initialize $c(s)$ with zeros for all $s \in S$ ▷ Visit Counts
5: **end procedure**

6: **procedure** PLAN-POLICY(M, s) ▷ Approximate planning from state s using model M
7: UCT-RESET()
8: **while** time available **do**
9: UCT-SEARCH($M, s, 0$)
10: **end while**
11: **end procedure**

12: **procedure** UCT-RESET() ▷ Lower confidence in v.f. since model changed
13: **for all** $s_{disc} \in S_{disc}$ **do** ▷ For all discretized states
14: **if** $c(s_{disc}) > resetCount \cdot |A|$ **then**
15: $c(s_{disc}) \leftarrow resetCount \cdot |A|$ ▷ $resetCount$ per action
16: **end if**
17: **for all** $a \in A$ **do**
18: **if** $c(s_{disc}, a) > resetCount$ **then**
19: $c(s_{disc}, a) \leftarrow resetCount$
20: **end if**
21: **end for**
22: **end for**
23: **end procedure**

24: **procedure** UCT-SEARCH(M, s, d) ▷ Rollout from state s at depth d using model M
25: **if** TERMINAL or d $= maxDepth$ **then**
26: **return** 0
27: **end if**
28: $s_{disc} \leftarrow$ DISCRETIZE($s, nBins, minVals, maxVals$) ▷ Get discretized version of state s
29: $a \leftarrow \text{argmax}_{a'} \left(Q(s_{disc}, a') + 2 \cdot \frac{r_{max} - r_{min}}{1-\gamma} \cdot \sqrt{\frac{\log c(s_{disc})}{c(s_{disc}, a')}} \right)$ ▷ Note: Ties broken randomly
30: $(s', r) \leftarrow M \Rightarrow$QUERY-MODEL($s, a$) ▷ Algorithm 4
31: $sampleReturn \leftarrow r + \gamma$UCT-SEARCH($M, s', d+1$) ▷ Continue rollout from state s'
32: $c(s_{disc}) \leftarrow c(s_{disc}) + 1$ ▷ Update counts
33: $c(s_{disc}, a) \leftarrow c(s_{disc}, a) + 1$
34: $Q(s_{disc}, a') \leftarrow \alpha \cdot sampleReturn + (1-\alpha) \cdot Q(s_{disc}, a')$
35: **return** $\lambda \cdot sampleReturn + (1-\lambda) \cdot \max_{a'} Q(s_{disc}, a')$ ▷ Use λ-returns
36: **end procedure**

planning and acting (Gelly et al 2008; Chaslot et al 2008; Méhat and Cazenave 2011) by performing multiple rollouts in parallel, but they have not incorporated parallel model learning as well.

For the three threads to operate properly, they must share information while avoiding race conditions and data inconsistencies. The model learning thread must know which new transitions to add to its model, the planning thread must access the model being learned and know what state the agent is currently at, and the action thread must access the policy being planned. RTMBA uses mutex locks to control access to these variables, as summarized in Table 1.

The action thread (lines 26 to 35) receives the agent's new state and reward, and adds the new transition experience, $\langle s, a, s', r \rangle$, to the *updateList* to be updated into the model. It then saves the agent's current state in *agentState* for use by the planner and returns the action determined by the agent's value function,

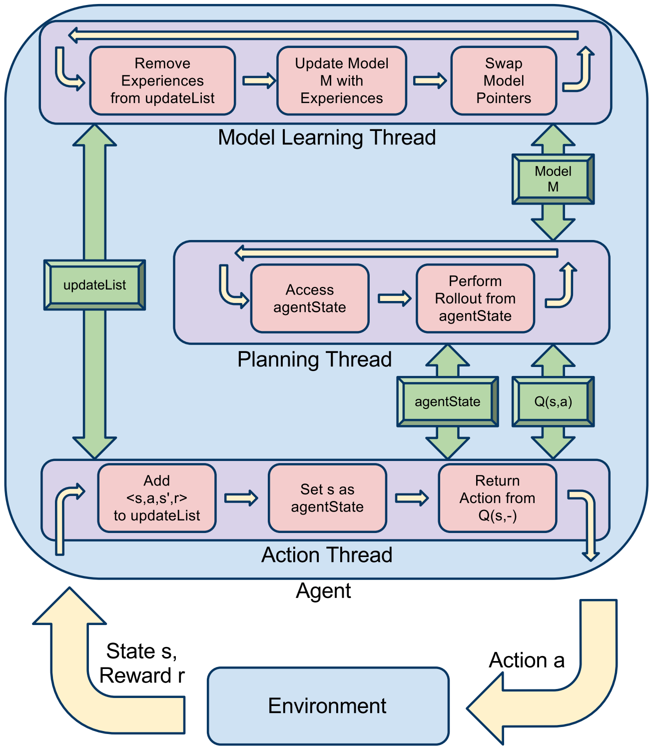


Fig. 2 A diagram of the proposed parallel architecture for real-time model-based RL.

Variable	Threads	Use
updateList	Action, Model Learning	Store experiences to be updated into model
agentState	Action, Planning	Set current state to plan from
$Q(s,a)$	Action, Planning	Update policy used to select actions
M	Planning, Model Learning	Latest model to plan on

Table 1 This table shows all the variables that are protected under mutex locks in the proposed architecture, along with their purpose and which threads use them.

Q. Since *updateList*, *agentState*, and Q are protected by mutex locks, it is possible that the action thread could have to wait for a mutex lock before it could proceed. However, *updateList* is only used by the model learning thread between model updates, *agentState* is only accessed by the planning thread between each rollout, and Q is under individual locks for each state. Thus, any given state is freely accessible most of the time. When the planner does happen to be using the same state the action thread wants, it releases it immediately after updating the values for that state. Therefore, there is never a long wait for mutex locks, and the action thread can return actions quickly when required.

Algorithm 3 Real-Time Model-Based Architecture (RTMBA)

1: **procedure** INIT ▷ Initialize variables
2: **Input:** $S, A, nBins, minVals, maxVals$ ▷ $nBins$ is the # of discrete values for each feature
3: Initialize s to a starting state in the MDP
4: $agentState \leftarrow s$
5: $updateList \leftarrow \emptyset$
6: Initialize M to empty model
7: UCT-INIT() ▷ Initialize Planner
8: **end procedure**

9: **procedure** MODELLEARNINGTHREAD ▷ Model Learning Thread
10: **loop** ▷ Loop, adding experiences to model
11: **while** $updateList = \emptyset$ **do**
12: Wait for experiences to be added to list
13: **end while**
14: $tmpModel \leftarrow M \Rightarrow$COPY ▷ Make temporary copy of model
15: $tmpModel \Rightarrow$UPDATE-MODEL($updateList$) ▷ Update model $tmpModel$ (Alg 4)
16: $updateList \leftarrow \emptyset$ ▷ Clear the update list
17: UCT-RESET() ▷ Less confidence in current values
18: $M \leftarrow tmpModel$ ▷ Swap model pointers
19: **end loop**
20: **end procedure**

21: **procedure** PLANNINGTHREAD ▷ Planning Thread
22: **loop** ▷ Loop forever, performing rollouts
23: UCT-SEARCH($M, agentState, 0$) ▷ Algorithm 2
24: **end loop**
25: **end procedure**

26: **procedure** ACTIONTHREAD ▷ Action Selection Thread
27: **loop**
28: $s_{disc} \leftarrow$ DISCRETIZE($s, nBins, minVals, maxVals$) ▷ Get discretized version of state s
29: Choose $a \leftarrow \text{argmax}_a Q(s_{disc}, a)$
30: Take action a, Observe r, s'
31: $updateList \leftarrow updateList \cup \langle s, a, s', r \rangle$ ▷ Add experience to update list
32: $s \leftarrow s'$
33: $agentState \leftarrow s$ ▷ Set agent's state for planning rollouts
34: **end loop**
35: **end procedure**

The model learning thread (lines 9 to 20) checks if there are any experiences in $updateList$ to be added to its model. If there are, it makes a copy of its model to $tmpModel$, updates $tmpModel$ with the new experiences, and clears $updateList$. Then it resets the planning visit counts to $resetCount$ to lower the planner's confidence in the out-dated value function, which was calculated on an old model. Finally, on line 18, it replaces the original model with the updated copy. The other threads can continue accessing the original model while the copy is being updated, since only the swapping of the models requires locking the model mutex. After updating the model, the model learning thread repeats, checking for new experiences to add to the model.

The model learning thread can call any type of model on line 15, such as a tabular model (Brafman and Tennenholtz 2001), a Gaussian Process regression model (Deisenroth and Rasmussen 2011), or the random forest model used by

TEXPLORE, which is described in Section 3.2. Depending on how long the model update takes and how fast the agent is acting, the agent can add tens or hundreds of new experiences to its model at a time, or it can wait for long periods for a new experience. When adding many experiences at a time, full model updates are not performed between each individual action. In this case, the algorithm's sample efficiency is likely to suffer compared to that of sequential methods, but in exchange, it continues to act in real time.

Though TEXPLORE uses a variant of UCT, the planning thread can use any MCTS planning algorithm. The thread retrieves the agent's current state ($agentState$) and its planner performs a rollout from that state. The rollout queries the latest model, M, to update the agent's value function. The thread repeats, continually performing rollouts from the agent's current state. With more rollouts, the algorithm's estimates of action-values improve, resulting in more accurate policies. Even if very few rollouts are performed from the current state before the algorithm returns an action, many of the rollouts performed from the previous state should have gone through the current state (if the model is accurate), giving the algorithm a good estimate of the state's true action-values.

3.2 Model Learning

While the parallel architecture presented above enables TEXPLORE to operate in real-time, the algorithm must learn an accurate model of the domain quickly to learn the task with high sample efficiency. Although tabular models are a common approach, they require the agent to take every action from each state once (or multiple times in stochastic domains), since they learn a prediction for each state-action separately. Instead, TEXPLORE uses supervised learning techniques to *generalize* the effects of actions across states, as has been done by some previous algorithms (Degris et al 2006; Jong and Stone 2007). Since the *relative* transition effects of actions are similar across states in many domains, TEXPLORE follows the approach of Leffler et al (2007) and Jong and Stone (2007) in predicting relative transitions rather than absolute outcomes. In this way, model learning becomes a supervised learning problem with (s, a) as the input and $s' - s$ and r as the outputs to be predicted. Model learning is sped up by the ability of the supervised learner to make predictions for unseen or infrequently visited states.

Like Dynamic Bayesian Network (DBN) based RL algorithms (Guestrin et al 2002; Strehl et al 2007; Chakraborty and Stone 2011), the algorithm learns a model of the factored domain by learning a separate prediction for each of the n state features and the reward, as shown in Algorithm 4. The MDP model is made up of n models to predict each feature ($featModel_1$ to $featModel_n$) and a model to predict reward ($rewardModel$). Each model can be queried for a prediction for a particular state-action ($featModel \Rightarrow \text{QUERY}(\langle s, a\rangle)$) or updated with a new training experience ($featModel \Rightarrow \text{UPDATE}(\langle s, a, out\rangle)$). In TEXPLORE, each of these models is a random forest, presented in Section 3.4 as Algorithm 7.

Algorithm 4 shows TEXPLORE's model learning algorithm. It starts by calculating the relative change in the state (s^{rel}) on line 9, then it updates the model for each feature with the new transition on line 11 and updates the reward model on line 13. Like DBN-based algorithms, TEXPLORE assumes that each of the state variables transitions independently. Therefore, the separate feature predictions can be combined to create a prediction of the complete state vector. The agent samples a prediction of the value of the change in each feature on line 18 and adds this vector, s^{rel}, to s to get a prediction of s'. The agent then samples a prediction of reward (line 21) and these sampled predictions are returned for planning with MCTS.

Algorithm 4 MODEL

1: **procedure** INIT-MODEL(n) ▷ n is the number of state variables
2: **for** $i = 1 \rightarrow n$ **do**
3: $featModel_i \Rightarrow$INIT() ▷ Init model to predict feature i
4: **end for**
5: $rewardModel \Rightarrow$INIT() ▷ Init model to predict reward
6: **end procedure**

7: **procedure** UPDATE-MODEL($list$) ▷ Update model with $list$ of experiences
8: **for all** $\langle s, a, s', r \rangle \in list$ **do**
9: $s^{rel} \leftarrow s' - s$ ▷ Calculate relative effect
10: **for all** $s_i^{rel} \in s^{rel}$ **do**
11: $featModel_i \Rightarrow$UPDATE($\langle s, a \rangle, s_i^{rel}$) ▷ Train a model for each feature
12: **end for**
13: $rewardModel \Rightarrow$UPDATE($\langle s, a \rangle, r$) ▷ Train a model to predict reward
14: **end for**
15: **end procedure**

16: **procedure** QUERY-MODEL(s, a) ▷ Get prediction of $\langle s', r \rangle$ for s, a
17: **for** $i = 1 \rightarrow$ LENGTH(s) **do**
18: $s_i^{rel} \leftarrow featModel_i \Rightarrow$QUERY($\langle s, a \rangle$) ▷ Sample a prediction for feature i
19: **end for**
20: $s' \leftarrow s + \langle s_1^{rel}, ..., s_n^{rel} \rangle$ ▷ Get absolute next state
21: $r \leftarrow rewardModel \Rightarrow$QUERY($\langle s, a \rangle$) ▷ Sample r from distribution
22: **return** $\langle s', r \rangle$ ▷ Return sampled next state and reward
23: **end procedure**

We tested the applicability of several different supervised learning methods to the task of learning an MDP model in previous work (Hester and Stone 2009). Decision trees, committees of trees, random forests, support vector machines, neural networks, nearest neighbor, and tabular models were compared on their ability to predict the transition and reward models across three toy domains after being given a random sample of experiences in the domain. Decision tree based models (single decision trees, committees of trees, and random forests) consistently provided the best results. Decision trees generalize broadly and can be refined to make accurate predictions at all states. Another reason decision trees perform well is that in many domains, the state space can be split into regions with similar dynamics. For example, on a vehicle, the dynamics can be split into different regions corresponding to which gear the car is in.

Based on these results, TEXPLORE uses decision trees to learn models of the transition and reward functions. The decision trees are learned using an implementation of Quinlan's C4.5 algorithm (Quinlan 1986). The inputs to the decision trees are treated both as numerical and categorical inputs, meaning both splits of the type **if** $\text{x} = 3$ and **if** $\text{x} > 3$ are allowed. The C4.5 algorithm chooses the split at each node of the tree based on information gain. TEXPLORE's 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.

The decision trees are the supervised learner that is called on lines 11, 13, 18, and 21 of Algorithm 4 to predict each feature and reward. Each tree makes predictions for the particular feature or reward it is given based on a vector containing the n features of the state s along with the action a: $\langle s_1, s_2, ..., s_n, a \rangle$. This same vector is used when querying the trees for the change in each feature on line 18 and for reward on line 21.

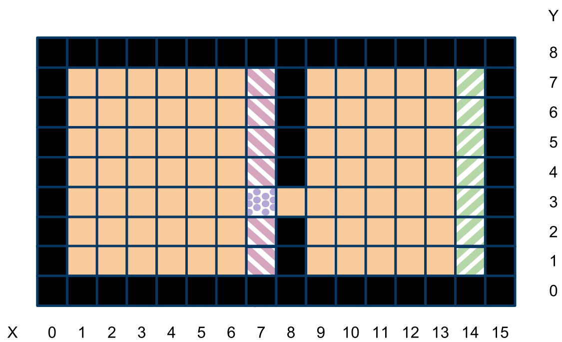
Figure 3 shows an example decision tree predicting the relative change in the x variable of the agent in the given gridworld domain. The decision tree can split on both the actions and the state of the agent, allowing it to split the state space up into regions where the transition dynamics are the same. Each leaf of the tree can make probabilistic predictions based on the ratio of experienced outcomes in that leaf. The grid is shaded 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 then it will generalize broadly, making predictions about large parts of the state space, such as what the EAST or WEST actions do. For unvisited state-actions, the tree will predict that the outcome is the same as that of similar state-actions (ones in the same leaf of the tree). It will continue to refine itself until it has leaves for individual states where the transition dynamics differ from the global dynamics.

3.2.1 Models of Continuous Domains

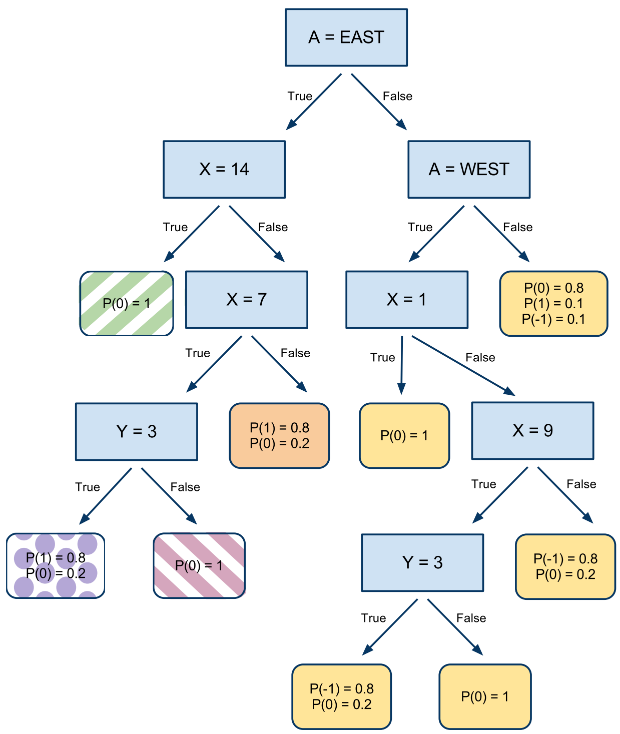
While decision trees work well for discrete domains, TEXPLORE needs to be capable of modeling continuous domains as well. Discretizing the domain is one option, but important information is lost in the discretization. Not only is noise added by discretizing the continuous state, but the discrete model does not model the function underlying the dynamics and thus cannot generalize predictions to unseen states very well.

To extend the discrete decision trees to the continuous case, TEXPLORE uses linear regression trees, learned using the M5 algorithm (Quinlan 1992). The M5 algorithm builds these decision trees in a similar manner to the C4.5 algorithm, greedily choosing each split to reduce the variance on each side. Once the tree is fully built, it is pruned by replacing some tree splits with linear regression models. Going up the tree from the leaves, a sub-tree is replaced by a linear regression model if the regression model has less prediction error on the training set than the sub-tree. The result is a smaller tree with regression models in each leaf, rather than each leaf making a discrete class prediction. The linear regression trees will fit a piecewise linear model to the dynamics of the domain. Similar trees have been used to approximate the value function (Munos and Moore 2002; Ernst et al 2005), but not for the approximating the transition and reward model of a domain.

Figure 4 shows an example of how the regression trees can result in simpler models that are faster to build and make more accurate predictions than discrete decision trees. Figure 4(a) shows the predictions of the discrete tree approximating the underlying function. The model requires examples of the output at each discrete level to make an accurate prediction and cannot generalize beyond these seen examples. In contrast, the regression trees make a piecewise linear prediction, with each leaf predicting a linear function. This type of model can fit the data

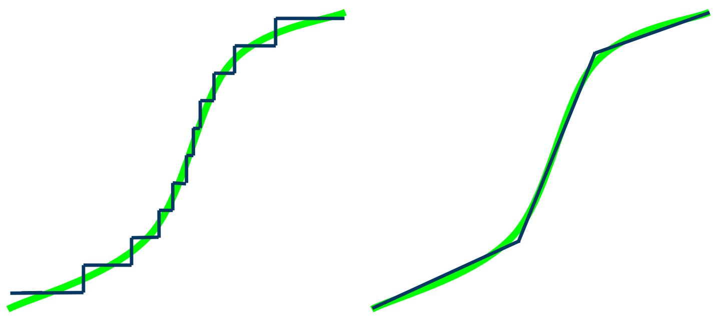


(a) Two room gridworld domain.



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

Fig. 3 This figure shows the decision tree model learned to predict the change in the x feature (or Δx). The two room gridworld is shaded 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 $x = 14$, the agent is hitting the right wall. This input falls into the leaf on the top left, where the probability of $\Delta x = 0$ is 1.



(a) Discrete Trees. (b) Regression Trees.

Fig. 4 An example of a function estimated by 4(a): discrete trees and 4(b): regression trees. Note that the regression tree is able to fit the function better than the discrete tree.

more closely and makes predictions for unseen parts of the space by extrapolating the linear function from nearby regions.

3.3 Domains with Delays

We are particularly interested in applying TEXPLORE to robots and other physical devices, but one common problem with these devices is that their sensors and actuators often have delays. For example, a robot's motors may be slow to start moving, and thus the robot may still be executing (or yet to execute) the last action given to it when the algorithm selects the next action. This is important, as the algorithm must take into account what the state of the robot will be when the action actually gets executed, rather than the state of the robot when the algorithm makes the action selection. TEXPLORE should model these delays and handle them efficiently.

Modeling and planning on domains with delay can be done by taking advantage of the k-Markov property (Katsikopoulos and Engelbrecht 2003). While the next state and reward in these domains is not Markov with respect to the current state, it is Markov with respect to the previous k states. TEXPLORE's approach to addressing delays is inspired by the U-TREE algorithm (McCallum 1996), using data from the last k experiences. The key insight of U-TREE is to allow its decision trees to split on previous states and actions in addition to the current state and action, enabling it to work in partially observable domains where the state alone is not enough to make an accurate prediction.

TEXPLORE adopts the same approach for delayed domains. The action thread is modified to keep a history of the last k actions (shown in Algorithm 5), which is sufficient to make the domain Markov. In addition to the current state and action, the thread appends the past k actions as inputs for each decision tree to use for its predictions. Any of these inputs can be used for splits in the decision tree. One of the advantages of decision trees over other models is that they can choose relevant inputs when making splits in the tree. Thus, even if the value of k input to the algorithm is higher than the true delay in the domain, the tree will ignore the extra inputs and still build an accurate model. Model learning approaches based on prediction suffix trees are similar, but require splits to be made in order on the most recent observations and actions first (Willems et al 1995; Veness et al 2011).

Algorithm 5 Action Thread with Delays

```
procedure ActionThread                                  ▷ Action Selection Thread
    history ← ∅
    loop
        s_disc ← DISCRETIZE(s, nBins, minVals, maxVals)
        Choose a ← argmax_a Q(s_disc, history, a)         ▷ Values of state-history-actions
        Take action a, Observe r, s′
        augState ← ⟨s, history⟩                          ▷ Augment state with history
        updateList ← updateList ∪ ⟨augState, a, s′, r⟩
        PUSH(history, a)                                 ▷ Keep last k actions
        if LENGTH(history) > k then
            POP(history)
        end if
        s ← s′
        agentState ← s                                   ▷ Set agent's state for planning rollouts
    end loop
end procedure
```

Similarly, TEXPLORE takes advantage of the k-Markov property for planning, by slightly modifying UCT(λ). Algorithm 6 shows the modified UCT(λ)-SEARCH algorithm. In addition to the agent's state, it also takes the history of k actions. While performing the rollout, it updates the history at each step (lines 9 to 12), and uses the augmented state including history when querying the model (line 8). States may have different optimal actions when reached with a different history, as different actions will be applied before the currently selected action takes place. This problem can be remedied by planning over an augmented state space that incorporates the k-action histories, shown in the visit count and value function updates in lines 14 to 16. Katsikopoulos and Engelbrecht (2003) have shown that solving this augmented MDP provides the optimal solution to the delayed MDP. However, the state space increases by a factor of $|A|^k$. While this would greatly increase the computation required by a planning method such as value iteration that iterates over all the states, UCT(λ) focuses its updates on the states (or augmented state-histories) the agent is likely to visit soon, and thus its computation time is not greatly affected. Note that with $k = 0$, the *history* will be $\emptyset$ and the action thread and UCT(λ) search methods presented here will exactly match the ones presented in Algorithms 3 and 2, respectively. Later, in Section 5.3, we evaluate the performance of TEXPLORE's approach for handling delays in comparison with other approaches.

This version of UCT(λ) planning on the augmented state space is similar to the approach taken for planning inside the MC-AIXI algorithm (Veness et al 2011). The difference is that their algorithm performs rollouts over a history of previous state-action-reward sequences, while TEXPLORE uses the current state along with only the previous k actions. One thing to note is that while TEXPLORE's approach is intended to address delays, it can also be used to address partial observability, if a sufficient k is chosen such that the domain is k-Markov.

Addressing action delays by utilizing k-action histories integrates well with TEXPLORE's approaches for model learning and planning. TEXPLORE's decision tree models select which delayed action inputs provide the most information gain while making splits in the tree, and can ignore the delayed actions that are not relevant for the task at hand. In addition, planning with UCT(λ) is easily modified to track histories while performing rollouts; planning with a method such as value iteration would require the agent to plan over a state space that is $|A|^k$ times bigger. Using k-action histories for delays is one example of how the various components of TEXPLORE are synergistic.

Algorithm 6 UCT(λ) with delays

procedure SEARCH($M, s, history, d$) ▷ Rollout from state s with *history*
    if TERMINAL or d $= maxDepth$ **then**
        return 0
    end if
    $s_{disc} \leftarrow$ DISCRETIZE($s, nBins, minVals, maxVals$)
    $a \leftarrow \text{argmax}_{a'} \left(Q(s_{disc}, history, a') + 2 \cdot \frac{r_{max}-r_{min}}{1-\gamma} \cdot \sqrt{\frac{\log c(s_{disc}, history)}{c(s_{disc}, history, a')}} \right)$
    $augState \leftarrow \langle s, history \rangle$
    $(s', r) \leftarrow M \Rightarrow$QUERY-MODEL($augState, a$)
    PUSH($history, a$) ▷ Keep last k actions
    if LENGTH($history$) $> k$ **then**
        POP($history$)
    end if
    $sampleReturn \leftarrow r + \gamma$SEARCH($M, s', history, d+1$)
    $c(s_{disc}, history) \leftarrow c(s_{disc}, history) + 1$ ▷ Update counts
    $c(s_{disc}, history, a) \leftarrow c(s_{disc}, history, a) + 1$
    $Q(s_{disc}, history, a') \leftarrow \alpha \cdot sampleReturn + (1-\alpha) \cdot Q(s_{disc}, history, a')$
    return $\lambda \cdot sampleReturn + (1-\lambda) \cdot \max_{a'} Q(s_{disc}, history, a')$
end procedure

3.4 Exploration

Our goal is to perform learning on robots, where taking hundreds or thousands of actions is impractical. Therefore, our learning algorithm needs to limit the amount of exploration it performs so that it has time to exploit its knowledge within this limited timeframe. On such domains with a constrained number of actions, it is better for the agent to quickly converge to a good policy than to explore more exhaustively to learn the optimal policy. With this idea in mind, our algorithm performs limited exploration, which is targeted on state-actions that appear promising for the final policy, while avoiding state-actions that are unlikely to be useful for the final policy.

Using decision trees to learn the model of the MDP provides TEXPLORE with a model that can be learned quickly with few samples. However, each tree represents just one possible hypothesis of the true model of the domain, which may be generalized incorrectly. Rather than planning with respect to this single model, our algorithm plans over a distribution of possible tree models (in the form of a random forest) to drive exploration. A random forest is a collection of decision trees, each of which differ because they are trained on a random subset of experiences and have some randomness when choosing splits at the decision nodes. Random forests have been proven to converge with less generalization error than individual tree models (Breiman 2001).

Algorithm 7 presents pseudo-code for the random forest model. Each of the m decision trees ($tree_1$ to $tree_m$) in the forest can be updated with a new input-output pair ($tree \Rightarrow$ UPDATE(in, out)) or queried for a prediction for a given input ($tree \Rightarrow$ QUERY(in)). This algorithm implements the MODEL that is called on lines 11, 13, 18, and 21 of Algorithm 4. Each tree is trained on only a subset of the agent's *experiences* ($\langle s, a, s', r\rangle$ tuples), as it is updated with each new experience with probability w (line 8). To increase stochasticity in the models, at each split in the tree, the best input is chosen from a random subset of the inputs, with each one removed from this set with probability f. When UCT(λ) requests a prediction from the random forest model, it only needs to return the prediction of a single tree in the forest, which saves some computation.

Algorithm 7 MODEL: Random Forest

1: **procedure** INIT(m) ▷ Init forest of m trees
2:   **for** $i = 1 \rightarrow m$ **do**
3:     $tree_i \Rightarrow$ INIT() ▷ Init tree i
4:   **end for**
5: **end procedure**

6: **procedure** UPDATE(in, out) ▷ Update forest with (in, out) example
7:   **for** $i = 1 \rightarrow m$ **do** ▷ For m trees in the random forest
8:     **if** RAND() $\leq w$ **then** ▷ Update each tree with prob. w
9:       $tree_i \Rightarrow$ UPDATE(in, out)
10:     **end if**
11:   **end for**
12: **end procedure**

13: **procedure** QUERY(in) ▷ Get prediction for in
14:   $i =$ RAND$(1, m)$ ▷ Select a random tree from forest
15:   $x \leftarrow tree_i \Rightarrow$ QUERY(in) ▷ Get prediction from tree i
16:   **return** x ▷ Return prediction
17: **end procedure**

There are a number of options regarding how to use the m hypotheses of the domain model to drive exploration. BOSS (Asmuth et al 2009) is a Bayesian method that provides one possible example. BOSS samples m model hypotheses from a distribution over possible models. The algorithm plans over actions from any of the models, enabling the agent to use the most optimistic model for each state-action. With m models, the value function is calculated as follows, with the subscript on Q_i, R_i, and P_i representing that it is from model i:

$$Q(s, a) = \max_i Q_i(s, a) \tag{3}$$

$$Q_i(s, a) = R_i(s, a) + \gamma \sum_{s'} P_i(s'|s, a) \max_{a'} Q(s', a'), \tag{4}$$

The policy of the agent is then:

$$\pi(s) = \text{argmax}_a Q(s, a). \tag{5}$$

The agent plans over the most optimistic model for each state-action. Since one of the models is likely to be optimistic with respect to the true environment in each state, the agent is guaranteed to explore enough to find the optimal policy in a polynomial number of steps.

Model Based Bayesian Exploration (MBBE) (Dearden et al 1999) is another Bayesian method that uses model samples for exploration. It samples and solves m models to get a distribution over action-values. The action-values for each model i are:

$$Q_i(s,a) = R_i(s,a) + \gamma \sum_{s'} P_i(s'|s,a) \max_{a'} Q_i(s',a'). \tag{6}$$

Note that this differs from BOSS in that the next state values are using the same model i, rather than a value from an optimistic merged model. The expected value, $E[Q(s,a)]$, for a particular state-action is then the average of its value for each model. Using the expected action-values, at any given state the agent has a best action a_1 and a second best action a_2. MBBE uses the distribution over action-values to calculate how much the agent's policy will improve if it learns that a particular model i is correct:

$$Gain_i(s,a) = \begin{cases} E[Q(s,a_2)] - Q_i(s,a), & \text{if } a = a_1 \text{ and } Q_i(s,a) < E[Q(s,a_2)], \\ Q_i(s,a) - E[Q(s,a_1)], & \text{if } a \neq a_1 \text{ and } Q_i(s,a) > E[Q(s,a_1)], \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

The first case is if model i predicts that the value of the best action, a_1, is not as good as expected and is less than the expected value of action a_2. The second case is if model i predicts that another action would have a better value than a_1. In either case the gain is the improvement in the value function for the given state action pair. This value of perfect information (VPI) for a state-action is then the average of the gains for that state-action for each model. This value is added to the expected action-values to calculate the action-values that the agent maximizes for its policy:

$$Q(s,a) = \frac{1}{m} \sum_{i=1}^{m} Q_i(s,a) + Gain_i(s,a). \tag{8}$$

When the sampled models are optimistic or pessimistic compared to the true MDP, the agent is encouraged to explore. With an optimistic model, the agent's policy would be improved if the model is correct and this improvement is reflected in the VPI for this model. With a pessimistic model, the agent would be driven to explore the state-action because it would gain the knowledge that its policy is poor and should not be followed. Thus, this approach drives the agent to explore state-actions thoroughly to find the optimal policy.

For the goal of learning on robots, learning in polynomial time is not fast enough. Both BOSS and MBBE explore thoroughly; on problems with very large (or continuous) state-action spaces, they could take many hundreds or thousands of time-consuming, expensive, and possibly dangerous actions to learn a policy. The key insight of our approach is to be *greedier* than these methods in order to learn in fewer actions. TEXPLORE performs less exploration than these approaches and thus exploits more of what it has learned. Since TEXPLORE is doing less exploration, the exploration it does perform must be targeted on state-actions that appear promising. In other words, with such limited exploration, TEXPLORE cannot afford to explore state-actions that may lead to low-valued outcomes (it decides *not* to explore such state-actions).

Rather than using exploration bonuses or optimistic models like BOSS and MBBE, TEXPLORE plans greedily with respect to a distribution of m model hypotheses. TEXPLORE's action-values are then:

$$Q(s,a) = \frac{1}{m}\sum_{i=1}^{m} R_i(s,a) + \gamma \frac{1}{m}\sum_{i=1}^{m}\sum_{s'} P_i(s'|s,a)\max_{a'} Q(s',a'). \tag{9}$$

Each decision tree in the random forest generalizes transitions differently, resulting in different hypotheses of the true MDP. As each tree model's predictions differ more, the predictions from the aggregate model become more stochastic. For example, if each of five trees predict a different next state, then the aggregate model will have a uniform distribution over these five possible next states. The aggregate model includes some probability of transitioning to the states and rewards predicted by the optimistic models as well as those predicted by the pessimistic ones. Thus, planning on the aggregate model makes the agent balance the likelihood that the transitions predicted by the optimistic and pessimistic model will occur. The agent will explore towards state-actions that some models predict to have higher values while avoiding those that are predicted to have low values.

Another benefit of planning on this aggregate model is that it enables TEXPLORE to explore multiple possible generalizations of the domain, as it can explore state-actions that are promising in any one of the hypotheses in the aggregate model. In contrast, if TEXPLORE acted using a single hypothesis of the task model, then it would not know about state-actions that are only promising in other possible generalizations of its past experience. Figure 5 shows a diagram of how the entire model learning system works. In Section 5.1, we evaluate TEXPLORE's exploration in comparison with other approaches.

Using an aggregate model provides a few other advantages compared to prior approaches. The aggregate random forest model provides less generalization error than simply sampling a single decision tree model and using it (Breiman 2001). Another advantage of TEXPLORE over BOSS and MBBE is that both of these methods require more planning, which can take more computation time. BOSS must plan over a state space with m times more actions than the true environment, while MBBE must plan for each of its m different models. In contrast, TEXPLORE plans on a single model with the original $|S||A|$ state-actions.

As an example, imagine TEXPLORE with $m = 5$ models is learning to control a humanoid robot to kick a ball by shifting its weight and swinging its leg. If it shifts its weight more than 5 cm to one side, the robot will fall over, resulting in a negative reward of -1000. If the robot kicks successfully, it gets a reward of 20. Until TEXPLORE has experienced the robot falling over, it will not predict it is possible. If TEXPLORE finds a successful kicking policy without ever falling over during its exploration, then it will have avoided falling over entirely. If it does experience falling over during exploration, then each of its tree models may generalize what causes the robot to fall over differently. For example, one tree model may predict that the robot falls with a 2 cm shift, another with a 5 cm shift, etc. For a state with a 4 cm shift, perhaps three of the models predict the

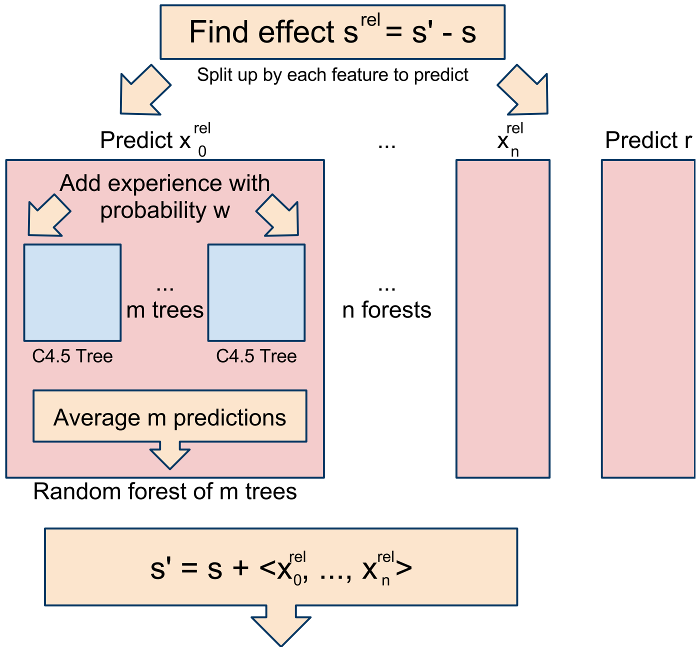


Fig. 5 *Model Learning.* This is how the algorithm learns a model of the domain. The agent calculates the difference between s' and s as the transition effect s^{rel}. Then it splits up the state vector and learns a random forest to predict each state feature. Each random forest is made up of stochastic decision trees, which get 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. Averaging the predictions makes the agent balance exploring the optimistic models with avoiding the pessimistic ones.

robot will fall over and receive -1000 reward, and two predict a successful kick with reward 20. Thus, the aggregate model predicts a reward of -592. This large negative reward will cause the agent to avoid exploring this and similar state-actions, and instead focus exploration on state-actions where some models predict successful kicks but none predict falling over. Avoiding these state-actions may lead the agent to learn a sub-optimal policy if the best kick requires the robot to shift its weight 4 cm, but it will also save the robot from many costly and possibly damaging exploration steps.

In contrast, BOSS would explore enough to guarantee optimality, which means it will explore many weight shifts that cause the robot to fall over. Since BOSS plans over the most optimistic model in each state (ignoring the others), at the 4 cm shift state, it will plan over the optimistic model that predicts a successful kick and reward 20, ignoring the fact that 3 of its 5 models predict the robot will fall over. As long as at least one model predicts high rewards, the agent will continue exploring these potentially damaging state-actions. In contrast, TEXPLORE performs limited exploration and thus would focus its exploration on other more promising state-actions while avoiding this one. MBBE would give a VPI bonus to state-actions which one of its models suggests has a higher value. These exploration bonuses are added to the expected value of the action, so the exploration should be less aggressive than BOSS's. Still, MBBE will explore many costly state-actions that may cause the robot to fall over.

It is important to note that the best exploration-exploitation trade off will depend highly on the domain. In the robotics domains we are focusing on, the agent has a limited number of time steps for learning, and thus must limit its exploration and start exploiting more quickly. In addition, exploring certain state-actions can be dangerous for the robot, providing another impetus to avoid exploring too much. However, in other domains such as simulated tasks where more time steps are available and actions are not damaging, it may be better to explore more (like BOSS and MBBE) to find a better final policy.

Similar to the prior that is created for Bayesian RL algorithms, TEXPLORE can be given some basic knowledge of the structure of the domain. TEXPLORE can be seeded with a few sample transitions from the domain, which it uses to initialize its models. For example, in an episodic task, a seed of the task's goal state can give the agent a general idea of the task at hand, instead of forcing it to search for an arbitrary goal state. The agent's performance is sensitive to these transition seeds since they bias the agent's expectations of the domain. TEXPLORE could be used as an apprenticeship learning algorithm if the seed experiences come from user-generated trajectories in the domain.

3.5 The Complete TEXPLORE Algorithm

After presenting each of the components of TEXPLORE, we now combine them together into one complete algorithm. TEXPLORE is constituted by the RTMBA architecture shown in Algorithms 3 and 5 combined with the random forest model learning approach shown in Algorithms 4 and 7 and the UCT(λ) planning method shown in Algorithms 2 and 6. Two separate versions of TEXPLORE can be run for discrete or continuous domains: *Discrete* TEXPLORE uses discrete decision trees in its random forest, while *Continuous* TEXPLORE uses linear regression trees to model continuous dynamics. For continuous domains, Discrete TEXPLORE requires the domain be discretized entirely, while Continuous TEXPLORE requires discrete states to maintain the value function, but learns models of the continuous dynamics. TEXPLORE also takes a parameter, k, that specifies the history length to handle delayed domains. When k is not defined, it is assumed to be 0 (the setting for non-delayed domains).

4 Related Work

Since TEXPLORE is addressing four different challenges, there is ample related work. However, to the best of our knowledge, none of the related work simultaneously addresses all four challenges or is easily adapted to do so. Section 4.1 examines the related work addressing Challenge 1 on sample efficiency and exploration. We look at work addressing Challenge 2 on continuous state spaces in Section 4.2, Challenge 3 on delayed actions and observations in Section 4.3, and Challenge 4 on real-time actions in Section 4.4. Finally, we summarize the related work and contrast it with TEXPLORE in Section 4.5.

4.1 Challenge 1: Sample Efficiency

For learning on robots or other real-world problems, sample efficiency is very important, because taking millions of samples to learn a task can also mean taking many real-world seconds to learn the task. For model-based methods, sample efficiency is mainly limited by how long it takes the agent to learn an accurate model of the domain. Exploration is very important for an agent to learn a model quickly. Therefore, we start by focusing on various exploration methods in Section 4.1.1, and then go into depth about Bayesian methods for exploration in Section 4.1.2.

4.1.1 Exploration

Many algorithms use ϵ-greedy exploration (Sutton and Barto 1998), which is one of the simplest approaches to exploration. Agents using it take what they think are the optimal actions most of the time, but take a random action ϵ of the time. Random exploration is guaranteed to explore the entire state space when given an infinite number of samples, but does not attempt to explore in any targeted way.

Boltzmann, or soft-max, exploration improves upon ϵ-greedy exploration, by taking better exploratory actions (Sutton and Barto 1998). Instead of taking a completely random action when exploring, the probability of selecting action a is weighted by its value relative to the other action-values using the following equation:

$$P(a) = \frac{e^{Q(a)/\tau}}{\sum_{b=1}^{n} e^{Q(b)/\tau}} \tag{10}$$

where τ is a temperature parameter determining the amount of exploration.

R-MAX (Brafman and Tennenholtz 2001) is a typical model-based approach that uses a tabular model and explores thoroughly by providing intrinsic rewards of R_{max} to all state-actions with fewer than m visits. These reward bonuses encourage the agent to visit all state-actions that are closer than states with maximal one-step reward. R-MAX is guaranteed to find the optimal policy in time polynomial in the number of states and actions, but exploring all the state-actions closer than the state with maximal one-step reward can be infeasible in larger domains.

With tabular models, the agent must explore each state-action in order to learn an accurate model for each one. In larger domains, however, it will not be feasible to visit every single state-action. In this case, it is better if the agent generalizes its model to unvisited state-actions. When using these models, the agent should efficiently explore where its model most needs improvement.

SLF-R-MAX (Strehl et al 2007), MET-R-MAX (Diuk et al 2009), and LSE-R-MAX (Chakraborty and Stone 2011) perform directed exploration on factored domains. They use a DBN to model the transition function where some features are only dependent on some subset of the features at the previous state. The methods 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 determine that some features are not relevant for the predictions of certain features. With fewer relevant features, the number of states with unique relevant features can be much less than the total number of states.

RAM-R-MAX is another approach that uses R-MAX-like exploration (Leffler et al 2007). In RAM-R-MAX, each state is mapped to a particular type, c. For a given type and action, the agent learns a model of the possible outcomes (for example, the relative change in state features). Using the state and the predicted outcome, the agent can predict the next state. Since the agent is given information about the types of all the states, it can easily generalize action effects across states with the same type. The authors demonstrate the RAM-R-MAX agent learning to navigate a robot across various terrains with different dynamics. While RAM-R-MAX's generalization gives it good sample efficiency, it requires the user to provide classifications for each state in the domain. In addition, it does not run in real-time.

Model Based Interval Estimation (MBIE) (Wiering and Schmidhuber 1998; Strehl and Littman 2005) is an approach that looks at the distribution over transition probabilities to drive exploration. 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.

Literature on active exploration provides more ideas on how RL agents could explore. Oudeyer et al (2007) present Intelligent Adaptive Curiosity (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. IAC splits the state space into regions and attempts to learn a model of the transition dynamics in each region. They maintain an error curve for each region and use the slope of this curve as the intrinsic reward for the agent, driving the agent to explore the areas where its model is improving the most. The resulting intrinsic motivation drive could provide efficient model learning, but their algorithm selects actions only to maximize the immediate reward, rather than the discounted sum of future rewards. In addition, their method has no way of incorporating external rewards or weighing their value in deciding what to explore.

Knows What It Knows (KWIK) (Li et al 2008) is a learning framework for efficient model learning. A learning algorithm that fits the KWIK framework 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 RL setting, as the agent can be driven to explore the states the model does not 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."

Fasel et al (2010) examine the InfoMax agent, which ignores external rewards and just tries to gain as much information as possible. The agent uses an intrinsic reward of the negative entropy of the agent's beliefs. They show that the agent can learn useful long-term policies, and learn to take multi-step trajectories to maximize information gain. While they want the agent to gain information to prepare it for future tasks, they do not use external rewards or have any way of trading off between exploration and exploitation.

4.1.2 Bayesian Methods

Model-based Bayesian RL methods 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. However, these methods have a few drawbacks. They must maintain a belief distribution over models, which can be computationally expensive. In order to generalize, the user must design a model parametrization that ties the dynamics of different states together in the correct way. In addition, the user must provide a well-defined prior for the model.

Duff (2003) presents an "optimal probe" that solves the exploration problem optimally, using an augmented state space 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 samples. 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.

Another approach to the exploration problem is Gaussian Process RL. Deisenroth and Rasmussen (2011) present one such approach called Probabilistic Inference for Learning Control (PILCO), where the agent maintains a model of the domain using Gaussian Process regression. This model generalizes experience to unknown situations and represents uncertainty explicitly. This approach has achieved great results on motor control problems such as the inverted pendulum and cart-pole problems. However, the algorithm requires ten minutes of computation time for every 2.5 seconds of experience when learning the cart-pole task. Also, rather than learning from an arbitrary reward function, the reward must encode a function of how far the agent is from the target state.

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, as previously described in Section 3.4, samples m models from the model posterior and merges them into a single model with the same state space, but an augmented action space of mA actions. Planning over this model allows the agent to select at each state an action from the most optimistic model. The agent will explore states where the model is uncertain because at least one of the sampled models is likely to be optimistic with respect to the true environment in these states. One drawback to this approach is that the agent ignores any possible costs to exploration, as the agent can always take the action from the most optimistic model, even if the other models all predict a negative outcome.

Model Based Bayesian Exploration (Dearden et al 1999) (MBBE) was also described in Section 3.4. It maintains a distribution over model parameters and samples and solves m models to get a distribution over action-values. This distribution is used to calculate the value of perfect information (VPI), which is added as a bonus value to actions to drive exploration.

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. In addition, the user must come up with a way for the model's predictions to be generalized across states or the agent will have to visit every state-action similar to the tabular approaches.

4.2 Challenge 2: Continuous Domains

Most of the model-based methods presented above are intended for discrete domains. This section looks at some of the related work on learning models for domains with continuous state spaces. The PILCO method presented earlier (Deisenroth and Rasmussen 2011) can handle continuous dynamics by using Gaussian Process regression for both learning a model and computing a policy.

Strehl and Littman (2007) introduce a linear regression model that provides its confidence in its predictions, which is useful for driving exploration. However, this model *only* works in domains that are linearly parametrized, whereas the linear regression tree model used by TEXPLORE works on those domains by learning a tree with a single leaf containing a linear function, *and* can also fit a piecewise linear function to any other domain that is not linear. In addition, the authors do not solve the problem of planning over a continuous state space, instead assuming they have a perfect planner. In later work (Walsh et al 2009b), they use the algorithm to predict a continuous reward function in a domain with discrete states, again avoiding the continuous state problem.

For planning over continuous domains, a common method is fitted value iteration (Gordon 1995), which adapts value iteration to continuous state spaces. It updates the values of a finite set of sampled states, and then fits a function approximator to their values. Like value iteration, it must iterate over the entire sampled state set which can be computationally expensive. In addition, this method only plans over the finite state set, while TEXPLORE, by using MCTS, can plan from the agent's real-valued state.

Jong and Stone (2007) present an extension of R-MAX to continuous domains called FITTED R-MAX. The authors use an instance based model and determine if a state is known based on the density of nearby visited states. The agent is driven to visit unknown states, like R-MAX. The policy is computed using fitted value iteration. While this method is a good extension of R-MAX to continuous domains, it suffers from the same over-exploration as R-MAX, while TEXPLORE focuses its exploration on parts of the state space that appear promising.

Finally, model-free methods can be extended to work in continuous domains by using function approximators to approximate the value function. For example, using Q-LEARNING or SARSA with neural networks or tile coding as a function approximator is a common approach for these problems. However, these model-free methods do not have the sample efficiency required to meet the first challenge of sample efficiency.

Munos and Moore (2002) use kd-trees to approximate the value function in continuous domains. In their approach, they incrementally refine the trees to improve their representation of the value function. They have specific value function based metrics to determine when is the best time to add new splits to the tree. While this method takes advantage of trees similar to TEXPLORE, it does it for value function approximation, instead of for approximating the transition and reward models.

4.3 Challenge 3: Observation and Action Delays

On real devices such as robots, there are frequently delays in both sensor readings and the execution of actions. This section presents some related work on handing delays in both actions and state observations, which are equivalent (Katsikopoulos and Engelbrecht 2003).

Walsh et al (2009a) develop a method called Model Based Simulation (MBS) for delayed domains. Given the domain's delay, k, as input, the algorithm can uncover the underlying MDP and learn a model of it. When the agent is selecting an action, MBS uses its model to simulate what state the selected action is likely to take effect in, and returns the action given by its policy for this state. The authors combine this approach with R-MAX learning the underlying model, creating an algorithm called MBS-R-MAX. The algorithm works well, but requires knowledge of the exact amount of delay, k, while TEXPLORE only requires an upper bound on the delay. Also, in stochastic domains, the agent may make poor predictions of the state where the action will take effect.

Methods with eligibility traces such as SARSA(λ) can be useful for delayed domains, because the eligibility traces spread credit for the current reward over the previous state-actions that may be have been responsible for it. Schuitema et al (2010) take this a step further, updating action-values for the *effective* action that was enacted at that state, rather than the action actually selected by the agent at the given state. However, the agent still selects actions based on its current state observation, so the values for which actions to select may not be correct.

The U-TREE (McCallum 1996) algorithm is the inspiration for TEXPLORE's approach of adding additional inputs to the decision trees used for learning the domain model. While TEXPLORE uses decision trees strictly for learning a model, U-TREE builds trees to represent a value function of the domain, with each leaf representing a set of states that have similar value. Value iteration is performed using each tree leaf as a state. TEXPLORE separates the policy representation from the model representation, as there are often cases where states have similar values but different transition dynamics (or vice versa).

The MC-AIXI algorithm (Veness et al 2011) takes a very similar approach to TEXPLORE, although theirs is intended for POMDPs rather than domains with delay. They use UCT to plan using a history of previous state-action-reward sequences, while TEXPLORE uses the current state augmented with the previous k actions. Both approaches take advantage of the ability of UCT to easily incorporate histories into its rollouts and focus planning on the relevant parts of the state space.

Outside of RL, there is some evidence that a mechanism similar to TEXPLORE's approach is used in the mammalian cerebellum. The cerebellum determines the proper control output on a delayed task by using different fibers which provide signals at various delays (Ohyama et al 2003).

4.4 Challenge 4: Real-Time Actions

Learning on a robot requires actions to be given at a specific control frequency, while maintaining sample efficiency so that learning does not take too long. Model-free methods typically return actions quickly enough, but are not very sample efficient, while model-based methods are more sample efficient, but typically take too much time for model updates and planning. This section describes related work that makes model-free methods more sample efficient as well as work making model-based methods run in less clock time.

Batch methods such as experience replay (Lin 1992), fitted Q-iteration (Ernst et al 2003), and LSPI (Lagoudakis and Parr 2003) improve the sample efficiency of model-free methods by saving experiences and re-using them in periodic batch updates. However, these methods typically run one policy for a number of episodes, stop to perform their batch update, and then repeat. While these methods take breaks to perform computation, RTMBA continues taking actions in real-time even while model and policy updates are occurring.

The DYNA framework (Sutton 1990) incorporates some of the benefits of model-based methods while still running in real-time. DYNA saves its experiences, and then performs l Bellman updates on randomly selected experiences between each action. Thus, instead of performing full value iteration each time, its planning is broken up into a few updates between each action. However, it uses a simplistic model (saved experiences) and thus does not have very good sample efficiency.

The DYNA-2 framework (Silver et al 2008) extends DYNA to use UCT as its planning algorithm. In addition, it maintains separate value function approximators for updates from real experience and sample-based updates, such that the sample-based planner can have a finer resolution in the region the agent is in. This improves the performance of the algorithm compared to DYNA. However, to be sample-efficient, DYNA-2 must have a good model learning method, which may require large amounts of computation time between action selections.

Real Time Dynamic Programming (RTDP) (Barto et al 1995) is a method for performing dynamic programming in real-time by performing rollouts, similar to

UCT. It simulates trajectories from the start of the task using Boltzmann exploration. For each state that it visits, it does a full backup on that's states values. It differs from TEXPLORE's version of UCT in that it is doing full one-step backups rather than λ-returns, and it is using Boltzmann exploration rather than upper confidence bounds. Still, it presents an intriguing alternative to UCT.

Walsh et al (2010) argue that with new compact representations for model-learning, many algorithms have PAC-MDP sample efficiency guarantees. The bottleneck is now that these methods require planning every step on a very large domain. Therefore, they want to replace traditional flat MDP planners with sample-based methods where computation time is invariant with the size of the state space. In order to maintain their PAC-MDP guarantees, they create a more conservative version of UCT that guarantees ϵ-accurate policies and is nearly as fast as the original UCT. They show that this new algorithm is still PAC-MDP efficient.

These methods all have drawbacks; they either have long pauses in learning to perform batch updates, or require complete model update or planning steps between actions. None of these methods accomplish both goals of being sample efficient and acting continually in real-time.

4.5 Summary

While there is a large body of work relating to each challenge that TEXPLORE addresses, none of these approaches address all four challenges together. A few methods come close. The PILCO algorithm (Deisenroth and Rasmussen 2011) is extremely sample efficient, targets exploration where the model needs improvement, and works on robots with continuous state spaces. However, it cannot take actions in real-time.

Policy search methods such as Policy Learning by Weighting Exploration with the Returns (PoWER) (Kober and Peters 2011) provide an alternative approach to applying RL to robots. In these approaches, the control policy is parametrized and the parameters for this policy are updated between each episode. With a good policy parametrization, a good policy can be learned in few samples. However, these methods require the user to create the policy parametrization and can take considerable time between each episode for computation.

The Horde architecture (Sutton et al 2011) takes a very different approach to learning on robots. In parallel, it learns to predict the values of many different sensors using general value functions. In addition, it learns policies to maximize those sensor values. Horde can learn these predictions while running in real-time on a robot that is following some other policy. While Horde adopts a parallel real-time architecture like TEXPLORE to learn predictions about the world, it cannot use these predictions as a model to plan more complicated policies. In addition, it is not particularly sample efficient, as it takes 8.5 hours of experience to learn a light-following policy. However, sample efficiency is less important in this scenario as Horde can learn while the robot is doing other things.

In contrast to these approaches, TEXPLORE addresses all of the desired criteria: it is sample-efficient, takes actions continually in real-time, works in domains with continuous state spaces, and can handle sensor and actuator delays. It also does not require much user input: a discretization size for continuous domains, an upper bound on the delay in the domain, and possibly seed experiences to bias initial

Fig. 6 The autonomous vehicle operated by Austin Robot Technology and The University of Texas at Austin.

learning. In the following section, the various aspects of the algorithm are examined empirically.

5 Empirical Results

This section presents experiments that examine TEXPLORE's solution to each challenge in isolation from the other parts. It examines a variety of options for each challenge while keeping the other components of the TEXPLORE algorithm fixed. Each component is demonstrated on a simulation of controlling an autonomous vehicle. First, Section 5.1 examines TEXPLORE's approach to Challenge 1: sample efficiency and exploration. Section 5.2 examines how TEXPLORE's models address Challenge 2 by modeling continuous domains. The use of k action histories to handle delays (Challenge 3) is explored in Section 5.3 and Section 5.4 examines the effects of using the real-time architecture, addressing Challenge 4. Finally, Section 5.5 shows the algorithm learning to control the physical autonomous vehicle, rather than the simulation.

Each component of the algorithm is examined on a simulation of a robot task: controlling the velocity of an autonomous vehicle (Beeson et al 2008). This task requires an algorithm to address all the challenges laid out in the introduction: it has a continuous state space and delayed action effects, and it requires learning that is both sample efficient (to learn quickly) and computationally efficient (to learn on-line while controlling the car).

The experimental vehicle is an Isuzu VehiCross (Figure 6) that has been upgraded to run autonomously by adding shift-by-wire, steering, and braking actuators to the vehicle. The brake is actuated with a motor physically moving the pedal, which has a significant delay. ROS (Quigley et al 2009) is used as the underlying middleware. Actions must be taken in real-time, as the car cannot wait for an action when a car stops in front of it or it approaches a turn in the road. To the best of our knowledge, no prior RL algorithm is able to learn in this domain *in real time*: with no prior data-gathering phase for training a model.

State	Desired Velocity, Current Velocity, Throttle Position, Brake Position
Actions	Do nothing, Increase Throttle position by 0.1, Decrease Throttle Position by 0.1 Increase Brake Position to 0.1, Decrease Brake Position to 0.1
Reward	-10.0 * |Desired Velocity - Current Velocity|

Table 2 Properties of the autonomous vehicle velocity control task.

Since the autonomous vehicle was already running ROS as its middleware, we created a ROS package for interfacing with RL algorithms similar to the message system used by RL-Glue (Tanner and White 2009). We created an RL Interface node that wraps sensor values into *states*, translates *actions* into actuator commands, and generates *reward*. This node uses a standard set of ROS messages to communicate with the learning algorithm. At each time step, the RL Interface node computes the current state and reward and publishes them as a ROS message to the RL agent. The RL agent can then process this information and publish an action message, which the interface will convert into actuator commands. Whereas RL agents using RTMBA respond with an action message immediately after receiving the state and reward message, sequential methods may have a long delay to complete model updates and planning before sending back an action message. In this case, the vehicle would continue with all the actuators in their current positions until it receives a new action message. The ROS messages we defined for communicating with an RL algorithm are available as a ROS package: `http://www.ros.org/wiki/rl_msgs`.

The task is to learn to drive the vehicle at a desired velocity by controlling the pedals. For learning this task, the RL agent's 4-dimensional state is the desired velocity of the vehicle, the current velocity, and the current position of the brake and accelerator pedals. For the discrete methods and the planner for the continuous methods, desired velocity is discretized into 0.5 m/s increments, current velocity into 0.25 m/s increments, and the pedal positions into tenths of maximum position. The agent's reward at each step is -10.0 times the error in velocity in m/s. Each episode is run at 10 Hz for 10 seconds. The agent has 5 actions: one does nothing (no-op), two increase or decrease the desired brake position by 0.1 while setting the desired accelerator position to 0, and two increase or decrease the desired accelerator position by 0.1 while setting the desired brake position to 0. While these actions change the desired positions of the pedals immediately, there is some delay before the brake and accelerator reach their target positions. The experiments are run with a discount factor of 0.95. None of the algorithms are given prior inputs or seed transitions before starting learning; the algorithms all start learning with no prior knowledge of this task. Table 2 formally defines the states, actions, and rewards for the domain.

5.1 Challenge 1: Sample Efficiency and Exploration

First, TEXPLORE's exploration and sample efficiency are compared against other possible approaches. We compare both with other exploration approaches utilized within TEXPLORE and with other existing algorithms such as BOSS and Gaussian Process RL. To fully examine the exploration of TEXPLORE, experiments are performed on both the simulated car control task and a gridworld domain designed to illustrate differences in exploration.

5.1.1 Simulated Vehicle Velocity Control

We examine TEXPLORE's exploration while keeping TEXPLORE's model learning, planning, and architecture constant. Its exploration is compared with a number of other approaches, including some that are inspired by Bayesian RL methods. By treating each of the regression tree models in the random forest as a sampled model from a distribution, we can examine the exploration approaches taken by some Bayesian RL methods, without requiring the computational overhead of maintaining a posterior distribution over models or the need to design a good model parametrization.

Bayesian DP (Strens 2000) was described in detail in Section 4.1.2. It samples a single model from the distribution, plans a policy on it, and uses it for a number of steps. We create a similar method for comparison by replacing the QUERY procedure in Algorithm 7 with the one shown in Algorithm 8. At the start of each episode, *curr* is set to a random number between 1 and m. The procedure returns the predictions of $tree_{curr}$ until a new model is chosen on the next episode.

Algorithm 8 Bayesian DP-like Approach

```
procedure QUERY(in)                              ▷ Get prediction for input in
    return tree_curr ⇒ QUERY(in)                 ▷ Prediction from model curr
end procedure
```

Best of Sampled Set (BOSS) (Asmuth et al 2009) was also described in detail in Section 4.1.2. It creates an augmented model with mA actions - a set of actions for each sampled model. By replacing QUERY in Algorithm 7 with Algorithm 9, we create a comparison method that takes a similar approach. The action that is passed in as part of *in* is used to determine which model to query.

Algorithm 9 BOSS-like Approach

```
procedure QUERY(in)                              ▷ Get prediction for input in
    ⟨s, a⟩ ← in
    model ← ROUND(a/m)                           ▷ Action a defines which model
    act ← a mod m                                ▷ And which action on that model
    input ← ⟨s, act⟩
    return tree_model ⇒ QUERY(input)             ▷ Prediction from tree model for action act
end procedure
```

In addition to the Bayesian-inspired approaches, we compare with the approach taken in the PILCO algorithm (Deisenroth and Rasmussen 2011) (described in Section 4.1.1), which adds a bonus reward into the model for state-actions where the predictions have the highest variance. This bonus reward encourages the agent to explore state-actions where its models disagree, and therefore where they need more experiences to learn a more accurate model. Each tree in the random forest model makes its own (possibly different) prediction of the next value of each feature and reward. The variances in the predictions made by the different trees are calculated, and the reward sample r returned by the QUERY-MODEL method for a given (s, a) of Algorithm 4 is modified by a value proportional to the average variance:

$$r = r + b\frac{1}{n+1}[\sigma^2 R(s,a) + \sum_{i=1}^{n} \sigma^2 P(s_i^{rel}|s,a)]. \tag{11}$$

Here, b is a coefficient which determines the bonus amount, $\sigma^2 R(s,a)$ is the variance in the reward predicted by each model, and $\sigma^2 P(s_i^{rel}|s,a)$ is the variance in the prediction of the change in each state feature. This VARIANCE-BONUS approach takes an exploration parameter, b, which adds or subtracts intrinsic rewards based on a measure of the variance in the model's predictions for each feature and reward. By setting $b < 0$, the agent will avoid states that the model is uncertain about; setting $b > 0$ will result in the agent being driven to explore these uncertain states. If $b = 0$, the agent will act greedily with respect to its model. Changing the parameter b affects how aggressive the agent is in trying to improve uncertainties in its model.

In total, we compare 7 different exploration approaches listed below:

1. Greedy w.r.t. aggregate model (TEXPLORE default)
2. ϵ-greedy exploration ($\epsilon = 0.1$)
3. Boltzmann exploration ($\tau = 0.2$)
4. VARIANCE-BONUS Approach $b = 1$ (Eq. 11)
5. VARIANCE-BONUS Approach $b = 10$ (Eq. 11)
6. Bayesian DP-like Approach (Alg. 8)
7. BOSS-like Approach (Alg. 9)

We do not run a version of MBBE because planning on m different models is too computationally inefficient to run at the frequency required by the car. Based on informal testing, all experiments with TEXPLORE are run with $\lambda = 0.05$, the probability that each experience is given to each model, w, set to 0.6, and the probability a feature is randomly removed from the set used for each split in the tree, f, set to 0.2. The values of ϵ and τ were also found through informal testing. All of these experiments are run with TEXPLORE's architecture and random forest model with the length of action histories, k, set to 2.

Figure 7 shows the average reward per episode for each of these exploration approaches. TEXPLORE's greedy approach, ϵ-greedy exploration, Boltzmann exploration, and the Bayesian DP-like approach are not significantly different. They all receive significantly more average rewards than the other three approaches after episode 24 ($p < 0.001$). Note that adding ϵ-greedy exploration, Boltzmann exploration, or Bayesian DP-like exploration on top of TEXPLORE's aggregate model does not significantly improve the rewards that it receives. Since the agent has a fairly limited number of steps in this task, the methods that explore more (the VARIANCE-BONUS approaches and the BOSS-LIKE approach) do not start exploiting in time to accrue much reward on this task. In contrast, TEXPLORE performs limited exploration using its aggregate random forest model and accrues equal or more reward than all the other methods.

In addition to comparing with methods using TEXPLORE's models, we compare with other methods that are state of the art for exploration, particularly Bayesian

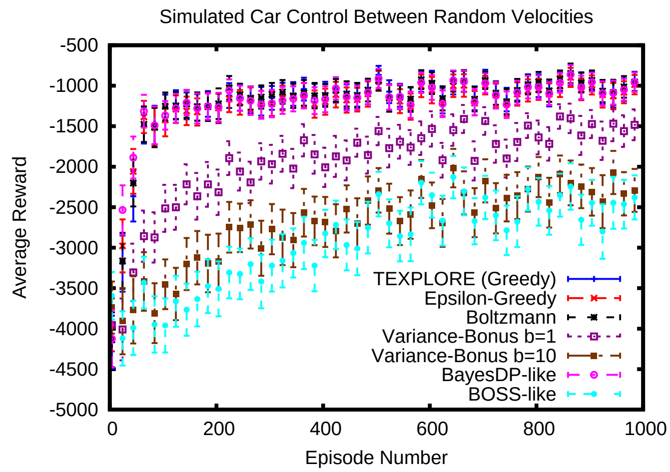


Fig. 7 Average reward over 1000 episodes on the simulated car control task. Results are averaged over 50 trials using a 5 episode sliding window and plotted with 95% confidence intervals. Note that TEXPLORE's exploration accrues the most reward.

methods. Here TEXPLORE is compared against the full versions of these methods, where sparse Dirichlet priors over models are maintained and sampled from. The parallel architecture is used to select actions in real-time. TEXPLORE is compared with the following 5 algorithms:

1. BOSS (Asmuth et al 2009)
2. Bayesian DP (Strens 2000))
3. PILCO (Deisenroth and Rasmussen 2011)
4. R-MAX (Brafman and Tennenholtz 2001)
5. Q-LEARNING using tile-coding (Watkins 1989; Albus 1975)

Both BOSS and Bayesian DP utilize a sparse Dirichlet prior over the discretized version of the domain as their model distribution (Strens 2000), while PILCO uses a Gaussian Process regression model and R-MAX uses a tabular model.

Results for these comparisons are shown in Figure 8. Here, TEXPLORE accrues significantly more rewards than all the other methods after episode 24 ($p < 0.01$). In fact, the Bayesian methods all fail to improve during this time scale (however, they would *eventually* learn an optimal policy). Thus, the combination of model learning and exploration approach used by TEXPLORE is the best for this particular domain.

5.1.2 Fuel World

Next, we created a novel domain called *Fuel World* to further examine exploration, shown in Figure 9. In it, the agent starts in the middle left of the domain and is trying to reach a terminal state in the middle right of the domain which has a reward of 0. The agent has a fuel level that ranges from 0 to 60. The agent's state vector, s, is made up of three features: its ROW, COL, and FUEL. Each step the agent takes reduces its fuel level by 1. If the fuel level reaches 0, the episode

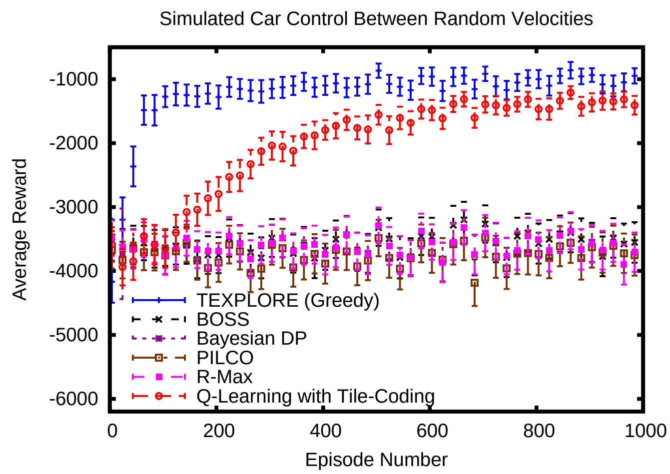


Fig. 8 Average reward over 1000 episodes on the simulated car control task. Results are averaged over 50 trials using a 5 episode sliding window and plotted with 95% confidence intervals. Note that TEXPLORE accrues the most reward.

State	Row, Column, Fuel Level
Actions	NORTH, EAST, SOUTH, WEST, NORTHEAST, SOUTHEAST, SOUTHWEST, NORTHWEST
Reward	Ranges from -400.0 to +20.0

Table 3 Properties of the Fuel World task.

terminates with reward -400. There are fuel stations along the top and bottom row of the domain which increase the agent's fuel level by 20. The agent can move in eight directions: NORTH, EAST, SOUTH, WEST, NORTHEAST, SOUTHEAST, SOUTHWEST, and NORTHWEST. The first four actions each move the agent one cell in that direction and have a reward of -1. The last four actions move the agent to the cell in that diagonal direction and have reward -1.4. An action moves the agent in the desired direction with probability 0.8 and in the two neighboring directions each with probability 0.1. For example, the NORTH action will move the agent north with probability 0.8, northeast with probability 0.1 and northwest with probability 0.1. The domain has 21×31 cells, each with 61 possible energy levels, and 8 possible actions, for a total of $317,688$ state-actions. The agent does not start with enough fuel to reach the goal, and must learn to go to one of the fuel stations on the top or bottom row before heading towards the goal state. The domain is formally defined in Table 3.

Actions from a fuel station have an additional cost, which is defined by:

$$R(x) = base - (x \bmod 5)a, \tag{12}$$

where $R(x)$ is the reward of a fuel station in column x, base is a baseline reward for that row, and a controls how much the costs vary across columns. There are two versions of the domain which differ in how much the costs of the fuel stations vary. The parameters for both the *Low variation* and *High variation Fuel World* are shown in Table 4.

The *Fuel World* domain was designed such that the center states have easily modeled dynamics and should be un-interesting to explore. The fuel stations all

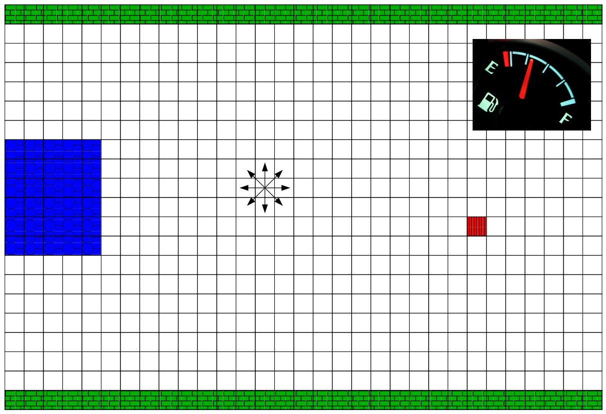


Fig. 9 The *Fuel World* domain. Starting states have blue hexagons, fuel stations have green brick patterns, and the goal state is shown in red with vertical lines. The possible actions the agent can take are shown in the middle. Here, the fuel stations are the most interesting states to explore, as they vary in cost, while the center white states are easily predictable.

Domain	Bottom Row		Top Row	
	base	*a*	*base*	*a*
Low variation Fuel World	-18	1	-21	1
High variation Fuel World	-10	5	-13	5

Table 4 Parameters for Equation 12 for the two versions of the *Fuel World* task.

have varying costs and are more interesting, but still only the fuel stations that may be useful in the final policy (i.e. the ones on a short path to the goal) should be explored. In addition, there is a clear cost to exploring, as some of the fuel stations are quite expensive.

The following 8 methods are compared:

1. Greedy w.r.t. aggregate model (TEXPLORE default)
2. ϵ-greedy exploration ($\epsilon = 0.1$)
3. Boltzmann exploration ($\tau = 0.2$)
4. VARIANCE-BONUS Approach $b = 10$ (Eq. 11)
5. Bayesian DP-like Approach (Alg. 8)
6. BOSS-like Approach (Alg. 9)
7. Bayesian DP with sparse Dirichlet prior (Strens 2000)
8. BOSS with sparse Dirichlet prior (Strens 2000)

The first six methods are the ones shown in the previous section that use the TEXPLORE model with various forms of exploration. The last two algorithms are Bayesian methods that are using models drawn from a sparse Dirichlet distribution. We did not run PILCO because this is a discrete domain (note that other Gaussian Process based methods can be run in discrete domains). We do not present results for Q-LEARNING and R-MAX because they performed so poorly on this task. All of these methods are run in real-time with actions taken at a rate of 10 Hz.

All of the algorithms are given seeding experiences in the domain. They are given two experiences from the goal state, two transitions from each row of fuel

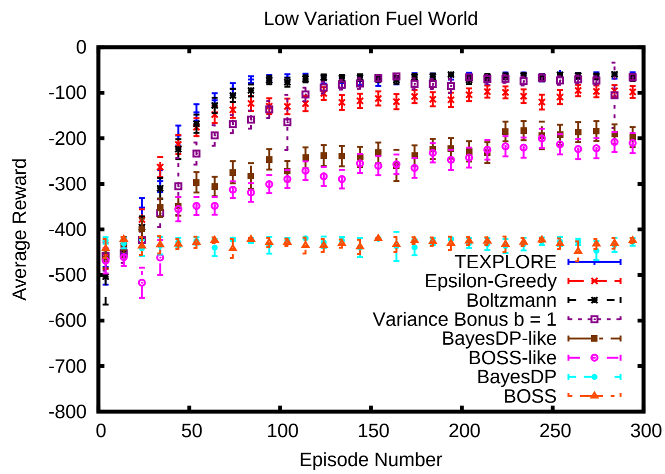


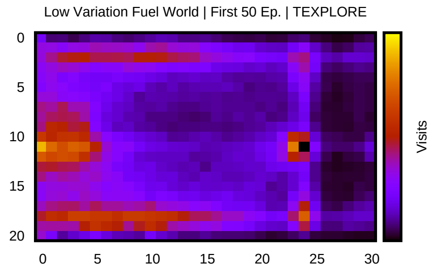
Fig. 10 Average reward over the first 300 episodes in *Low variation Fuel World*. Results are averaged over 50 trials using a 5 episode sliding window and plotted with 95% confidence intervals. TEXPLORE learns the policy faster than the other algorithms.

stations, and two experiences of running out of fuel for a total of eight seeding experiences. Since the sparse Dirichlet prior used by BOSS and BAYESIAN DP does not generalize, the sample experiences are only useful to them in the exact states they occurred in. In contrast, TEXPLORE's random forest models can generalize these experiences across state-actions.

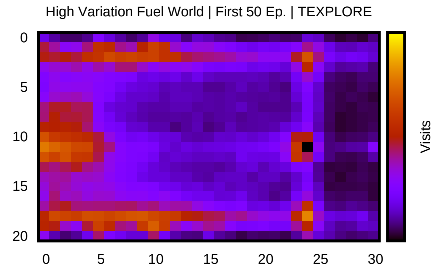
Figure 10 shows the average reward per episode over 50 trials for the methods in the *Low variation Fuel World* (Results are similar in the *High variation Fuel World*). TEXPLORE learns the fastest and accrues the most cumulative reward of any of the methods. TEXPLORE receives significantly more average rewards than all the other methods on episodes 20-32, 36-45, 68-91, and 96-110 ($p < 0.05$). TEXPLORE is not significantly worse than any other methods on any episode. All of the methods using TEXPLORE's model are able to learn the task to some degree, while the two Bayesian methods are unable to learn it within 300 episodes and their agents run out of fuel every episode.

To further examine how the agents are exploring, Figure 11 shows heat maps of which states the agents visited. The shading (color) represents the number of times the agent visited each cell in the domain (averaged over 50 trials and all fuel levels), with lighter shading (brighter color) meaning more visits.

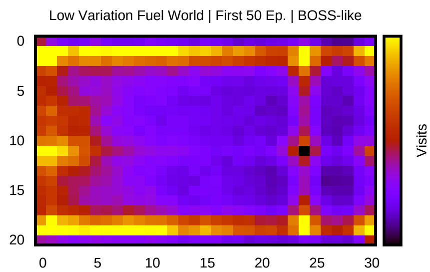
Figures 11(a) and 11(b) show the heat maps over the first 50 episodes for TEXPLORE in the *Low* and *High variation Fuel World* domains and Figures 11(e) and 11(f) show the heat maps over the final 50 episodes. First, the figures show that the algorithm is mainly exploring states near the fuel stations and the path to the goal, ignoring the space in the middle and right of the domain. Looking at the cells in the top and bottom rows between columns 5 and 10, Figure 11(a) shows that the agent in the *Low variation Fuel World* explores more of these fuel stations, while in the *High variation* world in Figure 11(b), the higher exploration costs cause it to quickly settle on the stations in column 5 or 10. The effects of the agent's different exploration in these two domains can be seen in its final policy



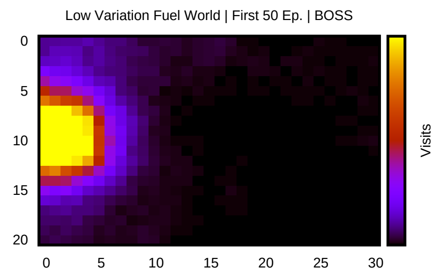
(a) TEXPLORE on *Low Variation Fuel World* over first 50 episodes.



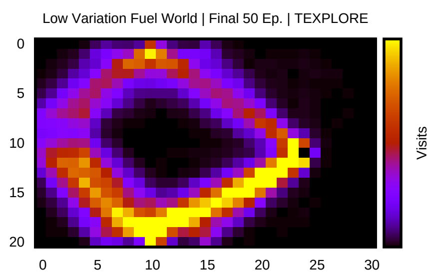
(b) TEXPLORE on *High Variation Fuel World* over first 50 episodes.



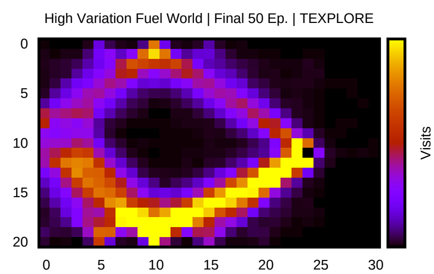
(c) BOSS-like on *Low Variation Fuel World* over first 50 episodes.



(d) BOSS on *Low Variation Fuel World* over first 50 episodes.



(e) TEXPLORE on *Low Variation Fuel World* over final 50 episodes.



(f) TEXPLORE on *High Variation Fuel World* over final 50 episodes.

Fig. 11 Heat maps displaying the average number of visits to each state over 50 episodes in the *Fuel World* domain, averaged over 50 trials and all fuel levels. With the higher fuel station costs in the *High Variation Fuel World*, TEXPLORE explores less there (Fig. 11(b)) than in the *Low Variation* domain (Fig. 11(a)). In either case, it explores less thoroughly than the BOSS-like algorithm (Fig. 11(c)) or the complete BOSS algorithm (Fig. 11(d)). The last two figures show the final policies for TEXPLORE in the two versions of the domain. In the *High Variation* domain (Fig. 11(f)), TEXPLORE explores less and converges to more final policies, while the the *Low Variation* version (Fig. 11(e)), it explores more and converges to fewer final policies across the 30 trials.

in each domain. Since the agents in the *Low variation Fuel World* explore more thoroughly than in the *High variation* world, they settle on better (and fewer) final policies. In the *High variation* task, the agent explores less after finding a cheap station and thus the various trials settle on a number of different policies. Since the reward within one fuel row can vary up to 20.0 in the *High variation* domain, it is not worthwhile for the agent to receive this additional cost while exploring, only to find a fuel station that is minimally better than one it already knows about.

The reason that TEXPLORE out-performs the other methods is that they explore too thoroughly and are unable to start exploiting a good policy within the given number of episodes. In contrast, TEXPLORE explores much less and starts exploiting earlier. Since TEXPLORE explores in a limited fashion, it uses these limited exploratory steps wisely, focusing its exploration on fuel stations rather than the other states. In contrast, the VARIANCE-BONUS, BAYESIAN DP-like, and BOSS-like approaches explore all of the state space. As an example, Figure 11(c) shows the exploration of the BOSS-like method on the *Low Variation Fuel World*. This approach is very optimistic and explores most of the cells near the start and near the fuel stations. The two complete Bayesian algorithms perform poorly because their sparse Dirichlet distribution over models does not generalize across states. Therefore, they explore each state-action separately and are only able to explore the starting states in the first 300 episodes, as shown in Figure 11(d). When acting in such a limited time frame, it is better to perform little exploration and target this exploration on useful state-actions. When given more time, it would be better to explore more thoroughly, as all of the other exploration methods will converge to the optimal policy if given enough time.

5.2 Challenge 2: Modeling Continuous Domains

Next, we examine the ability of TEXPLORE's model learning method to accurately predict state transitions and rewards on the continuous simulated vehicle velocity control task. In order to separate the issues of planning and exploration from the model learning, we train the model on a random sampling of experiences from the domain and then measure its accuracy on predicting the next state and reward for a randomly sampled 10,000 experiences in the domain.

Seven different model types are compared:

1. Regression Tree Forest (TEXPLORE Default)
2. Single Regression Tree
3. Decision Tree Forest
4. Single Decision Tree
5. Tabular Model
6. KWIK Linear Regression (Strehl and Littman 2007)
7. Gaussian Process Regression (PILCO model) (Deisenroth and Rasmussen 2011)

The first four are variants of TEXPLORE's regression tree forest model, while the last three are typical benchmark approaches.

To compare the accuracy of the models, the Euclidean distance between the next state the model predicted most likely and the true most likely next state is

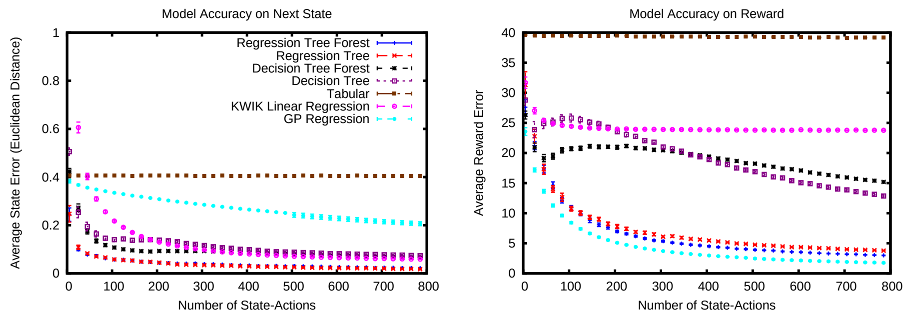


Fig. 12 Average error in the prediction of the next state and reward for each model, averaged over 50 trials and plotted with 95% confidence intervals. Each model is trained on random experiences from the domain and tested on its ability to predict 10,000 random experiences from the domain. The state error is the average Euclidean distance between the most likely predicted state and the true most likely next state and the reward error is the error in expected reward. Note that TEXPLORE's model, a random forest of regression trees, is the most accurate for next state predictions and second best for reward prediction.

used. For reward, the average error between the expected reward predicted by the model and the true expected reward in the simulation is calculated.

Figure 12 shows the average next state and reward prediction error for each model. For prediction of the next state, the regression tree forest and single regression tree have significantly less error than all the other models ($p < 0.001$). The single regression tree and the forest are not significantly different. For reward prediction, Gaussian process regression is significantly better than the other models ($p <= 0.001$). The regression tree forest has the next lowest error and is significantly better than all other models (including the single regression tree) after training on 205 state-actions ($p < 0.001$). While Gaussian process regression has the lowest error on reward prediction, its prediction of the next state is very poor, likely due to discontinuities in the function mapping the current state to the next state. These results demonstrate that TEXPLORE's model is well-suited to the robot learning domain: it makes accurate predictions, generalizes well, and has significantly less error in predicting states than the other models.

5.3 Challenge 3: Delayed Actions

Next, we examine the effects of TEXPLORE's approach for dealing with delays on the simulated car velocity control task. As described in Section 3.3, TEXPLORE takes a k-Markov approach, adding the last k actions as extra inputs to its models and planning over states augmented with k-action histories. The other components of TEXPLORE are particularly suited to this approach, as UCT(λ)'s rollouts can easily incorporate histories and the tree models can correctly identify which delayed inputs to use.

We evaluate TEXPLORE's approach using values of k ranging from 0 to 3. In addition, we compare with Model Based Simulation (MBS) (Walsh et al 2009a), which represents the main alternative to handling delays with a model-based method. MBS requires knowledge of the exact value of k to uncover the true MDP for model learning. MBS then uses its model to simulate forward to the state where the action will take effect and uses the policy at that state to select the action. MBS is combined with TEXPLORE's parallel architecture and models. In addition, to show the unique advantages of using regression trees for modeling, we compare with an approach using tabular models. Since the tabular models do not generalize, the agent must learn a correct model for every history-state-action. The following variations are compared:

1. TEXPLORE $k = 0$
2. TEXPLORE $k = 1$
3. TEXPLORE $k = 2$
4. TEXPLORE $k = 3$
5. MBS $k = 1$
6. MBS $k = 2$
7. MBS $k = 3$
8. Tabular model $k = 2$

The delay in the velocity control task comes from the delay in physically actuating the brake pedal (which is modeled in the simulation). The brake does not have a constant delay; it is slow to start moving, then starts moving quickly before slowing as it reaches the target position. MBS is not well suited to handle this type of delay, as it expects a constant delay of exactly k. In contrast, TEXPLORE's model can potentially use the previous k actions to model the changes in the brake's position.

The average reward for each method on the simulated car control task is shown in Figure 13. The TEXPLORE methods using $k = 1, 2$, and 3 receive significantly more average rewards than the other methods after episode 45 ($p < 0.005$). These three delay levels are not significantly different, however, TEXPLORE with $k = 1$ learns faster, receiving more average rewards through episode 80, but TEXPLORE with $k = 2$ learns a better policy and has the best average rewards after that. TEXPLORE with $k = 0$ learns a poor policy, while the methods using MBS and the TABULAR model do not learn at all.

5.4 Challenge 4: Real-Time Action

In this section, we demonstrate the effectiveness of the RTMBA architecture to enable the agent to act in real-time. The goal is for the agent to learn effectively on-line while running continuously on the robot in real-time, without requiring any pauses or breaks for learning. This scenario conforms to the eventual goal of performing lifelong learning on a robot without pauses or breaks. TEXPLORE's RTMBA architecture enables this by employing a multi-threaded approach along with UCT(λ) planning.

Various approaches for real-time action selection are evaluated on the simulated vehicle velocity control task. We compare with three other approaches: one that also does approximate planning in real-time, one that does exact planning in real-time, and one that does not select actions in real-time at all. All four approaches use TEXPLORE's model and exploration:

1. RTMBA (TEXPLORE)
2. Real Time Dynamic Programming (RTDP) (Barto et al 1995)

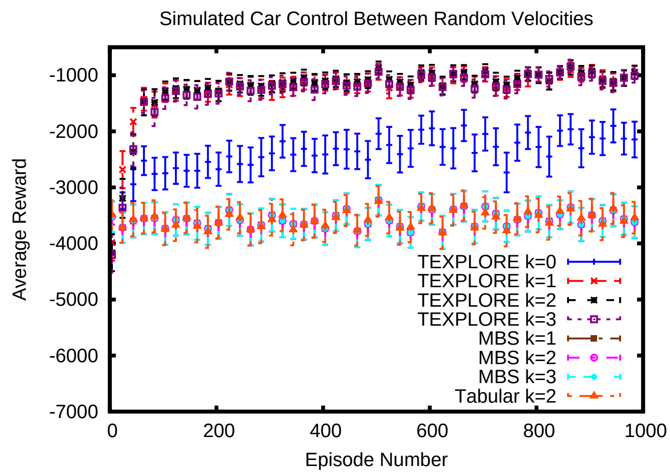


Fig. 13 Average reward over 1000 episodes for each method on the simulated vehicle control task. Results are averaged over 50 trials using a 5 episode sliding window and plotted with 95% confidence intervals. TEXPLORE with $k = 2$ performs the best, but not significantly better than TEXPLORE with $k = 1$ or $k = 3$. These three approaches all perform significantly better than than using no delay ($k = 0$) or using another approach to handling delay ($p < 0.005$). Note that the curves for all three MBS methods and the Tabular method are on top of each other.

3. Parallel Value Iteration
4. Value Iteration

RTDP is an alternative way to do approximate planning instead of using UCT. In contrast to UCT, RTDP does full backups on each state of its rollout and performs action selection differently. The implementation of RTDP still uses TEXPLORE's multi-threaded architecture to enable parallel model learning and planning, but uses RTDP for planning instead of UCT.

For a comparison with a method doing exact planning and still acting in real-time, we implemented a multi-threaded version of value iteration (Parallel Value Iteration) that runs model updates and value iteration in a parallel thread while continuing to act using the most recently calculated policy.

Finally, we compare with value iteration run sequentially, to show what happens when actions are not taken in real-time. Since this architecture is sequential, there could be long delays between action selections while the model is updated and value iteration is performed. If the vehicle does not receive a new action, its throttle and brake pedals remain in their current positions.

In addition to these four different architectures, we also compare with DYNA (Sutton 1990) and Q-LEARNING with tile-coding (Watkins 1989; Albus 1975). DYNA saves experiences and updates its value function by performing Bellman updates on randomly sampled experiences. The implementation of DYNA performs as many Bellman updates as it can between actions while running at 10 Hz. Q-LEARNING with tile-coding for function approximation could select actions faster than 10 Hz, but the environment only requests a new action from it at 10 Hz. Both DYNA and Q-LEARNING perform Boltzmann exploration with $\tau = 0.2$, which performed the best based on informal tests.

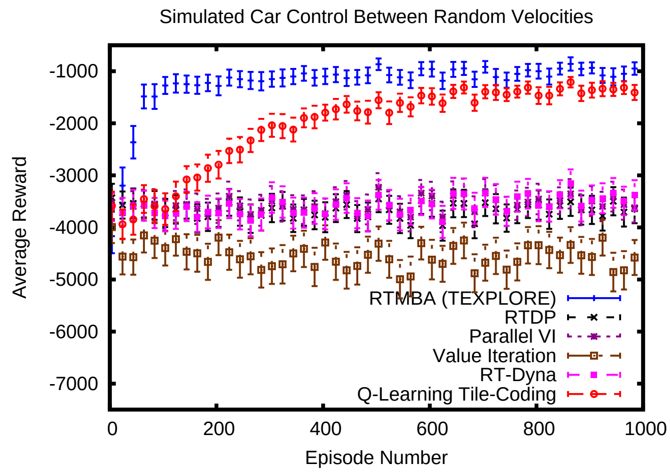


Fig. 14 Average reward over 1000 episodes for each method on the simulated vehicle control task. Results are averaged over 50 trials using a 5 episode sliding window and plotted with 95% confidence intervals. Note that TEXPLORE performs the best.

Figure 14 shows the average rewards for each of these approaches over 1000 episodes and averaged over 50 trials while controlling the simulated vehicle. TEXPLORE's architecture receives significantly more average rewards per episode than the other methods after episode 29 ($p < 0.01$). While RTDP is out-performed by TEXPLORE's architecture here, recent papers have shown modified versions of RTDP to be competitive with UCT (Kolobov et al 2012). Both TEXPLORE and RTDP are run with $k = 2$. Since running value iteration on this augmented state space would result in 25 times more state-actions to plan on, the value iteration approaches are run with $k = 0$. Still, they perform significantly worse than TEXPLORE with $k = 0$ (not shown) after episode 41 ($p < 0.001$). This issue provides another demonstration that k-Markov histories work well with UCT(λ) planning but make methods such as value iteration impractical.

In addition to these experiments, in previous work (Hester et al 2012), we further analyzed the trade-offs of using the multi-threaded architecture. We showed that at slow enough action rates (10 - 25 Hz), RTMBA does not perform significantly worse than using sequential architectures which have unlimited computation time between actions.

5.5 On the Autonomous Vehicle

After demonstrating each aspect of TEXPLORE on the simulated vehicle control task, this section demonstrates the complete algorithm learning on the physical autonomous vehicle. Due to the time, costs, and dangers involved, only TEXPLORE is tested on the physical vehicle. Five trials of TEXPLORE with $k = 2$ are run on the physical vehicle learning to drive at 5 m/s from a start of 2 m/s. Figure 15 shows the average rewards over 20 episodes. In all five trials, the agent learns the

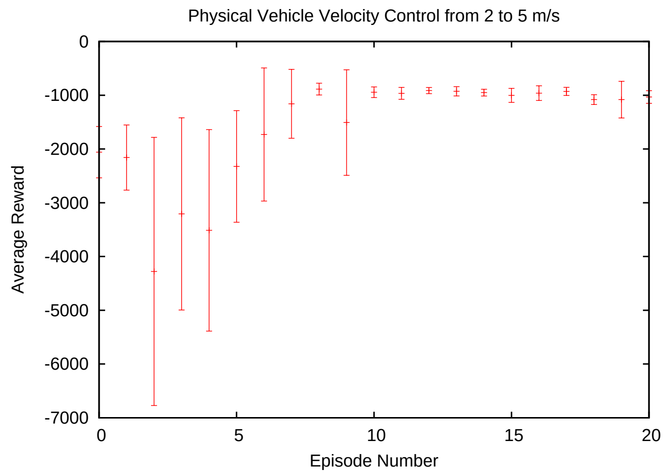


Fig. 15 Average rewards of TEXPLORE learning to control the physical vehicle from 2 to 5 m/s. Results are averaged over 5 trials and plotted with 95% confidence intervals. In every trial, the agent successfully learns the task by episode 10.

task within 11 episodes, which is less than 2 minutes of driving time. In 4 of the trials, the agent learns the task in only 7 episodes.

As the first author was physically present in the vehicle for the learning experiments, we can report on the typical behavior of the agent while learning to drive the car. Typically, on the first episode or two, the agent takes actions mostly randomly, and the car's velocity simply drifts from its starting velocity. Then on the next few trials, the learning algorithm explores what happens when it pushes the throttle or brake all the way down (by alternatively pushing the throttle or brake to the floor for a few seconds). Next, the agent starts trying to accelerate to the target velocity of 5 m/s. For the remaining episodes, the agent learns how to track the target velocity once it is reached and makes improvements in the smoothness of its acceleration and tracking. This experiment shows that TEXPLORE can learn on a task requiring all the challenges presented in the introduction.

6 Discussion and Conclusion

We identify four properties required for RL to be practical for continual, on-line learning on a broad range of robotic tasks: it must (1) be sample-efficient, (2) work in continuous state spaces, (3) handle sensor and actuator delays, and (4) learn while taking actions continually in real-time. This article presents TEXPLORE, the first algorithm to address all of these challenges.

TEXPLORE addresses challenge 1 by learning random forest models that generalize transition and reward dynamics to unseen states. Unlike methods that guarantee optimality by exploring more exhaustively, TEXPLORE learns faster by limiting its exploration to states that are promising for the final policy. Instead of exploring more broadly, it quickly moves to exploiting what it has learned to accrue good rewards in a limited time frame.

TEXPLORE works in continuous domains (addressing challenge 2) by learning regression tree models of the domain. For the 3rd challenge: learning in domains with delayed sensors and actuators, TEXPLORE provides its models with histories of actions to learn the delay of the domain. This approach requires the user to provide an upper bound on the delay in the domain. TEXPLORE is uniquely suited for this approach because its models are capable of determining which delayed inputs to use for predictions, and UCT(λ) can plan over histories instead of the full state space.

Challenge 4 is for the agent to learn while taking actions continually in real-time. TEXPLORE addresses this challenge by using sample-based planning and a multi-threaded architecture (RTMBA). In addition, RTMBA enables the algorithm to take advantage of the multi-core processors available on many robotic platforms.

In addition to addressing these challenges, TEXPLORE requires minimal user input. Unlike many methods that require users to define model priors or model parametrization, TEXPLORE only requires a discretization (for continuous domains), an upper bound on the sensor/actuator delay (typically 0), and possibly some seed experiences to bias learning (optional).

For each of these challenges, TEXPLORE's solution is compared with other approaches on the task of controlling the velocity of a simulated vehicle. In each case, its approach is shown to be the best one and leads to the most rewards. Finally, the algorithm is shown to successfully learn to control the actual vehicle, a task which requires an algorithm that addresses all four challenges.

The empirical results show that TEXPLORE addresses each challenge and, in fact, outperforms many other methods designed to address just one of the challenges. Not only does TEXPLORE address each challenge, but it addresses all of them *together* in one algorithm. The approach taken for each challenge meshes well with the other components of the algorithm, enabling the entire algorithm to work well as a whole.

While TEXPLORE out-performs other methods on the comparison tasks, it is important to note that TEXPLORE cannot outperform all of these methods all of the time. In domains where the effects of actions do not generalize across states, TEXPLORE's random forest model will not learn as quickly and other methods are likely to outperform TEXPLORE. In addition, TEXPLORE is not guaranteed to converge to an optimal policy, and may not explore fully enough to find arbitrarily located high-rewarding state-actions that cannot be predicted from neighboring states. Such states are guaranteed to be found by methods with convergence guarantees that explore every state-action, however, in many real-world domains, this exploration is not feasible. In these cases, the assumptions that TEXPLORE makes (i.e. that similar states have similar dynamics) are more practical, and enable it to learn a good policy very quickly. The key trade-off is that in domains with a limited number of time-steps, it is better to perform more targeted and limited exploration like TEXPLORE does, while with more steps, it is better to explore more thoroughly to learn a better final policy.

There are a few other issues with learning on robots that we have not addressed. For example, many real world tasks are partially observable. With a sufficiently high value of k, the k action histories TEXPLORE uses for delay could also handle partially observable domains. In addition, one of the advantages of the model learning and UCT planning approaches that TEXPLORE uses is that they can utilize a rich feature space without it greatly affecting the computation time needed for model learning or planning. The state could be made up of a rich set of features from the robot, including both sensor values and internally calculated features such as estimated poses. Thus, most of the state would be observable, although unknown aspects of the environment would still be unobservable. These unknown aspects could be treated as stochastic transitions, enabling the robot to plan for and react to a range of possible environments that it may encounter. Incorporating a better solution for POMDPs remains an area for future work.

There are a number of other possible directions for future work. For now TEXPLORE uses discrete actions as an approximation to the continuous actuators that most robots have, but addressing the issue of continuous actuators remains an area for future work. Another direction we are interested in pursuing is using the algorithm for developmental and lifelong learning. In a domain with limited or no external rewards, exploration rewards from the model could be used to provide intrinsic motivation for a developing, curious agent. The resetting of the visit counts for UCT(λ) when the model changes could be improved by setting the values based on how much the model has changed. In addition, we intend to perform more empirical testing on larger, more complex robot learning tasks. Finally, there are many opportunities for us to further parallelize the architecture to take advantage of robots with multiple cores. Each tree of the random forest could be learned on a separate core, and many UCT(λ) rollouts can be performed at the same time in parallel as well.

In summary, this article presents four main contributions:

1. The use of regression trees to model continuous domains.
2. The use of random forests to provide targeted, limited exploration for an agent to quickly learn good policies.
3. A novel multi-threaded architecture that is the first to parallelize model learning in addition to planning and acting.
4. The complete implemented TEXPLORE algorithm, which is the first to address all of the previously listed challenges together in a single algorithm, and is publicly available online.

By addressing all the challenges laid out in the introduction, TEXPLORE represents an important step towards the applicability of RL to larger and more real-world tasks such as robotics problems. The algorithm can work on a large variety of problems, and act continually in real-time while maintaining high sample efficiency. Because of its sample efficiency, TEXPLORE is particularly useful on problems where the agent has a very limited number of samples in which to learn.

References

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

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)

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

Barto AG, Bradtke SJ, Singh SP (1995) Learning to act using real-time dynamic programming. Artificial Intelligence 72(12):81 – 138

Beeson P, O'Quin J, Gillan B, Nimmagadda T, Ristroph M, Li D, Stone P (2008) Multiagent interactions in urban driving. Journal of Physical Agents 2(1):15–30

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)

Chaslot G, Winands MHM, van den Herik HJ (2008) Parallel Monte-Carlo tree search. In: The 6th International Conference on Computers and Games (CG 2008), pp 60–71

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)

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

Ernst D, Geurts P, Wehenkel L (2003) Iteratively extending time horizon reinforcement learning. In: Proceedings of the Fourteenth European Conference on Machine Learning (ECML), pp 96–107

Ernst D, Geurts P, Wehenkel L (2005) Tree-based batch mode reinforcement learning. Journal of Machine Learning Research 6:503–556

Fasel I, Wilt A, Mafi N, Morris C (2010) Intrinsically motivated information foraging. In: Proceedings of the Ninth International Conference on Development and Learning (ICDL)

Gelly S, Silver D (2007) Combining online and offline knowledge in UCT. In: Proceedings of the Twenty-Fourth International Conference on Machine Learning (ICML), pp 273–280

Gelly S, Hoock JB, Rimmel A, Teytaud O, Kalemkarian Y (2008) The parallelization of Monte-Carlo planning. In: ICINCO 2008, Proceedings of the Fifth International Conference on Informatics in Control, Automation and Robotics, Intelligent Control Systems and Optimization, pp 244–249

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

Hester T, Stone P (2009) An empirical comparison of abstraction in models of Markov Decision Processes. In: Proceedings of the ICML/UAI/COLT Workshop on Abstraction in Reinforcement Learning

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 (2012) RTMBA: A real-time model-based reinforcement learning architecture for robot control. In: Proceedings of the 2012 IEEE International Conference on Robotics and Automation (ICRA)

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)

Katsikopoulos K, Engelbrecht S (2003) Markov Decision Processes with delays and asynchronous cost collection. IEEE Transactions on Automatic Control 48(4):568 – 574

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

Kober J, Peters J (2011) Policy search for motor primitives in robotics. Machine Learning 84(1-2):171–203

Kocsis L, Szepesvári C (2006) Bandit based Monte-Carlo planning. In: Proceedings of the Seventeenth European Conference on Machine Learning (ECML)

Kohl N, Stone P (2004) Machine learning for fast quadrupedal locomotion. In: Proceedings of the Nineteenth AAAI Conference on Artificial Intelligence

Kolobov A, Mausam, Weld D (2012) Lrtdp versus uct for online probabilistic planning. URL `https://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/4961/5334`

Lagoudakis M, Parr R (2003) Least-squares policy iteration. Journal of Machine Learning Research 4:1107–1149

Leffler B, Littman M, Edmunds T (2007) Efficient reinforcement learning with relocatable action models. In: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, pp 572–577

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

Lin LJ (1992) Reinforcement learning for robots using neural networks. PhD thesis, Pittsburgh, PA, USA

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

Méhat J, Cazenave T (2011) A parallel general game player. Knstliche Intelligenz 25(1):43–47

Munos R, Moore A (2002) Variable resolution discretization in optimal control. Machine Learning 49:291–323

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

Ohyama T, Nores WL, Murphy M, Mauk MD (2003) What the cerebellum computes. Trends in Neurosciences 26(4):222–7

Oudeyer PY, Kaplan F, Hafner VV (2007) Intrinsic motivation systems for autonomous mental development. IEEE Trans Evolutionary Computation 11(2):265–286

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

Quigley M, Conley K, Gerkey B, Faust J, Foote T, Leibs J, Wheeler R, Ng A (2009) ROS: an open-source robot operating system. In: ICRA Workshop on Open Source Software

Quinlan R (1986) Induction of decision trees. Machine Learning 1:81–106

Quinlan R (1992) Learning with continuous classes. In: 5th Australian Joint Conference on Artificial Intelligence, World Scientific, Singapore, pp 343–348

Schuitema E, Busoniu L, Babuska R, Jonker P (2010) Control delay in reinforcement learning for real-time dynamic systems: A memoryless approach. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp 3226 –3231

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

Silver D, Sutton R, Muller M (2012) Temporal difference search in computer go. Machine Learning 87

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, Littman M (2007) Online linear regression and its application to model-based reinforcement learning. In: Advances in Neural Information Processing Systems (NIPS) 20

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, Barto A (1998) Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA

Sutton R, Modayil J, Delp M, Degris T, Pilarski P, White A, Precup D (2011) Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction.

In: Proceedings of the Tenth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)
Tanner B, White A (2009) RL-Glue : Language-independent software for reinforcement-learning experiments. Journal of Machine Learning Research 10:2133–2136
Veness J, Ng KS, Hutter M, Uther WTB, Silver D (2011) A Monte-Carlo AIXI approximation. Journal of Artificial Intelligence Research 40:95–142
Walsh T, Nouri A, Li L, Littman M (2009a) Learning and planning in environments with delayed feedback. Autonomous Agents and Multi-Agent Systems 18:83–105
Walsh T, Szita I, Diuk C, Littman M (2009b) Exploring compact reinforcement-learning representations with linear regression. In: Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence (UAI)
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
Watkins C (1989) Learning from delayed rewards. PhD thesis, University of Cambridge
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
Willems FMJ, Shtarkov YM, Tjalkens TJ (1995) The context tree weighting method: Basic properties. IEEE Transactions on Information Theory 41:653–664