Studies in Computational Intelligence 503

Todd Hester

TEXPLORE: Temporal Difference Reinforcement Learning for Robots and Time-Constrained Domains

Studies in Computational Intelligence

Volume 503

Series Editor

J. Kacprzyk, Warsaw, Poland

For further volumes:

http://www.springer.com/series/7092

Todd Hester

TEXPLORE: Temporal Difference Reinforcement Learning for Robots and Time-Constrained Domains

Springer

Todd Hester
Department of Computer Science
University of Texas at Austin
Austin, Texas
USA

ISSN 1860-949X ISSN 1860-9503 (electronic)
ISBN 978-3-319-01167-7 ISBN 978-3-319-01168-4 (eBook)
DOI 10.1007/978-3-319-01168-4
Springer Cham Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013942265

© Springer International Publishing Switzerland 2013
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To my family, especially my lovely wife Beth

Foreword

It is a great pleasure and honor to be able to write the foreward for this book, representing the culmination of Todd Hester's Ph.D. thesis research at The University of Texas at Austin. It was my good fortune to be Todd's advisor during his years as a Computer Science graduate student. I therefore was able to participate in and enjoy the adventure of starting with the kernel of an idea and fully developing it into a full-blown dissertation.

Todd's research is in the area of Reinforcement learning (RL), a machine learning paradigm that focuses on enabling computers and robots to learn to perform sequential tasks. Though grounded in some very theoretically elegant results, and showing great promise for enabling learning-based robots that could be deployed in the real world, most RL algorithms to date require either too much data (training experience) or too much computation to be practical on real-world problems.

This book introduces TEXPLORE, one of the first RL algorithms to be both data-efficient and computation-efficient enough to work on real robots in the real world. The core idea of TEXPLORE is the realization that scaling up to large domains in real time requires actively reasoning about which states *not* to explore. Most RL algorithms to date still insist on exhaustive exploration: visiting every state, or in continuous settings, every region of the state space. Doing so is necessary if the goal is finding the *optimal* policy because in principle, any unvisited state could be a "gold mine." However when accepting that perfection can sometimes be the enemy of the good, it becomes clear that exploration must be more focused.

The distinguishing characteristic of TEXPLORE is that it learns *in real-time, while continuing to act.* As this book documents fully, it has been demonstrated to learn a speed-control task on a real autonomous vehicle during the course of two minutes of continual driving. In addition to this driving task, the algorithms introduced in this book have been validated on real humanoid robots, and in carefully controlled, simulation environments.

This book also includes an investigation into the application of TEXPLORE to the idea of intrinsically motivated RL. Analogous to curiosity on the part of human learners, intrinsically motivated RL requires guiding exploration by properties of the environment, rather than based on external reward.

This book is important for the field for these novel algorithms themselves, and also for the fact that it opens up several exciting directions for future research. By releasing the associated source code as an RL package within the Robot

Operating System (ROS) development environment, Todd has made it easy for future researchers to build upon his contributions.

Overall, for both newcomers to the field, and for practitioners looking for nuanced detail, this book has plenty to offer. Whichever your perspective, I trust you will enjoy reading it!

Austin, Texas
June, 2013

Peter Stone

Preface

This book presents the main results of the research I conducted for my Ph.D. thesis at the University of Texas at Austin. The main focus of the research is on developing new reinforcement learning methods that enable fast and robust learning on robots in real-time.

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. 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) is a paradigm for learning sequential decision making processes and could solve the problems of learning and adaptation on robots. This book identifies four key challenges that must be addressed for an RL algorithm to be practical for robotic control tasks. These *RL for Robotics Challenges* are: 1) it must learn in very few samples; 2) it must learn in domains with continuous state features; 3) it must handle sensor and/or actuator delays; and 4) it should continually select actions in real time. This book focuses on addressing all four of these challenges. In particular, this book is focused on *time-constrained domains* where the first challenge is critically important. In these domains, the agent's lifetime is not long enough for it to explore the domain thoroughly, and it must learn in very few samples.

Although existing RL algorithms successfully address one or more of the *RL for Robotics Challenges*, no prior algorithm addresses all four of them. To fill this gap, this book introduces TEXPLORE, the first algorithm to address all four challenges. TEXPLORE is a model-based RL method that learns a random forest model of the domain which generalizes dynamics to unseen states. Each tree in the random forest model represents a hypothesis of the domain's true dynamics, and the agent uses these hypotheses to 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 each component of TEXPLORE in comparison with other state-of-the-art approaches. In addition, we present modifications of TEXPLORE's exploration mechanism for different types of domains. The key result of this book is a demonstration of TEXPLORE learning to control the velocity of an autonomous vehicle on-line, in real time, while running on-board the robot. After

controlling the vehicle for only two minutes, TEXPLORE is able to learn to move the pedals of the vehicle to drive at the desired velocities. The work presented in this book represents an important step towards applying RL to robotics and enabling robots to perform more tasks in society. By enabling robots to learn in few actions while acting on-line in real time on robots with continuous state and actuator delays, TEXPLORE significantly broadens the applicability of RL to robots.

This book would not have been possible without help from a great number of people. First and foremost, I want to thank my advisor Peter Stone, whose guidance, advice, and support has been invaluable. There are also many other graduate students who helped and collaborated with me on this research. In particular, Nick Jong let me assist on an AAMAS paper on reinforcement learning in my first year as a graduate student and gave me a great start on RL.

Austin, Texas
April, 2013

Todd Hester

Table of Contents

1 Introduction **1**
1.1 Time-Constrained Domains 3
1.2 Algorithm Overview 5
1.3 Contributions 7
1.4 Overview 7

2 Background and Problem Specification **11**
2.1 Markov Decision Problems 11
2.2 Value Function Reinforcement Learning 13
2.2.1 Model-Free Methods 13
2.2.2 Model-Based Methods 14
2.2.3 Factored Models 17
2.2.4 Planning 18
2.3 Time-Constrained Domains 19
2.4 A Specific Problem 21
2.5 Chapter Summary 23

3 Real Time Architecture **25**
3.1 Monte Carlo Tree Search (MCTS) Planning 26
3.1.1 Domains with Delay 29
3.2 Parallel Architecture 31
3.3 Chapter Summary 34

4 The TEXPLORE Algorithm **35**
4.1 Model Learning 35
4.1.1 Models of Continuous Domains 38
4.1.2 Domains with Delays 38
4.1.3 Dependent Feature Transitions 41
4.2 Exploration 43
4.3 The Complete TEXPLORE Algorithm 48
4.4 Chapter Summary 49

5 Empirical Evaluation **51**
5.1 Challenge 1: Sample Efficiency and Exploration 52
5.1.1 Simulated Vehicle Velocity Control 53
5.1.2 Fuel World 56

5.2 Challenge 2: Modeling Continuous Domains . . . 62
5.2.1 Simulated Vehicle Velocity Control . . . 62
5.2.2 Continuous Task Performance . . . 63
5.3 Challenge 3: Delayed Actions . . . 67
5.3.1 Simulated Vehicle Velocity Control . . . 67
5.3.2 Delayed Gridworld . . . 69
5.4 Challenge 4: Real Time Action . . . 73
5.4.1 Simulated Vehicle Velocity Control . . . 73
5.4.2 Mountain Car . . . 75
5.5 Dependent Transitions . . . 78
5.6 TEXPLORE on a Physical Robot . . . 83
5.7 Chapter Summary . . . 84

6 Further Examination of Exploration . . . 85
6.1 Explicit Exploration . . . 87
6.1.1 Methodology . . . 87
6.1.2 Empirical Evaluation . . . 88
6.2 Variance and Novelty Intrinsic Rewards . . . 94
6.2.1 Methodology . . . 95
6.2.2 Empirical Evaluation . . . 98
6.3 On-Line Learning of Exploration Parameters . . . 105
6.3.1 Methodology . . . 105
6.3.2 Empirical Evaluation . . . 108
6.4 Empirical Comparison . . . 115
6.5 Chapter Summary . . . 119

7 Related Work . . . 121
7.1 Sample Efficiency . . . 121
7.1.1 Exploration . . . 122
7.1.2 Intrinsic Motivation . . . 124
7.1.3 Bayesian Methods . . . 126
7.1.4 Models . . . 128
7.2 Continuous Domains . . . 129
7.3 Observation and Action Delays . . . 130
7.4 Real-Time Architectures . . . 131
7.5 Real-World Problem Domains . . . 133
7.6 Chapter Summary . . . 134

8 Discussion and Conclusion . . . 137
8.1 Summary . . . 137
8.2 Contributions . . . 139
8.3 Discussion . . . 140
8.4 Future Work . . . 142
8.4.1 Expanded Applicability of RL . . . 142
8.4.2 Exploration . . . 144
8.4.3 Opponent Modeling . . . 145

8.4.4 Lifelong Learning 146
8.4.5 Summary 147
8.5 Conclusion 147

A TEXPLORE Pseudo-code **149**

B Evaluation Domains **155**

References **159**

1 Introduction

This chapter presents the motivation and objectives for this book, and an overview of the work presented in the book. I begin by presenting the motivation for applying reinforcement learning (RL) to robots. Next, I present four specific challenges for applying RL to robotics problems. Then I describe a particular challenge of learning in few enough samples to be effective on domains with limited, expensive samples such as robots. I then provide a brief overview of the TEXPLORE *algorithm introduced in this book and how it addresses these issues. Finally I present the contributions of this book and preview of each chapter of the book.*

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) is a paradigm for learning in sequential decision making processes that could solve the problems of learning and adaptation on robots. In RL, an agent is seeking 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).

My motivation for this work is to develop a new RL algorithm that applies to real-world problems such as controlling robots. The number of robots being used in society is continually growing. However, most of these robots require someone to pre-program them for their specific task (e.g. vacuum cleaning robots, gutter cleaning robots), or require a user to tele-operate them (e.g. rescue robots, bomb detection robots). Developing an RL algorithm that applies naturally and practically to robots, and then applying it to them would make robots more useful in three ways: 1) robots would learn to improve their performance on their task *on-line*, while performing the task; 2) robots would generalize their knowledge to new situations and environments for which they were not pre-programmed; and 3) robots would require less hand-coding, as more of their skills could be left for them to learn.

However, learning on robots presents a number of challenges for existing RL algorithms, because a successful method must learn in few actions while running

T. Hester, *TEXPLORE: Temporal Difference Reinforcement Learning for Robots and Time-Constrained Domains*, Studies in Computational Intelligence 503,
DOI: 10.1007/978-3-319-01168-4_1, © Springer International Publishing Switzerland 2013

on the robot. In addition, the method must handle continuous state as well as noisy and/or delayed sensors and actuators. RL algorithms have 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 book, we identify four properties of an RL algorithm that would make it generally applicable to a broad range of robot control tasks, which we will henceforth call the *RL for Robotics Challenges*:

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 select 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 would not only make an RL algorithm applicable to more robotic control tasks, but it would also make such an algorithm applicable to many other real-world tasks. We note that robots also typically have continuous action spaces. We leave addressing continuous actions for future work as we have found that using a discretized action space works well in many domains.

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. In Table 1.1, we provide a listing of related work, each of which addresses some of these challenges, but not all four of them. We say that an algorithm *addresses* a challenge if it is explicitly focused on that challenge or if its approach to that challenge is applicable to robotics problems. However, even some of the methods that address a particular challenge may not do so in a way that is effective for all domains. I describe these algorithms in further detail in Chapter 7, 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 while learning to control 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 parameterization that will be useful for generalization. While Table 1.1 only shows methods that learn from scratch, there are also related works for robot learning that start with experience from an expert user (Ng et al., 2003; Bagnell and Schneider, 2001) or from robot simulation (Kolter et al., 2010).

Table 1.1. This table shows state-of-the-art learning algorithms that address some of the *RL for Robotics Challenges* required for performing reinforcement learning on-line on robots. None of the methods prior to this book address all four challenges and even the challenges that are addressed by these methods may not be addressed in a way sufficient for the robotic domains we are interested in.

Algorithm	Citation	Sample Efficient	Real Time	Contin-uous	Delay
R-MAX	(Brafman and Tennenholtz, 2001)	Yes	**No**	**No**	**No**
Q-LEARNING	(Watkins, 1989)	**No**	Yes	**No**	**No**
with F.A.	(Sutton and Barto, 1998)	**No**	Yes	Yes	**No**
SARSA	(Rummery and Niranjan, 1994)	**No**	Yes	**No**	**No**
PILCO	(Deisenroth and Rasmussen, 2011)	Yes	**No**	Yes	**No**
NAC	(Peters and Schaal, 2008)	Yes	**No**	Yes	**No**
BOSS	(Asmuth et al., 2009)	Yes	**No**	**No**	**No**
Bayesian DP	(Strens, 2000)	Yes	**No**	**No**	**No**
MBBE	(Dearden et al., 1999)	Yes	**No**	**No**	**No**
SPITI	(Degris et al., 2006)	Yes	**No**	**No**	**No**
MBS	(Walsh et al., 2009a)	Yes	**No**	**No**	Yes
U-TREE	(McCallum, 1996)	Yes	**No**	**No**	Yes
DYNA	(Sutton, 1990)	**No**	Yes	**No**	**No**
DYNA-2	(Silver et al., 2008)	**No**	Yes	Yes	**No**
KWIK-LR	(Strehl and Littman, 2007)	Yes	**No**	**Partial**	**No**
FITTED R-MAX	(Jong and Stone, 2007)	Yes	**No**	Yes	**No**
DRE	(Nouri and Littman, 2010)	Yes	**No**	Yes	**No**
TEXPLORE	This book	Yes	Yes	Yes	Yes

Our objective with this work is to develop a reinforcement learning algorithm that can run on-board a robot and learn to control it in real time without pauses for off-line computation. Such an algorithm would be useful for performing long-term in-situ learning on the robot. For an algorithm to be capable of this objective, it must meet all four *RL for Robotics Challenges*.

1.1 Time-Constrained Domains

While we desire the algorithm to solve all four challenges, the first challenge, learning in very few samples, is a particular focus of this book. On many problems, each action the agent takes can be very expensive in terms of money, time, and labor. For example, robots are expensive and suffer from wear and tear, short battery life, and potentially overheating. In addition, performing learning on them often requires human supervision, and perhaps even particular environmental conditions (e.g. good lighting for vision, good weather for driving, etc.). Along with expensive samples, such problems are often very large, with high-dimensional continuous state and action spaces. The combination of large state-action spaces and expensive samples means that learning in few enough samples to be useful can be very difficult.

Addressing Challenge 1 of learning in few samples is applicable not just to robots, but to many other RL problems. Many real-world problems also have very expensive samples and large state-action spaces (a sample of these problems can be found in Section 7.5). Users will not apply RL algorithms to these problems unless the algorithms can learn in a very small number of actions. In this section, we formally characterize the class of domains where addressing Challenge 1 is critical as *time-constrained* domains. The TEXPLORE algorithm that we present in this book is meant to address this challenge in these domains. In this class of domains, the agent has a short lifetime relative to the size of the domain, and does not have enough actions in its lifetime to guarantee that it can find an optimal policy. Thus in this class of domains, it is important for the agent to find a good policy quickly, in contrast to spending more time learning and exploring to find an optimal policy.

An important criterion for algorithm performance is the *sample complexity* of the algorithm, or the number of actions it must take to find a near-optimal policy. The sample complexity *of exploration* is the number of sub-optimal exploratory actions the agent must take. Kakade (2003) proves the lower bound for this sample complexity is $O(\frac{NA}{\epsilon(1-\gamma)}log\frac{1}{\delta})$ for stochastic domains, where N is the number of states, A is the number of actions, γ is the discount factor, and the algorithm finds an ϵ-optimal policy with probability $1-\delta$. There are many cases where this lower bound is already an unacceptable number of actions. For example, if the problem has billions of states or actions, then the NA factor above is already too big. Alternatively, on a robotic task, actions may take minutes to complete, such that even requiring a few thousand actions to solve the problem is unacceptable. What can we do in these cases where we do not have enough actions to guarantee convergence to an optimal policy? This book focuses on this problem by addressing the following question:

How should an on-line reinforcement learning agent act in time-constrained domains?

In this book, we seek to address the problem of acting in *time-constrained* domains, which we define as domains where the agent is limited to two orders of magnitude fewer actions than the lower bound presented above (time-constrained domains are formally defined and examined in Chapter 2). In addition, the agent should act in real time, at whatever action frequency the problem requires. In time-constrained domains, TEXPLORE will find a better policy and accrue more cumulative reward in its lifetime than other methods. When given a longer lifetime (such that the problem is not a time-constrained one), other methods may find the optimal policy while TEXPLORE will not.

Essentially, we are focused on problems where we cannot guarantee that the agent will learn an optimal policy. Instead, the algorithm must limit its exploration of the domain and start exploiting its knowledge earlier. Since it is only exploring a limited part of the domain, it must make some assumptions about the other parts of the domain. In particular, instead of assuming that the transition and reward dynamics of each state may be arbitrarily different than the

others, TEXPLORE generalizes these dynamics between states. It then performs limited, targeted exploration to improve its model and quickly starts exploiting this model to accrue high rewards within its limited lifetime.

1.2 Algorithm Overview

This book introduces the TEXPLORE algorithm, which addresses the question presented in the previous section. In addition, it addresses all four *RL for Robotics Challenges*. Importantly, not only does TEXPLORE solve each of the four challenges, it does so while ensuring that each solution meshes well with the others, to form a complete algorithm for performing RL on robots. The TEXPLORE algorithm has been 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. This section presents a brief overview of the algorithm, which is presented in detail in Chapters 3 and 4.

For Challenge 1 of being sample efficient, particularly in domains where the agent has a limited lifetime, the agent must learn a high-rewarding (but not necessarily optimal) policy in as few actions as possible, so it can use its remaining lifetime to exploit what it has learned. This lifetime constraint means the agent must be relatively greedy compared to many other RL methods (i.e. it must switch from exploring to exploiting quickly). This approach gives up guarantees of optimality (and thus the need to explore every state-action) in order to find a high-rewarding policy in a very small number of actions. Since it only performs limited exploration, its exploration must be efficient and targeted so that it can learn a model very quickly.

The TEXPLORE algorithm learns a model of its domain through its experience, and then uses this model to plan a policy to follow in the domain. Following such a model-based approach enables TEXPLORE to plan multi-step exploration trajectories as well as update its value function through internal simulations using its model. Since TEXPLORE has a limited number of time steps for exploration, it must make some assumptions about the parts of the domain it is not exploring. Therefore, instead of a typical tabular model, which learns a separate model for each state-action, TEXPLORE incorporates generalization into the model learning, such that the transition and reward effects of actions are generalized across states. This generalization speeds up learning by providing the agent with an estimate of the model for unseen states. Unlike the typical use of function approximation in RL, where the value function is approximated, TEXPLORE uses generalization in the model, while maintaining an accurate value function based on this model. Others have developed algorithms with model generalization before, e.g. using instance-based models (Jong and Stone, 2007) or decision trees (Degris et al., 2006) (detailed further in Section 7.1.4).

The particular method that TEXPLORE uses for model learning is decision trees (Quinlan, 1986). Decision trees perform well in many domains by splitting the state space into regions with similar transition dynamics. In addition, they

provide natural solutions to Challenge 2 of acting in domains with continuous state and Challenge 3 of acting in domains with actuator or sensor delays. For domains with continuous state representations, TEXPLORE's decision trees can be replaced with regression trees (Quinlan, 1992), which learn a regression model in each leaf of the tree. These regression tree models can then make predictions about continuous state. For domains with actuator or sensor delays, TEXPLORE adds its previous k actions as inputs to the tree model, allowing it to model and predict the delay in the system.

Finally, TEXPLORE should apply to realistic tasks where actions must be taken frequently, addressing Challenge 4. The agent must be ready to select an action when required and cannot be slow to respond because it is performing batch processing or updates to its model. For example, if an agent is driving a car, it cannot wait for a few seconds to think about what to do when a car in front of it slams on its brakes. It must respond immediately when an action is requested. In Chapter 3 of this book, I present a real time architecture for model-based reinforcement learning. This architecture enables real time action by performing model updates, planning, and action selection in parallel threads, such that acting is not constrained by the time required for model updates or planning.

After presenting the algorithm, in Chapter 5 I present evaluations of it on time-constrained tasks, including both discrete and continuous domains. In addition, I present TEXPLORE learning a decision-making task on a physical robot while running in real time on-board the robot. We compare the cumulative reward and average reward that the algorithm achieves during a limited lifetime with the reward accrued by other state of the art approaches. Since TEXPLORE is not guaranteed to find an optimal policy, its final policies in the limit may be worse than other approaches, but we show that its accrued rewards and learned policies are better than those of other methods when working with a constrained lifetime.

In Chapter 6, I provide a deeper examination of exploration. The best exploration strategy for an RL algorithm varies depending on the task at hand. In this chapter, I examine exploration strategies for two opposite types of domains. First, I present an approach for domains where transitions and rewards are located arbitrarily and the best the agent can do is to explore each state-action. Next, I present an approach for domains where there are richer, more complex state features that the agent can use to explore more intelligently. Then I present an approach that can learn to use the best of a set of exploration strategies on-line. Finally, I present some empirical comparisons of these exploration approaches with TEXPLORE's approach on a few domains.

In summary, in this book I present a model-based RL algorithm called TEXPLORE that performs limited, targeted exploration to learn a good policy quickly. It generalizes the effects of actions across states, allowing it to limit its exploration more than approaches that do not make such assumptions. The algorithm forgoes guarantees of optimality, instead targeting its exploration on particular state-actions that may be most useful to the model learning to accrue rewards as quickly as possible. TEXPLORE is capable of modeling actuator and sensor

delays and continuous states. In addition, it utilizes a novel real time parallel architecture that enables it to act in real time.

1.3 Contributions

This book provides the following six major contributions to the field:

1. TEXPLORE: The TEXPLORE algorithm, which is the first algorithm to address all four of the *RL for Robotics Challenges* together simultaneously in the same algorithm. In addition, TEXPLORE is effective at learning good policies and accruing high rewards on time-constrained domains. The TEXPLORE algorithm is not only presented in this book, but has been publicly released as an open-source ROS package at:
`http://www.ros.org/wiki/rl-texplore-ros-pkg`.
2. *Generalized Models*: Methods for learning MDP models that: 1) generalize transition and reward dynamics across state-actions; 2) provide a measure of uncertainty in their predictions; 3) can model continuous domains; 4) can model domains with sensor or actuator delays; and 5) can learn accurate models of dependent feature transitions in factored domains.
3. *Targeted Exploration*: An examination of exploration methods for RL agents with models that generalize across state-actions. This examination includes methods to drive the agent to perform limited, targeted exploration, methods to explore uncertain or novel states, and intrinsically motivated exploration for domains with little or no external rewards.
4. *Real Time Architecture*: A parallel real time model-based RL agent architecture that enables model-based RL agents to act in real time, without being constrained by the time required for model updates or planning. In addition, this architecture is capable of planning in both continuous domains and domains with sensor or actuator delays.
5. *ROS RL Interface*: We developed a RL interface for ROS (Robot Operating System) to make it easy to integrate RL with existing robots already using ROS. The interface defines messages for the agent to send and receive from the environment to perform learning. This interface is available as part of our ROS package at: `http://www.ros.org/wiki/rl_msgs`.
6. *Evaluation*: Empirical evaluation of TEXPLORE learning in a variety of time-constrained domains, and in particular, evaluation of TEXPLORE learning to control a physical robot while running in real time on-board the robot.

Each of these contributions is described in detail in the remainder of this book.

1.4 Overview

The rest of this book is organized as follows.

Chapter 1: I present background on Markov Decision Processes and reinforcement learning. I present some typical model-based and model-free RL algorithms.

I continue into more detail about model-based methods and present a sample-based planning algorithm. Finally, we define the set of *time-constrained* problems that this book is focused on.

Chapter 2: I present the real time model based architecture (RTMBA) that TEXPLORE utilizes. This architecture separates the model learning, planning, and acting into three parallel threads such that the agent can act as fast as necessary without being constrained by the time required for model learning or planning. The architectures uses sample-based planning methods such as Monte Carlo tree search to perform anytime planning.

Chapter 4: This chapter presents the TEXPLORE algorithm, in particular its approach to model learning and exploration. TEXPLORE uses decision trees to learn models of the transition and reward dynamics of the domain. For exploration, TEXPLORE utilizes random forests of these tree models, where each tree represents a different hypothesis of the true dynamics of the domain.

Chapter 5: In this chapter, we empirically evaluate TEXPLORE's solution to each of the *RL for Robotics Challenges*. For each challenge, we compare TEXPLORE's solution with other possible approaches, both on a simulated robotics task and a second example task. Finally, we present experiments demonstrating TEXPLORE's ability to learn to control a real robot while running on-board the robot.

Chapter 6: I further examine possible approaches to exploration in this chapter. I present TEXPLORE with explicit exploration (TEXPLORE-EE) for domains that require exploration of each state-action to find arbitrarily located transitions or rewards. Then I present the TEXPLORE with variance and novelty intrinsic rewards (TEXPLORE-VANIR) algorithm that performs more intelligent, targeted exploration in domains with richer, more informative state features. I also present the LEO algorithm for learning the best exploration strategy on-line. Finally, I evaluate these various exploration approaches on a handful of tasks.

Chapter 7: I present work related to the various aspects of TEXPLORE in this chapter. For each of the *RL for Robotics Challenges*, I present related work and other potential solutions to the challenge. I also present work that does not address any individual challenge, but is focused on robotics or other real-world learning problems.

Chapter 8: In this chapter, I summarize the contributions of this book. Then, I discuss various interesting issues that the book raises and indicate directions for future work.

Appendix A: This appendix presents comprehensive pseudo-code for the complete TEXPLORE algorithm.

Appendix B: This appendix lists all the evaluation domains used in the book, along with their properties.

Each chapter is dependent on the background chapter (Chapter 2). In addition, for Chapter 5, which presents empirical evaluations of the algorithm, it would be useful to read Chapters 3 and 4 on the TEXPLORE algorithm. For Chapter 6, which examines exploration further, it would be useful to already be familiar with the exploration of the TEXPLORE algorithm, which is presented in Chapter 4. The other chapters are largely self-contained and an interested reader can read them without first reading the previous chapters.

2 Background and Problem Specification*

This chapter presents background on sequential decision making and reinforcement learning as well as the specification of the problems that this book is addressing. I begin by presenting a formal description of sequential decision making problems as Markov Decision Processes. Then I describe the reinforcement learning problem. Next, I explain the difference between model-free and model-based approaches and present example algorithms of each class. I present details on using model-based RL in factored domains. In Section 2.2.4, I present an important aspect of model-based RL, planning, along with the UCT *planning algorithm. In the next section, I formally define the class of domains this book is focused on: time-constrained domains where learning in very few samples is critical. Finally, I present a specific example of a domain from this class and demonstrate how each of the* RL for Robotics Challenges *are present in this domain.*

Many tasks that we would desire a robot or agent to perform can be defined as *sequential decision making* problems. There is an agent interacting in some *environment*. The agent is making a series of decisions that possibly affect the environment, and is evaluated based on these decisions through a scalar reward signal. In *reinforcement learning*, as the agent interacts with the environment, it is learning to improve its decision making, with the goal of maximizing the reward it receives. Typically, the agent begins with no or very little prior information about the environment or the task. The sequential decision making problem is defined formally below.

2.1 Markov Decision Problems

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 $T(s, a, s') = P(s'|s, a)$. We define the number of states $N = |S|$. In each state $s \in S$, the agent takes an action $a \in A$. As shown in Figure 2.1, upon taking this action, the agent receives a reward $R(s, a)$ and transitions to 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 $\pi = P(a|s)$ specifies, for each state, a distribution over actions that the agent will take.

* This chapter contains material from two publications: (Hester and Stone, 2011, 2012b).

T. Hester, *TEXPLORE: Temporal Difference Reinforcement Learning for Robots and Time-Constrained Domains*, Studies in Computational Intelligence 503,
DOI: 10.1007/978-3-319-01168-4_2, © Springer International Publishing Switzerland 2013

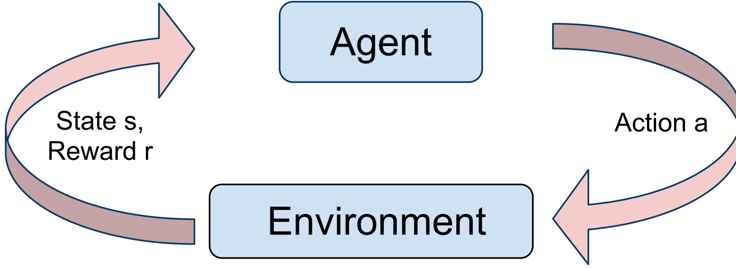


Fig. 2.1. How the reinforcement learning agent interacts with the environment

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:

$$J = \sum_t \gamma^t r_t, \tag{1}$$

where $0 < \gamma < 1$ is the discount factor and r_t is the reward obtained at time step t. One set of approaches to this problem, called *policy search* methods, search in the space of policies directly for policies that accumulate high rewards. Alternatively, *value function* methods learn to predict the value of discounted reward that will be received from any state-action and use this value function to calculate a policy. Since value function methods make more use of information through the calculation of the value function, they are typically more sample efficient than policy search methods when the MDP is discrete, finite, and fully observable. In addition, they have a string of theoretical results proving their convergence (Watkins, 1989; Brafman and Tennenholtz, 2001). In this work, one of our goals is to apply RL to domains such as robots where samples are very expensive. Therefore, we require methods with low sample complexity, and we follow the value function approach in this work.

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 optimal value function $Q^*(s,a)$ provides maximal values in all states and is determined by solving the Bellman equation:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q^*(s',a'). \tag{2}$$

The optimal policy π^* is then as follows:

$$\pi^*(s) = \text{argmax}_a Q^*(s,a). \tag{3}$$

We have stated that the goal of the agent is to maximize its expected total discounted reward. There are a number of ways an agent's performance can be evaluated. First, we can look at how many actions it takes the agent to learn an ϵ-optimal policy (a policy whose expected return is within ϵ of that of the

optimal policy π^*). Second, we can measure the total discounted reward accrued over the agent's lifetime, including while it is learning. Third, we can look at the final performance achieved by the agent. In this work, we evaluate all three criteria.

2.2 Value Function Reinforcement Learning

Value function based RL methods fall into two general classes: model-free (direct) and model-based (indirect) methods. *Model-free* methods update their value function directly from experience in the environment. *Model-based* methods however, perform their updates from a model of the domain, rather than from experience in the domain itself. Both of these classes of methods are explained in more detail in the following sections.

2.2.1 Model-Free Methods

Model-free RL methods learn by updating their value function directly from experience in the environment. Two commonly used model-free RL methods are SARSA (Rummery and Niranjan, 1994) and Q-LEARNING (Watkins, 1989). Pseudo-code for Q-LEARNING is shown in Algorithm 2.1. The $x \stackrel{\alpha}{\leftarrow} y$ operator is shorthand for $x \leftarrow \alpha(y - x) + (1 - \alpha)x$, which is the incremental stochastic approximation update of x towards the sample y. Q-LEARNING makes incremental updates to the value function based on its experiences through use of the Bellman equation. Q-LEARNING is proven to converge to the optimal value function (and thus optimal policy) when visiting each state-action infinitely often and with an appropriate annealing of the learning rate. Q-LEARNING is a representative model-free algorithm and is used for comparison in our experiments in Chapter 5. It was chosen because it is one of the most straightforward and theoretically grounded model-free RL algorithms. However, it is important to note that it is not the most practical approach for the types of problems we wish to address, as model-free methods are not particularly sample efficient.

Algorithm 2.1. Q-LEARNING

1: **Input:** S, A, α ▷ S: state space, A: action space, α: learning rate
2: Initialize Q arbitrarily for all S, A
3: Initialize policy π randomly
4: Initialize s to a starting state in the MDP
5: **loop**
6: Choose $a \leftarrow \pi(s)$
7: Take action a, observe r, s'
8: $Q(s,a) \stackrel{\alpha}{\leftarrow} r + \gamma \max_{a \in A} Q(s', a)$
9: $\pi(s) \leftarrow \text{argmax}_{a \in A} Q(s, a)$
10: $s \leftarrow s'$
11: **end loop**

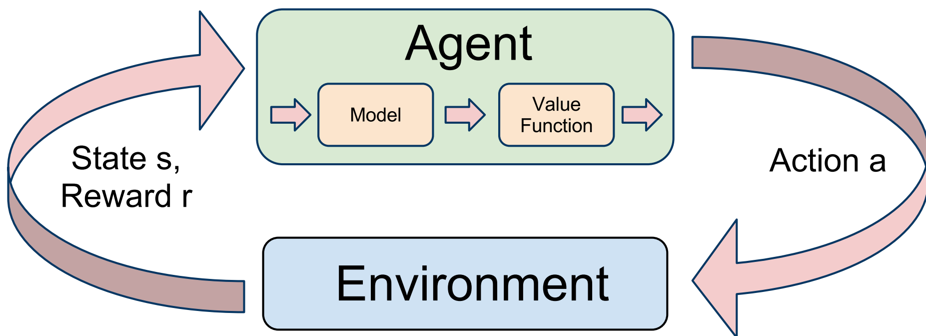


Fig. 2.2. Model-based RL agents use their experiences to first learn a model of the domain, and then use this model to compute their policy

2.2.2 Model-Based Methods

In contrast to model-free methods, *Model-based* RL methods perform their updates from a model of the domain, rather than from experience in the domain itself. Instead, the model is learned from experience in the domain, and then the value function is updated by planning over the learned model. This sequence is shown in Figure 2.2. This planning can take the form of simply running a model-free method on the model, or it can be a method such as *value iteration* (Sutton and Barto, 1998) or *Monte Carlo Tree Search* (Kocsis and Szepesvári, 2006).

The models learned by these methods can vary widely. Models can be learned entirely from scratch, the structure of the model can be given so that only parameters need to be learned, or a nearly complete model can be provided. A common approach is to use a tabular model where the agent learns a separate model for each state-action based on the frequencies of different outcomes at each state. The agent could also learn the model using any supervised learning technique, such as decision trees (Degris et al., 2006) or Gaussian Process regression (Deisenroth and Rasmussen, 2009).

If the algorithm can learn an accurate model quickly enough, model-based reinforcement learning can be more sample efficient than model-free methods. Once an accurate model is learned, an optimal policy can be planned without requiring any additional experiences in the world. For example, when an agent first discovers a goal state, the values of its policy can be updated at once through planning over its new model that represents that goal. Conversely, a model-free method would have to follow a trajectory to the goal many times for the values to propagate all the way back to the start state. The better sample efficiency of model-based methods typically comes at the cost of more computation for learning the model and planning a policy and more space to represent the model.

Another advantage of models is that they provide an opportunity for the agent to perform *targeted* exploration. The agent can plan a policy using its model to drive the agent to explore particular states; these states can be states it has not

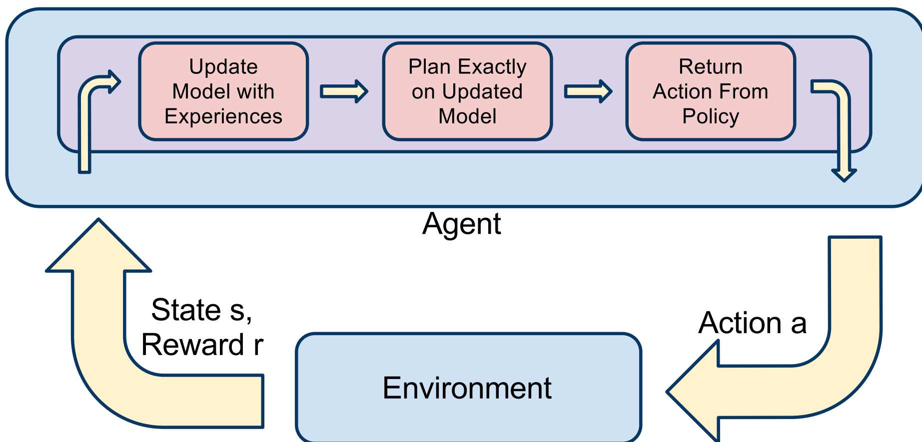


Fig. 2.3. Typically, model-based agents interleave model learning and planning sequentially, first completing an update to the model, and then planning on the updated model to compute a policy

visited or is uncertain about. Methods such as R-MAX (Brafman and Tennenholtz, 2001) modify the agent's model of the domain with artificial rewards to encourage it to explore. A key to learning a model quickly is acquiring the right experiences needed to learn the model (similar to *active learning*). Various methods for exploring in this way exist, leading to fast learning of accurate models, and thus good sample efficiency.

There are a number of ways to combine model learning and planning in a model-based RL agent. Typically, as the agent interacts with the environment, its model gets updated at every time step with the latest transition, $< s, a, r, s' >$. Each time the model is updated, the algorithm re-plans on it with its planner (as shown in Figure 2.3). This approach is taken by many algorithms (Brafman and Tennenholtz, 2001; Degris et al., 2006). However, due to the computational complexity of learning the model and planning on it, it is not always feasible. Another approach is to do model updates and planning in batch mode, only performing updates after every episode or every k actions. Due to the high action frequency required to control robots, this approach is used by many algorithms that perform learning on robots (Deisenroth and Rasmussen, 2011; Kober and Peters, 2011). However, this approach means that the agent must stop acting for long pauses while it performs batch updates, which may not be acceptable in some problems.

R-MAX is a representative model-based approach that uses a tabular model and explores thoroughly by driving the agent to visit each state-action m times (Brafman and Tennenholtz, 2001). Pseudo-code for R-MAX is shown in Algorithm 2.2. R-MAX uses a tabular maximum-likelihood model, keeping counts of the number of times each action was taken and which outcomes were seen. All state-actions with fewer than m visits are considered *unknown* and are given a reward of R_{max} (the maximum reward in the domain) to encourage the agent

Algorithm 2.2. R-MAX

```
1: Input: S, A, m, R_max
2: Initialize s_r as absorbing state with reward R_max
3: Initialize all counts C to 0
4: Initialize s to a starting state in the MDP
5: loop
6:     Choose a = π(s)
7:     Take action a, observe r, s'
8:     Increment C(s, a, s'), C(s, a)                ▷ Update model
9:     Rsum(s, a) ← Rsum(s, a) + r
10:    if C(s, a) ≥ m then                           ▷ Known state
11:        R(s, a) ← Rsum(s, a)/C(s, a)
12:        for all s' ∈ C(s, a, ·) do
13:            T(s, a, s') ← C(s, a, s')/C(s, a)
14:        end for
15:    else                                          ▷ Unknown state
16:        R(s, a) ← R_max
17:        T(s, a, s_r) ← 1
18:    end if
19:    Call VALUE-ITERATION                          ▷ Plan updated policy
20:    s ← s'
21: end loop
```

to explore them. After each update to the model, R-MAX re-plans on its model using a method such as value iteration to calculate a new policy. R-MAX is guaranteed to find the optimal policy in time polynomial in the number of states and actions, but it may still result in an inordinate amount of time spent exploring the domain. We use R-MAX for comparison with our algorithms in our experiments in Chapter 5, because it is a straightforward, theoretically grounded representative of the class of model-based RL algorithms. However, in the real world problems we are focused on, R-MAX can be too computationally expensive and can explore too much to accrue good rewards on domains with a limited number of samples.

As a comparison of the sample efficiency of model-free versus model-based RL methods, we present some results here comparing Q-LEARNING and R-MAX as representative algorithms from each class. R-MAX was run with $m = 1$ and Q-LEARNING was run with $\alpha = 0.3$ and with Q-values initialized optimistically to 0. We ran the algorithms on a 10×5 grid world domain with two rooms. The agent received a reward of -1 each step until it reached the goal state, when the episode terminated with a reward of 0. All the transitions were deterministic and the discount factor was 0.98. Experiments were run on a Dell XPS laptop with a 2.8 GHz Intel Core i7-2640M processor and 8 GB of RAM.

We compared both the number of episodes and amount of wall clock time each algorithm took to learn a 0.2-optimal policy. We ran each algorithm for 2000 episodes on the domain, and averaged our results over 30 trials. On average, it took Q-LEARNING 592.27 episodes to learn a 0.2-optimal policy, while it took

R-MAX only 12.10 episodes (48.9 times faster). In contrast, it took Q-LEARNING an average of 0.0039 seconds to learn a 0.2-optimal policy, while R-MAX took 0.7962 seconds (204.1 times slower). These results show that model-based RL methods can be much more sample efficient than model-free methods, but at the cost of more computation time. For the real-world domains we are interested in, we require methods that are *both* sample and computationally efficient. In addition, there is progress to be made to make model-based methods such as R-MAX work efficiently on more complex, stochastic domains with limited samples.

2.2.3 Factored Models

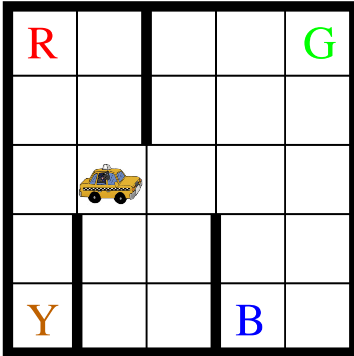
As specified earlier in Section 2.1, in many tasks, the agent's state can be represented by a set of state features that describe the world. Many RL algorithms (Guestrin et al., 2002; Degris et al., 2006; Strehl et al., 2007; Chakraborty and Stone, 2011) take advantage of these factored representations to accelerate model learning by learning Dynamic Bayes Network (DBN) or decision tree models of the domain. A key assumption that helps these approaches learn models faster is that they predict each feature independently based on the agent's previous state and action. This simplifies the model learning problem and reduces the number of experiences required for the agent to learn an accurate model. This assumption that features can be predicted independently is made by all of these factored methods.

Learning such a *factored* model can reduce the amount of data required to learn an accurate model of the domain. In the DBN model, each feature of the next state may only be dependent on some subset of features from the previous state and action. The features that a given state feature are dependent on are called its *parents*. If the features have fewer parents than the total number of features, then the DBN model can be learned faster than a tabular model. When using a DBN transition model, it is assumed that each feature transitions independently of the others. The probability of a particular next state is the product of the probabilities of each of its features:

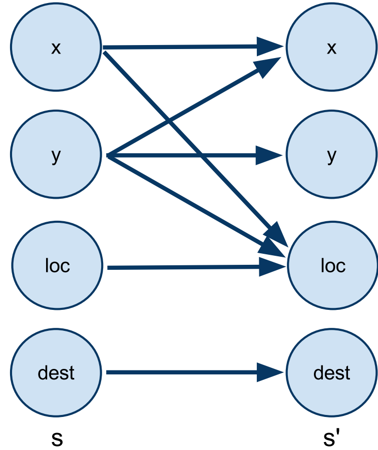
$$P(s'|s,a) = \Pi_{i=0}^{n} P(s'_i|s,a). \quad (4)$$

Learning the structure of this DBN transition model is known as the *structure learning problem*. Once the structure of the DBN is learned, the *conditional probabilities* for each edge must be learned.

Figure 2.2.3 shows an example DBN for the *Taxi* domain (Dietterich, 1998), a popular toy domain in the RL community. In the *Taxi* domain, the agent's state is made up of four features: its X and Y location, the passenger's LOCATION, and the passenger's DESTINATION. The DESTINATION is one of four colored landmarks (i.e. R, G, B, Y), and the LOCATION is one of these landmarks or in the taxi. The agent's goal is to navigate the taxi to the passenger, pick up the passenger, navigate to her destination, and drop off the passenger. The Y location of the taxi is only dependent on its previous Y location, and not its X location or the LOCATION or DESTINATION of the passenger. Because of the vertical walls in



(a) Taxi Domain.



(b) DBN Transition Structure.

Fig. 2.4. 2.4(a) shows the Taxi domain, where the agent must navigate the taxi to the passenger, pick her up, and then navigate to her destination and drop her off. 2.4(b) shows the DBN transition model for this domain. Here the X feature in state s' is only dependent on the X and Y features in state s and the Y feature is only dependent on the previous Y. The passenger's DESTINATION is only dependent on her previous DESTINATION, and her LOCATION is dependent on her previous LOCATION and the taxi's previous (X,Y) coordinates.

the domain, the X location of the taxi is dependent on both X and Y. The LOCATION of the passenger only changes if the PICKUP action is performed, and is dependent on the taxi's (X,Y) location and the passenger's LOCATION. If this structure is known, it makes the model learning problem much easier, as the same model for the transition of the X and Y variables can be used for any possible value of the passenger's LOCATION and DESTINATION.

2.2.4 Planning

One of the most computationally expensive steps for model-based methods is to plan on their updated model to compute a new policy. Typical model-based methods use exact planning methods such as value iteration to plan a new policy every time the model changes. Value iteration iterates over all the states in the domain, updating their values using the Bellman equations. In anything but the smallest domains, this process can be quite slow. For model-based methods to work in these large domains, we need a faster planning method. UCT (Kocsis and Szepesvári, 2006) is one such method. UCT is a Monte Carlo planning method that works by sampling trajectories from the agent's current state using the agent's model, thus focusing its updates on states the agent is likely to visit soon.

UCT searches from the start state to a maximum depth or terminal state, selecting actions based on upper confidence bounds using the UCB1 algorithm (Auer et al., 2002). Algorithm 2.3 shows pseudo-code for the UCT algorithm. This

Algorithm 2.3. UCT (s, d)

1: **Inputs:** α, r_{range}
2: **if** TERMINAL or d == MAXDEPTH **then**
3:     **return** 0
4: **end if**
5: $a \leftarrow \text{argmax}_{a'}(Q^d(s,a') + 2 \cdot \frac{r_{max}-r_{min}}{1-\gamma} \cdot \sqrt{\log\left(c(s,d)\right)/c(s,a',d)})$
6: $(s', r) \leftarrow$ SAMPLENEXTSTATE(s, a) ▷ Sample from model
7: $update \leftarrow r +$ UCT$(s', d+1, \alpha)$
8: Increment $c(s, d)$
9: Increment $c(s, a, d)$
10: $Q^d(s, a') \overset{\alpha}{\leftarrow} update$
11: **return** *update*

function is called from the agent's current state with a depth of 0. The algorithm maintains a count, $C(s, d)$, of visits to each state at a given depth in the search, d, as well as a count, $C(s, a, d)$, of the number of times action a was taken from that state at that depth. These counts are used to calculate the upper confidence bound to select the action. The action selected at each step is calculated using the upper tail of the confidence interval on line 5. By selecting actions using the upper tail of the confidence interval, the algorithm mainly samples good actions, while still exploring when other actions have a higher upper confidence bound.

After sampling a trajectory out to a maximum depth or a terminal state, the algorithm updates the values of all the state-actions encountered along the trajectory. This process constitutes 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). Modified versions of UCT have had great success in the world of Go algorithms as a planner with the model of the game already provided (Wang and Gelly, 2007). UCT is also used as the planner inside several model-based reinforcement learning algorithms (Silver et al., 2008, 2012). We make use of UCT in this book as the planning method used in the real time architecture presented in Chapter 3.

2.3 Time-Constrained Domains

Now that background on reinforcement learning has been presented, we can define the *time-constrained* domains that this book addresses. Time-constrained domains were introduced briefly in Chapter 1, but we will formally define them in this section. Time-constrained domains are a class of domains where the agent has a very small number of actions relative to the size of the domain, requiring the agent to learn a good policy very quickly.

We will define time-constrained domains using the sample complexity of exploration. The sample complexity of exploration is the number of sub-optimal exploratory actions an agent must take. For a given domain, Kakade (2003)

proved that the sample complexity of exploration is at least $O(\frac{NA}{\epsilon(1-\gamma)}log\frac{1}{\delta})$ actions. This bound means that at best, an algorithm must take at least that many actions before it can be guaranteed to start acting optimally in a worst-case environment. Even in deterministic domains, the agent must take at least $O(\frac{NA}{1-\gamma})$ exploratory actions, which can be an unacceptable number of actions in many large domains.

For continuous domains, it has been proven that under certain assumptions, Q-LEARNING with function approximation will converge to the optimal policy when the Markov chain is uniformly ergodic, $\pi(s,a) > 0$ for all $a \in A$ and μ_x-almost all $x \in X$, which essentially says that the agent must visit all state-action pairs infinitely often (Melo et al., 2008).

A few algorithms such as R-MAX (Brafman and Tennenholtz, 2001) or MET-RMAX (Diuk et al., 2009) are considered *efficient* RL algorithms because they require a number of actions polynomial in N, A, $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, and $\frac{1}{1-\gamma}$. Our work is focused on domains where these algorithms take too many actions to be useful. In fact, our focus is on domains where even an algorithm that takes only the provable minimum required number of actions to find an optimal policy is taking too long. For our problems, we will consider an agent *lifetime*, L, or the number of actions we expect the agent to have, in addition to its MDP definition. For example, in a robotic task where actions take many minutes, the agent may only get a few dozen actions. On the other hand, in a domain simulated on a computer, the agent may get millions of actions. We will define *time-constrained* domains to be ones where the lifetime $L < 2NA$. This lifetime is two orders of magnitude less than the lower bound for deterministic domains with $\gamma = 0.99$, which is $\frac{NA}{1-\gamma}$, and even less than the lower bound for stochastic domains. This book seeks to provide algorithms that will provide reasonable solutions in these time-constrained domains.

In continuous domains, there are an infinite number of states, and the constraint on lifetime would be infinite. In order to put a practical bound on the lifetime in these domains, we estimate the number of states needed to represent the optimal policy in the domain. If we knew the Lipschitz constant, K, defining the smoothness of the domain, we could calculate the error in the value function for a given discretization of the state space (Chow and Tsitsiklis, 1991). Since we do not know the value of K for any domain a priori, we find the number of states required to represent an optimal policy empirically. To do so, we discretize the continuous domain and run Q-LEARNING on the discretized version. Then we find the number of states required for Q-LEARNING to learn an ϵ-optimal policy, with $\epsilon = 0.9$. Table 2.1 shows the number of states required for a number of typical continuous RL domains using this method.

For a few common RL domains, Table 2.2 shows the minimum number of actions required to learn an optimal policy and the maximum lifetime that will make them time-constrained. The minimum bound numbers assume $\gamma = 0.99$, $\epsilon = 0.2$, and $\delta = 0.2$.

An agent acting in a time-constrained domain is quite limited because it does not have enough actions to guarantee that it can learn an optimal policy.

Table 2.1. This table shows the number of states required for Q-LEARNING to learn an ϵ-optimal policy, with $\epsilon = 0.9$

Domain	No. States
Mountain Car	10,000
Puddle World	400
Cart-Pole Balancing	160,000

Table 2.2. This table shows the maximum value of L for which these domains would classify as *time-constrained* domains

Domain	States	Actions	State-Actions	Min Bound Deterministic	Min Bound Stochastic	Maximum L
Taxi	500	6	3,000	300,000	1,050,000	6,000
Four Rooms	100	4	400	40,000	140,000	800
Two Rooms	51	4	204	20,400	72,400	408
Fuel World	39,711	8	317,688	31,768,800	111,190,800	635,376
Mountain Car	10,000	3	30,000	300,000	10,500,000	60,000
Puddle World	400	4	1,600	160,000	560,000	3,200
Cart-Pole Balancing	160,000	2	320,000	32,000,000	11,200,000	640,000

In addition, it most likely does not have a long enough lifetime to visit every state-action in the domain and any unvisited state-action may turn out to be arbitrarily rewarding. For an algorithm to learn effectively in these domains, it must make some assumptions about the domain. In this book, we assume that the effects of actions are similar across nearby states, enabling the agent to generalize predictions to unvisited state-actions rather than requiring the agent to visit each one.

2.4 A Specific Problem

Now that we have formally defined the set of domains we are interested in, we will present a specific example of one of these domains: controlling the velocity of an autonomous vehicle (the *Vehicle Velocity Control* task) (Beeson et al., 2008). This task requires an algorithm to address all of the *RL for Robotics Challenges*: 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, shown in Figure 2.5, 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

Fig. 2.5. The autonomous vehicle operated by Austin Robot Technology and The University of Texas at Austin

learn in this domain *in real time*: with no prior data-gathering phase for training a model or pauses for batch computation.

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 (DES-VEL), the current velocity (CURR-VEL), and the current position of the BRAKE and ACCELERATOR pedals. The agent's reward at each step is -10.0 times the error in velocity in m/s. The agent receives new sensor information at 10 Hz, and thus should provide actions at 10 Hz as well. 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. Table 2.3 formally defines the states, actions, and rewards for the domain. We utilize this task for some of our empirical evaluations later in Chapter 5.

Using the methodology from Section 2.3, we calculated an estimate of the number of discrete states required to learn in a simulated version of this domain without any brake or accelerator delays. We found the broadest discretization where Q-LEARNING could learn an ϵ-optimal policy, which was $43,615$ discrete states. As shown in Table 2.3, the number of state-actions is then $218,075$, and thus the maximum lifetime for this domain to be a time-constrained one is $436,150$.

Applying RL to this task requires solving all four of the *RL for Robotics Challenges* presented in Chapter 1. Samples on the car are very expensive, as they

Table 2.3. Properties of the *Vehicle Velocity Control* task. Note that each episode is 100 actions long, as it is 10 seconds of control of the car with actions taken at 10 Hz.

State	DES-VEL, CURR-VEL, BRAKE, ACCELERATOR
Actions	NO-OP, ACC-UP, ACC-DOWN, BRAKE-UP, BRAKE-DOWN
Reward	$-10.0 *$ |DES-VEL - CURR-VEL|
# State-Actions	218, 075
Time-Constrained Lifetime	436, 150 actions, 4, 361 episodes

require human supervision for safety, as well as good road conditions. In addition, each action takes real world time, and the car may break down, overheat, or run out of gas. Therefore, it is very important for the agent to learn in very few samples, addressing Challenge 1. The task has a continuous state space, so the agent must also address Challenge 2. Unlike many simulated RL tasks where actions taken by the agent have instantaneous effects, on this task, there is significant delay before the brake pedal gets to the position requested by the agent. Thus, the agent must learn good policies even with unknown sensor and actuator delays, addressing Challenge 3. Finally, the car requires the agent to take actions continually in real time, addressing Challenge 4. If the agent does not provide new actions to the car at the required rate, its pedals will remain in their current positions, which can be very bad. For example, if the car is approaching a red light with its throttle pushed down, it is necessary for the agent to send a new action to the car immediately.

2.5 Chapter Summary

In this chapter, I have presented background material on Markov Decision Processes and Reinforcement Learning. I presented the two main classes of value function RL methods: model-free and model-based. This chapter included pseudo-code for a representative algorithm from each class as well as a comparison between them. I presented more details on model-based learning approaches, including learning factored models and planning methods. I have formally defined the set of domains that this book is focused on, where sample efficiency is critical. Finally, I presented an example domain from this class and demonstrated how each of the four *RL for Robotics Challenges* is present in this domain. This material will serve as the foundation for the research presented in the rest of this book. In the next chapter, I will begin presenting the TEXPLORE algorithm, which addresses all of these challenges.

3 Real Time Architecture*

This chapter presents a brief summary of the TEXPLORE *algorithm before fully describing and presenting the real time RL architecture. First, I present a typical example of a sequential model-based RL architecture. Then I present details on using Monte Carlo Tree Search for planning, including a description of the modified version of the* UCT *algorithm (Kocsis and Szepesvári, 2006) that we use for planning. In Section 3.2, I present the parallel architecture for real time action, which puts model learning, planning, and acting on three parallel threads, such that actions can be taken as fast as required without being constrained by how long model updates or planning take. Finally, I summarize the chapter in Section 3.3.*

In this book, I introduce TEXPLORE, a sample-efficient model-based real time RL algorithm. When learning on robots, agents typically have 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 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 the final policy. 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 Chapter 4. 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, it is desirable for the agent to be capable of learning on-line in-situ for the lifetime of the robot, adapting to new states and situations without pauses for computation. TEXPLORE combines a multi-threaded architecture with Monte

* This chapter contains material from two publications: (Hester et al., 2012; Hester and Stone, 2012b).

T. Hester, *TEXPLORE: Temporal Difference Reinforcement Learning for Robots and Time-Constrained Domains*, Studies in Computational Intelligence 503,
DOI: 10.1007/978-3-319-01168-4_3, © Springer International Publishing Switzerland 2013

Algorithm 3.1. Sequential Model-Based Architecture

```
1: Input: S, A                          ▷ S: state space, A: action space
2: Initialize M to empty model
3: Initialize policy π randomly
4: Initialize s to a starting state in the MDP
5: loop
6:     Choose a ← π(s)
7:     Take action a, observe r, s′
8:     M ⇒ UPDATE-MODEL(⟨s, a, s′, r⟩)    ▷ Update model M with experience
9:     π ← PLAN-POLICY(M)               ▷ Exact planning on updated model
10:    s ← s′
11: end loop
```

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.

In this chapter, I introduce TEXPLORE's parallel architecture, enabling it to return actions in real time, addressing Challenge 4 of the *RL for Robotics Challenges*. Most current model-based RL methods use a sequential architecture such as the one shown in Figure 2.3 in Chapter 2. Pseudo-code for the sequential architecture is shown in Algorithm 3.1. In this sequential architecture, the agent receives a new state and reward; updates its model with the new transition $\langle s, a, s', r \rangle$ (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 (Sutton and Barto, 1998) or prioritized sweeping (Moore and Atkeson, 1993)); 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 approach 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) and 2) developing a novel parallel architecture (presented in Section 3.2). I later evaluate this planning method and parallel architecture in comparison with other approaches in Section 5.4.

3.1 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 or prioritized sweeping. This section describes TEXPLORE's use of UCT for approximate planning as well as the modifications we have made to the algorithm. The standard UCT algorithm was presented in Section 2.2.4, but here 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. While prioritized sweeping (Moore and Atkeson, 1993) improves upon the efficiency of value iteration by propagating value backups backwards through the state space, it still iterates over much of the state space rather than focusing computation on the states the agent is likely to visit soon.

The particular MCTS method that TEXPLORE uses is a variant of UCT (Kocsis and Szepesvári, 2006), which was presented in Algorithm 2.3 in Chapter 2. Our variation of UCT, called UCT(λ), is shown in Algorithm 3.2 and 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 2.3, 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 Chapter 4 and shown in Algorithm 4.1). 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 3.1, which performs exact planning, with PLAN-POLICY from Algorithm 2.3, which performs approximate planning, the sequential architecture could be made faster. TEXPLORE's real time architecture, which is presented later in Algorithm 3.4, 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 expect to ever visit the same real-valued state twice, nor can it 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.

Algorithm 3.2. 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 ∈ S, a ∈ A
3:     Initialize c(s, a) with ones for all s ∈ S, a ∈ A          ▷ To avoid divide-by-zero
4:     Initialize c(s) with zeros for all s ∈ S                    ▷ Visit Counts
5: end procedure

6: procedure PLAN-POLICY(M, s)        ▷ Approx. 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 ∈ S_disc do                         ▷ For all discretized states
14:        if c(s_disc) > resetCount · |A| then
15:            c(s_disc) ← resetCount · |A|                     ▷ resetCount per action
16:        end if
17:        for all a ∈ A do
18:            if c(s_disc, a) > resetCount then
19:                c(s_disc, a) ← 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 ← DISCRETIZE(s, nBins, minVals, maxVals)          ▷ Discretize state s
29:    a ← argmax_a' ( Q(s_disc, a') + 2 · (r_max − r_min)/(1 − γ) · sqrt( log c(s_disc) / c(s_disc, a') ) )   ▷ Ties broken randomly
30:    (s', r) ← M⇒QUERY-MODEL(s, a)                             ▷ Algorithm 4.1
31:    sampleReturn ← r + γ UCT-SEARCH(M, s', d + 1)    ▷ Continue rollout from state s'
32:    c(s_disc) ← c(s_disc) + 1                                  ▷ Update counts
33:    c(s_disc, a) ← c(s_disc, a) + 1
34:    Q(s_disc, a') ← α · sampleReturn + (1 − α) · Q(s_disc, a')
35:    return λ · sampleReturn + (1 − λ) · max_a' Q(s_disc, a')        ▷ Use λ-returns
36: end procedure
```

Then the algorithm updates the value and visit counts for the discretized state on Lines 32 to 34.

3.1.1 Domains with Delay

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 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

Algorithm 3.3. UCT(λ) with delays

```
1: procedure SEARCH(M, s, history, d)              ▷ Rollout from state s with history
2:     if TERMINAL or d = maxDepth then
3:         return 0
4:     end if
5:     s_disc ← DISCRETIZE(s, nBins, minVals, maxVals)
6:     a ← argmax_a' ( Q(s_disc, history, a') + 2 · rmax/(1−γ) · sqrt( log c(s_disc, history) / c(s_disc, history, a') ) )
7:     augState ← ⟨s, history⟩
8:     (s', r) ← M ⇒ QUERY-MODEL(augState, a)
9:     PUSH(history, a)                                 ▷ Keep last k actions
10:    if LENGTH(history) > k then
11:        POP(history)
12:    end if
13:    sampleReturn ← r + γ SEARCH(M, s', history, d + 1)
14:    c(s_disc, history) ← c(s_disc, history) + 1        ▷ Update counts
15:    c(s_disc, history, a) ← c(s_disc, history, a) + 1
16:    Q(s_disc, history, a') ← α · sampleReturn + (1 − α) · Q(s_disc, history, a')
17:    return λ · sampleReturn + (1 − λ) · max_a' Q(s_disc, history, a')
18: end procedure
```

action given to it when the algorithm selects the next action. This delay 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 takes advantage of the k-Markov property for planning by slightly modifying UCT(λ). Algorithm 3.3 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 increase 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 (demonstrated empirically in Section 5.3). Note that

with $k = 0$, the *history* is $\emptyset$ and the action thread and UCT(λ) search methods presented here exactly match the ones presented in Algorithms 3.4 and 2.3, 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.

Not only does this k-Markov approach to handling delay work well with UCT planning, it also works with our model learning approach. Later, in Section 4.1.2, we will describe how this approach applies to model learning.

3.2 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. 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 3.1. 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 3.4. 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 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 3.1.

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, 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

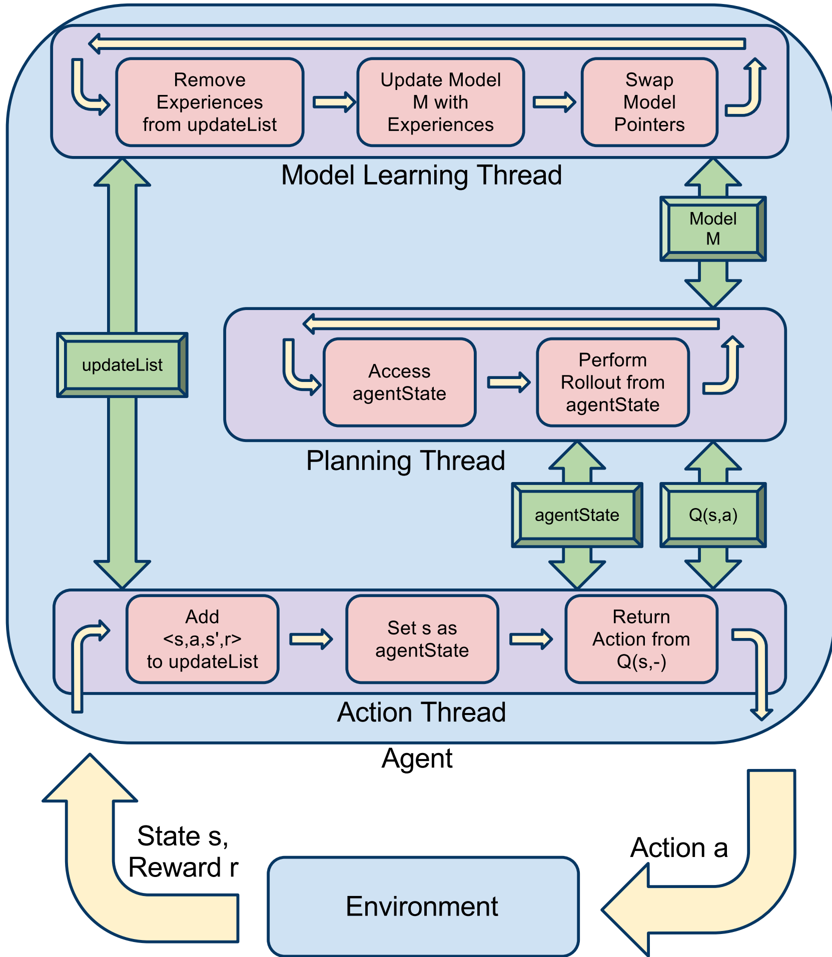


Fig. 3.1. A diagram of the parallel real time architecture for model-based RL

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.

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

Algorithm 3.4. 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 4.1)
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 2.3
24: **end loop**
25: **end procedure**

26: **procedure** ACTIONTHREAD ▷ Action Selection Thread
27: **loop**
28: $s_{disc} \leftarrow$ DISCRETIZE($s, nBins, minVals, maxVals$) ▷ Discretize 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**

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

Table 3.1. This table shows all the variables that are protected under mutex locks in the real time architecture, along with their purpose and which threads use them

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

model (Deisenroth and Rasmussen, 2011), or the random forest model used by TEXPLORE, which is described in Chapter 4. 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.

3.3 Chapter Summary

In this chapter, I have presented TEXPLORE's parallel real time architecture for model-based RL. This architecture parallelizes model learning, planning, and acting into three separate threads so that action selection can happen in real time, even if model learning or planning take more computation time. The architecture utilizes a sample-based anytime planning method, which improves as it is given time for more planning rollouts. In the next chapter, I will present the model learning method that is used within this architecture in the TEXPLORE algorithm.

4 The TEXPLORE Algorithm*

This chapter presents the TEXPLORE *algorithm, which uses the architecture presented in the previous chapter. First,* TEXPLORE*'s model learning approach is presented in Section 4.1.* TEXPLORE *utilizes a factored model, making a separate prediction about the next value of each state feature and reward. It builds decision trees to model each feature, enabling it to generalize the effects of actions across states. In Section 4.1.1, I describe how* TEXPLORE*'s decision tree models can be extended to regression tree models to model domains with continuous state. Next, in Section 4.1.2 I describe how* TEXPLORE*'s trees can model domains with sensor or actuator delays by providing them with the agent's previous k actions as additional inputs. I describe how to modify* TEXPLORE*'s model for domains with dependent feature transitions in Section 4.1.3.*

Section 4.2 presents TEXPLORE*'s approach to performing limited, targeted exploration. In* TEXPLORE*'s approach, the agent acts greedily with respect to a random forest model, which aggregates multiple decision tree models together. This approach enables the agent to balance each of its hypotheses of the true dynamics of the domain in a natural way. Then, I describe how the various components presented in this chapter along with the architecture from Chapter 3 can be combined into the full* TEXPLORE *algorithm. Finally, I summarize the chapter in Section 6.5.*

While the parallel architecture presented in the previous chapter enables TEXPLORE to operate in real time, the algorithm must learn the task with high sample efficiency. This objective requires the agent to learn a model of the transition and reward functions in the domain very quickly, and explore intelligently to improve that model. I present TEXPLORE's model learning in the next section, and its exploration in Section 4.2.

4.1 Model Learning

To learn a high quality behavior in few samples, TEXPLORE must learn an accurate model of the domain quickly. 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

* This chapter contains material from two publications: (Hester and Stone, 2010, 2012b).

T. Hester, *TEXPLORE: Temporal Difference Reinforcement Learning for Robots and Time-Constrained Domains*, Studies in Computational Intelligence 503,
DOI: 10.1007/978-3-319-01168-4_4, © Springer International Publishing Switzerland 2013

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 4.1. 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, shown later in Algorithm 4.3.

Algorithm 4.1 shows TEXPLORE's model learning algorithm. It starts by calculating the relative change in the state (s^{rel}) on Line 12, then it updates the model for each feature with the new transition on Line 14 and updates the reward model on Line 16. Like DBN-based algorithms, TEXPLORE assumes that each of the state variables transitions independently (however, I present an extension for dependent feature transitions in Section 4.1.3). 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 23 and adds this vector, s^{rel}, to s to get a prediction of s'. The agent then samples a prediction of reward (Line 27) and these sampled predictions are returned for planning with MCTS.

We tested the applicability of several different supervised learning methods to the task of learning an MDP model in previous work (Hester and Stone, 2009a). 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 refine their predictions to smaller regions as they learn. Starting with a broad representation and refining it over time has been shown to be effective in other areas such as value function approximation (Munos and Moore, 2002). 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. Another advantage of using decision trees is that they can learn *context-specific feature independence*, meaning that they can learn that a prediction is independent of some features given that other features have specific values (Boutilier et al., 2000).

Algorithm 4.1. MODEL

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

 7: procedure UPDATE-MODEL(list)                 ▷ Update model with list of experiences
 8:     for all ⟨s, a, s′, r⟩ ∈ list do
 9:         s^rel ← s′ − s                                 ▷ Calculate relative effect
10:         for all s_i^rel ∈ s^rel do
11:             featModel_i⇒UPDATE(⟨s, a⟩ , s_i^rel)      ▷ Train a model for each feature
12:         end for
13:         rewardModel⇒UPDATE(⟨s, a⟩ , r)          ▷ Train a model to predict reward
14:     end for
15: end procedure

16: procedure QUERY-MODEL(s, a)                       ▷ Get prediction of ⟨s′, r⟩ for s, a
17:     for i = 1 → LENGTH(s) do
18:         s_i^rel ← featModel_i⇒QUERY(⟨s, a⟩)    ▷ Sample a prediction for feature i
19:     end for
20:     s′ ← s + ⟨s_1^rel, ..., s_n^rel⟩                        ▷ Get absolute next state
21:     r ← rewardModel⇒QUERY(⟨s, a⟩)                   ▷ Sample r from distribution
22:     return ⟨s′, r⟩                          ▷ Return sampled next state and reward
23: end procedure
```

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 the 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** X $=$ 3 and **if** X $>$ 3 are allowed. The C4.5 algorithm chooses the split at each node of the tree based on information gain. While the C4.5 algorithm builds entire trees in batch updates, 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. If the new experience would not change the tree, then the tree remains unchanged.

The decision trees are the supervised learner that is called on Lines 14, 16, 23, and 27 of Algorithm 4.1 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 23 and for reward on Line 27.

Figure 4.1 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 updated 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.

4.1.1 Models of Continuous Domains

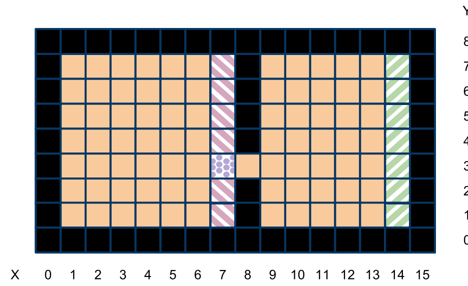
While decision trees work well for discrete domains, TEXPLORE needs to be capable of modeling continuous domains to meet Challenge 2 of the *RL for Robotics Challenges*. Discretizing the domain is one option, but important information is lost in the discretization. Not only is noise added by discretizing 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 smaller 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 approximating the transition and reward model of a domain.

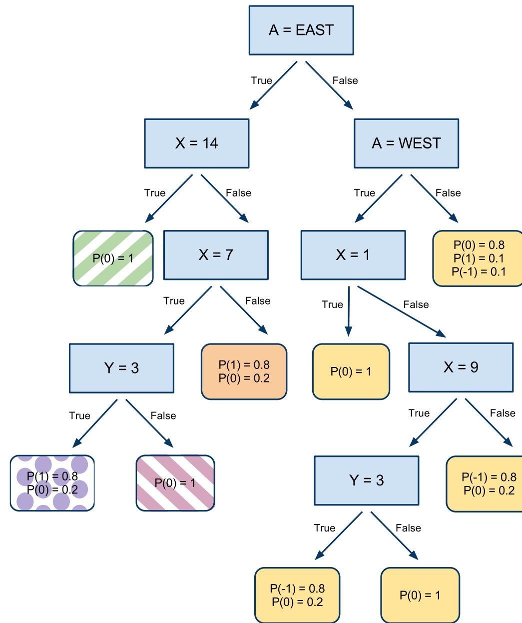
Figure 4.2 shows an example of how the regression trees can result in simpler models that are faster to build and make more accurate predictions than discrete decision trees. Figure 4.2(a) shows the predictions of the discrete tree approximating the underlying function. The model requires examples of 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 more closely and makes predictions for unseen parts of the space by extrapolating the linear function from nearby regions.

4.1.2 Domains with Delays

As described in Section 3.1.1, one of TEXPLORE's objectives is to address Challenge 3 of working well in domains with sensor or actuator delays. Addressing

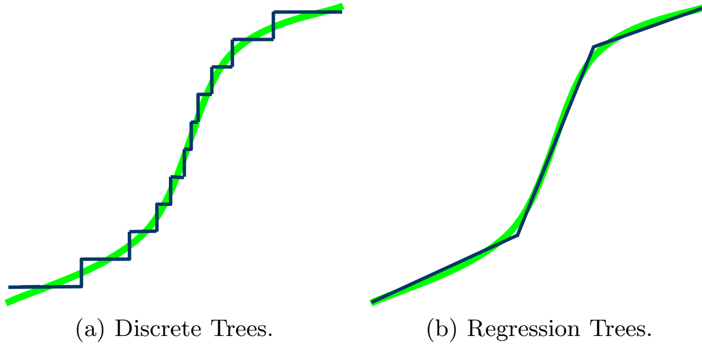


(a) Two room gridworld domain.



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

Fig. 4.1. 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. 4.2. An example of a function (the thick green line) estimated by 4.2(a): discrete trees and 4.2(b): regression trees. Note that the regression tree is able to fit the function better than the discrete tree.

this challenge is important to make TEXPLORE applicable to many physical systems such as robots. Similar to the approach taken in Section 3.1.1 to perform UCT planning with delays, we will take a k-Markov approach to handling delay in the model learning as well.

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 4.2), 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 can ignore the extra inputs and still build an accurate model. This benefit is demonstrated empirically in Section 5.3. 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).

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

Algorithm 4.2. Action Thread with Delays

```
 1: procedure ActionThread                                  ▷ Action Selection Thread
 2:     history ← ∅
 3:     loop
 4:         s_disc ← DISCRETIZE(s, nBins, minVals, maxVals)
 5:         Choose a ← argmax_a Q(s_disc, history, a)  ▷ Values of state-history-actions
 6:         Take action a, Observe r, s'
 7:         augState ← ⟨s, history⟩                    ▷ Augment state with history
 8:         updateList ← updateList ∪ ⟨augState, a, s', r⟩
 9:         PUSH(history, a)                                   ▷ Keep last k actions
10:         if LENGTH(history) > k then
11:             POP(history)
12:         end if
13:         s ← s'
14:         agentState ← s              ▷ Set agent's state for planning rollouts
15:     end loop
16: end procedure
```

relevant for the task at hand. In addition, as shown in Section 3.1.1, 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. Thus, using k-action histories for delays is one example of how the various components of TEXPLORE are synergistic.

4.1.3 Dependent Feature Transitions

Thus far, TEXPLORE's model learning approach has assumed that it can predict each state feature independently of the others, based on the agent's previous state and action. This assumption simplifies the model learning problem and reduces the number of experiences required for the agent to learn an accurate model and is a common assumption made by all factored RL methods (Guestrin et al., 2002; Degris et al., 2006; Strehl et al., 2007; Chakraborty and Stone, 2011). While this independence assumption proves useful in speeding up learning, it may not always be valid. In some domains, subsets of features may transition dependently with each other and thus cannot be accurately predicted independently. In these cases, the models learned by methods that assume feature independence will be wrong and consequently could lead to low-value policies.

As described in Section 2.2.3, typically, the DBN learned by a factored model predicts each feature independently, as shown in Figure 2.4(b). To model dependent transitions, we add synchronic arcs between the predicted features, as illustrated in Figure 4.3. Now the value of each feature is dependent on its parents in the previous time step as well as the predicted value of some of the other features. Since the features are correlated in their changes, there is no order on them, and the synchronic arcs can be placed in any order. We arbitrarily order the synchronic arcs in the same order the features are given from the domain. Later, in Section 5.5, we empirically demonstrate that the ordering does not

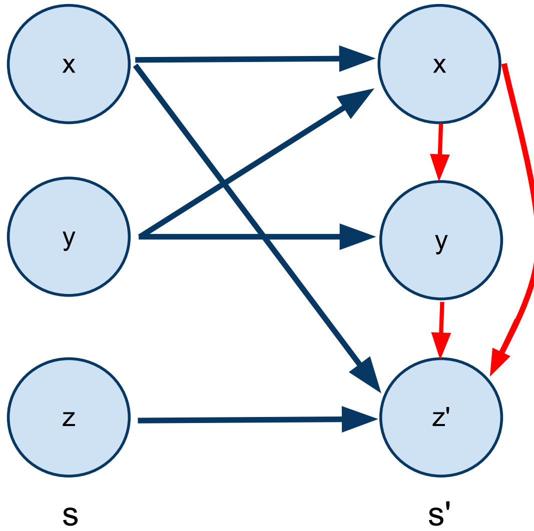


Fig. 4.3. DBN model with added synchronic arcs

matter. This solution applies not only to TEXPLORE, but also to other methods for factored domains that make this independence assumption (Guestrin et al., 2002; Degris et al., 2006; Strehl et al., 2007; Chakraborty and Stone, 2011).

Adding these synchronic arcs to the structure of the DBN model changes what inputs each model is given. For example, TEXPLORE's decision tree models are typically given the features of the previous state and the action as inputs and can make splits and decisions based on any of these features. In this new model, they are also given the values/predictions for the lower-ordered features of the current state, and can split on these features as well. However, if these features are not required, the tree model does not have to split on them. So while TEXPLORE's models are being given additional inputs, they may not be used and thus the new model does not necessarily require the agent to explore more or use more samples than it would otherwise (shown empirically in Section 5.5).

For factored methods using DBN models, applying the DBN with synchronic arcs is a simple change. When training a model to predict feature s'_i, the original DBN model trains on the following input (the previous state and k actions):

$$\langle s_1, ..., s_n, a_t, ..., a_{t-k} \rangle . \tag{5}$$

For clarity, we will assume $k = 0$ in the following equations and only consider a_t, but these models all extend to the case where $k > 0$ and instead of a dependence on a_t they are dependent on $a_t, ..., a_{t-k}$. The input for the DBN model with synchronic arcs additionally includes the values of current features with index $< i$:

$$\left\langle s_1, ..., s_n, a_t, s'_1, ..., s'_{i-1} \right\rangle . \tag{6}$$

When these models are making predictions, the values of the current features, s'_1 to s'_{i-1}, are replaced by their predicted values. When using the DBN with added synchronic arcs, the probability of each feature s'_i is dependent on the predicted values of the lower indexed features in addition to the previous state and action:

$$P(s'_i|s, a_t, s'_1, ..., s'_{i-1}). \tag{7}$$

The probability of the vector s' being $\langle s'_1, ..., s'_n \rangle$ given $\langle s, a_t \rangle$ is:

$$P(s'_1|s, a_t) \cdot P(s'_2|s, a_t, s'_1) \cdot ... \cdot P(s'_n|s, a_t, s'_1, ..., s'_{n-1}). \tag{8}$$

The prediction for the original DBN model without synchronic arcs would be:

$$P(s'_1|s, a_t) \cdot P(s'_2|s, a_t) \cdot ... \cdot P(s'_n|s, a_t). \tag{9}$$

Thus when using the DBN with added synchronic arcs, the prediction of each feature is dependent on the predicted values of all the lower indexed features, while the basic DBN model predicts each feature completely independently.

4.2 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 can exploit its knowledge within this limited time frame. 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 toward 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). Another advantage of random forests is that their convergence rate is only affected by the number of relevant input features and not on the number of extraneous noise features (Biau, 2012). When providing the model the previous k actions to handle delayed domains, or extra features to handle dependent transitions, this property enables the model to ignore any unnecessary features without a drop in performance.

Algorithm 4.3 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

Algorithm 4.3. MODEL: Random Forest

```
1: procedure INIT(m)                                  ▷ Init forest of m trees
2:     for i = 1 → m do
3:         tree_i ⇒ 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 → m do                ▷ For m trees in the random forest
8:         if RAND() ≤ w then           ▷ Update each tree with prob. w
9:             tree_i ⇒ 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 ← tree_i ⇒ QUERY(in)                 ▷ Get prediction from tree i
16:    return x                                      ▷ Return prediction
17: end procedure
```

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 14, 16, 23, and 27 of Algorithm 4.1. 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 for a rollout, it only needs to return the prediction of a single randomly selected tree in the forest, which saves some computation.

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 subscripts on Q_i, R_i, and P_i representing that they are from model i:

$$Q(s,a) = \max_i Q_i(s,a) \tag{10}$$

$$Q_i(s,a) = R_i(s,a) + \gamma \sum_{s'} P_i(s'|s,a) \max_{a'} Q(s',a'), \tag{11}$$

The policy of the agent is then:

$$\pi(s) = \text{argmax}_a Q(s,a). \tag{12}$$

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:

$$Q_i(s,a) = R_i(s,a) + \gamma \sum_{s'} P_i(s'|s,a) \max_{a'} Q_i(s',a'). \quad (13)$$

Note that this equation 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:

$$Gain_i(s,a) = \begin{cases} E[Q(s,a_2)] - Q_i(s,a), & \text{if } a = a_1 \text{ and } Q_i(s,a) < E[Q(s,a_2)], \\ Q_i(s,a) - E[Q(s,a_1)], & \text{if } a \neq a_1 \text{ and } Q_i(s,a) > E[Q(s,a_1)], \\ 0, & \text{otherwise.} \end{cases} \quad (14)$$

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:

$$Q(s,a) = \frac{1}{m} \sum_{i=1}^{m} Q_i(s,a) + Gain_i(s,a). \quad (15)$$

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 distinguishing characteristic of our approach is that it is *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:

$$Q(s,a) = \frac{1}{m}\sum_{i=1}^{m} R_i(s,a) + \gamma \frac{1}{m}\sum_{i=1}^{m}\sum_{s'} P_i(s'|s,a) \max_{a'} Q(s',a'). \tag{16}$$

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 4.4 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

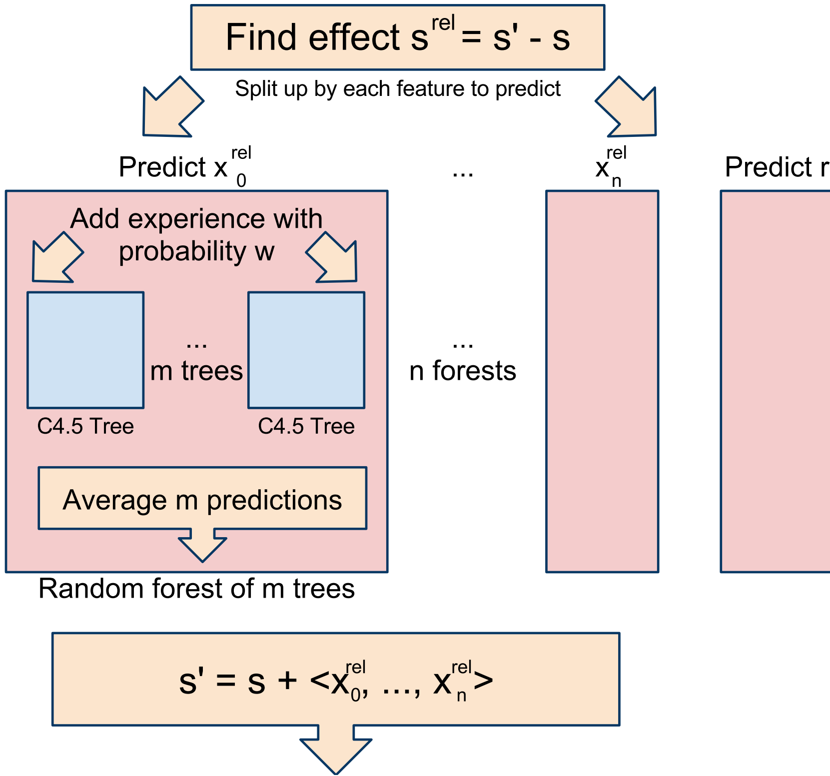


Fig. 4.4. *Model Learning.* This diagram shows how TEXPLORE learns a model of the domain. The agent calculates the difference between s' and s as the transition effect s^{rel}. Then it splits up the state vector and learns a random forest to predict each state feature. Each random forest is made up of 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.

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 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 time-constrained domains we are focused 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, when learning on robots, 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.

We have thus discussed the advantages of TEXPLORE over other methods such as BOSS and MBBE. It is useful to also note that 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. Smart and Kaelbling (2002) argue that for RL to be effective on robots, the agent *must* be given prior information about the task. They suggest providing the agent with experiences from a human or human-programmed controller running the robot. Here, we are more conservative, only providing the agent with a few example transitions from the domain. 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. In many domains, we do not provide any seed transitions and let TEXPLORE learn from scratch.

4.3 The Complete TEXPLORE Algorithm

Having presented each of the components of TEXPLORE, we now combine them together into one complete algorithm. Since each component of TEXPLORE

was presented separately, for clarity we have placed all of the pseudo-code for TEXPLORE in Appendix A. This appendix contains a comprehensive set of the code for TEXPLORE, incorporating the solutions for handling sensor and actuator delays, dependent feature transitions, and continuous states. TEXPLORE is constituted by the RTMBA architecture shown in Algorithm A.1, which uses the UCT(λ) planning method shown in Algorithm A.2. TEXPLORE learns random forest models of each state feature and reward, as shown in Algorithms A.3 and A.4. 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 use to handle delayed domains. When k is not defined, it is assumed to be 0 (the setting for non-delayed domains). All of the versions of the TEXPLORE algorithm are freely available in our open-source ROS package at: `http://www.ros.org/wiki/rl-texplore-ros-pkg`.

4.4 Chapter Summary

In this chapter, I have presented the TEXPLORE algorithm, including its approaches to both model learning and exploration. First, I presented the model learning approach employed by TEXPLORE. TEXPLORE assumes a factored domain, where the state is represented by a set of state features. TEXPLORE makes predictions about the next value of each feature using decision trees. These trees enable TEXPLORE to generalize the effects of actions across states. This model can be extended for domains with continuous state, sensor or actuator delays, or dependent feature transitions. For continuous domains, the decision trees can be modified to be regression trees, which have regression models in each leaf of the tree to make predictions about continuous values. For domains with delay, TEXPLORE's model is given the agent's previous k actions, so that the tree can predict based on the action actually affecting the observation on the current time step. Finally, for domains with dependent features, the decision tree predicting each feature can be given the other predicted features as input, so it can predict each feature dependently on the lower-ordered ones.

After presenting TEXPLORE's model learning, I presented TEXPLORE's approach to exploration in Section 4.2. TEXPLORE acts greedily with respect to a model that aggregates multiple predictions about the true dynamics of the domain in the form of a random forest. This approach enables TEXPLORE to naturally balance the trade-off between exploring the state-actions that are predicted to be good by the optimistic models while avoiding potentially costly state-actions as predicted by the more pessimistic models. In the next chapter, we will empirically evaluate the TEXPLORE algorithm and the choices we made for each component of it in comparison to other state of the art approaches.

5 Empirical Evaluation*

In this chapter, I empirically evaluate TEXPLORE *in comparison with other state-of-the-art methods. First, I analyze* TEXPLORE*'s sample efficiency and exploration on two tasks. Then, I examine how its models perform in three different continuous domains. In the third section, I look at how* TEXPLORE*'s k-Markov approach to handling delays performs in two domains. In Section 5.4, I evaluate* RTMBA *in comparison with other approaches to act in real time and examine the trade-off between computation time and sample efficiency. Next, I look at* TEXPLORE*'s solution for domains with dependent feature transitions and what its impact is in domains where dependent feature transitions are not an issue. In Section 5.6, I demonstrate* TEXPLORE *learning to control the velocity of the physical autonomous vehicle in real time, while running on-board the robot. Finally, I summarize the chapter in Section 5.7.*

This chapter presents experiments that examine TEXPLORE's solution to each of the *RL for Robotics Challenges* 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 task that exemplifies that challenge. In addition, each component is also evaluated on a simulation of the *Vehicle Velocity Control* task presented earlier in Section 2.4. All significance results are calculated using a Student's t-test. All of the domains used in this chapter are listed in Appendix B and are freely available in our open-source ROS package: `http://www.ros.org/wiki/rl_env`.

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. Section 5.5 analyzes our approach to handling domains where state features transition dependently. Finally, Section 5.6 shows the complete algorithm learning to control the physical autonomous vehicle, rather than the simulation.

Each component of the algorithm is examined on a simulation of the robot task presented in Section 2.4: controlling the velocity of an autonomous vehicle (Beeson et al., 2008). The properties of this task, including the time-constrained lifetime as defined in Section 2.3, are listed in Table 2.3. This task requires an algorithm to address all of the *RL for Robotics Challenges*. The task is to learn

* This chapter contains material from three publications: (Hester and Stone, 2010; Hester et al., 2012; Hester and Stone, 2012b).

T. Hester, *TEXPLORE: Temporal Difference Reinforcement Learning for Robots and Time-Constrained Domains*, Studies in Computational Intelligence 503,
DOI: 10.1007/978-3-319-01168-4_5, © Springer International Publishing Switzerland 2013

to drive the vehicle at a desired velocity by controlling the pedals. The agent's state is made up of the pedal positions and the desired and current velocity of the car, and it has actions to move the brake or throttle up or down. 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.

Since the autonomous vehicle was already running ROS (Quigley et al., 2009) 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. The actuators of the car remain in the same positions until it receives an actuator command from the RL agent. The ROS messages we defined for communicating with an RL algorithm are publicly available in our ROS package: `http://www.ros.org/wiki/rl_msgs`.

Instead of reinforcement learning, another approach to this problem would be to use classical control methods such as Proportional-Integral-Derivative (PID) control. However, PID controllers are notoriously difficult to tune and existing tuning methods such as the Ziegler-Nichols method only work for devices with a single actuator (Ziegler and Nichols, 1942). This problem has many properties that make it difficult for PID control (Sung and Lee, 1996; Atherton and Majhi, 1999), as it is non-symmetric and non-linear, and the vehicle acts differently at different desired velocities. In addition, PID control does not handle the brake delay very well. If the controller has a non-zero integrative term to account for possible control errors, it also causes the controller to brake excessively and overshoot target velocities when decelerating.

As presented in Table 2.3, the time-constrained lifetime for this task is $436,150$ actions, which with exactly 100 actions per episode equals $4,361$ episodes. Note that all of the experiments on the simulation of this domain were run for $1,000$ episodes, well within the time-constrained lifetime of $4,361$ episodes.

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 parameterization.

Bayesian DP (Strens, 2000) will be described in further detail in Section 7.1.3. It samples a single model from the distribution over models, 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 4.3 with the one shown in Algorithm 5.1. 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 5.1. Bayesian DP-like Approach

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

Best of Sampled Set (BOSS) (Asmuth et al., 2009) will also be described in detail in Section 7.1.3. It samples m models from the distribution and creates an augmented model with mA actions—a set of actions for each sampled model. BOSS then plans over this augmented model, enabling it to use the most optimistic model in each part of the state space. By replacing QUERY in Algorithm 4.3 with Algorithm 5.2, 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 5.2. BOSS-like Approach

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

In addition to the Bayesian-inspired approaches, we compare with the approach taken in the PILCO algorithm (Deisenroth and Rasmussen, 2011), 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 4.1 is modified by a value proportional to the average variance:

$$r = r + v\frac{1}{n+1}[\sigma^2 R(s,a) + \sum_{i=1}^{n} \sigma^2 P(s_i^{rel}|s,a)]. \tag{17}$$

Here, v is a coefficient that 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, v, 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 $v < 0$, the agent will avoid states that the model is uncertain about; setting $v > 0$ will result in the agent being driven to explore these uncertain states. If $v = 0$, the agent will act greedily with respect to its model. Changing the parameter v affects how aggressive the agent is in trying to resolve 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 $v = 1$ (Eq. 17)
5. VARIANCE-BONUS Approach $v = 10$ (Eq. 17)
6. Bayesian DP-like Approach (Alg. 5.1)
7. BOSS-like Approach (Alg. 5.2).

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 and the number of trees in each forest, m, set to 5.

Figure 6.10 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

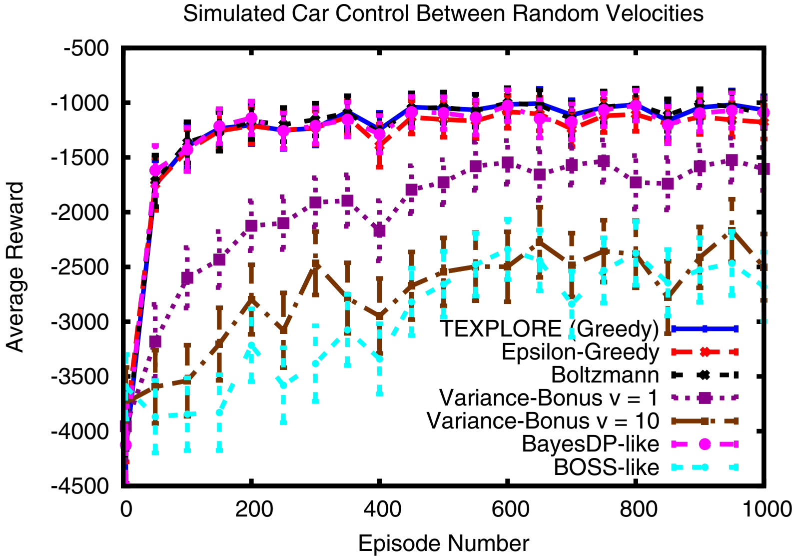


Fig. 5.1. Average reward over 1000 episodes on *Simulated Vehicle Velocity Control*. Results are averaged over 50 trials using a 5 episode sliding window and plotted with 95% confidence intervals. Note that TEXPLORE's exploration accrues the most reward.

(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 model, we compare with methods using different models that are state of the art for exploration, particularly Bayesian 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 5.2. Here, TEXPLORE accrues significantly more rewards than all the other methods after episode 24 ($p < 0.01$). In addition, TEXPLORE learns well within the time-constrained lifetime for this domain of $436,150$ steps (or $4,361$ episodes). In fact, the Bayesian

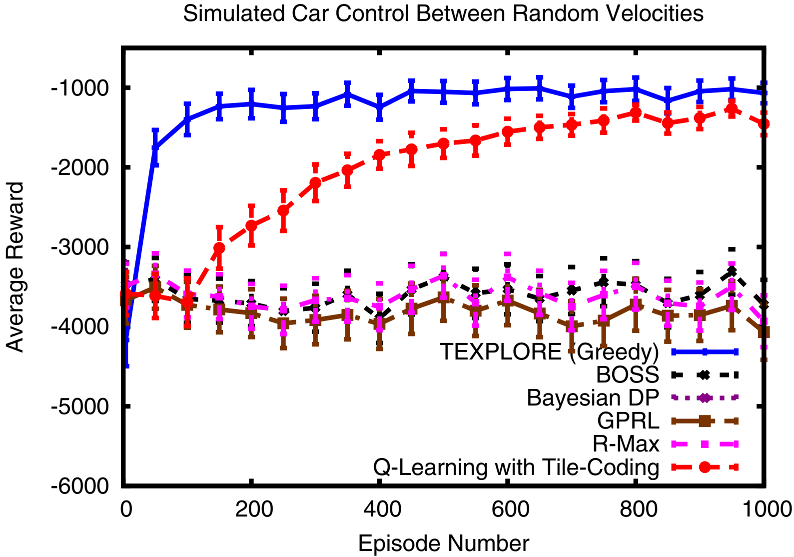


Fig. 5.2. Average reward over 1000 episodes on *Simulated Vehicle Velocity Control*. Results are averaged over 50 trials using a 5 episode sliding window and plotted with 95% confidence intervals. Note that TEXPLORE accrues the most reward.

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 5.3. 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 that 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 terminates with reward -400. There are fuel stations along the top and bottom row of the domain that 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,

Table 5.1. Properties of the *Fuel World* task

State	ROW, COL, FUEL
Actions	NORTH, EAST, SOUTH, WEST, NORTHEAST, SOUTHEAST,SOUTHWEST, NORTHWEST
Reward	Ranges from -400.0 to $+20.0$
# State-Actions	$317,688$
Time-Constrained Lifetime	$635,376$ actions

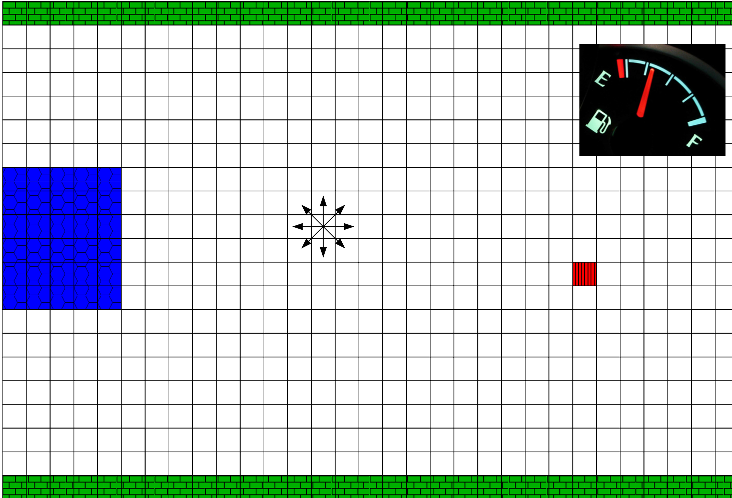


Fig. 5.3. The *Fuel World* domain. Starting states have blue hexagons, fuel stations have green brick patterns, and the goal state is shown in red with vertical lines. The possible actions the agent can take are shown in the middle. Here, the fuel stations are the most interesting states to explore, as they vary in cost, while the center white states are easily predictable.

each with 61 possible energy levels, and 8 possible actions, for a total of $317,688$ state-actions. The agent starts with a random amount of fuel between 14 and 18, which is not enough 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 properties of this domain are shown in Table 5.1.

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

$$R(x) = base - (x \bmod 5)a, \tag{18}$$

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 that 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 5.2.

Table 5.2. Parameters for Equation 18 for the two versions of the *Fuel World* task

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

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 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 $v = 10$ (Eq. 17)
5. Bayesian DP-like Approach (Alg. 5.1)
6. BOSS-like Approach (Alg. 5.2)
7. Bayesian DP with sparse Dirichlet prior (Strens, 2000)
8. BOSS with sparse Dirichlet prior (Asmuth et al., 2009).

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 domain is discrete (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 from the domain. They are given two experiences from the goal state, two transitions from each row of fuel 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 5.4 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. TEXPLORE learns the task within the time-constrained lifetime of $635,376$ steps.

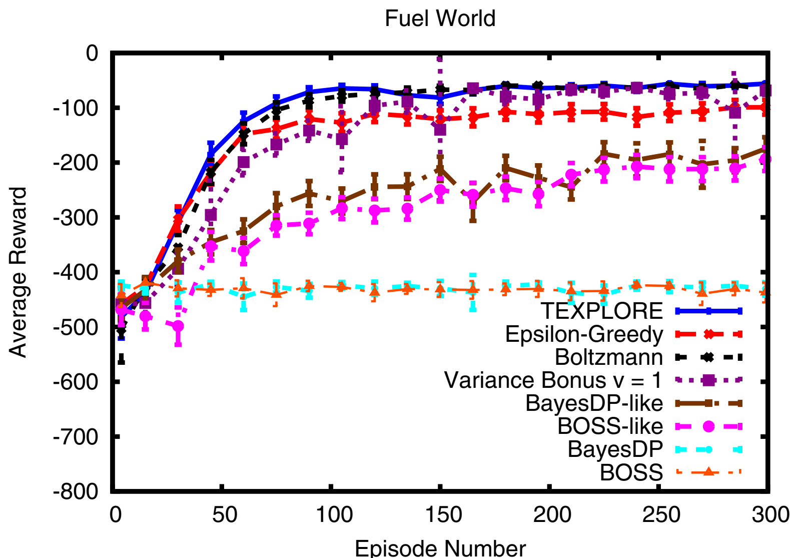
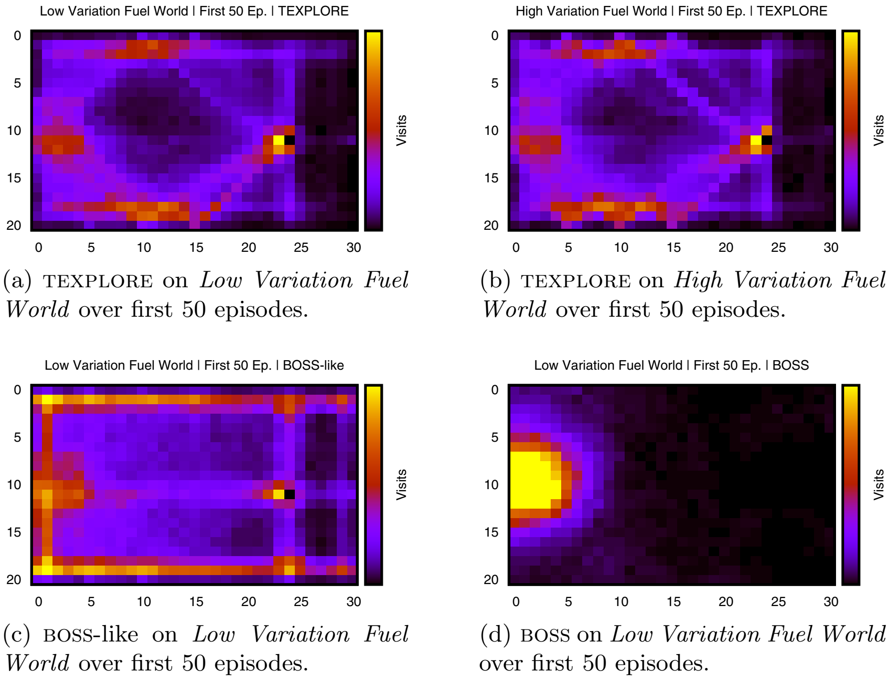


Fig. 5.4. Average reward over the first 300 episodes in *Low Variation Fuel World*. Results are averaged over 50 trials using a 5 episode sliding window and plotted with 95% confidence intervals. TEXPLORE learns the policy faster than the other algorithms.

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, we kept track of every state the agents visited while learning in a deterministic version of the *Fuel World* domain. We used a deterministic version so that it is clear that the agent is *exploring* to visit particular states, rather than being driven there by stochasticity. Figure 5.5 shows heat maps of which states the agents visited during their first 50 episodes in the domain and Figure 5.6 shows their visits during the final 50 episodes. 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 5.5(a) and 5.5(b) show the heat maps over the first 50 episodes for TEXPLORE in the *Low* and *High Variation Fuel World* domains. 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 5.5(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 5.5(b), the higher exploration costs cause it to quickly settle on the stations in Column 5, 10, or 15. The effects of the agent's different exploration in these two domains can be seen in its final policy in each domain, shown in Figures 5.6(a) and 5.6(b).



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

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

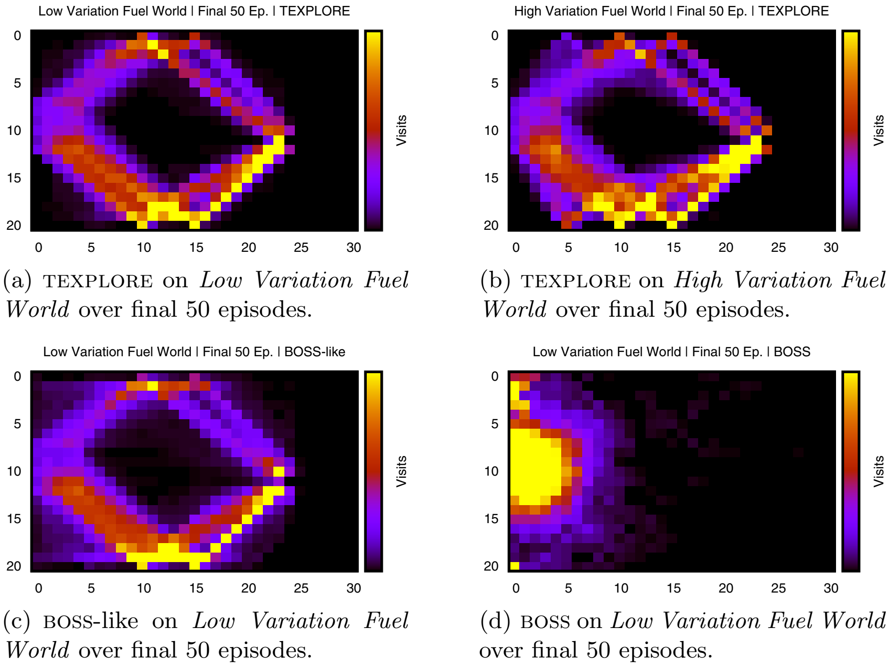
(c) BOSS-like on *Low Variation Fuel World* over first 50 episodes.

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

Fig. 5.5. Heat maps displaying the average number of visits to each state over the first 50 episodes in the deterministic *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. 5.5(b)) than in the *Low Variation* domain (Fig. 5.5(a)). In either case, it explores less thoroughly than the BOSS-like algorithm (Fig. 5.5(c)) or the complete BOSS algorithm (Fig. 5.5(d)).

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 than the agents in the *High Variation* domain. 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, with more policies going through the fuel stations in Column 5. 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,



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

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

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

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

Fig. 5.6. Heat maps displaying the average number of visits to each state over the final 50 episodes in the deterministic *Fuel World* domain, averaged over 50 trials and all fuel levels. These figures show which states the agents visited while following their final learned policies. In the *High Variation* domain (Fig. 5.6(b)), TEXPLORE explores less and converges to a larger number of final policies across the 30 trials than it does in the *Low Variation* version (Fig. 5.6(a)). In contrast, the BOSS-LIKE method in the *Low Variation* domain (Fig. 5.6(c)) explores more and settles on fewer policies, while the actual BOSS algorithm (Fig. 5.6(d)) has only learned to go to the top fuel stations to survive by the end of the 300 episodes.

Figure 5.5(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. Although the BOSS-like agent learns similar final policies to TEXPLORE (shown in Figure 5.6(c)), the extra costs it accrues while exploring result in it receiving less cumulative rewards.

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 50 episodes, as shown in Figure 5.5(d). By the end of the 300 episodes, the BOSS algorithm has discovered how to reach the top fuel stations to survive, but not how to reach the goal (shown in Figure 5.6(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 the other exploration methods like BOSS 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 continuous state model learning method, presented in Section 4.1.1, to accurately predict state transitions and rewards on continuous tasks. First, we examine the accuracy of the learned models on the *Simulated Vehicle Velocity Control* task. 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. Then in Section 5.2.2, we examine the performance of the continuous models when used inside the full algorithm on two continuous RL domains from the literature: *Mountain Car* and *Cart-Pole Balancing*.

5.2.1 Simulated Vehicle Velocity Control

In this section, we measure the accuracy of TEXPLORE's approach to learning models of continuous domains in comparison with six other approaches. Each model is trained on a random sample of experiences from the *Simulated Vehicle Velocity Control* task. Then, the Euclidean distance between the next state the model predicted most likely and the true most likely next state is calculated to measure the accuracy of the models. For reward, the average error between the expected reward predicted by the model and the true expected reward in the simulation is calculated. 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, the tabular model is a typical benchmark approach, and the last two are state-of-the-art approaches for continuous domains.

Figure 5.7 shows the average next state prediction error for each model. The regression tree forest and single regression tree have significantly less error than all the other models in predicting the next state ($p < 0.001$). The single regression tree and the forest are not significantly different. Figure 5.8 shows the average reward prediction error for each model. For this 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

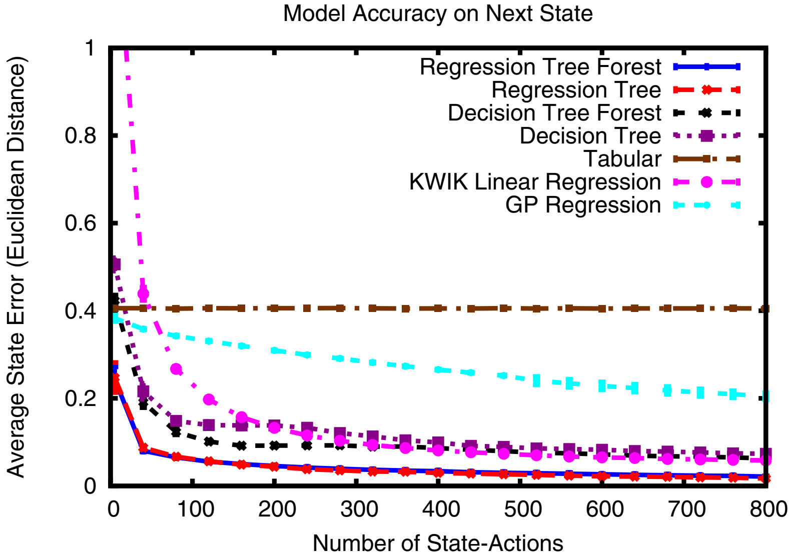


Fig. 5.7. Average error in the prediction of the next state for each model on the *Simulated Vehicle Velocity Control* task, averaged over 50 trials and plotted with 95% confidence intervals. Each model is trained on random experiences from the domain and tested on its ability to predict 10,000 random 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. Note that TEXPLORE's model, a random forest of regression trees, is the most accurate for next state predictions.

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.

5.2.2 Continuous Task Performance

While we showed in the previous section that TEXPLORE's M5 linear regression models learn accurate predictions of next state and reward from random samples of experience, it is important that this model works well within the RL algorithm. Thus, we examine the advantages of using the M5 linear regression model within the TEXPLORE algorithm. We test the algorithm on two benchmark continuous domains from the literature: *Mountain Car* and *Cart-Pole Balancing*. On both tasks, we compare six algorithms:

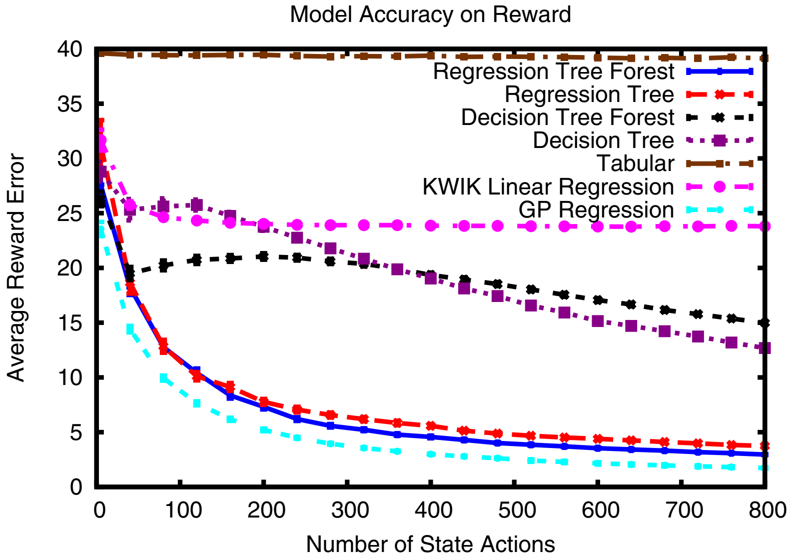


Fig. 5.8. Average error in the prediction of the reward for each model on the *Simulated Vehicle Velocity Control* task, averaged over 50 trials and plotted with 95% confidence intervals. Each model is trained on random experiences from the domain and tested on its ability to predict 10,000 random experiences from the domain. The reward error is the error in expected reward. Note that TEXPLORE's model, a random forest of regression trees, is the second best at reward prediction.

1. Discrete TEXPLORE (using C4.5 trees on a discretized state space)
2. Continuous TEXPLORE (using M5 trees)
3. R-MAX
4. FITTED R-MAX (Jong and Stone, 2007)
5. Tabular Q-LEARNING
6. Q-LEARNING with tile coding.

We chose these methods to compare discrete and continuous versions of TEXPLORE, R-MAX (a representative model-based method), and Q-LEARNING (a representative model-free method).

Mountain Car. The first domain we tested the algorithms on is *Mountain Car*, shown in Figure 5.9. *Mountain Car* is a commonly used testbed for learning continuous tasks (Moore, 1990; Sutton and Barto, 1998), where the agent controls an under-powered car that does not have enough power to drive directly up the hill to the goal. Instead, it must go up the left slope to gain momentum first. The agent has three actions to accelerate the car in different directions: LEFT, RIGHT, NONE. The agent's state is made up of two features: its POSITION and its VELOCITY. The agent receives a reward of -1 each time step until it reaches the

Table 5.3. Properties of the *Mountain Car* task

State	POSITION, VELOCITY
Actions	LEFT, RIGHT, NONE
Reward	−1 each step, 0 upon reaching goal
# State-Actions	30,000
Time-Constrained Lifetime	60,000 actions

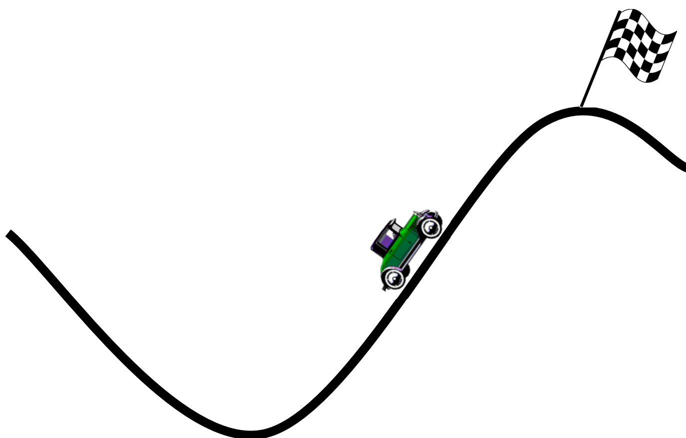


Fig. 5.9. The *Mountain Car* domain. The under-powered car must be driven up the left hill first to gain enough momentum to reach the goal at the top of the right hill.

goal, when the episode terminates with a reward of 0. For the methods requiring discretization, we discretize both state features into 50 values each. Each algorithm is initialized with one seed experience ($\langle s, a, s', r \rangle$ tuple) of the car reaching the goal to jump-start learning. The properties of the domain are listed in Table 5.3.

The average reward per episode of each algorithm on the *Mountain Car* task is shown in Figure 5.10. For each method, the continuous version outperforms the discrete version, as the discrete version cannot model the continuous transition dynamics as quickly as the continuous methods. Continuous TEXPLORE learns the fastest, accruing significantly more reward on the first 8 episodes ($p < 0.0005$) and learning the task well within the time-constrained lifetime. It quickly learns an accurate model of the task, and has a near-optimal policy after only 3 episodes.

Cart-Pole. Next, we performed experiments in the *Cart-Pole Balancing* domain, shown in Figure 5.11. *Cart-Pole Balancing* is another domain typically used for testing continuous state RL agents (Sutton and Barto, 1998), where the agent must learn to balance a pole on top of a cart by applying force to the cart.

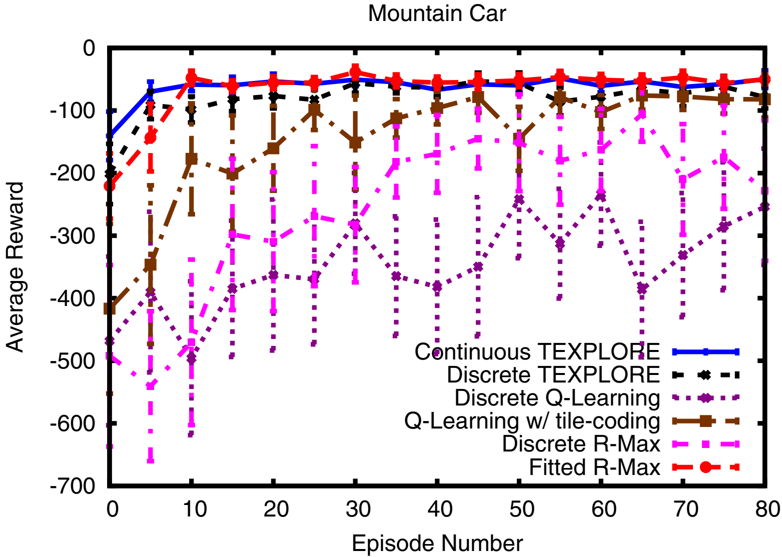


Fig. 5.10. Average reward per episode on the *Mountain Car* task, averaged over 30 trials. Note that Continuous TEXPLORE learns the fastest.

Table 5.4. Properties of the *Cart-Pole Balancing* task

State	CART-POS, CART-VEL, POLE-POS, POLE-VEL
Actions	LEFT, RIGHT
Reward	+1 each step until episode terminates
# State-Actions	320, 000
Time-Constrained Lifetime	640, 000 actions

The state is made up of four features: the cart's position (CART-POS) and velocity (CART-VEL) and the pole's angle (POLE-POS) and velocity (POLE-VEL). The agent has two actions that apply force to the cart in either the LEFT or RIGHT direction. The episode ends when either: 1) the pole falls; 2) the cart goes off the track; or 3) 1,000 time steps have passed. The agent receives a reward of +1 each step until the end of the episode. In this simulated task, the pole is 1 meter long and weighs 0.1 kg, the cart weighs 1.0 kg, the actions exert 10 N of force, and the task is simulated at 50 Hz. For the methods requiring discretization, each feature is discretized into 10 values. Since this task does not have a goal state, no seed experiences are provided to the algorithms. Table 5.4 lists the properties of this domain.

Figure 5.12 shows the average rewards per episode on the *Cart-Pole Balancing* task. Continuous TEXPLORE greatly outperforms the other algorithms and approaches the limit of balancing the pole for 1,000 time steps very quickly. Discrete TEXPLORE also learns quickly, but does not learn a model accurate enough

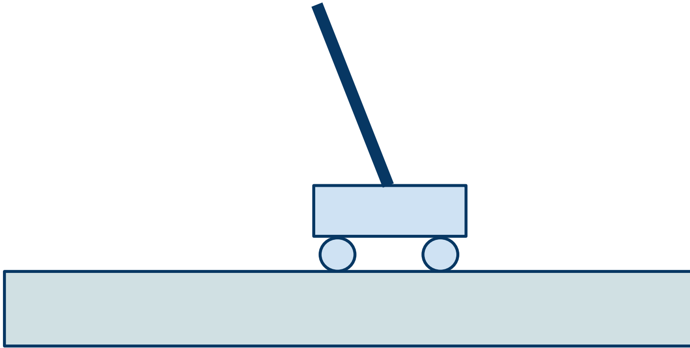


Fig. 5.11. The *Cart-Pole Balancing* task. The agent must apply forces to the cart to balance the pole while keeping the cart on the track.

to balance the pole for 1,000 steps. Meanwhile, the other methods take too long exploring to learn within 100 episodes. Continuous TEXPLORE gains significantly more rewards than the other methods from episode 30 onward ($p < 0.0005$). Continuous TEXPLORE learns the task within the time-constrained lifetime for the task and in considerably fewer episodes than previous approaches, which took 200-750 episodes (Riedmiller, 2005; Lagoudakis and Parr, 2003). In addition, whereas continuous TEXPLORE is controlling the cart-pole continually at 50 Hz, these methods have off-line learning phases where they stop controlling the cart.

5.3 Challenge 3: Delayed Actions

Next, we examine the effects of TEXPLORE's approach for dealing with delays, presented in Section 4.1.2, on two different tasks. First, in Section 5.3.1, we evaluate the algorithm on the *Simulated Vehicle Velocity Control* task. Then we evaluate it on a simulated gridworld task with delayed actions in Section 5.3.2. As described in Section 4.1.2, 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 random forest models can correctly identify which delayed inputs to use.

5.3.1 Simulated Vehicle Velocity Control

First, we evaluate TEXPLORE's approach to actuator and sensor delays on the *Simulated Vehicle Velocity Control* task. 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

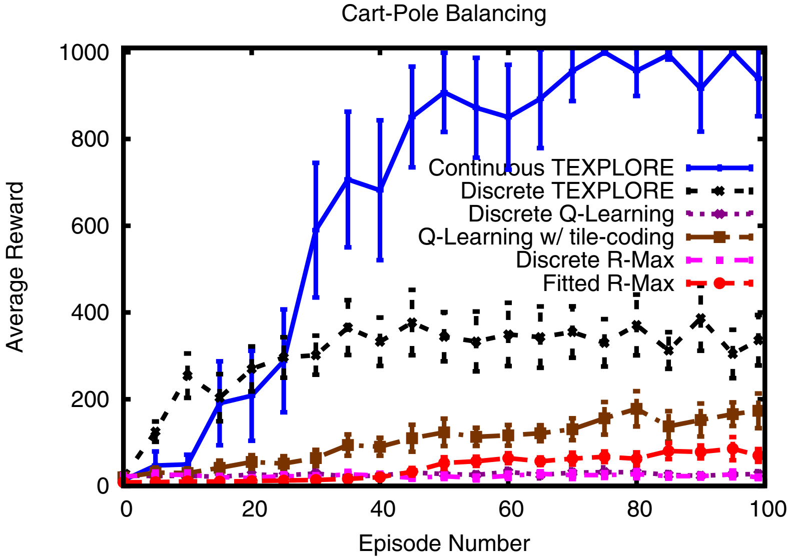


Fig. 5.12. Average reward per episode on the *Cart-Pole Balancing* task, averaged over 30 trials. Note that Continuous TEXPLORE gains significantly more rewards than the other methods from episode 30 onward ($p < 0.0005$).

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 tuple. 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 this 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.

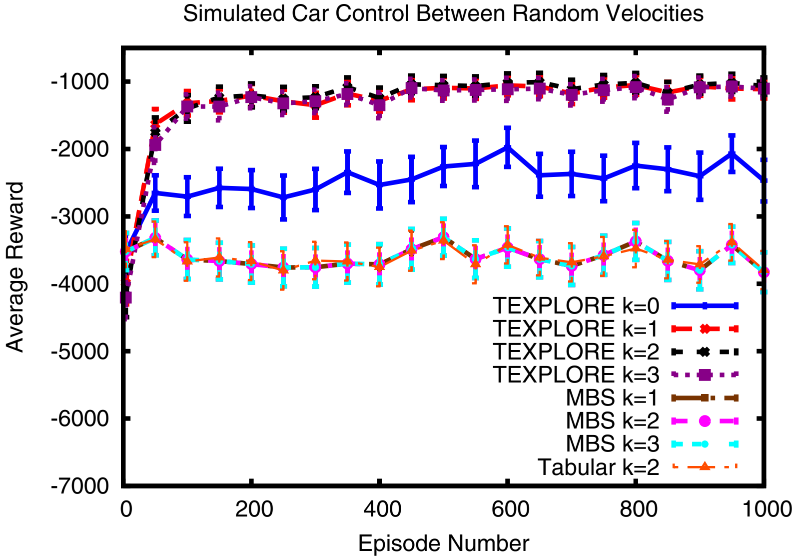


Fig. 5.13. Average reward over 1000 episodes for each method on the *Simulated Vehicle Velocity Control* task. Results are averaged over 50 trials using a 5 episode sliding window and plotted with 95% confidence intervals. TEXPLORE with $k = 2$ performs the best, but not significantly better than TEXPLORE 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.

The average reward for each method on the simulated car control task is shown in Figure 5.13. The TEXPLORE methods using $k = 1, 2,$ and 3 receive significantly more average rewards than the other methods after episode 45 ($p < 0.005$). The results with 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.

5.3.2 Delayed Gridworld

Next, we examine the effects of TEXPLORE's approach for dealing with delays in a *Delayed Gridworld* domain, shown in Figure 5.14. In this domain, the agent starts in a random state in the right room and has to navigate to the goal state on the left. The agent is given 4 actions (NORTH, SOUTH, EAST, and WEST), each of which move the agent in the given direction with probability 0.8, and to either side with probability 0.1. Actions are delayed two time steps before taking effect (the agent does not move for the first two steps). All of the agents are initialized

Table 5.5. Properties of the *Delayed Gridworld* task

State	ROW, COL
Actions	NORTH, EAST, SOUTH, WEST
Reward	-1 each step, 0 upon reaching goal
# State-History-Actions	$3,264$
Time-Constrained Lifetime	$6,528$ actions

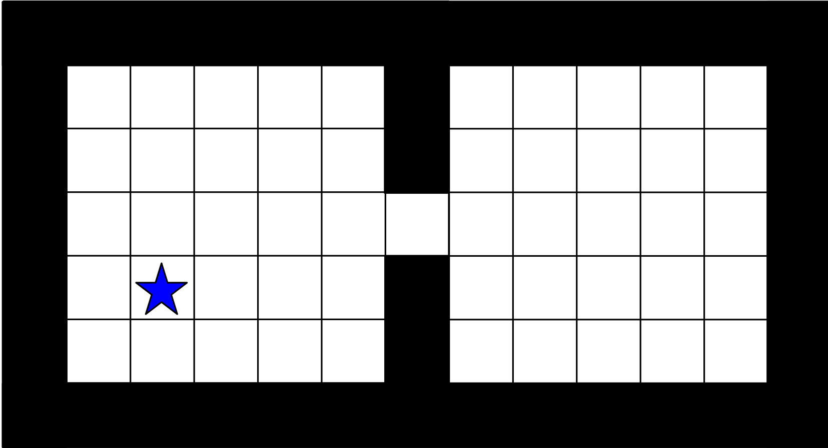


Fig. 5.14. The *Delayed Gridworld* domain. The agent starts at a random cell in the right room and must reach the state marked with the star. Each action the agent selects is delayed 2 steps before taking effect.

with two seed experiences: one of the agent entering the doorway and one of it reaching the goal. The properties of this domain are shown in Table 5.5.

We ran experiments with SARSA(λ) with $\lambda = 0.9$ because the eligibility traces give it some ability to credit reward over previous actions. In addition, we ran MBS-R-MAX (Walsh et al., 2009a) given both the correct delay of $k = 2$ and an incorrect delay of $k = 3$. As described in Section 5.3.1, MBS-R-MAX uses knowledge of the amount of delay to uncover and solve the true underlying MDP. We also ran TEXPLORE with history lengths of $k = 0, 2, 3, 4$, and 5. Figure 5.15 shows the results. Since the random forests used by TEXPLORE can select the relevant features for predictions while ignoring the unnecessary ones, the algorithm performs well even when given extra history features (when $k > 2$) that are not relevant. As the delay input to TEXPLORE is increased, its performance eventually degrades as it must plan over a larger augmented state space and select from a larger set of features. However, it still learns the task even with $k = 4$ and $k = 5$. In contrast, when MBS-R-MAX is given the wrong delay ($k = 3$), the underlying MDP that is uncovered is incorrect and the agent is unable to learn. This distinction is important, as on a robot where the delay may be

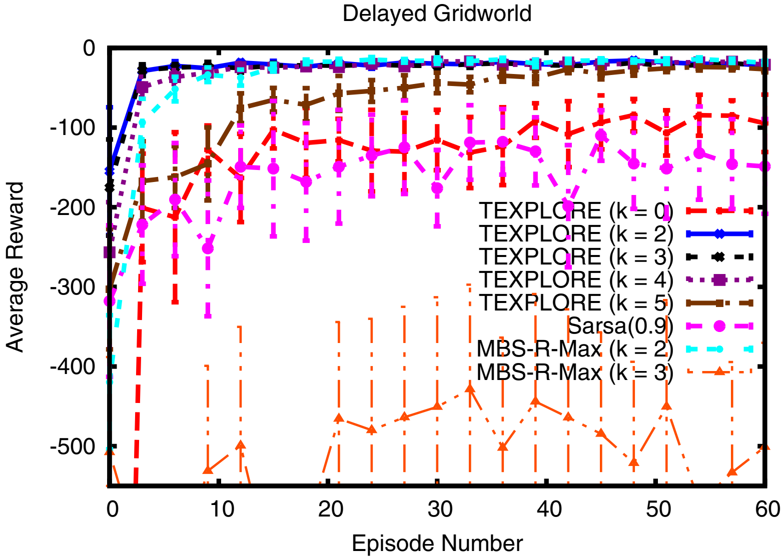


Fig. 5.15. Average reward per episode on the *Delayed Gridworld* with 2 step delay, averaged over 30 trials. Note that TEXPLORE receives significantly more reward than MBS-R-MAX ($p < 0.0005$), even when given an incorrect delay of $k = 2$ or $k = 3$.

unknown, MBS-R-MAX requires the exact delay, while TEXPLORE only requires that we provide an upper bound on the delay. However, if TEXPLORE is given an input that is lower than the true delay (when $k = 0$), it fails as well. Even when MBS-R-MAX is given the correct delay, TEXPLORE with $k = 2$ and $k = 3$ both gain significantly more reward per episode than it for the first 13 episodes ($p < 0.0005$).

Another benefit of making model predictions based on a history of previous actions is that they can be used to uncover hidden state in partially observable domains. As an example of this ability, we created another gridworld domain called *Multi-Goal* with a second possible goal, shown in Figure 5.16. In this task, one of these two states is randomly selected as the goal before each episode. The agent's actions take affect instantly. Although which goal was active for a particular episode is not observable by the agent, by keeping a history of whether it had visited the other goal state, it could uncover the true goal state.

Average results for the algorithms in this task are shown in Figure 5.17. In this domain, once the agent visited the incorrect goal, its model could predict the reward for the other goal state based on the current state, action, and history of actions. Thus, TEXPLORE with $k = 2$ and $k = 3$ performed well on this task. In contrast, MBS-R-MAX learns a model for a fully observable MDP, and is not able to uncover the hidden state of which goal is active.

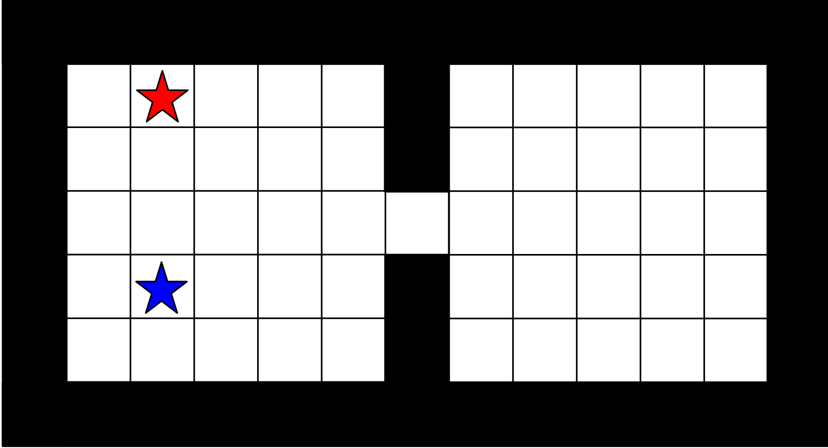


Fig. 5.16. The *Multi-Goal* domain. Each episode, the goal is randomly selected from the two starred states. The agent starts at a random cell in the right room and must reach the goal state.

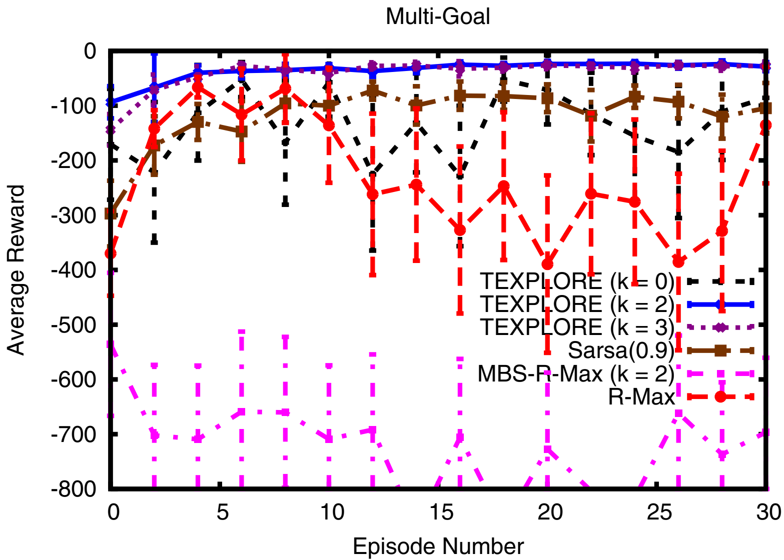


Fig. 5.17. Average reward per episode on the *Multi-Goal* domain, averaged over 30 trials. Note that TEXPLORE with $k = 2$ and $k = 3$ out-performs the other methods.

5.4 Challenge 4: Real Time Action

In this section, we demonstrate the effectiveness of the RTMBA architecture, presented in Chapter 3, 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 real time learning by employing a multi-threaded approach along with UCT(λ) planning.

To demonstrate the effectiveness of RTMBA, we performed experiments on two problems. Our first experiment measures the performance gains due to RTMBA on the simulation of the autonomous vehicle, where real time actions are absolutely necessary. The second set of experiments measure the cost of parallelization in terms of environmental reward compared to a traditional sequential architecture. We use a simulated domain, which can wait as long as necessary for the agent to return an action (or it can execute actions as fast as the algorithm returns them).

5.4.1 Simulated Vehicle Velocity Control

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)
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.

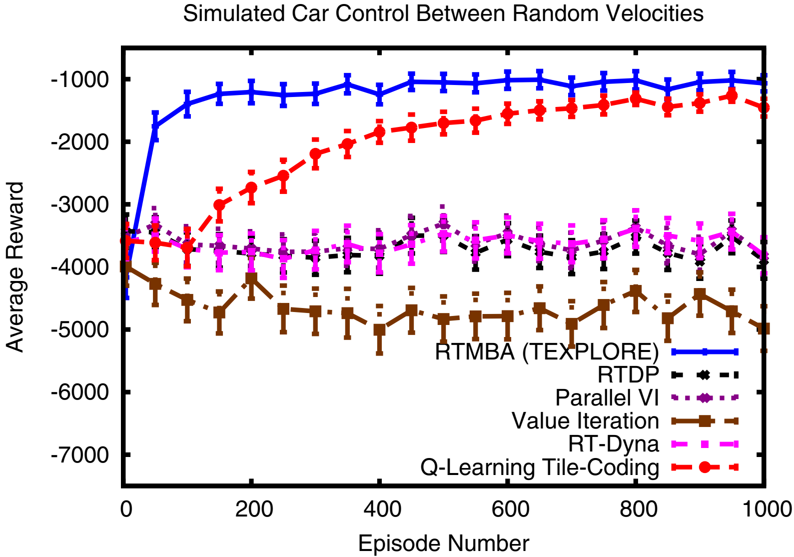


Fig. 5.18. Average reward over 1000 episodes for each method on the *Simulated Vehicle Velocity Control* task. Results are averaged over 50 trials using a 5 episode sliding window and plotted with 95% confidence intervals. Note that TEXPLORE performs the best.

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.

Figure 5.18 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.

5.4.2 Mountain Car

In this section, we demonstrate the effectiveness of the RTMBA architecture on the *Mountain Car* task, presented in Section 5.2.2. In this task, the simulated environment can wait for the agent to return an action (or it can execute actions as fast as the algorithm returns them). Our experiments measure the cost of parallelization in terms of environmental reward compared to a traditional sequential architecture where model learning and planning can each take as long as necessary.

We ran experiments comparing Q-LEARNING, DYNA and TEXPLORE. All methods were run on a discretized version of the domain, with both state features discretized into 100 values. Five algorithms were run with the TEXPLORE model using the following architectures:

1. Sequential Architecture (Alg. 3.1) with Value Iteration planning
2. Sequential Architecture (Alg. 3.1) with UCT planning
3. RTMBA (Alg. 3.4) at 10Hz
4. RTMBA (Alg. 3.4) at 25Hz
5. RTMBA (Alg. 3.4) at 100Hz.

Note that the last three algorithms, since they are using RTMBA, are the TEXPLORE algorithm. We ran two versions of DYNA: DYNA performed updates on 1,000 saved experiences between each action; and RT-DYNA performed as many updates as it could while returning actions at 25 Hz. Between each action, the two sequential methods performed a full model update, then planned on their model by running value iteration to convergence or performing UCT rollouts for 0.1 seconds. Each algorithm is initialized with one seed experience ($\langle s, a, s', r \rangle$ tuple) of the car reaching the goal to jump-start learning. We ran 30 trials of each algorithm, with Q-LEARNING run for 2,000,000 episodes, DYNA for 4,000 episodes, and the remaining methods run for 1,000 episodes. Each trial was run on a single core of a machine with 2.4 - 2.66 GHz Intel Xeon processors and 4 GB of memory.

Our aim was to compare the real time algorithms with the sequential methods when they are given the time needed to fully complete their computation between each step. Thus we can examine the performance lost by the real time algorithms due to acting quickly. In contrast, the model-free methods could act as fast as they wanted, resulting in learning that took little wall clock time but many more samples. To perform these experiments, the environment waited for each algorithm to return its action, thus benefiting the sequential algorithms. Waiting this way is only possible in simulation, whereas on a real robot, the action rate is defined by the robot rather than the algorithm.

Figure 5.19 shows the average reward per episode for each algorithm over the first 50 episodes in the domain and Figure 5.20 shows the reward plotted against clock time in seconds (note the log scale on the x axis). The first plot shows that the two sequential methods perform better than RTMBA in sample efficiency, in particular, receiving significantly more reward per episode than RTMBA running at 25 and 100 Hz over the first 5 episodes ($p < 0.05$). RTMBA running at 10 Hz did

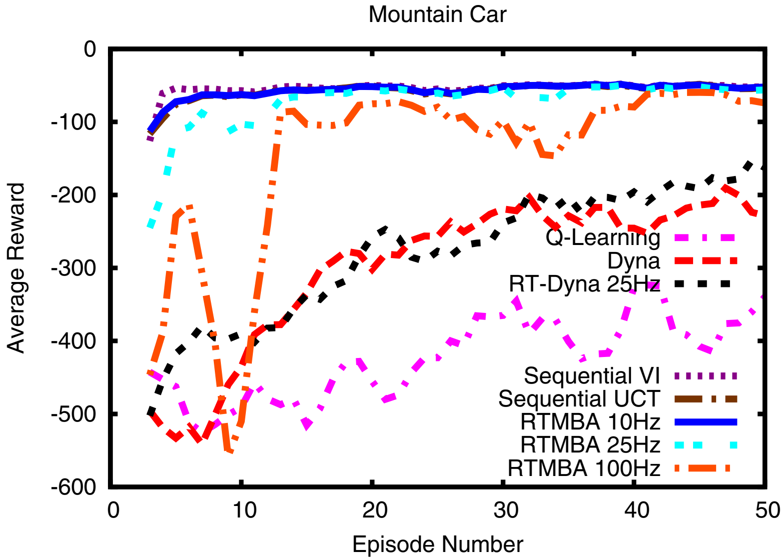


Fig. 5.19. Average reward per episode on *Mountain Car*, averaged over 30 trials. Results are averaged over a 4 episode sliding window.

not perform significantly worse than the sequential method using UCT. However, Figure 5.20 shows that better performance of the sequential methods came at the cost of more computation time. For the sequential methods, switching from exact to approximate planning reduces the time to complete the first episode from 1541 to 142 seconds, but the UCT method is still restricted by the need to perform complete model updates between actions. This restriction is removed with RTMBA, and all three versions using it complete the first episode within 20 seconds. In fact, all three RTMBA methods start performing well after 90 seconds, likely because they all took this much time to learn an accurate domain model. Compared with the sequential methods, RTMBA is only slightly worse in sample efficiency, and acts much faster, meeting our requirement of continual real time action selection.

The model-free approaches, Q-LEARNING and DYNA, select actions extremely quickly and converge to the optimal policy in less wall clock time than any version of RTMBA. However, Figure 5.19 shows that they are not as sample efficient. While RTMBA converges to the optimal policy within tens of episodes, DYNA takes approximately 650 episodes to converge, and Q-LEARNING takes approximately 22,000. Although RT-DYNA performs more planning updates between actions than DYNA, it is still not as sample efficient as TEXPLORE, taking approximately 300 episodes to converge. These methods learn in less wall clock time simply because they can take many more actions than RTMBA in a given amount of time. On an actual robot, it will not be possible to take actions faster than the robot's control frequency, and the poor sample efficiency of these methods will

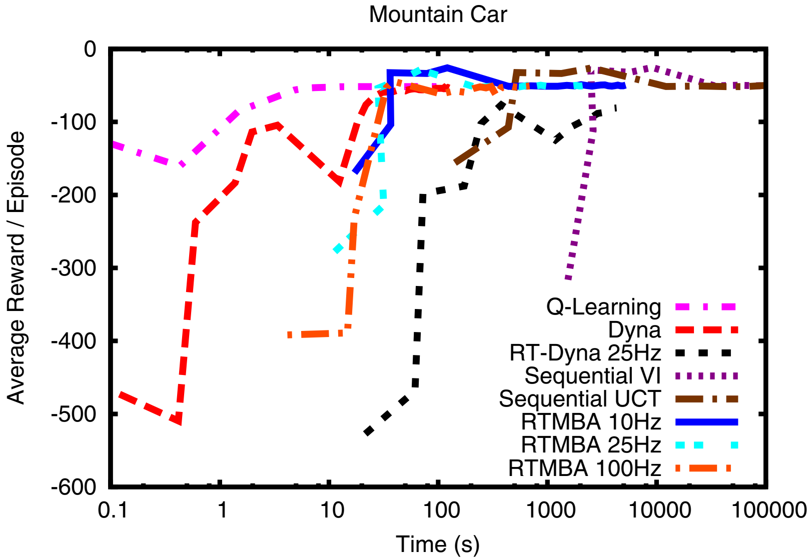


Fig. 5.20. Average reward versus clock time on *Mountain Car*, averaged over 30 trials. Q-LEARNING was run for 2,000,000 episodes, DYNA was run for 4,000 episodes, and the other algorithms were run for 1000 episodes. The line for each algorithm starts when the first episode was *completed*. Note that the x-axis is in log scale.

result in longer wall clock learning times as well. In comparison, RTMBA learns in fewer samples, meeting our requirement of sample efficiency even while running at reasonable robot control rates between 10 and 100 Hz.

In addition to enabling real time learning, another benefit of RTMBA is its ability to take advantage of multi-core processors, because each parallel thread can run on a separate core. We ran experiments comparing the performance of RTMBA when running on one versus multiple cores. These experiments were performed on a machine with four 2.6 GHz AMD Opteron processors. Figure 5.21 shows the average reward per episode for these experiments, running at 25 Hz. For comparison, we ran the sequential method using UCT as a planner on the multi-core machine. It had unlimited time for model updates and then planned for 0.04 seconds (the same time given to RTMBA for both computations). Since the sequential architecture only has a single thread, it only used a single core even on the multi-core machine. Meanwhile, RTMBA utilized three processors with each thread running on its own core. Using the extra processors allowed the parallel version to perform more model updates and planning rollouts between actions than the single core version. Due to these advantages, the multi-core version performs better than the single core version, receiving significantly more rewards on every episode ($p < 0.005$). In addition, it even performs better than the sequential method on episodes 3 to 14 ($p < 0.01$), even though the sequential method is given unlimited time for model updates.

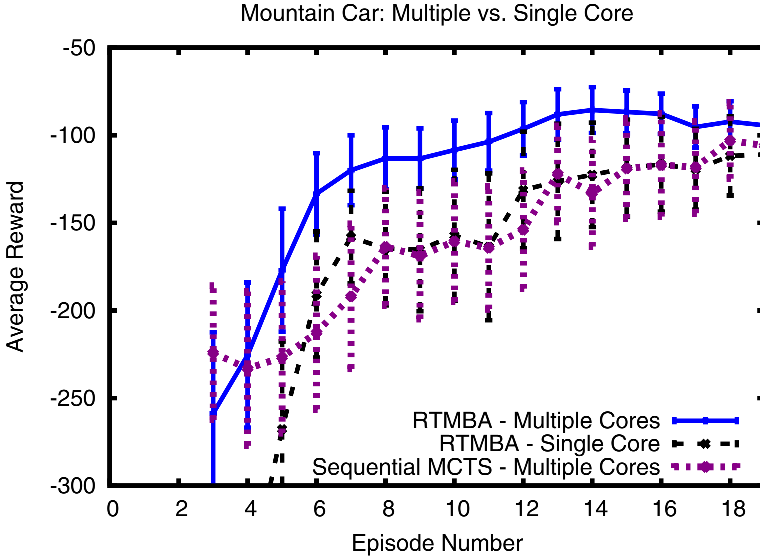


Fig. 5.21. Comparisons of the methods using a multiple core machine. Each method is averaged over 30 trials on *Mountain Car*. Results are averaged over a 4 episode sliding window.

These results demonstrate that RTMBA enables algorithms to maintain sample efficiency, even while acting in real time. In addition, we have demonstrated that while using approximate planning reduces the time required by model-based methods, they do not reach real time performance without RTMBA.

5.5 Dependent Transitions

In this section, we evaluate TEXPLORE's solution to handling domains with dependent feature transitions, which was presented in Section 4.1.3. Since TEXPLORE's solution of adding synchronic arcs to the DBN works with other factored algorithms, we also tested it with FACTORED R-MAX (Guestrin et al., 2002). FACTORED-R-MAX is given the structure of the DBN and only has to learn conditional probabilities, while TEXPLORE uses random forests to learn both the structure and probabilities of the model. FACTORED R-MAX was run with $m = 20$. TEXPLORE was run with $m = 5$ trees per forest, $f = 0.2$, and $w = 0.55$. All experiments were run with the discount factor $\gamma = 0.998$. For all of these experiments, each algorithm was initialized with one seed experience ($\langle s, a, s', r \rangle$ tuple) of the goal state to jump-start learning.

While the features in many domains often do transition dependently, it is surprisingly difficult to find domains where the resulting modeling error from assuming feature independence affects the learned policy. As an example where having such an incorrect model affects the learned policy, we created a domain

Table 5.6. Properties of the *Trap Room* task

State	ROW, COL
Actions	NORTH, EAST, SOUTH, WEST
Reward	−1 each step, 0 on goal, −250 on trap
# State-Actions	252
Time-Constrained Lifetime	504 actions

called the *Trap Room*, shown in Figure 5.22. It is a typical gridworld domain: the agent starts in the top left of the gridworld, and receives a reward of −1 each step until it reaches the goal state, where it terminates with a reward of 0. However, there are two "trap" states that provide large negative rewards of −250 when the agent passes through them. The agent has four actions to move it NORTH, SOUTH, EAST, and WEST. The agent's actions are stochastic: they move the agent in the intended direction with probability 0.8 and in either perpendicular direction with probability 0.1. The optimal policy for the domain is for the agent to follow the white arrows shown in Figure 5.22. However, if the agent assumes that its features transition independently, then its model predicts that there is some chance of it moving diagonally into the trap above the goal on its last step into the goal state. With this incorrect model, it will instead follow the solid red arrows, navigating around the other trap to approach the goal from the south. The properties of this domain are listed in Table 5.6.

Figure 5.23 shows the average rewards per episode for the algorithms averaged over 30 trials in the *Trap Room*. The versions of TEXPLORE and FACTORED R-MAX using models that predict features independently perform poorly, with FACTORED R-MAX converging to a policy that receives an average of −32.7 reward per episode and TEXPLORE converging to a policy averaging −26.5 reward. Meanwhile, when their models are modified with synchronic arcs to model the dependence between state features, the agents learn the optimal policy, converging to policies that receive −9.0 reward per episode. Both methods with synchronic arcs received significantly more rewards than both methods without synchronic arcs after episode 70 ($p < 0.0005$). We applied the synchronic arcs between the X and Y feature in both directions, and the results show that the ordering of these features does not result in a difference in the learned policy. Finally, for comparison, we show results for R-MAX (Brafman and Tennenholtz, 2001), which does not learn a factored model. It learns the optimal policy, but takes more episodes to learn it than the factored approaches. Algorithms using the DBN with the added synchronic arcs still achieve higher sample efficiency than using a tabular model, even while removing the feature independence assumption.

Next, in Figure 5.24 we show results in a similar domain that has no traps, called the *No Trap Room*. Although the features still transition dependently and thus the independent feature models are incorrect, the policy calculated by them is still the optimal one. Here we see that both the dependent and independent model achieve the optimal policy. For FACTORED R-MAX, there is some additional exploration required for the added synchronic arcs, resulting in the agent taking

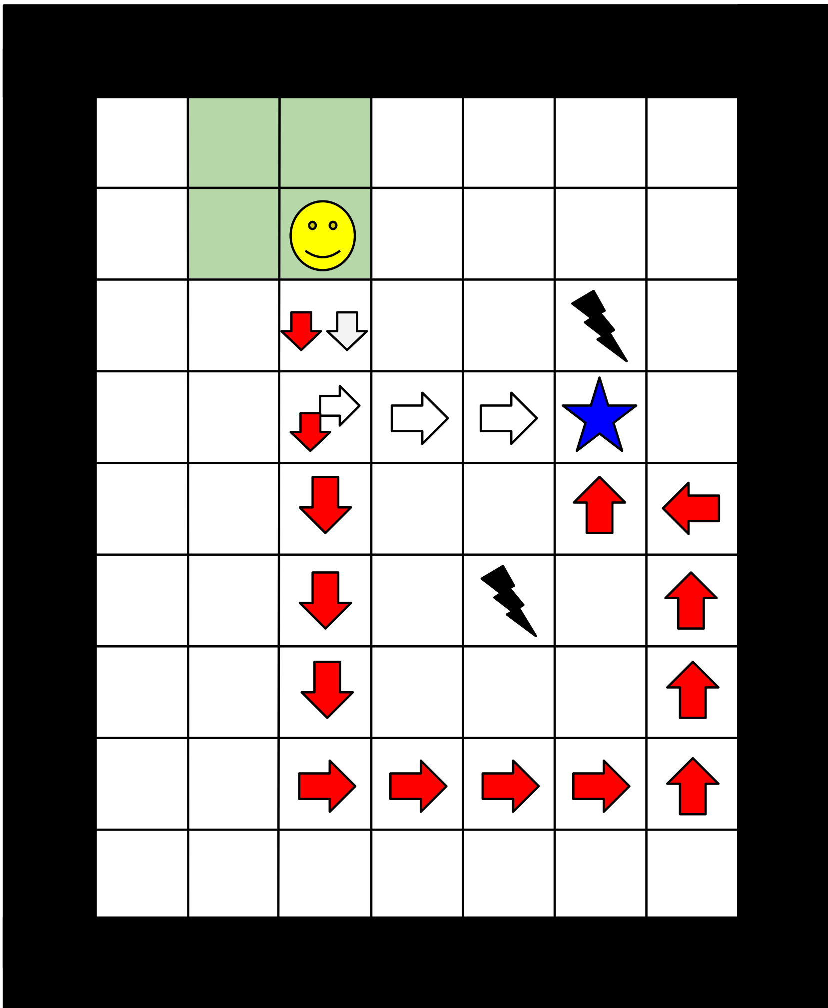


Fig. 5.22. The *Trap Room* domain. The goal state is marked by the star, and the negative rewarding "trap" states are marked by lightning bolts. The optimal policy is represented by the white arrows, while the policy learned by models assuming feature independence is shown by the solid red arrows.

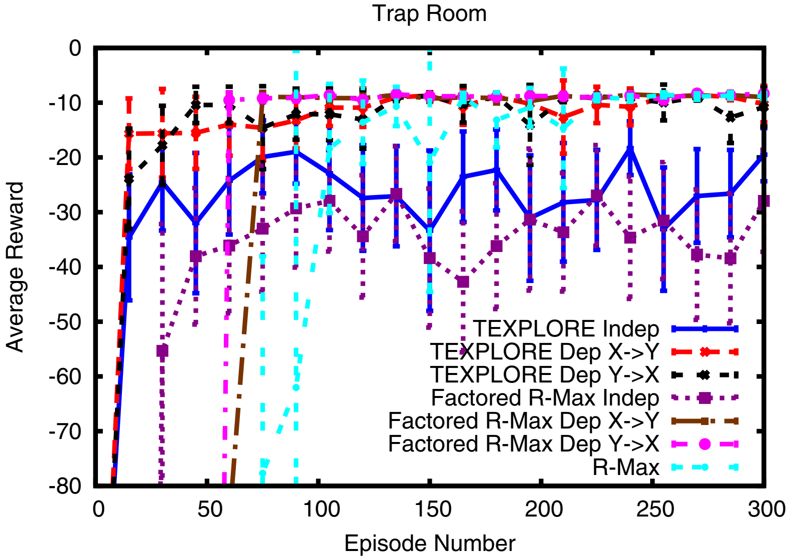


Fig. 5.23. Average rewards for each method over 300 episodes in the *Trap Room* domain. Results are averaged across 30 trials and using a 20 episode sliding window. Note that TEXPLORE with dependent transitions performs the best, and is not affected by the order of the dependencies.

approximately 28 extra episodes to learn the task. However, even with the extra exploration required for the added synchronic arcs, FACTORED R-MAX still learns the task faster than R-MAX. Unlike FACTORED R-MAX, which takes the DBN structure as input, TEXPLORE is only taking possible parents for each feature and learning the structure on its own. Thus, TEXPLORE learns the task in the same number of episodes with or without the additional features as inputs to its model, as the convergence rate of the random forest model is not affected by the extra inputs (Biau, 2012).

Finally, we created a third domain called *Independent Feature Trap Room* by modifying the transition dynamics of the *Trap Room* domain so that features transition independently. The agent moves in the intended direction with probability 0.8, and diagonally to either side with probability 0.1 each. Thus, the features transition independently as one feature always transitions, and the other transitions randomly. Figure 5.25 shows results for this domain. Again, all the methods converge to the optimal policy. For FACTORED R-MAX, the method with the added synchronic arcs takes longer to learn than the one without synchronic arcs, but still learns faster than R-MAX. TEXPLORE performs the same with or without the added features, as it correctly learns a DBN structure that ignores them.

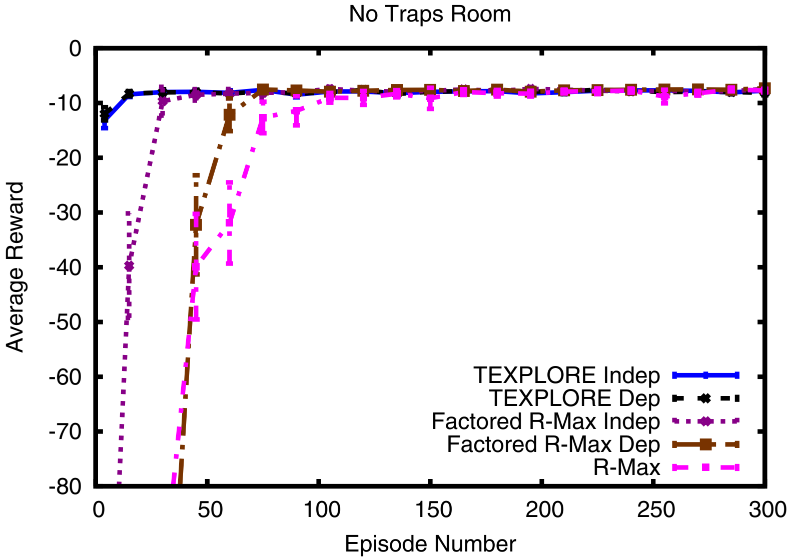


Fig. 5.24. Average rewards for each method over 300 episodes in the *No Trap Room*. Results are averaged across 30 trials and using a 20 episode sliding window. Note that the extra dependencies in TEXPLORE's model do not affect it negatively.

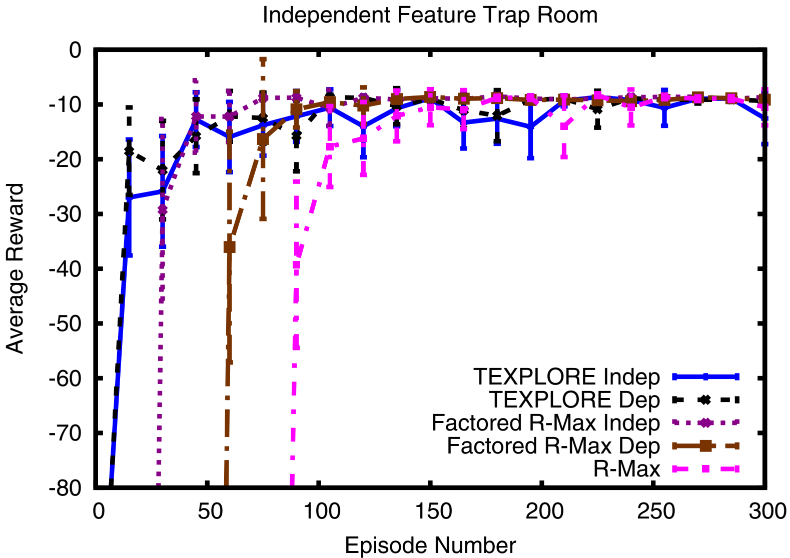


Fig. 5.25. Average rewards for each method over 300 episodes in the *Independent Feature Trap Room*. Results are averaged across 30 trials and using a 20 episode sliding window.

Our results show that adding synchronic arcs to the DBNs used by many factored models enables them to learn correct models in domains where features transition dependently. We have also shown that these extra arcs do not slow down TEXPLORE's learning, as its random forest model is able to effectively select the relevant features to split on.

5.6 TEXPLORE on a Physical Robot

After demonstrating each aspect of TEXPLORE on the simulated vehicle control task, this section demonstrates the complete algorithm learning on the physical autonomous vehicle. This domain is the main motivation for the *RL for Robotics Challenges* that TEXPLORE addresses, and these experiments are the culminating results of this book. 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 5.26 shows the average rewards over 20 episodes. In all five trials, the agent learns the 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. Since there is only a single target velocity for these experiments, the number of state-actions in the domain is considerably less, and the time-constrained lifetime for this task is $33,550$ steps, or 335 episodes. Still, TEXPLORE easily learns the task within this time frame.

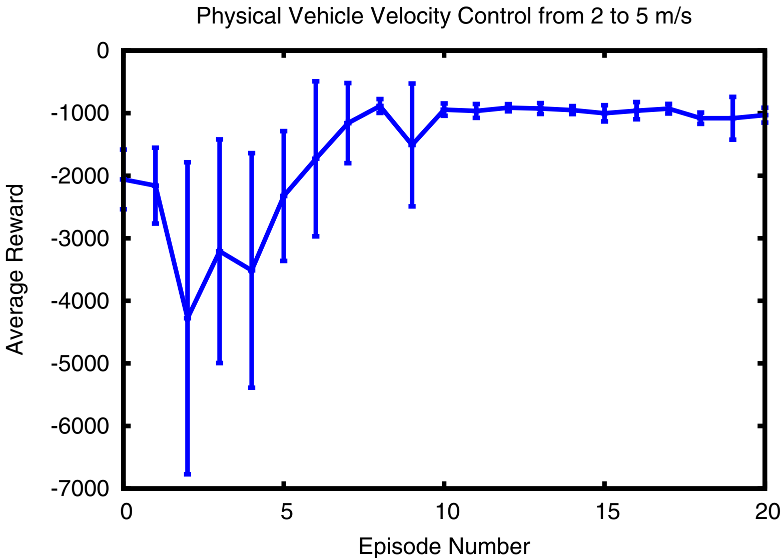


Fig. 5.26. Average rewards of TEXPLORE learning to control the physical vehicle from 2 to 5 m/s. Results are averaged over 5 trials and plotted with 95% confidence intervals. In every trial, the agent successfully learns the task by episode 10.

As I was physically present in the vehicle for the learning experiments, I 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.

5.7 Chapter Summary

In this chapter, I have thoroughly evaluated TEXPLORE and its approach to solving each of the *RL for Robotics Challenges*. For Challenge 1 (sample efficiency), I showed that its approach to model learning and exploration enable it to learn two tasks in fewer samples than other state-of-the-art approaches. For Challenge 2 of learning in continuous domains, I show that TEXPLORE's regression tree models learn more accurate predictions than other possible models and that it performs better than other methods on both *Mountain Car* and *Cart-Pole*. For the third challenge, handling sensor and actuator delays, I show that TEXPLORE's k-Markov solution works well on two different domains and only requires the user to provide an upper bound on the amount of delay, k. Next, I demonstrated that TEXPLORE's real time architecture works better than the alternatives and demonstrated that it does not cost TEXPLORE much in sample efficiency to run at reasonable real time frame rates. In Section 5.5, I evaluated TEXPLORE's approach to learning in domains with dependent feature transitions, and showed that the added synchronic arcs to not adversely affect the number of samples required for TEXPLORE to learn. Finally, I demonstrated TEXPLORE learning to control the velocity of a physical autonomous vehicle in real time, while being run on-board the robot. For all of these domains, TEXPLORE learns to perform well on the task within the time-constrained lifetime. In the next chapter, I further explore other possibilities for exploration within the TEXPLORE algorithm.

6 Further Examination of Exploration*

In this chapter, I examine other approaches to exploration that could be combined with TEXPLORE*'s model. First, I introduce three domain classes that each suggest a different type of exploration. Then, in Section 6.1, I look at how to perform exploration in domains where a needle-in-a-haystack search is required to find an arbitrarily located reward or transition. In the next section, I look at the opposite case: can we explore better in a domain with a richer, more informative set of state features? Finally, in Section 6.3, I present an algorithm that can learn which of these exploration approaches to adopt on-line, while interacting with the environment. Then I present some empirical comparisons of these approaches against* TEXPLORE *in Section 6.4, before summarizing the chapter in Section 6.5.*

This book is focused on applying RL to *time-constrained* domains, where the agent is not given enough steps to guarantee that it can learn an optimal policy. In such domains, exploring intelligently is critical. Since the agent cannot explore exhaustively, it must target its exploration on the particular state-actions that are most likely to give it the knowledge to perform well in its limited lifetime. In this chapter, I examine various approaches to exploration and how they perform in different types of domains.

The best exploration strategy for an agent to employ varies greatly depending on the exact properties of the domain. To aid with our analysis of exploration in such domains, we will define three different classes of domains: *haystack* domains, *prior information* domains, and *informative* domains. Domains may belong to multiple classes or none of the three classes.

Haystack domains are ones where the agent needs to find an arbitrarily located state that has an unusual transition or reward. We define a state as having an unusual transition or reward if the image of the transition or reward function from that state is unusual in some way compared to the images of these functions from other states in the domain. For example, the image of the transition function may be unusual because the state is a terminal state, or because it has a different form of distribution over next states than do the images from other states. The image of the reward function may be unusual because of its magnitude or sign relative to the images of the function from other states in the domain. Henceforth, we refer to states with such unusual transition or reward function images as being *unusual states*. Many domains have unusual states that the agent must discover to perform well. In these domains, the best the agent can

* This chapter contains material from four publications: (Hester and Stone, 2009b; Hester et al., 2010; Hester and Stone, 2012a; Hester et al., 2013).

T. Hester, *TEXPLORE: Temporal Difference Reinforcement Learning for Robots and Time-Constrained Domains*, Studies in Computational Intelligence 503,
DOI: 10.1007/978-3-319-01168-4_6, © Springer International Publishing Switzerland 2013

do is to perform a "needle-in-the-haystack" search, visiting every state-action in the domain until it finds the desired state. Many toy domains in the RL literature such as the Taxi domain (Dietterich, 1998) and Puddle World (Sutton, 1996) are *haystack* domains.

Prior Information domains are ones where the agent is given some information about the location of unusual states. An example is a tourist in a city looking for a particular landmark. Rather than driving to every intersection to see if the landmark is there, the tourist can use a map to locate the position of the landmark on their own. For an RL agent, this is implemented by giving the agent a partial model of the domain, or giving it a few example transitions to initialize its model, as is done with TEXPLORE. In these domains, rather than exploring each state-action for a particular transition or reward function, the agent knows the positions of these interesting state-actions and instead needs to learn the dynamics of the domain to find the best policy.

Informative domains are ones where the agent is given some informative state features that predict the positions of unusual states. An example of this class of domain is a robot with distance sensors. These sensors enable it to sense the location of doorways in a wall without having to try moving through the wall at each position. In these domains, the agent is given less information than in the *prior information* domains, as it must learn what its sensors mean, what they predict, and how to use them. Still, agents in these types of domains can use their sensors to perform more specific, targeted exploration than agents in *haystack* domains.

Domains can belong to more than one of these classes. For example, a grid-world domain could have sensors that provide the agent with information about the locations of walls and doorways, but have an arbitrarily located goal state. In this case, the domain is both an *informative* and *haystack* domain. Domains also may not belong to any of these classes. For example, domains may not have individual states with unusual transition or reward function images. Instead, some domains have transition or reward functions that vary smoothly with the state features. For example, the transition function in the *Cart-Pole Balancing* domain varies smoothly with the pole angle, rather than there being individual states that have unusual transition function images.

Most of the domains that we looked at in Chapter 5 were *prior information* domains. In this chapter, we present extensions to TEXPLORE to modify its exploration for both *haystack* and *informative* domains. In Section 6.1 we present an approach for *haystack* domains that explores each state-action in the domain. Then, in Section 6.2 we look at how an agent can make use of the state features in *informative* domains to explore more intelligently. Sometimes, it may be difficult to determine which type a domain is and thus which exploration strategy to select. For these domains, we present a method for learning the best exploration strategy on-line while interacting with the domain in Section 6.3. Finally, we compare these approaches against TEXPLORE on a couple of domains and analyze when each should be adopted. All the domains presented in this chapter are listed in Appendix B and are available in our ROS

package: `http://www.ros.org/wiki/rl_env`. Throughout the chapter, significance results are calculated using a Student's t-test.

6.1 Explicit Exploration

One of our goals in developing the TEXPLORE algorithm was to have a method that would not have to visit every state-action in the domain. However in *haystack* domains, where there are arbitrarily located states with unusual transition or reward function images, such exploration is required. For example, in a typical grid world with an arbitrarily located goal state, the best the agent can do is try each state-action until it finds the goal. Therefore, in this section we present an exploration method that addresses *haystack* domains. While this method is driving the agent to explore each state-action, we still desire it to make use of TEXPLORE's model generalization and learn faster than tabular methods such as R-MAX (Brafman and Tennenholtz, 2001).

6.1.1 Methodology

In *haystack* domains, we desire the agent to use R-MAX-like exploration (visiting every state-action) until it finds the arbitrarily located transition or reward, and then switch to acting greedily with respect to its model. We extend the TEXPLORE method into an algorithm called TEXPLORE with explicit exploration (TEXPLORE-EE)[1] to use an explicit exploration mode. The general approach is to plan a policy on the model, calculate the expected value of this policy, and use this value to decide whether to explore or exploit. TEXPLORE-EE differs from TEXPLORE in two ways. First, rather than using a random forest of trees to drive exploration, it explicitly chooses when to explore or exploit. Second, it uses a model with a *single* decision tree for each feature rather than a random forest, as the agent is not building or using multiple hypotheses of the domain. Pseudo-code for the algorithm is shown in Algorithm 6.1.

In this approach, the user sets one parameter, V_{min}, that specifies the minimum acceptable value for a learned policy. The agent learns a model of the domain using a single decision tree for each feature. Whenever the agent updates its model, it plans on the model and checks if its policy achieves the minimum value V_{min} (Line 6). If the agent's policy does not achieve the minimal value, the agent goes into *exploration* mode, where it is driven to explore each state-action. Otherwise, it remains in *exploitation* mode, where it takes what it believes is the optimal action at each step. In this approach, the agent will start out exploring each state-action until it has learned enough to plan a policy with value $V^{\pi} \geq V_{min}$. At this point, it will stop exploring and exploit its model to receive these rewards.

In exploration mode, the agent follows a policy similar to R-MAX (Brafman and Tennenholtz, 2001). When planning, the algorithm ignores the task reward and

[1] Note that TEXPLORE-EE was called RL-DT in (Hester and Stone, 2009b; Hester et al., 2010).

Algorithm 6.1. TEXPLORE with explicit exploration (TEXPLORE-EE)

```
1: Input: S, A, V_min        ▷ S: state space, A: action space, V_min: threshold
2: Initialize M to empty model
3: Initialize s to a starting state in the MDP
4: loop
5:     π_exploit, V_exploit ← PLAN-POLICY(M)                    ▷ Plan on model
6:     if V_exploit(s) < V_min(s) then               ▷ Go into exploration mode
7:         π_explore, V_explore ← PLAN-POLICY(M_explore)   ▷ Use modified model
8:         Choose a ← π_explore(s)
9:     else
10:        Choose a ← π_exploit(s)
11:    end if
12:    Take action a, observe r, s'
13:    M ⇒ UPDATE-MODEL(⟨s, a, s', r⟩)                        ▷ Update model M
14:    s ← s'
15: end loop
```

gives all state-actions with the minimum number of visits a reward of 1.0. Unlike R-MAX, which assumes these unknown transitions are terminal, this approach can plan policies to go through a series of unvisited state-actions to achieve the highest possible intrinsic rewards. In addition, this approach will visit all unvisited state-actions first, then state-actions that have been visited only once, and continue in order, rather than treating all state-actions with fewer than m visits equally.

This approach has both benefits and drawbacks. It performs more limited exploration than methods such as R-MAX by stopping exploration as soon as it finds a sufficient policy. In addition, TEXPLORE-EE does take some advantage of the generalization capabilities of the decision tree models, as it uses its predictions of unvisited state-actions to plan trajectories through sequences of unvisited state-action for the most efficient exploration. However, it is still not *targeting* its exploration in any way: even if the agent has evidence that certain state-actions are not useful or provide negative rewards, it will still explore them if they have the fewest visits. In addition, TEXPLORE-EE requires the user to tune a threshold of the minimum value policy that is acceptable, which will need to be tuned separately for each domain.

6.1.2 Empirical Evaluation

We evaluate this exploration approach on two different domains. First we look at at the *Taxi* domain, where are there are multiple arbitrarily located goals. This task makes the benefits of exploring individual state-actions clear. We then evaluate the algorithm on a robot control task, controlling a Nao robot learning to score penalty kicks. This task is similar to *Taxi*, as the robot can only score goals from a few specific states in the domain.

Table 6.1. Properties of the *Taxi* task

State Features	X, Y, PASSENGER, DESTINATION
Actions	EAST, WEST, NORTH, SOUTH, PICK-UP, DROP-OFF
Reward	−1 normally, +20 upon completion −10 for bad PICK-UP or DROP-OFF action
# State-Actions	3,000
Time-Constrained Lifetime	6,000 actions
Domain Class	*Haystack*

Taxi. Our first set of experiments is on the Taxi domain (Dietterich, 1998), which was first introduced in Section 2.2.3. In this domain, the agent must navigate to the landmark where the passenger is located, pick her up, and then navigate to her destination landmark and drop her off. These landmarks are at one of four arbitrarily located cells in the grid world. Another aspect of this domain is that calling the PICK-UP or DROP-OFF action in the wrong state results in a negative reward of −10, while the agent normally receives a reward of −1 each step. The properties of this domain are shown in Table 6.1.

On this domain, we compared four methods: TEXPLORE, TEXPLORE-EE, Q-LEARNING, and R-MAX. TEXPLORE-EE was run with the threshold $V_{min} = -10.0$. Q-LEARNING was run with ϵ-greedy exploration with $\epsilon = 0.1$ and R-MAX was run with $m = 10$. None of the algorithms are given any seed experience transitions to initialize their model.

The cumulative rewards received by each algorithm over 200 episodes are shown in Figure 6.1. TEXPLORE-EE earns significantly more cumulative rewards than the other three algorithms ($p < 0.001$). Importantly, it learns the task much faster than R-MAX. Although exploration of each state-action is required for this task, TEXPLORE-EE can still take advantage of its model generalization to learn faster. Another important point to note is that TEXPLORE-EE's choice between exploration and exploitation modes is dependent on which landmark it is trying to navigate to. In each episode, the passenger's location and destination are selected randomly from the four landmarks. On episodes where these landmarks are new to TEXPLORE-EE, it will not find a good policy and choose to explore. On other episodes where it has seen the landmarks before, it can exploit its knowledge, even though it still may not know some landmark locations.

TEXPLORE performs very poorly on this domain. Not only are the landmarks in arbitrary locations, but TEXPLORE's tree models all quickly agree on the fact that the PICK-UP and DROP-OFF actions provide large negative rewards and TEXPLORE stops trying them, selecting randomly from the four navigation actions. Later, in Section 6.4, I will examine how TEXPLORE performs in this domain when I modify it to be a *prior information* domain by giving the agent information about the landmark locations. In addition, I will also look at a different *haystack* domain where exploratory actions are not penalized.

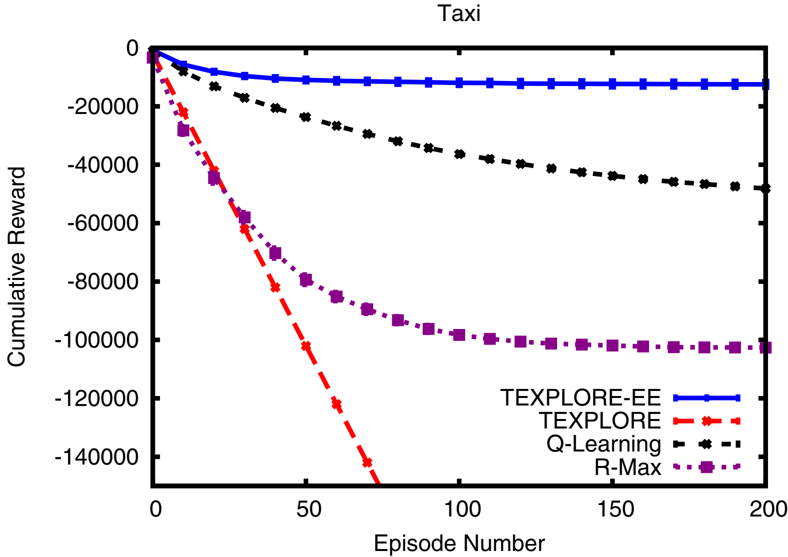


Fig. 6.1. Cumulative reward of the algorithms on the *Taxi* domain, averaged over 30 trials. Note that TEXPLORE-EE performs the best here, while TEXPLORE performs poorly on this task because it requires the agent to explore every individual state-action.

Penalty Kicks. Our second evaluation task for TEXPLORE-EE is to train an Aldebaran Nao humanoid robot to score penalty kick goals. This scenario takes place in the domain of the RoboCup Standard Platform League (SPL). RoboCup is an annual robot soccer competition with the goal of developing an autonomous humanoid robot soccer team that can defeat the world champion human team by 2050. Games in the SPL use the Aldebaran Nao humanoid robot, which is 58 centimeters tall and has 21 degrees of freedom. The robot has two cameras in its head and computation is performed on the robot using its AMD Geode processor.

When an elimination game in the SPL ends in a tie score, the winner is determined by best of five penalty kicks. In the penalty kick, the defending robot starts in the middle of the goal on the goal line, while the offensive robot starts at mid field. The ball is placed on a white cross located 1.8 meters from the goal. The robot has one minute to walk up to the ball and score a goal.

Penalty kicks can be critical to success in the SPL as many of the teams are evenly matched and historically many games end in a tie and are decided by penalty kicks. At RoboCup 2009 in Graz, Austria, 28 of the 64 games (43.75%) ended in a tie.[2] The frequency of games ending in ties makes penalty kicks an important aspect of the games.

Even though most teams employed a stationary goal keeper for the kicks in 2009, teams rarely scored on penalty kicks as lining up and aiming the ball past the keeper proved to be particularly difficult. Out of the 9 games that were

[2] `http://www.tzi.de/spl/bin/view/Website/Results2009`

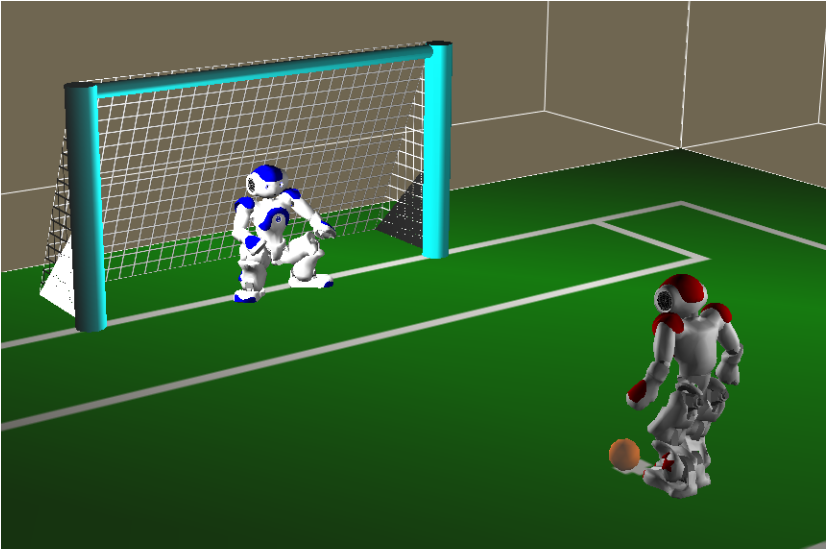


Fig. 6.2. The experimental setup of the *Penalty Kick* task in the Webots simulator, with the robot learning to aim its kick past the keeper to score penalty kicks.

decided by penalty kicks (the rest were left as a draw), only 3 had goals scored during the best of five penalty kicks. In total, there were only 7 goals scored in 90 penalty kick attempts, resulting in a low scoring percentage of 7.8%.

Our goal is to have the robot learn how to score penalty goals against a typical stationary keeper. We set up the experiments with the ball on the penalty mark 1.8 meters from the goal as specified in the SPL penalty kick rules (Röfer et al., 2009). The robot is placed facing the goal with the center of its feet 15 cm behind the penalty mark. In an actual penalty kick, the robot starts at midfield, but here we are strictly trying to learn to aim the kick and we assume the robot has walked up to ball. The keeper is placed in a crouched position in the center of the goal. Every episode begins with this exact setup, shown in Figure 6.2.

For each episode, the agent starts our normal kick engine (Hester et al., 2009), standing on its right leg and looking down at the ball. The learning algorithm then controls the free left leg with three available actions: MOVE-OUT, MOVE-IN, and KICK. The robot's state consists of two state features: the X coordinate of the ball in the robot's camera image and the distance the free foot is shifted out from the robot's hip in millimeters (FOOT-SHIFT), demonstrated in Figure 6.3. Each feature is discretized: the ball's image coordinate is discretized into bins of two pixels each, while the leg distance is discretized in 4 millimeter bins. The MOVE-OUT and MOVE-IN actions each moved the leg 4 mm in or out from the robot's body. The agent receives a reward of -1 for each action moving the leg in or out and -20 if the action causes the robot to fall over (by shifting its leg too far in either direction). When kicking, the agent receives a reward of $+20$ if it scores a goal, and -2 if it does not. The agent's goal is to learn exactly how

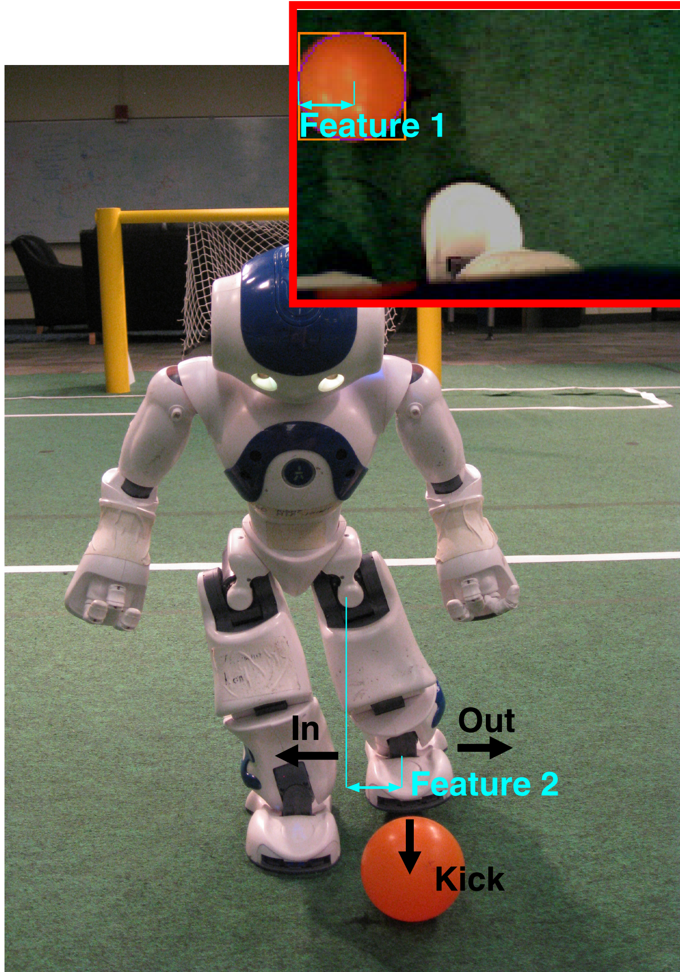


Fig. 6.3. For the *Penalty Kick* task, the robot's state consisted of the x coordinate of the ball in the robot's camera image and the distance the robot's foot was shifted out from its hip. The actions available to the robot were to move the leg in, out, or kick.

Table 6.2. Properties of the *Penalty Kick* task

State Features	X, FOOT-SHIFT
Actions	MOVE-IN, MOVE-OUT, KICK
Reward	Ranges from -20 to $+20$
# State-Actions	$3,360$
Time-Constrained Lifetime	$6,720$ actions
Domain Class	*Haystack*

far to shift its leg before kicking such that it will kick the ball at an angle past the keeper. The properties of this *Penalty Kick* task are shown in Table 6.2.

Due to the difficulties and time involved in performing learning experiments on the physical robot, we started by performing experiments in the Webots simulator from Cyberbotics.[3] Webots is a robotics simulator that uses the Open Dynamics Engine (ODE) for physics simulation. It simulates all the joints and sensors of the robot, including the camera. After running experiments in the simulator, we ran experiments on the physical robot to validate our results.

On this task, we compared Q-LEARNING, R-MAX and TEXPLORE-EE. TEXPLORE-EE was run with the threshold $V_{min} = 8.0$. Q-LEARNING was run with ϵ-greedy exploration with $\epsilon = 0.1$ and R-MAX was run with $m = 10$.

On each trial, the ball is placed in a random location relative to the robot to simulate the noisiness of the robot's approach to the ball. We first determined the range of ball locations where it was possible to score from and then randomly placed the ball in this region, which was between 0 and 34 mm or between 74 and 130 mm left of the penalty mark. In this experiment, the robot could use the state feature about the ball's location in its camera image to determine how far it needed to shift its leg to line up the ball properly to score a goal.

Plots of the average cumulative reward for the three algorithms over 30 trials are shown in Figure 6.4. It is possible to score at many positions without shifting the leg and Q-LEARNING performs well by quickly learning to score consistently at these positions. While Q-LEARNING performs well early, its final policy is not as good as that of the other two algorithms. Figure 6.5 shows the percentage of tries that each algorithm scored at each ball position during the final 200 episodes. Q-LEARNING does very well on the positions where there was no leg shift required, but is unable to learn to score on more difficult positions, such as when the ball is offset between 74 and 84 mm. The two model-based methods perform more exploration and learn to score from these positions. TEXPLORE-EE explores fast enough that it is able to accumulate enough reward from its better policy to surpass the cumulative reward of Q-LEARNING and it has a better policy than R-MAX at the end of the 1500 episodes.

Following these experiments, we ran one trial of TEXPLORE-EE on the physical robot. In this case, we manually reset the robot and ball to the correct positions between each episode, and attempted to place the ball at a constant offset of

[3] http://www.cyberbotics.com

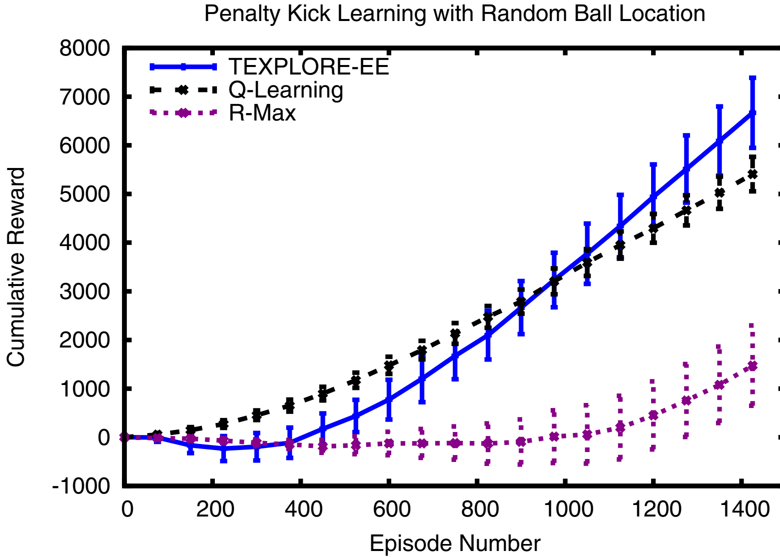


Fig. 6.4. Cumulative reward of the learning agents on the *Penalty Kick* task with a random ball location in the Webots simulator. While Q-LEARNING has success early, TEXPLORE-EE surpasses it and has the highest cumulative reward after 1500 episodes.

30 mm from the penalty spot. The cumulative reward plot for the one physical robot trial is shown in Figure 6.6 along with the cumulative reward averaged over 30 trials of TEXPLORE-EE in the simulator for comparison. The results the algorithm achieves on the real robot are very similar to its performance in the simulator, validating that these experiments do cross over to the physical robot.

The results on both the *Taxi* and *Penalty Kick* domains demonstrate that TEXPLORE-EE can successfully learn on *haystack* domains, which require exploration of each state-action. Although TEXPLORE-EE is exploring each state-action when necessary, it is still able to out-perform R-MAX on both domains.

6.2 Variance and Novelty Intrinsic Rewards

While using explicit exploration and exploitation modes is useful in *haystack* domains, it does not make much sense in *informative* domains. *Informative* domains have informative state features that provide some information and guidance about where unusual states may be. For example, an agent in a grid world could have a wall sensor that detects the distance to a wall, or sensors that tell an agent if it is near an object or goal. Although the agent must discover the meaning and use of these informative features, it could utilize them to explore in a more targeted and intelligent way. In addition to enabling the agent to learn the task more efficiently, such a method could also be useful for providing motivation for a developing curious agent in domains

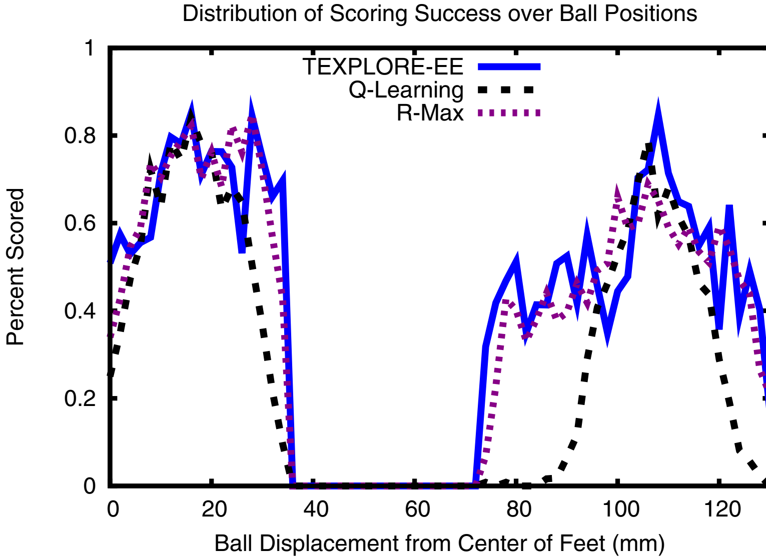


Fig. 6.5. This graph shows the percentage of tries that each algorithm scored at each ball position in the last 200 episodes of the *Penalty Kick* task. Note that the ball's initial position was never between 34 and 74 mm because these locations were impossible to score from.

with little or no external rewards. In this section, we present an extension to TEXPLORE that provides intrinsic rewards to drive exploration in a more targeted way in *informative* domains. This extension is called TEXPLORE with variance and novelty intrinsic rewards, or TEXPLORE-VANIR. TEXPLORE-VANIR has also been publicly released in the same package as the TEXPLORE algorithm: `http://www.ros.org/wiki/rl-texplore-ros-pkg`. After presenting the algorithm in Section 6.2.1, we then present some experiments demonstrating the efficacy of this approach in Section 6.2.2.

6.2.1 Methodology

One approach to driving exploration is to use *intrinsic* rewards to drive the agent to particular states. For example, R-MAX uses intrinsic rewards to drive the agent to state-actions with fewer than m visits. We hypothesize that the best intrinsic rewards to use to improve the efficiency of model-learning are highly dependent on the type of model being learned. With the random forest model TEXPLORE uses, we hypothesize that the following two intrinsic motivations will perform the best: 1) preferring to explore areas of the state space where there is a large degree of uncertainty in the model, and 2) preferring regions of the state space that have the most different state features from previously explored states (regardless of how certain the model is).

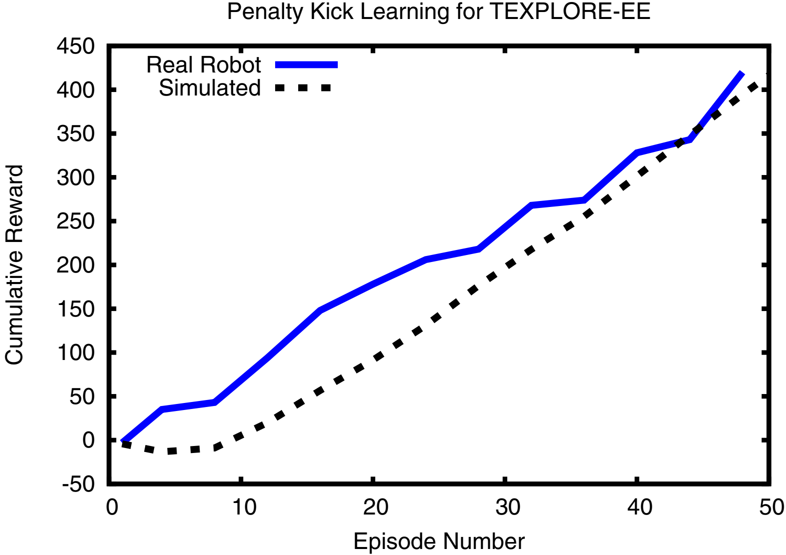


Fig. 6.6. Cumulative reward of TEXPLORE-EE on one trial on the real robot and averaged over 30 trials in the simulator with a standard ball location on the *Penalty Kick* task.

The variance of the predictions of each of the trees in the forest can be used to motivate the agent towards the state-actions where its models disagree, similar to the *query by committee* approach from active learning (Seung et al., 1992). Each tree in the random forest can be considered as a different hypothesis of the true dynamics of the domain. Therefore, the state-actions where the trees' predictions differ are the ones where there are still multiple hypotheses of the true model of the domain. TEXPLORE-VANIR calculates a measure of the variance in the predictions of the change in each state feature for a given state-action:

$$D(s,a) = \sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{m} D_{KL}(P_j(x_i^{rel}|s,a)||P_k(x_i^{rel}|s,a)), \quad (19)$$

where for every pair of models (j and k) in the forest, it sums the KL-divergences between the predicted probability distributions for each feature i. $D(s,a)$ measures how much the predictions of the different models disagree. This measure is different than just measuring where the predictions are noisy, as $D(s,a)$ will be 0 if all the tree models predict the same stochastic outcome distribution. An intrinsic reward proportional to this variance measure, the VARIANCE-REWARD, is incorporated into the agent's model for planning:

$$R(s,a) = vD(s,a), \quad (20)$$

where v is a coefficient determining how big this reward should be. By setting $v < 0$, the agent will avoid states that the model is uncertain about; setting $v > 0$

will result in the agent being driven to explore these uncertain states. If $v = 0$, the agent will act greedily with respect to its model. Changing the parameter v affects how aggressive the agent is in trying to improve uncertainties in its model. This reward can be combined with other rewards (intrinsic or extrinsic) to drive the agent. This reward for variance in the agent's models is similar to the approach taken in the PILCO algorithm (Deisenroth and Rasmussen, 2011) (described in Section 7.1.1), which adds a bonus reward into the model for state-actions where the predictions have the highest variance.

The VARIANCE-REWARD will drive the agent to the state-actions where its models have not yet converged to a single hypothesis of the world's true dynamics. However, there will still be cases where all of the agent's models make incorrect predictions. Therefore, TEXPLORE-VANIR also needs a measure of how likely it is for the model's predictions to be incorrect. For the random forest model that TEXPLORE-VANIR uses, the model is more likely to be incorrect when it has to generalize its predictions farther from the experiences it is trained on. Therefore, TEXPLORE-VANIR utilizes a second intrinsic reward based on the L_1 distance in feature space from a given state-action and the nearest one that the model has been trained on. This distance is calculated separately for each action. For an action a, X_a is the set of all the states where this action was taken. Then, $\delta(s, a)$ is the L_1 distance from the given state s to the nearest state where action a has been taken:

$$\delta(s, a) = \min_{s_x \in X_a} ||s - s_x||_1, \tag{21}$$

where each feature is normalized to range from 0 to 1. A reward proportional to this distance, the NOVELTY-REWARD, drives the agent to explore the state-actions that are the most novel compared to the previously visited state-actions:

$$R(s, a) = n\delta(s, a), \tag{22}$$

where n is a coefficient determining how big this reward should be. One nice property of this reward is that given enough time, it will drive the agent to explore *all* the state-actions in the domain, as any unvisited state-action is different in some feature from the visited ones. However, it will start out driving the agent to explore the state-actions that are the most different from ones it has seen.

The TEXPLORE with Variance-And-Novelty-Intrinsic-Rewards algorithm (TEXPLORE-VANIR) is completed by combining these two intrinsic rewards. Algorithm 6.2 shows TEXPLORE-VANIR's model learning approach, replacing Algorithm 4.1 used by TEXPLORE. TEXPLORE-VANIR's intrinsic rewards can be combined with different weightings of their coefficients (v and n) to drive the agent to both explore novel state-actions where its model may have generalized incorrectly and state-actions where its model is uncertain. A combination of these two intrinsic rewards should drive the agent to learn a model more efficiently, as well as explore in a developing and curious way: seeking out novel and interesting state-actions, while exploring increasingly complex parts of the domain. The next section presents experiments comparing this exploration approach to others on a domain with complex dynamics and rich features.

Algorithm 6.2. TEXPLORE-VANIR's model learning

```
1: procedure INIT-MODEL(n, A)                    ▷ n: Num. state features, A: Num. actions
2:     for i = 1 → n do
3:         featModel_i ⇒ INIT()                  ▷ Init model to predict feature i
4:     end for
5:     rewardModel ⇒ INIT()                      ▷ Init model to predict reward
6:     for i = 1 → A do
7:         X_i ← ∅                               ▷ Init visited state set for each action to ∅
8:     end for
9: end procedure

10: procedure UPDATE-MODEL(list)                 ▷ Update model with list of experiences
11:     for all ⟨s, a, s', r⟩ ∈ list do
12:         s^rel ← s' − s                       ▷ Calculate relative effect
13:         for all s_i^rel ∈ s^rel do
14:             featModel_i ⇒ UPDATE(⟨s, a⟩, s_i^rel)   ▷ Train a model for each feature
15:         end for
16:         rewardModel ⇒ UPDATE(⟨s, a⟩, r)      ▷ Train a model to predict reward
17:         X_a ← X_a ∪ s                        ▷ Add state s to visited set
18:     end for
19: end procedure

20: procedure QUERY-MODEL(s, a)                  ▷ Get prediction of ⟨s', r⟩ for s, a
21:     D ← 0
22:     for i = 1 → LENGTH(s) do
23:         s_i^rel ← featModel_i ⇒ QUERY(⟨s, a⟩)         ▷ Sample a prediction for feature i
24:         D ← D + featModel_i ⇒ CALCD(⟨s, a⟩)           ▷ Sum D for each feature (Eq. 19)
25:     end for
26:     s' ← s + ⟨s_1^rel, ..., s_n^rel⟩         ▷ Get absolute next state
27:     r ← rewardModel ⇒ QUERY(⟨s, a⟩)          ▷ Sample r from distribution
28:     δ ← min_{s_x ∈ X_a} ||s − s_x||_1        ▷ Calculate δ (Eq. 21)
29:     r ← r + vD + nδ                          ▷ Add intrinsic rewards
30:     return ⟨s', r⟩                           ▷ Return sampled next state and reward
31: end procedure
```

6.2.2 Empirical Evaluation

In this section, we evaluate TEXPLORE-VANIR on an *informative* domain under the usual RL framework. In addition, we analyze its benefits for driving a developing curious agent in a domain without external rewards. Rather than attempting to accrue reward on a given task, a curious agent's goal is better stated as preparing itself for any task. We therefore evaluate TEXPLORE-VANIR in four ways on a complex domain with no external rewards. First, we measure the accuracy of the agent's learned model in predicting the domain's transition dynamics. Second, we test whether the learned model can be used to perform tasks in the domain when given a reward function. Third, we examine the agent's exploration to see if it is exploring in a developing, curious way. Finally, we demonstrate

Table 6.3. Properties of the *Light World* domain

State Features	ID, X, Y, KEY, LOCKED, RED-E, RED-W, RED-N, RED-S, GREEN-E, GREEN-W, GREEN-N, GREEN-S, BLUE-E, BLUE-W, BLUE-N, BLUE-S,
Actions	EAST, WEST, NORTH, SOUTH, PRESS, PICKUP
Reward	0 each step, +10 when leaving room
# State-Actions	1,464
Time-Constrained Lifetime	2,928 actions
Domain Class	*Informative*

that TEXPLORE-VANIR can combine its intrinsic rewards with external rewards to learn faster than if it was given only external rewards. These results demonstrate that the intrinsic rewards and model learning approach TEXPLORE-VANIR uses are sufficient for the agent to explore in a developing curious way and to efficiently learn a transition model that is useful for performing tasks in the domain.

The agent is tested on the *Light World* domain (Konidaris and Barto, 2007), shown in Figure 6.7. In this domain, the agent goes through a series of rooms. Each room has a door, a lock, and possibly a key. The agent must go to the lock and press it to open the door, at which point it can then leave the room. It cannot go back through the door in the opposite direction. If a key is present, it must pickup the key before pressing the lock. Open doors, locks, and keys each emit a different color light that the agent can see. The agent has sensors that detect each color light in each cardinal direction. The sensors have a maximal value of 1 when the agent is at the light, and their values decrease linearly to 0 when the light is 20 steps away. The agent's state is made up of 17 different features: its X and Y location in the room, the ID of the room it is in, whether it has the KEY, whether the door is LOCKED, as well as the values of the 12 light sensors, which detect each of the three color lights in the four cardinal directions. The agent can take six possible actions: it can move in each of the four cardinal directions, PRESS the lock, or PICKUP the key. The first four actions are stochastic; they move the agent in the intended direction with probability 0.9 and to either side with probability 0.05 each. The PRESS and PICKUP actions are only effective when the agent is on top of the lock and the key, respectively, and then only with probability 0.9. The agent starts in a random state in the top left room in the domain, and can proceed through the rooms indefinitely. The properties of the domain are presented in Table 6.3.

This domain is well-suited for this task because the domain is *informative* and has complex dynamics. There are simple actions that move the agent, as well as more complex actions (PICKUP and PRESS) that interact with objects in different ways. There is a progression of the complexity of the uses of these two actions. Picking up the key is easier than pressing the lock, as the lock requires the agent to have already picked up the key and not yet unlocked the door.

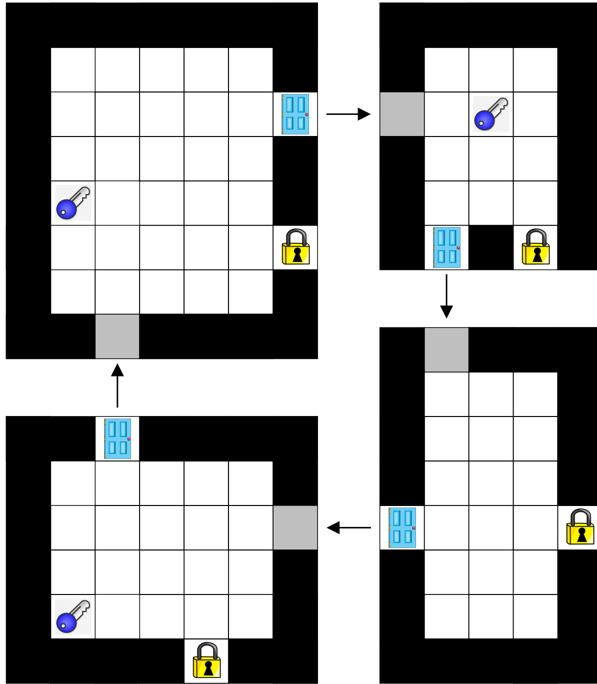


Fig. 6.7. The *Light World* domain. In each room, the agent must navigate to the key, PICKUP they key, navigate to the lock, PRESS it, and then navigate to and exit through the door to the next room.

Based on informal testing, we set TEXPLORE-VANIR's parameters to $v = 1$ and $n = 5$. TEXPLORE-VANIR is tested against the following agents:

1. Agent that selects actions *randomly*
2. Agent that is given an intrinsic motivation for regions with more *competence progress* (based on R-IAC (Baranes and Oudeyer, 2009))
3. Agent that is given an intrinsic motivation for regions with more *prediction errors*
4. Agent that uses R-MAX style rewards (terminal reward of R_{max} for state-actions with fewer than m visits)
5. Agent that acts randomly with a *tabular* model
6. R-MAX algorithm (Brafman and Tennenholtz, 2001).

These six algorithms provide four different ways to explore using TEXPLORE-VANIR's random forest model, as well two approaches using a tabular model. The tabular model is initialized to predict self-transitions for state-actions that have not been visited.

One of the more well-known intrinsic motivation algorithms is Robust Intelligent Adaptive Curiosity (R-IAC) (Baranes and Oudeyer, 2009). R-IAC does not

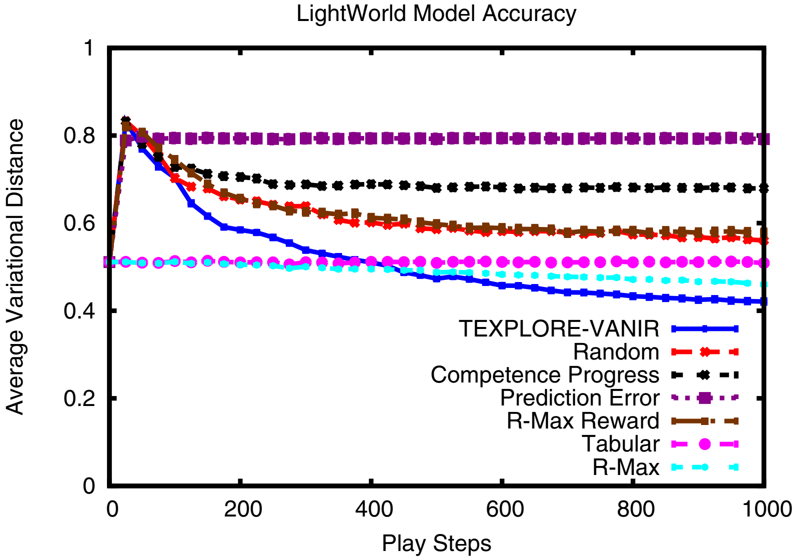


Fig. 6.8. Accuracy of each algorithm's model of the *Light World* domain plotted versus number of steps the agent has taken, averaged over 30 trials and 5000 randomly sampled state-actions. TEXPLORE-VANIR learns the most accurate models.

adopt the RL framework, but is similar in many respects. R-IAC splits the state space into regions and learns a model of the transition dynamics in each region. It maintains an error curve for each region and uses 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 (rewarding *competence progress*). This approach is intended for very large multi-dimensional continuous domains where learning may take many thousands of steps. We have created a method based on this idea to compare with our approach (the *Competence Progress* method). This method splits the state space into random regions at the start, maintains error curves in each region, and provides intrinsic rewards based on competence progress within a region. These intrinsic rewards are combined with the same TEXPLORE model learning approach as the other methods. As another comparison, the *Prediction Error* method uses the same regions, but rewards areas with high prediction error.

All the algorithms are run in the *Light World* domain for 1000 steps without any external reward. During this phase, the agent is free to play and explore in the domain, all the while learning a model of the dynamics of this world. For some of the experiments, a second phase of the experiment is run with external rewards to see if the agent's learned model is useful. All of the algorithms use the RTMBA parallel architecture and take 2.5 actions per second.

First, we examine the accuracy of the agent's learned model. After every 25 steps, 5000 state-actions from the domain are randomly sampled and the

variational distance between the model's predicted next state probabilities are compared with the true next state probabilities. Figure 6.8 shows the variational distance between these distributions, averaged over the 5000 sampled state-actions. This figure shows that TEXPLORE-VANIR learns significantly more accurate models than the other methods ($p < 0.025$). The next best algorithm is R-MAX. However, using R-MAX style reward with the TEXPLORE model strategy is worse than acting randomly. This result illustrates our point that the best intrinsic reward is dependent on the particular model learning approach that is used. The method rewarding visiting regions with high prediction error performs poorly, possibly because it is not visiting the right state-actions within these regions.

While TEXPLORE-VANIR and R-MAX appear to learn fairly accurate models, it is more important for the algorithms to be accurate in the interesting and useful parts of the domain than for them to be accurate about every state-action. Therefore, we next test if the learned models are useful to perform a task. After the algorithms learned models without rewards for 1000 steps, they are provided with a reward function for a task. The task is for the agent to continue moving through the rooms (requiring it to use the keys and locks). The reward function is a reward of 10 for moving from one room to the next, and a reward of 0 for all other actions. In this second phase, the agents act greedily with respect to their previously learned transition models and the given external reward function with *no* intrinsic rewards for 3000 steps.

Figure 6.9 shows the cumulative external reward received by each algorithm over the 3000 steps of the task. Again, TEXPLORE-VANIR performs the best, slightly out-performing R-MAX and significantly out-performing the other methods ($p < 0.001$). Learning an accurate transition model appears to lead to good performance on the task, as both TEXPLORE-VANIR and R-MAX perform well on the task.

Next, the exploration of the TEXPLORE-VANIR agent is examined. In addition to learning an accurate and useful model, we desire the agent to exhibit a developing curiosity. Precisely, the agent should progressively learn more complex skills in the domain, rather than explore randomly or exhaustively. Figures 6.10(a) and 6.10(b) show the cumulative number of times that TEXPLORE-VANIR and the random agent select the PRESS action in various states over 1000 steps in the task with no external rewards, averaged over 30 trials. Comparing the two figures shows that TEXPLORE-VANIR calls the PRESS action many more times than the random agent. Figure 6.10(a) also shows that TEXPLORE-VANIR tries PRESS on objects more often than on random states in the domain. In contrast, Figure 6.10(b) shows that the random agent tries PRESS on arbitrary states more often than it uses it correctly.

Analyzing the exploration of TEXPLORE-VANIR further, Figure 6.10(a) shows that it initially tries PRESS on the key, which is the easiest object to access, then tries it on the lock, and then on the door. The figure also shows that TEXPLORE-VANIR takes longer to learn the correct dynamics of the lock, as it continues to PRESS the lock incorrectly, either without the key or with the door already

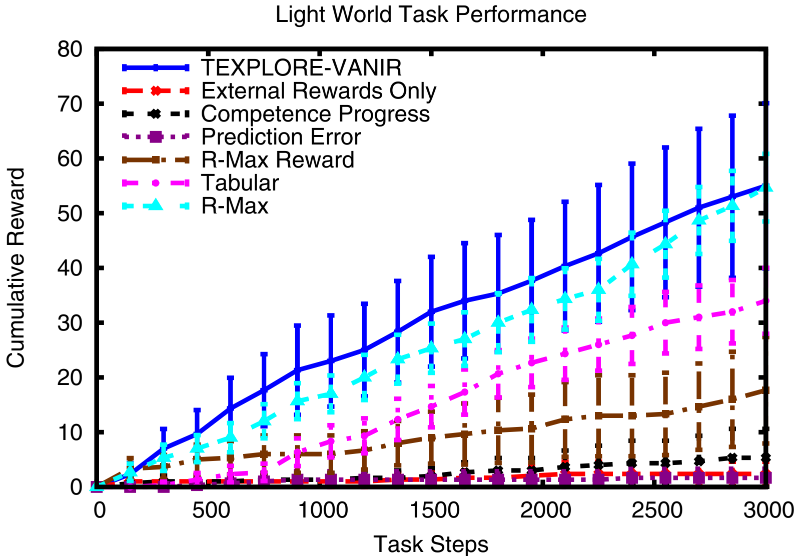
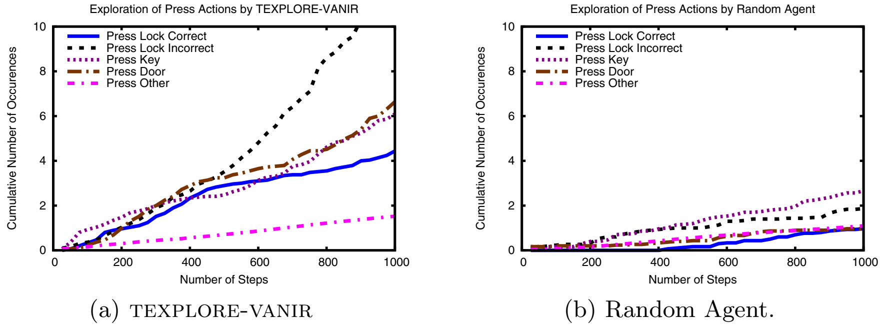


Fig. 6.9. Cumulative rewards received by each algorithm over 3000 steps in the *Light World* domain, averaged over 30 trials. Agents act greedily with respect to their previously learned transition model and the given external reward function. TEXPLORE-VANIR receives the most reward.

unlocked. These plots show that TEXPLORE-VANIR is acting in an intelligent, curious way, trying actions on the objects in order from the easiest to hardest to access, and going back to the lock repeatedly to learn its more complex dynamics.

Finally, not only should the agent's intrinsic rewards be useful when learning in tasks without external rewards, they should also make an agent in a domain with external rewards learn more efficiently. For this experiment, the algorithms are run for 3000 steps with their intrinsic rewards added to the previously used external reward function that rewards moving between rooms. Instead of an agent acting randomly, we instead have one agent acting using only the external rewards, and one performing Boltzmann, or soft-max, exploration with temperature $\tau = 0.2$. Figure 6.11 shows the cumulative external reward received by each agent over the 3000 steps of the task. TEXPLORE-VANIR receives significantly more reward than the other algorithms ($p < 0.001$), followed by R-MAX. Now that exploration and exploitation are no longer separated into separate phases, the exploration of R-MAX is too aggressive and costs it external reward.

These results show that TEXPLORE-VANIR's intrinsic rewards out-perform other exploration approaches and intrinsic motivations combined with the TEXPLORE model. TEXPLORE-VANIR performs similarly to R-MAX when exploration and exploitation are split into separate phases, but out-performs R-MAX significantly when combining intrinsic and external rewards together. TEXPLORE-VANIR explores the domain in a curious, developing manner progressing from



(a) TEXPLORE-VANIR (b) Random Agent.

Fig. 6.10. This plot shows the cumulative number of times that TEXPLORE-VANIR and a Random Agent select the PRESS action in various states over 1000 steps in *Light World* with no external rewards, averaged over 30 trials. Note that the random agent attempts the PRESS action much less than TEXPLORE-VANIR does. TEXPLORE-VANIR starts out trying to PRESS the key, which is the easiest object to find, and eventually does learn to press the lock, but has difficulty learning when to press the lock (it must be with the key but without the door already being open). The agent does not try calling the PRESS action on random states very often. In contrast, the random agent calls PRESS action on random states more often than it calls it correctly on the lock.

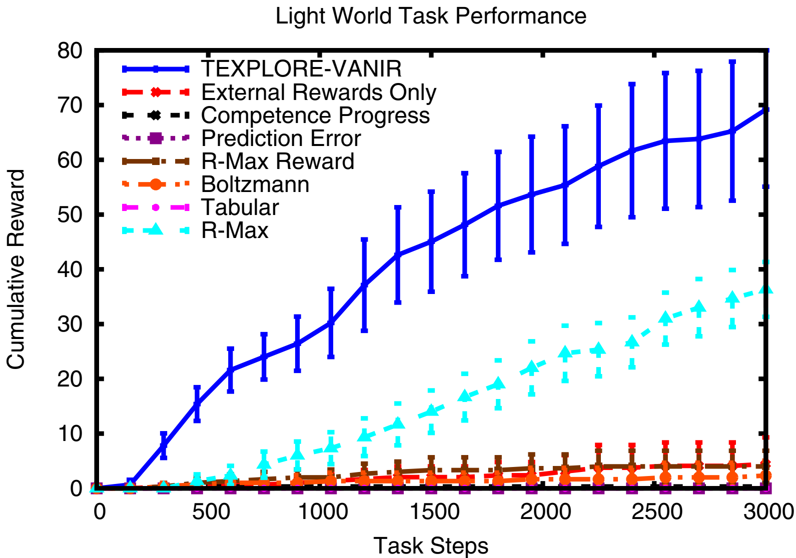


Fig. 6.11. Cumulative rewards received by each algorithm, using intrinsic and external rewards combined, over 3000 steps in the *Light World* domain, averaged over 30 trials. TEXPLORE-VANIR receives the most reward, while the agent using only external rewards performs very poorly.

state-actions with easier dynamics to those that are more difficult. Finally, in a task with external rewards, TEXPLORE-VANIR can use its intrinsic rewards to speed up learning with respect to an algorithm using only external rewards.

It is important to note that the best intrinsic rewards are dependent on the learning algorithm and the domain. For example, the competence progress rewards used by R-IAC are intended to be used in complex high-dimensional domains where learning is slow. It takes quite a few samples in one region to get an reasonable estimate of the derivative of the error. In the *Light World* domain, by the time the algorithm has determined error is improving in a region, the agent has already learned a model of that region and no longer needs to explore there. When using other model learning methods, the best intrinsic reward will vary as well, for example, in these experiments, the R-MAX reward works well for a tabular model, but not for a random forest model.

6.3 On-Line Learning of Exploration Parameters[4]

Both the TEXPLORE-EE algorithm for *haystack* domains and the TEXPLORE-VANIR algorithm for *informative* domains worked well on the desired domains. However, requiring the user to determine the domain type, and to select and tune different exploration parameters for each domain is not desirable. Instead, it would be ideal if the RL agent could learn which exploration strategy is best for a task on-line, while interacting with the task. In this section, we combine the TEXPLORE algorithm with an approach for learning exploration strategies on-line called LEO (Learning Exploration On-line), forming the TEXPLORE-LEO algorithm. We present results showing that TEXPLORE-LEO performs well across a set of tasks where no single exploration strategy performs well across all the tasks.

6.3.1 Methodology

In this section, we present our algorithm, LEO, for learning the best exploration strategies on-line. While it was designed to work with TEXPLORE, it is a general approach that works with any model-based RL method. LEO is given a set of different *exploration strategies* and its goal is to choose the best exploration strategy for each task while interacting with the environment on-line. Since we are concerned with on-line performance of the algorithm, LEO evaluates the performance of each exploration strategy based on the rewards received by the agent while following that strategy. Thus, LEO chooses the exploration strategies that find the rewards and goals the fastest, limiting the costs of exploration by exploring efficiently.

LEO treats each of these exploration strategies like one of the arms in a multi-armed bandit problem (Auer et al., 2000). Pseudo-code for our approach is shown in Algorithm 6.3. Briefly, the agent follows these steps: 1) it selects one of the

[4] This section presents work done jointly with Manuel Lopes.

strategies based on the past payouts received from following it; 2) it follows the selected strategy while tracking the similarity of the other strategies to the one it is following; and 3) at the end of the episode, it updates the expected payouts for each strategy (even the ones not followed). Each step of this process is explained in detail below.

The algorithm is given a set of strategies, E. Each strategy has a weight, w_e, which is an estimate of the expected normalized return for an episode when following that strategy. At the start of each episode, LEO uses these weights to compute a soft-max distribution over the set of strategies, similar to the EXP4 bandit algorithm (Auer et al., 2000):

$$P(e) \leftarrow \frac{e^{\beta(w_e - min(w))}}{\sum_j e^{\beta(w_j - min(w))}}. \tag{23}$$

After calculating this distribution, RUN-EPISODE is called on Line 5. RUN-EPISODE runs the agent through one episode, sampling strategies from this distribution every 10 steps. 10 was chosen through informal experiments, as it was important for the agent to follow a given exploration strategy for multiple steps, but following a bad strategy for an entire episode could greatly impact the agent's performance. At the end of the episode, RUN-EPISODE returns the normalized discounted reward received on the episode and the similarity of each strategy to the followed strategy. This similarity is calculated using importance sampling (Precup et al., 2000; Sutton and Barto, 1998) and is the likelihood of the followed trajectory under this strategy's policy.

After an episode is completed, the estimate of the expected normalized discounted return for each strategy is updated with the following equation on Line 6:

$$w_e \leftarrow w_e + \eta \cdot \frac{sim_e}{\sum_f sim_f}(\hat{J} - w_e). \tag{24}$$

The weight changes are divided between the strategies based on each strategy's proportion of the total similarity, $\frac{sim_e}{\sum_f sim_f}$, so that the sum of the weight changes for all strategies is η, the learning rate. Thus, strategies that were more similar to the followed policy in an episode are moved closer to the return from that episode than strategies that were not similar to the followed policy. These updated weights then affect the new distribution over strategies calculated before the next episode.

Algorithm 6.4 shows what LEO does during an episode. Every 10 steps, the algorithm selects a new strategy from the distribution over strategies (Line 7). Typically, one of these strategies is to act greedily with respect to the learned model of external reward in the task, and the other strategies' policies maximize other intrinsic rewards for exploration. Through informal testing, we found that strictly following any one of these exploration strategies can lead to poor performance in the task, as they are followed even if they contradict knowledge of the external rewards in the task. Thus, the algorithm plans a separate execution policy, π_x, on Line 10. This execution policy combines exploration and exploitation by maximizing both the intrinsic rewards of the selected strategy e as well

Algorithm 6.3. Learning Exploration On-line (LEO)

```
1: Input: E                                              ▷ Set of strategies E
2: w_e ← 1.0, ∀e ∈ E                                     ▷ Initialize strategy weights
3: loop                                                  ▷ Loop over episodes
4:     P(e) ← e^{β(w_e − min(w))} / Σ_j e^{β(w_j − min(w))}   ▷ Dist. over strategies
5:     sim, Ĵ ← RUN-EPISODE(P(e))
6:     w_e ← w_e + η · (sim_e / Σ_f sim_f)(Ĵ − w_e), ∀e ∈ E
7: end loop
```

Algorithm 6.4. LEO: RUN-EPISODE($P(e)$)

```
1: Input: S, A, E              ▷ S: state space, A: action space, E: set of strategies
2: i ← 0.0
3: sim_e ← 1.0, ∀e ∈ E                                   ▷ Reset strategy weights
4: J ← 0.0                                               ▷ Discounted return
5: while Episode Not Over do
6:     if i MOD 10 = 0 then
7:         Sample strategy b from P(e ∈ E)
8:     end if
9:     π_e ← PLAN-POLICY(e), ∀e ∈ E
10:    π_x ← PLAN-POLICY(b + task)                       ▷ Plan exec. pol.
11:    Sample action a from π_x(s, a ∈ A)
12:    Take action a, observe r, s'
13:    M ← UPDATE-MODEL(M ⟨s, a, s', r⟩)
14:    sim_e ← sim_e * π_e(s, a), ∀e ∈ E                 ▷ Update sim.
15:    s ← s'
16:    J ← J + γ^i * r
17:    i ← i + 1
18: end while
19: Ĵ ← (J − r_min/(1−γ)) / (r_max/(1−γ) − r_min/(1−γ))  ▷ Calculate normalized return
20: return sim, Ĵ
```

as the model of task rewards in the domain. The task reward is added in at this phase rather than into each exploration strategy itself so that each exploration strategy remains independent for similarity calculations.

While a particular strategy is being followed, the algorithm tracks the similarity of *all* the strategies, so that their weights can be updated even if they were not selected. Updating values of policies that are not being followed is called *off-policy* learning, and LEO uses a version of importance sampling to address this problem (Sutton and Barto, 1998; Precup et al., 2000). To track the similarity of the other strategies, at every step, a separate soft-max policy is planned for each exploration strategy with the call to PLAN-POLICY on Line 9. When an action is taken in the domain, each strategy's similarity is updated by the probability that it would have taken the selected action, $\pi_e(s, a)$:

$$sim_e \leftarrow sim_e * \pi_e(s,a). \tag{25}$$

Thus, at the end of the episode, the algorithm has a similarity of each strategy's policy to the policy that was actually followed by the agent.

Throughout the episode, LEO tracks the discounted reward, J, that the agent has received. At the end of the episode, it calculates a normalized return $\hat{J}$, where the minimum possible discounted return in the domain is 0 and the maximum possible discounted return in the domain is 1:

$$\hat{J} \leftarrow \frac{J - \frac{r_{min}}{1-\gamma}}{\frac{r_{max}}{1-\gamma} - \frac{r_{min}}{1-\gamma}}. \tag{26}$$

This normalized return is calculated so that the return has some meaning for how well the agent performed across tasks. It is then returned to Algorithm 6.3 and used to update the weights of the strategies.

6.3.2 Empirical Evaluation

In this section, we evaluate LEO in comparison with pre-defined exploration strategies combined with the TEXPLORE algorithm across a set of domains. While a hand-picked exploration strategy can perform well on one domain, the domains were selected so that it would be difficult to find one exploration strategy that was the best across all domains. In addition, finding the best strategy even for a single domain can require a lot of hand-tuning, whereas LEO self-tunes on-line automatically.

Exploration Strategies. We evaluate the LEO algorithm combined with TEXPLORE's model learning and planning approaches, forming the TEXPLORE-LEO algorithm. For our experiments, TEXPLORE-LEO is given the following strategies:

1. Maximize model of task reward
2. Use VARIANCE intrinsic reward
3. Use NOVELTY intrinsic reward
4. Reward exploring UNVISITED state-actions
5. Reward maximizing/minimizing individual state features.

The first strategy is to maximize the model of the task reward, which is a purely exploitative policy. This strategy is what is employed by TEXPLORE, and was presented in Chapter 4. The inclusion of this strategy enables the agent to learn the exploration-exploitation trade-off on-line, as it can choose to take an exploitative strategy. The second and third strategies are the two from the TEXPLORE-VANIR algorithm presented in Section 6.2. The fourth strategy is similar to the exploration performed in the explicit exploration mode of the algorithm presented in Section 6.1. This strategy provides intrinsic rewards for any state-actions that the agent has not visited yet. A parameter, u, defines how much reward is given to unvisited state-actions. Finally, we give the agent strategies that reward or punish particular state features. For example, the agent's reward may be the

value of the first state feature, encouraging the agent to maximize this feature, or it could be the negative value of the first feature, encouraging the agent to minimize this feature. For the number of state features in the domain, n, there will be $2n + 4$ strategies: n strategies that maximize the value of each feature, n strategies that minimize the value of each feature, and the first 4 strategies presented above.

We compared against TEXPLORE-VANIR using six static parameterizations of the VARIANCE, NOVELTY, and UNVISITED exploration strategies:

1. Greedy $(v = 0, n = 0, u = 0)$
2. VARIANCE only $(v = 5, n = 0, u = 0)$
3. NOVELTY only $(v = 0, n = 5, u = 0)$
4. UNVISITED only $(v = 0, n = 0, u = 5)$
5. LOW V-N $(v = 5, n = 5, u = 0)$
6. HIGH V-N $(v = 80, n = 80, u = 0)$.

These six options give us a variety of exploration strategies that were shown to work well in Sections 5.1, 6.1.2, and 6.2.2. There are three versions that are only using a single exploration strategy (Num. 2-4), one using no exploration (Num. 1), and two that combine the VARIANCE and NOVELTY strategies with different weights compared to the task reward (Num. 5 and 6).

Domains. We evaluated our algorithm over a set of four domains. We chose a set of domains where no single exploration strategy should perform well across all domains. Rather than hand-tuning the best exploration strategy for each domain, our algorithm can learn the best strategy in each domain on-line without any parameter tuning. We expect that while using the best strategy for one domain will perform better than TEXPLORE-LEO on that domain, none of the individual strategies will perform well across all four domains.

The first task we tested is *Fuel World*, presented earlier in Section 5.1.2. Since the agent is given example transitions of this task, it is a *prior information* domain. This domain has fuel stations of varying costs on the top and bottom rows of the grid world. As the agent explores some of the fuel stations, each of its trees may make different hypotheses about this cost function. Therefore, we hypothesize that the VARIANCE exploration will be the best, although its unclear how the VARIANCE reward should be weighted relative to exploiting the task reward.

The second domain is an example of an *informative* domain. It is a modification of the *Light World* domain (Konidaris and Barto, 2007) presented in Section 6.2.2. We modified the domain to be episodic, with the episode terminating as soon as the agent left the first room. We also slightly modified the reward function for this task, providing the agent with a reward of -1 each step until it successfully terminates the episode, at which point it receives a reward of $+10$. Since each of the objects in this domain has a related sensor feature, we hypothesize that a few different exploration strategies will work well on this task. Strategies that reward higher sensor features may help drive the agent to

Table 6.4. Properties of the *Sensor Goal* and *Arbitrary Goal* domains

State Features (*Sensor Goal*)	X, Y, SENSE-N, SENSE-E, SENSE-S, SENSE-W
State Features (*Arbitrary Goal*)	X, Y, GOAL-ID
Actions	EAST, WEST, NORTH, SOUTH
Reward	-1 each step, $+2$ upon reaching goal
# State-Actions	$58,564$
Time-Constrained Lifetime	$117,128$ actions
Domain Class (*Sensor Goal*)	*Informative*
Domain Class (*Arbitrary Goal*)	*Haystack*

the correct objects, and in Section 6.2.2, we showed that strategies that utilize the NOVELTY reward promote useful exploration.

The last two domains are similar in nature, but they represent two different classes of domains and thus the best exploration for each of them varies. In both domains, the agent is in a 11 by 12 grid world. It can navigate through the grid with the usual actions: NORTH, SOUTH, EAST, and WEST, each of which move the agent in the desired direction with probability 0.8 and in either perpendicular direction with probability 0.1. In this task, there is a goal state that is in a *different* random location each episode. Essentially, each new episode is a new exploration problem for the agent. It can use what it has learned from past episodes about *which* exploration strategies are the best, but none of its knowledge about the locations of the goal in the previous episodes translate to the current episode. The agent receives a reward of -1 each step until reaching the goal state, when its episode terminates with a reward of $+2$.

The first version of the domain is an *informative* domain called *Sensor Goal.* In this task, the agent's state is made up of six state features: the agent's X and Y location in the domain, and four sensor features telling it the distance to the goal in each of the four cardinal directions. In this version of the task, both the strategies that reward minimizing these sensor features and the strategy rewarding novel states should be successful.

The second version of the domain is a *haystack* domain called *Arbitrary Goal.* In this domain, the agent has no sensors of the goal's location, but instead has a state feature indicating the version of the domain it is in without providing any information about the goal location. In this version of the domain, the best exploration the agent can do is to visit every state in the domain until it finds the randomly located goal. The properties of both the *Arbitrary Goal* and *Sensor Goal* domains are shown in Table 6.4.

Results. In this section, we show the results for the algorithms across the four domains. Figure 6.12 shows the cumulative rewards accrued by the algorithms over 200 episodes on the *Fuel World* domain. As expected for this task, the best strategy is the VARIANCE only strategy, which drives the agent to explore its various hypotheses about the costs of the fuel stations. The next best strategies are

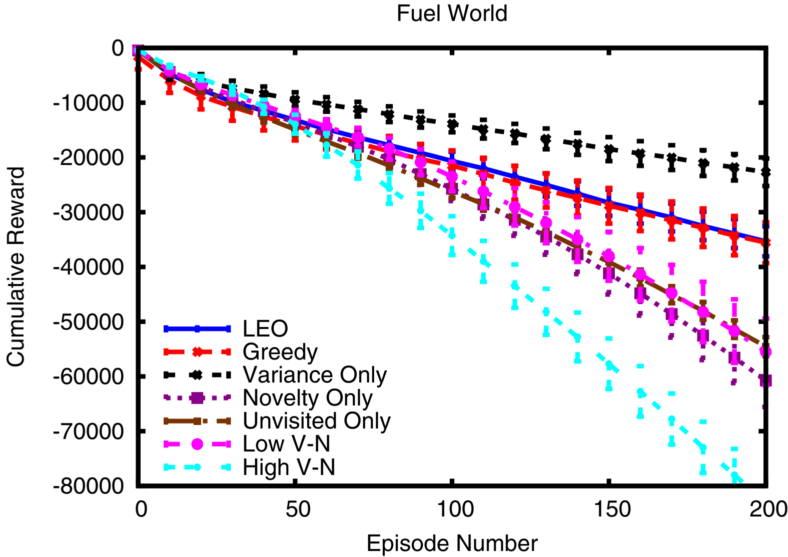


Fig. 6.12. This figure shows the cumulative rewards for the seven exploration strategies on the *Fuel World* domain, averaged over 30 trials. TEXPLORE-LEO performs second best.

LEO and Greedy. All three of these strategies accrue significantly more rewards than the others ($p < 0.001$).

Figure 6.13 shows the weights TEXPLORE-LEO learned for the different strategies over the first 50 episodes. TEXPLORE-LEO learns the highest weight for the model of task reward, followed by the variance strategy, which makes sense as it performed the best on the domain. The third highest weight is on maximizing the FUEL feature, as TEXPLORE-LEO has learned to keep the fuel level high to accrue rewards. It also puts positive weight on the strategy of maximizing the COL feature, which will lead it closer to the goal from its start state.

The cumulative rewards of the algorithms on the second domain, *Light World*, are shown in Figure 6.14. On this task, the HIGH V-N exploration strategy performed the best, followed by NOVELTY only and LEO. While TEXPLORE-LEO does not perform the best on this task or *Fuel World*, comparing Figures 6.12 and 6.14 show that it is the only method to perform well on both domains. The two methods that performed similar to or better than TEXPLORE-LEO on *Fuel World* (TEXPLORE-VANIR with VARIANCE only and Greedy) fail completely on *Light World*, never learning to accomplish the task. Conversely, the two methods that perform similar to or better than TEXPLORE-LEO on *Light World* (TEXPLORE-VANIR with NOVELTY only and HIGH V-N) perform the worst on *Fuel World*. The two domains require completely different exploration strategies, and only TEXPLORE-LEO is able to perform well on both tasks.

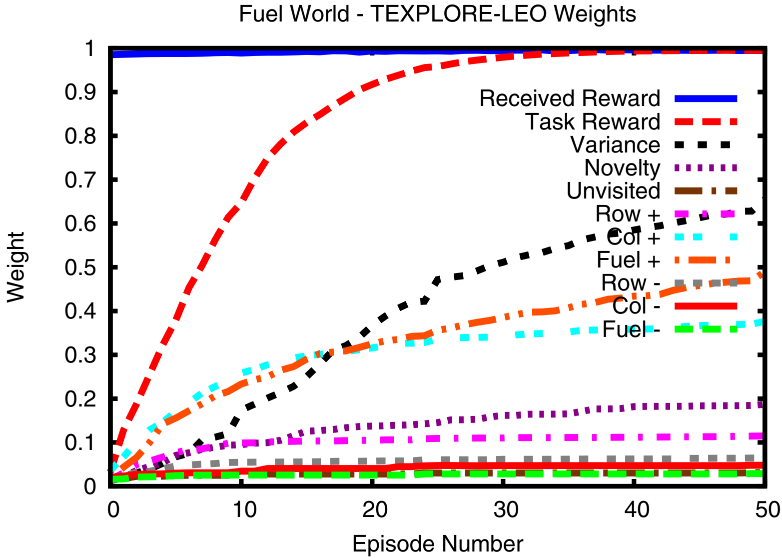


Fig. 6.13. This figure shows the weights learned by the TEXPLORE-LEO algorithm on the *Fuel World* domain, averaged over 30 trials

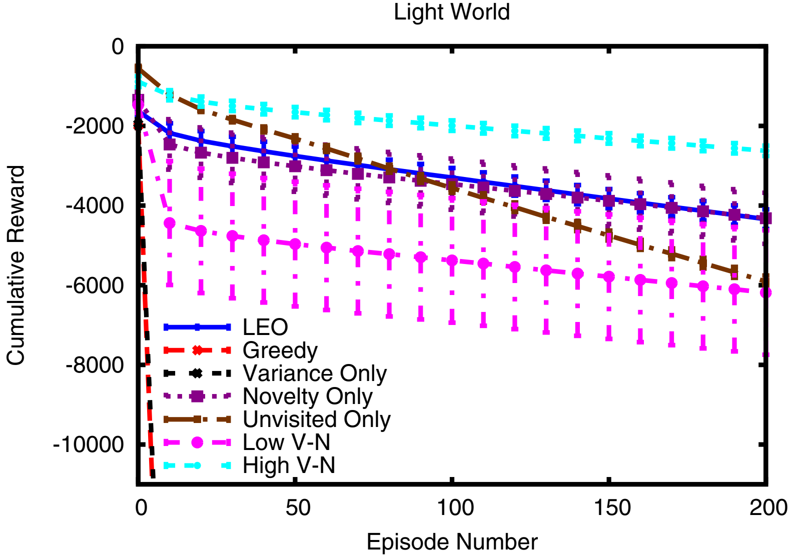


Fig. 6.14. This figure shows the cumulative rewards for the seven exploration strategies on the *Light World* domain, averaged over 30 trials. TEXPLORE-LEO performs reasonably well on this task, while the best algorithms on *Fuel World* fail completely on this task.

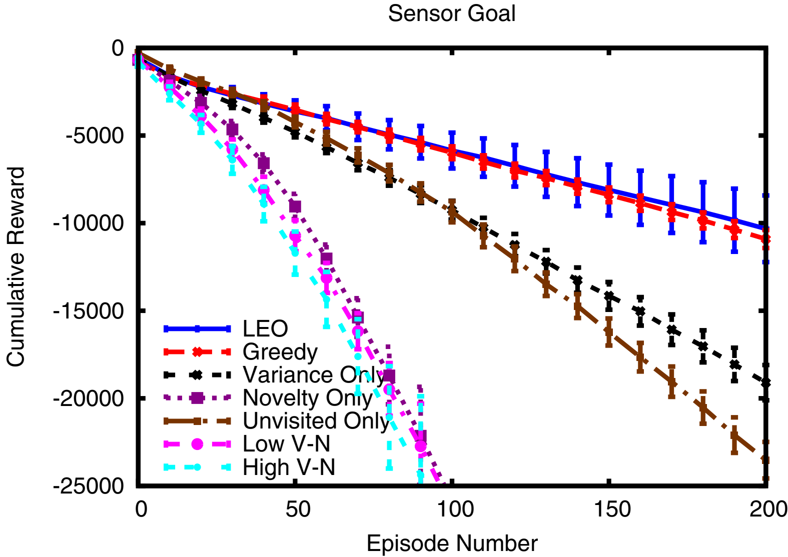


Fig. 6.15. This figure shows the cumulative rewards for the seven exploration strategies on the *Sensor Goal* domain, averaged over 30 trials

Figure 6.15 shows the cumulative rewards on the *Sensor Goal* domain. On this task, TEXPLORE-LEO performs the best, accruing significantly more rewards than the other algorithms ($p < 0.005$). Finally, cumulative rewards for the *Arbitrary Goal* domain are shown in Figure 6.16. As expected, on this task, the best strategy was to explore unvisited states to find the goal (the UNVISITED only strategy). After the UNVISITED only strategy, the NOVELTY only and LOW V-N strategies did well, followed by LEO. While TEXPLORE-LEO is out-performed by these algorithm on this task, none of them did significantly better than TEXPLORE-LEO on the other four tasks.

In addition to cumulative rewards, a successful algorithm should learn good final policies. Table 6.5 shows the average rewards each exploration strategy received on its final five episodes in each task, as well as how that average reward ranked compared with the other six strategies for that task. TEXPLORE-LEO has an average rank of 2.5 on the four domains, as it was the best on the *Sensor Goal* task and second on the *Arbitrary Goal* task. This rank is much better than the ranks of the other algorithms, as the next best method, TEXPLORE-VANIR with the UNVISITED only strategy, has an average rank of 3.5, as it performed very poorly on all the tasks but *Arbitrary Goal.* TEXPLORE-LEO was only significantly out-performed by other algorithms on one domain, *Light World*, where the methods with NOVELTY rewards performed the best. These results demonstrate that TEXPLORE-LEO performs well across a set of different domains requiring various exploration strategies, while none of the other methods perform well across all four domains. Instead, performing well on these domains would require a user to

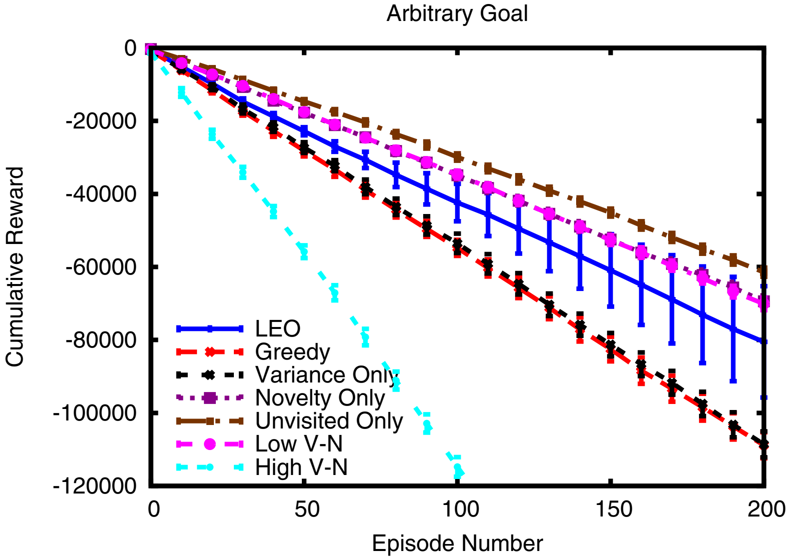


Fig. 6.16. This figure shows the cumulative rewards for the seven exploration strategies on the *Arbitrary Goal* domain, averaged over 30 trials

hand-tune the exploration parameters for each domain. In contrast, TEXPLORE-LEO is more robust, not requiring hand-tuning and capable of learning the best exploration strategy for each domain. In addition, it can adapt its strategy parameters on-line as its model changes.

In this work, our goal was to maximize on-line rewards, and therefore we evaluated the quality of an exploration strategy based on the rewards received while following it. The received rewards indicate how quickly the exploration led the agent to find the rewarding transitions in the domain. While this approach works well in practice, it would be ideal to evaluate an exploration strategy based on the long-term rewards received *after* following it. One challenging possibility for future work is to separate exploration and exploitation, and evaluate exploration

Table 6.5. This table shows the reward each exploration strategy achieved on the final five episodes of each task, averaged over the 5 episodes and 30 trials. * indicates that TEXPLORE-LEO received significantly more rewards than this method ($p < 0.01$) and $^+$ indicates methods that received significantly more rewards than TEXPLORE-LEO ($p < 0.01$). The table also shows the rank of each average reward compared to the other methods for each task.

Domain	LEO		Greedy		VARIANCE only		NOVELTY only		UNVISITED only		LOW V-N only		HIGH V-N only	
	Reward	Rank	Reward	Rank	Reward	Rank	Reward	Rank	Reward	Rank	Reward	Rank	Reward	Rank
Fuel World	-127.4	3	-121.7	2	-86.1	1	-405.6^*	6	-308.1^*	4	-392.3^*	5	-481.8^*	7
Light World	-10.2	4	-1735.6^*	6	-1794.1^*	7	-8.9^+	3	-22.9^*	5	-8.1^+	2	-7.0^+	1
Sensor Goal	-53.1	1	-53.8	2	-98.2^*	3	-406.5^*	6	-140.3^*	4	-408.0^*	7	-159.2^*	5
Arbitrary Goal	-313.5	2	-538.4	5	-548.1	6	-401.7	4	-308.5	1	-323.7	3	-975.7^*	7
Average	-126.1	2.5	-612.4	3.75	-631.6	4.25	-305.7	4.75	-195.0	3.5	-283.0	4.25	-405.9	5.0

strategies by the agent's performance on a later evaluation episode where it exploits the model it learned while exploring. Another alternative is to evaluate the exploration strategies by how much they improve the agent's model accuracy, addressing the pure exploration problem. However, both of these alternatives have an off-line phase; we believe that the approach taken by LEO makes the most sense when the goal is to maximize on-line rewards.

6.4 Empirical Comparison

The results in the previous three sections show that the best exploration strategy depends on the task at hand. In this section we present a few empirical evaluations to analyze when default TEXPLORE performs better or worse than these other methods.

In Section 6.1.2, we showed that TEXPLORE performs poorly on the *Taxi* domain, while TEXPLORE-EE performs well. We hypothesized that this performance was because of the combination of two factors. First, the *Taxi* domain is a *haystack* domain, for which one would ideally want the agent to explore every state-action. Second, the important actions to discover the landmark locations in *Taxi* have high penalties, causing TEXPLORE to stop attempting them. In this section, we look at what happens when we remove either of these two factors.

First, we perform experiments in the exact same *Taxi* domain, but give each algorithm one example transition of picking up a passenger from each of the four landmarks, and one example transition of dropping off a passenger at each of the four locations. These eight example transitions provide the algorithm with information about the location of each landmark, changing it from a *haystack* domain to a *prior information* domain.

We believe that the exploration required for good performance on *prior information* domains like this one is more useful that that needed for *haystack* domains. In *haystack* domains, the best exploration strategy is simply to explore every state-action. This solution is not going to scale up to tasks such as controlling robots, where the agent's lifetime is too short to explore every state-action. In contrast, in *prior information* or *informative* domains, the agent must make decisions about what to generalize, what state-actions to explore, and how to balance the risk and reward of exploration. The exploration required for these types of tasks is more likely to scale up to larger, more complex domains. Since exploring every state-action is often impossible on robotics tasks, Smart and Kaelbling (2002) argue that for RL to be effective on robots, the agent *must* be given prior information about the task. They suggest providing the agent with experiences from a human or human-programmed controller running the robot. Here, we are more conservative, only providing the agent with eight example transitions from the domain.

For these experiments we compare the following seven methods:

1. NO-EXPLORATION (Greedy w.r.t. a single tree model)
2. TEXPLORE (Greedy w.r.t. random forest model)
3. TEXPLORE-EE (Uses a single tree model)

4. TEXPLORE-VANIR (with $v = 5, n = 5$)
5. TEXPLORE-LEO
6. Q-LEARNING
7. R-MAX.

These algorithms enable us to compare the three methods presented in this chapter (TEXPLORE-EE, TEXPLORE-VANIR, and TEXPLORE-LEO) with TEXPLORE and a method performing NO-EXPLORATION. In addition, we compare against Q-LEARNING and R-MAX as representatives of typical model-free and model-based methods. As with the previous experiments on the *Taxi* domain, TEXPLORE-EE is run with $V_{min} = -10.0$, Q-LEARNING is run with ϵ-greedy exploration with $\epsilon = 0.1$ and R-MAX is run with $m = 10$.

The cumulative rewards for each algorithm on the *Taxi* domain when given example transitions of each landmark are shown in Figure 6.17. All of the TEXPLORE based methods significantly out-perform Q-LEARNING and R-MAX ($p < 0.001$). In addition, now that TEXPLORE is given example transitions of the landmarks, it receives significantly more cumulative rewards than TEXPLORE-EE and NO-EXPLORATION ($p < 0.01$).

Second, we performed experiments on the *Two Room* domain, which is exactly the same as the *Delayed Gridworld* domain presented in Section 5.3.2, but the actions take effect immediately. No seed transitions were provided to the algorithms, making it a *haystack* domain. Here, all the actions provide a reward of -1 so there is no incentive for TEXPLORE to stop exploring particular actions. The cumulative

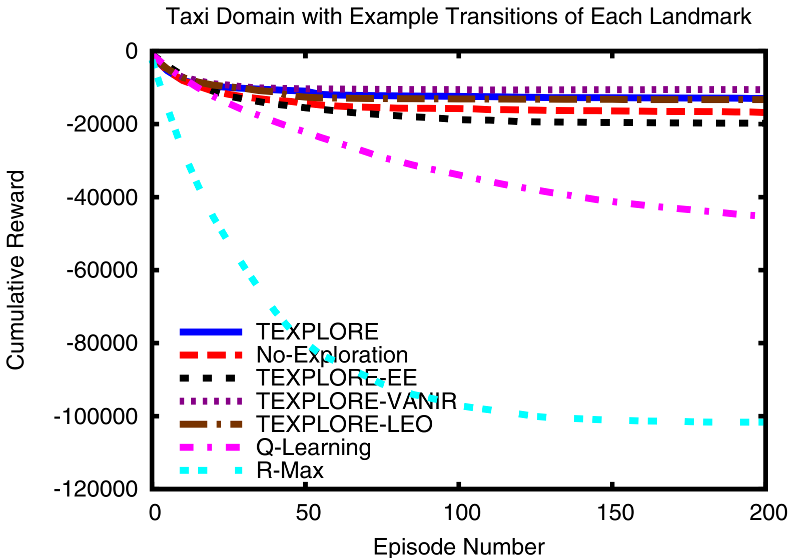


Fig. 6.17. Cumulative reward of the algorithms on the *Taxi* domain when given example transitions of each landmark, averaged over 30 trials. Note that TEXPLORE now receives significantly more reward than TEXPLORE-EE ($p < 0.01$)

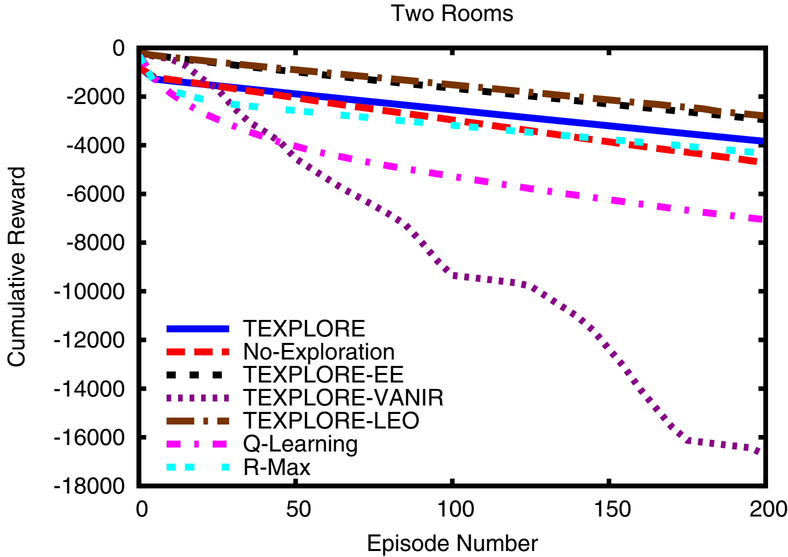


Fig. 6.18. Cumulative reward of the algorithms on the *Two Room* domain, averaged over 30 trials. TEXPLORE-EE performs the best and is closely followed by TEXPLORE-LEO and TEXPLORE.

rewards of each method over 200 episodes on this task are shown in Figure 6.18. TEXPLORE-LEO and TEXPLORE-EE both perform well on this domain, receiving significantly more cumulative rewards than all the other algorithms ($p < 0.01$). However, TEXPLORE also performs well on this task, out-performing the remaining four algorithms. Although this domain requires an algorithm to explore each state-action, TEXPLORE will explore randomly until finding the doorway and goal and is still able to take advantage of its random forest model. These results demonstrate that TEXPLORE can perform well in *haystack* domains. Note that if this domain is made into a *prior information* domain by giving the agent seed transitions of the doorway and goal, TEXPLORE receives more cumulative rewards over the 200 episodes than any of the other algorithms.

Finally, we perform experiments on a completely different domain: the *Stock Trading* domain (Strehl et al., 2007). Unlike *Taxi* and *Two Room*, this domain is non-episodic and not a grid world. It fits none of the three domain classes as there are not individual states with transition or reward function images that are unusual compared to those of the other states. The *Stock Trading* domain consists of e sectors of o stocks each. The values of the stocks are represented by a vector of $e \times o$ boolean variables representing whether each stock is rising or falling. The status of each stock is determined based on the stocks in its sector on the last time step. For a stock in sector i, the probability that it is rising is:

$$P(rising) = 0.1 + 0.8 \frac{\text{Number of sector } i \text{ stocks rising}}{o}. \tag{27}$$

Table 6.6. Properties of the *Stock Trading* domain

State Features	OWN-SEC-1, OWN-SEC-2, OWN-SEC-3, STOCK-1-1 STOCK-1-2, STOCK-2-1, STOCK-2-2, STOCK-3-1, STOCK-3-2
Actions	BUY-SEC-1, SELL-SEC-1, BUY-SEC-2 SELL-SEC-2, BUY-SEC-3, SELL-SEC-3
Reward	Ranges from -6 to $+6$
# State-Actions	$3,072$
Time-Constrained Lifetime	$6,144$ actions
Domain Class	*None*

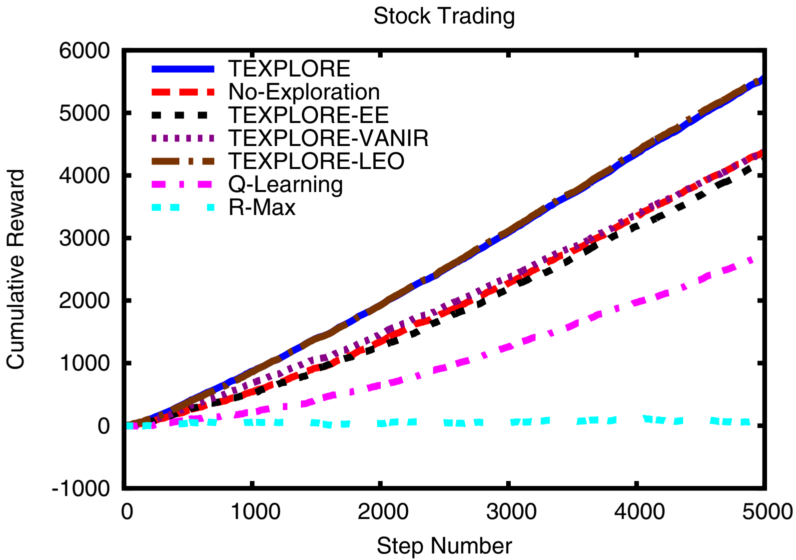


Fig. 6.19. Cumulative reward of the algorithms on the *Stock Trading* domain, averaged over 30 trials. TEXPLORE and TEXPLORE-LEO perform the best.

The agent's reward is determined based on which sectors it owns. For stocks in sectors that it owns, it receives $+1$ reward for stocks that are rising and -1 reward for stocks that are not. It receives no reward from sectors that it does not own. On each time step, the agent can BUY or SELL a sector, or do nothing. For our experiments, we used the same parameters as Strehl et al. (2007), with $e = 3$ sectors and $o = 2$ stocks per sector. These parameters result in 9 boolean state features and a total of 6 actions available to the agent. The properties of this domain are shown in Table 6.6.

For this domain, the V_{min} parameter for TEXPORE-EE is set to 0. The cumulative rewards of the algorithms on the *Stock Trading* domain are shown in Figure 6.19. TEXPLORE and TEXPLORE-LEO receive significantly more reward than the other methods ($p < 0.001$) while not performing significantly differently from each other.

These results show that TEXPLORE works well across many types of domains, and that TEXPLORE-LEO may be the the best approach in general. However, in certain domains, other exploration approaches may perform better. For example, in *haystack* domains where the agent's lifetime is not constrained, it may be best for the agent to explore every individual state-action. Alternatively, in *informative* domains, the agent can take advantage of these features to explore more intelligently.

6.5 Chapter Summary

In this chapter, I examined various approaches to exploration, as good exploration is critical for an agent to learn a task within a time-constrained lifetime. I presented three domain classes: 1) *haystack* domains, where unusual states are located arbitrarily; 2) *prior information* domains, where the agent has some prior information about the locations of unusual states; and 3) *informative* domains, where the agent has access to state features that predict the locations of unusual states.

After extensively examining *prior information* domains in Chapters 4 and 5, in this chapter I examined *haystack* and *informative* domains. I examined an approach for *haystack* domains that explicitly chooses to explore until it has found a sufficiently rewarding policy. However, this approach does not target its exploration in any way, requires the user to specify what a sufficiently rewarding policy is, and may not be effective with a limited agent lifetime. In Section 6.2, I presented intrinsic rewards that could be used to drive the agent's exploration in a more targeted way when acting in *informative* domains. These rewards drive the agent to explore state-actions where its various decision tree models disagree with each other, and that have the most different state features from the transitions its model was trained on. Finally, I presented an approach to learning the best exploration methods on-line through the course of the agent's lifetime.

I then presented empirical comparisons of these approaches on a few domains. These comparisons showed that TEXPLORE performs the best in many domains. However, the best exploration for a given domain differs. *Haystack* domains require the agent to explore every state-action to find arbitrarily located states with unusual transition or reward function images, while *informative* domains provide richer state features that enable the agent to explore more intelligently. Rather than require the user to select from among these strategies, the TEXPLORE-LEO algorithm for selecting exploration strategies on-line works well across the entire set of domains. In the next chapter, I will present work related to each component of TEXPLORE as well as work related to these exploration approaches.

7 Related Work

In this chapter, I present work related to this book, and particularly work related to the TEXPLORE *algorithm.* TEXPLORE *is the first algorithm to address all four of the* RL for Robotics Challenges *challenges together in one algorithm. However, for each individual challenge, there is ample related work. I present related work on each challenge in turn. First, work on sample efficiency, then continuous tasks, then domains with sensor and actuator delays, and finally architectures for real time action selection. In addition, I examine other work on applying RL methods to real-world problems such as robot control, clinical studies, and the environment.*

The TEXPLORE algorithm presented in this book is the first algorithm to address all four of the *RL for Robotics Challenges* together. Since TEXPLORE is addressing many different challenges, there is ample related work. However, to the best of our knowledge, none of the related work simultaneously addresses all four *RL for Robotics Challenges* or is easily adapted to do so. Section 7.1 examines the related work addressing Challenge 1 on sample efficiency and exploration. I look at work addressing Challenge 2 on continuous state spaces in Section 7.2, Challenge 3 on delayed actions and observations in Section 7.3, and Challenge 4 on real time actions in Section 7.4. In Section 7.5, I look at other work on applying RL methods to real-world problems. Finally, I summarize the related work and contrast it with TEXPLORE in Section 7.6.

7.1 Sample Efficiency

In this book, we are focused on time-constrained domains (as defined formally in Section 2.3), where the agent has a limited number of time steps in which to learn the task. Learning on robots and other real-world problems are good examples of time-constrained domains because on these problems taking millions of samples to learn a task can take many real-world hours, days, or weeks. In addition, these samples can be very expensive or dangerous. Thus, in these domains, it is critical that the agent addresses Challenge 1 and is able to learn in very few samples (i.e. it is sample efficient). For model-based methods, sample efficiency is mainly limited by how long it takes the agent to learn an accurate model of the domain. The speed of the model learning is affected by both the model learning approach being used and the exploration of the agent to get the necessary samples to improve its model. Therefore, I start by focusing on exploration methods in Section 7.1.1. In Section 7.1.2, I look at intrinsic motivation for exploration, and

T. Hester, *TEXPLORE: Temporal Difference Reinforcement Learning for Robots and Time-Constrained Domains*, Studies in Computational Intelligence 503,
DOI: 10.1007/978-3-319-01168-4_7, © Springer International Publishing Switzerland 2013

then in Section 7.1.3, I examine Bayesian methods for exploration. Finally, I look at other methods that incorporate generalization into their model learning in Section 7.1.4.

7.1.1 Exploration

One of the benefits of model-based methods is that they are able to plan multi-step exploration trajectories. Exploration is critically important in RL, as the agent's ability to learn a task is dependent on which states and transitions it experiences. In addition, the agent must decide how to balance exploring to improve its knowledge of the world with exploiting what it already knows about the world. Exploring typically costs the agent immediate reward, while exploiting may mean that the agent is not exploring some state that could provide the agent with more reward in the future. In this section, I briefly describe some of the common exploration approaches.

A very common and simple approach to exploration is to occasionally take random actions. One benefit of this type of exploration is that it works with model-free methods, not requiring a model or planning a multi-step trajectory. 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. The two most common random exploration approaches are ϵ-greedy exploration and Boltzmann (or soft-max) exploration (Sutton and Barto, 1998). Agents using ϵ-greedy exploration take what they think are the optimal actions most of the time, but take a random action ϵ of the time. Boltzmann exploration improves upon ϵ-greedy exploration by taking better exploratory actions. 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:

$$P(a) = \frac{e^{Q(a)/\tau}}{\sum_{b=1}^{n} e^{Q(b)/\tau}} \tag{28}$$

where τ is a temperature parameter determining the amount of exploration. While these approaches are simple and are guaranteed to visit every state-action in the limit, they do not generally result in high enough sample efficiency to address Challenge 1 as TEXPLORE does.

The most common approach to exploration for model-based methods is to employ "optimism in the face of uncertainty." The principle here is that the agent assumes that any parts of the world it is unsure about are very good, and therefore it should explore all parts of the world thoroughly so as to not miss out on anything. One approach that applies this principle is R-MAX (Brafman and Tennenholtz, 2001). It splits the state-actions into those that are *known* (visited at least m times) or *unknown* (visited fewer than m times) and adds intrinsic rewards of R_{max} in the model to drive the agent to explore the unknown state-actions. These reward bonuses encourage the agent to explore each state-action until it finds a policy that can reach the maximal one-step reward. R-MAX

is guaranteed to find the optimal policy in time polynomial in the number of states and actions, but this exploration is typically infeasible within a time-constrained lifetime for an agent. ZETA-R-MAX extends R-MAX to classify states as known based on the empirical measure of progress in model learning and provides similar convergence guarantees (Lopes et al., 2012).

Another way to perform optimistic exploration is to follow the approach of Model Based Interval Estimation (MBIE) (Wiering and Schmidhuber, 1998; Strehl and Littman, 2005), which maintains statistical confidence intervals over the transition and reward probabilities in the model, such that transitions that have been sampled more often have tighter distributions around the mean. When selecting actions, the algorithm computes the value function according to the transition 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. This way, the agent is assuming the model is as optimistic as it finds plausible. In contrast with the R-MAX approach, these distributions will converge smoothly to a single likely model, rather than having either optimistic rewards or true rewards. One drawback of these methods is that they can be too optimistic, choosing to explore state-actions because they are unknown, even if they are unlikely to have an effect on the final policy. Thus, these approaches can cause the agent to explore too much to learn within a time-constrained lifetime.

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.

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."

Although all of these methods address the issue of sample efficiency through exploration, most of them explore too much to learn a good policy within the time-constrained lifetime of an agent. In addition, none of these methods address the challenges of acting in real time, handling continuous state, or handling actuator and sensor delays. In contrast, TEXPLORE addresses all four *RL for Robotics Challenges*.

7.1.2 Intrinsic Motivation

Many model-based RL algorithms use "exploration bonus" rewards to drive the agent to explore particular parts of the state space and learn more efficiently. All of these algorithms can be considered as intrinsically motivated algorithms, as they are providing artificial intrinsic rewards during planning to drive exploration. As one example, the reward that R-MAX (Brafman and Tennenholtz, 2001) provides for state-actions that have been visited less than m times is an intrinsic reward. In this section, I review work that is focused on using intrinsic motivation to guide exploration and speed learning.

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

One of the more well-known intrinsic motivation algorithms is Robust Intelligent Adaptive Curiosity (R-IAC) (Oudeyer et al., 2007; Baranes and Oudeyer, 2009). R-IAC does not adopt the RL framework, but is similar in many respects. IAC splits the state space into regions and learns a model of the transition dynamics in each region. It maintains an error curve for each region and uses 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 (rewarding *competence progress*). This approach is intended for very large multi-dimensional continuous domains where learning may take many thousands of steps. One drawback of

this approach is that the intrinsic reward signal may be too slow to be useful. For example, by the time the model shows improvement in prediction errors, it may already have learned to make accurate predictions.

Jonsson and Barto (2007) take a similar approach to TEXPLORE-VANIR, in that they also learn trees to model the domain. Their method learns conditional trees using Bayesian Information Criterion to perform splits. Since having a uniform distribution over input values will provide the best information for making splits in the tree, their method provides intrinsic motivation for actions that would increase the uniformity of the inputs to the tree. This reward only drives local exploration, but does enable the agent to quickly learn accurate models of certain tasks. This work was extended to perform more global exploration by adding options to set each state feature to any possible value (Vigorito and Barto, 2010). The agent selected options to set features to values where it could then take actions to better improve the uniformity of input features to its trees. However, unlike TEXPLORE, this approach assumes that the agent can set each feature of the domain independently and learn options to do so.

Singh et al. (2005) present an approach to learning a broad set of reusable skills in a playroom domain. They learn option models for a variety of skills and show that the agent progresses from learning easier to more difficult skills. However, the skills the agent is to learn are pre-defined, rather than being entirely intrinsically motivated.

Şimşek and Barto (2006) present an approach for the pure exploration problem, where there is no concern with receiving external rewards. They provide a Q-LEARNING agent (Watkins, 1989) with intrinsic rewards for where its value function is most improving. This reward speeds up the agent's learning of the true task. However, such a reward to make the agent perform more value backups on its value function is not necessary for model-based algorithms like TEXPLORE, which can perform all the necessary backups using their models without having to re-visit each state-action. Stout and Barto (2010) extend this work to the case where the agent is learning multiple tasks and must balance the intrinsic rewards that promote the learning of each skill.

Singh et al. (2010) present an interesting perspective on intrinsically motivated learning. They argue that in nature, intrinsic rewards come from evolution and exist to help us perform any task. Agents using intrinsic rewards combined with external rewards should be able to perform better on tasks than those using solely external rewards. For two different algorithms and tasks, they search over a broad set of possible task and agent specific intrinsic rewards and find rewards that make the agent learn faster than if it solely used external rewards.

The Policy Gradient Reward Design algorithm (PGRD) learns the best intrinsic rewards on-line for cases where the true reward function is given and the agent is *limited* in some way (Sorg et al., 2011; Bratman et al., 2012). PGRD uses its knowledge of the true reward function to calculate the gradient of intrinsic rewards to agent return. Using this gradient, intrinsic rewards are found that enable the best agent performance given its limitations. For example, if the agent has a limited planning depth, then even with the true reward function, it cannot

perform well. However, good intrinsic rewards can make up for this deficiency. This work does not apply to agents without limitations, as providing the agent with the reward function effectively solves the problem. Similarly, in tasks where the reward function is not given, then the gradient cannot be calculated and this method does not work. PGRD addresses a different problem from the LEO algorithm presented in Section 6.3 as it is given the true reward function and is only useful when the agent is limited.

Reward shaping algorithms are another set of approaches that use intrinsic rewards. These methods provide the agent with intrinsic rewards for improving performance rather than only providing the agent external rewards when the goal has been achieved. These shaping rewards are intended to improve the learning speed of the agent. Shaping rewards have been used to enable RL agents to learn to ride a bicycle (Randløv and Alstrøm, 1998) and speed up learning on gridworld tasks (Ng et al., 1999). Typically, the shaping rewards are created heuristically by the user based on their knowledge of the domain (Sam Devlin and Kudenko, 2011). These methods affect the agent's exploration in the domain to speed up learning, but they typically require the user to have specific knowledge about what constitutes improvement in the task. In addition, if shaping rewards are used that are not *potential-based*, they can cause the agent to learn sub-optimal policies, diverging into cycles that receive lots of shaping reward without accruing any external reward (Ng et al., 1999).

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.

As we demonstrated empirically in Section 6.2.2, the correct intrinsic motivation is dependent on the type of algorithm. For example, with a Q-LEARNING agent (Watkins, 1989), it makes sense to give intrinsic rewards for where the value backups will have the largest effect, as done in (Şimşek and Barto, 2006). When learning with a tabular model, the agent must gain enough experiences in each state-action to learn an accurate model of it. Thus it makes sense to use intrinsic motivation to drive the agent to acquire these experiences, as done by R-MAX (Brafman and Tennenholtz, 2001). When using a model learning approach that generalizes as TEXPLORE's does, the best intrinsic rewards are different again.

7.1.3 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 parameterization 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 approach 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 parameterizing 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, while TEXPLORE can select actions in real time, PILCO 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. We evaluated both Bayesian DP (Strens, 2000) and Best of Sampled Set (BOSS) (Asmuth et al., 2009) in Section 5.1. Both algorithms 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 4.2, 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 4.2. 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 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. In contrast, the random forest model used by TEXPLORE avoids these problems, while still providing multiple decision tree models that can be used similar to the Bayesian model samples for driving exploration.

7.1.4 Models

One of the ways that TEXPLORE is able to learn models efficiently is by incorporating generalization into its model learning. There are some other examples of algorithms that take the same approach. For example, a few of the methods from Sections 7.1.1 and 7.1.3 incorporate generalization into their model learning. MET-R-MAX (Diuk et al., 2009) and LSE-R-MAX (Chakraborty and Stone, 2011) both take an R-MAX approach and apply it to factored models, enabling their models to generalize over different state features. The PILCO algorithm (Deisenroth and Rasmussen, 2011) learns a Gaussian Process regression model of the domain.

The SPITI algorithm (Degris et al., 2006) is similar to TEXPLORE as it also uses decision trees to learn models of the domain. The SPITI model differs in three major ways. First, SPITI models absolute rather than relative transitions which often makes it more difficult to generalize the effects of actions across states. Second, SPITI explores using ϵ-greedy exploration on a single three model, while TEXPLORE acts greedily with respect to a random forest model. Thus, TEXPLORE can explore in a more targeted way by comparing the trees in its forest,

while SPITI is exploring randomly. Finally, SPITI uses a traditional sequential architecture, meaning it cannot act in real-time as TEXPLORE does.

While TEXPLORE uses random forests of decision trees to learn models that represent multiple hypotheses of the domain, there are other ensemble methods that could work as well. One particularly interesting approach is the DECORATE algorithm (Melville and Mooney, 2003). This algorithm explicitly tries to maximize the diversity of predictions in its ensemble of learners. Essentially, additional training experiences are added to make at least one learner make a different prediction from the rest of the ensemble for unseen instances. Building such a model with more diverse predictions of the true dynamics of the world might be useful to drive exploration more efficiently.

While TEXPLORE applies decision trees to approximate the transition and reward functions, there are a few methods that apply similar techniques to approximating the value function in a model-free algorithm. The G algorithm (Chapman and Kaelbling, 1991) learns a value function using decision trees. Munos and Moore (2002) use kd-trees to adaptively approximate the value function. Similarly, Whiteson et al. (2007) use adaptive tile coding to represent the value function. Both of these methods are similar to decision trees, starting with a broad generalization and refining it over time. However, unlike TEXPLORE, they are approximating the value function rather than the model. By learning a model, TEXPLORE is able to learn more efficiently and plan multi-step exploration trajectories using its model.

All of the algorithms presented in this section address the issue of sample efficiency, either through exploration, intrinsic motivation, Bayesian approaches, or through model approximation. However, very few of these methods would be able to learn a good policy within the time-constrained lifetime of an agent. In addition, while TEXPLORE addresses all four *RL for Robotics Challenges*, very few of these methods address any of the other three challenges. Only the model-free methods are able to act in real-time, only a few of the algorithms such as PILCO handle continuous state, and none of the methods handle actuator and sensor delays.

7.2 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, addressing Challenge 2. 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 parameterized, 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. Learning models of the domain enables TEXPLORE to plan multi-step exploration trajectories and learn in a small number of samples.

7.3 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). Handling these delays addresses Challenge 3.

Walsh et al. (2009a) develop a method called Model Based Simulation (MBS) for delayed domains, which we evaluated empirically in Section 5.3. 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 approach 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. In contrast, TEXPLORE's model can learn the delay in the domain and select actions accordingly.

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 it is intended for POMDPs rather than domains with delay. MC-AIXI uses 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 to perform motor control under delay. TEXPLORE provides its models with a history of previous actions, and lets the model determine which delayed input is the correct one to use for predictions. Similarly, in the cerebellum, different fibers provide signals that have been delayed by different amounts. The cerebellum then uses these delayed inputs to determine the correct control outputs (Ohyama et al., 2003).

7.4 Real-Time Architectures

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. These modifications improve 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.

Silver et al. (2012) present a method very similar to our modified version of UCT(λ) called TD SEARCH. This approach combines UCT with eligibility traces, like our method, and additionally utilizes value function approximation. They demonstrate their algorithm on the task of computer Go.

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 state's 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. We demonstrated empirically in Section 5.3 that RTDP is not as effective as the version of UCT used by TEXPLORE.

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.

7.5 Real-World Problem Domains

One of the goals of this work is to develop an RL algorithm that is capable of working on more real-world problems, where sample efficiency and real-time actions are an issue. In particular, our focus is on the problem of controlling robots. There have been other methods that addressed some robot control problems, but relatively few considering the seemingly natural match between RL and robotics.

Ng et al. (2003) used a reinforcement learning approach to learn to control a model helicopter. First, they collected data from the helicopter while it was being controlled by an expert pilot and used this data to learn a model of the dynamics. Then they used the PEGASUS policy search method (Ng and Jordan, 2000) to learn policies on this model. While this approach was a great success, it did require a human expert to gather the right training experiences. In addition, this approach does not meet Challenge 4 of acting continually in real-time, as computation was performed off-line

Similar to PEGASUS, other policy search methods have proven to perform well on robotics tasks such as maximizing the power output of a micro wind turbine (Kolter et al., 2012) or having a robot arm perform a ball-in-a-cup task (Kober and Peters, 2011). These methods utilize a parameterized policy. After every episode, the gradient of reward with respect to the policy parameters is calculated and new parameters are calculated. With a good parameterization, a good policy can be learned in few samples. However, these methods require the user to create the policy parameterization and also do not act continually in real-time, as they 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.

Kolter et al. (2010) present an approach for learning to control an autonomous vehicle in extreme situations. Their algorithm is given two models: one fairly simple model that can be used for planning in normal situations, but may fail in more extreme scenarios; and one trajectory of an expert controlling the vehicle at the limits of its control. Their algorithm balances the benefits of the two

models, using each in the appropriate part of the domain, to perform extreme maneuvers on an autonomous car. Unlike our work, in this work the algorithm is given these two models and computation is performed off-line.

In addition to robots, RL has been applied to computer games such as backgammon (Tesauro, 1995) and Go (Silver et al., 2012). One advantage of performing RL on games is that the rules of play are known, and the agent can simulate many games against itself for planning purposes. Performing many planning style rollouts combined with temporal difference updates and value function approximation has proven to be successful in both backgammon and Go.

Clinical studies are another area where RL has been applied. For example, RL has been used to learn a controller for deep brain stimulation of patients with epilepsy (Guez et al., 2008) and to optimize treatment policies in clinical decision making (Shortreed et al., 2011). In both of these works, data was collected ahead of time through experiments or clinical trials. FITTED-Q-ITERATION was then applied to this data to learn new policies. Improvement is shown using the collected data, but no real-world evaluations of these new policies are made. Unlike our work, there is no opportunity for exploration here, as the data has already been collected. There are also no computational constraints as the policy calculations can be performed off-line.

RL has been applied to some environmental problems as well, such as deciding which actions to take in maintaining a forest (Crowley and Poole, 2011), or managing the populations of interacting endangered species (Chades et al., 2007). In both of these works, the authors start out with a simulator of the respective domain, given by domain experts. The simulator is then used by the RL algorithm to learn a new policy. Similar to the clinical studies above, there is no way to evaluate the new policy other than in the simulator. The time scale of the actions in these domains is often many years, so the real-time aspect that is present in our work is not an issue here.

There are a few general trends in these successful applications of RL to real-world problems. First, nearly all of them are given expert trajectories, a model, or a simulator ahead of time. They are not required to explore the domain on their own or solve the exploration-exploitation trade-off. Secondly, they all perform computation off-line, either once on the collected data or in batch mode. While this approach works for learning specific tasks, to have robots be fully utilized in society, they cannot be stopping every few seconds or minutes to perform batch computations. They will need to learn new tasks and adapt to their environments on-line, while acting in the environment, which is enabled by TEXPLORE.

7.6 Chapter Summary

While there is a large body of work relating to each challenge that TEXPLORE addresses, none of these approaches address all four *RL for Robotics Challenges* together. The PILCO algorithm (Deisenroth and Rasmussen, 2011) probably comes closest as it meets Challenges 1 and 2. PILCO is extremely sample efficient, targets exploration where the model needs improvement, and works on robots with

continuous state spaces. However, it cannot select actions in real-time or handle delays (Challenges 3 and 4).

In contrast, 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 learning. In addition, I have demonstrated that TEXPLORE works well on time-constrained domains and robotic control tasks. In the next chapter, I summarize this book, and look ahead to future work.

8 Discussion and Conclusion

This chapter concludes the book. First I summarize the TEXPLORE *algorithm presented in this book and the book itself. Next, I summarize the contributions of this book. Then, in Section 8.3, I discuss the limitations and applicability of the* TEXPLORE *algorithm and some aspects of the exploration problem. In the following section, I present some directions for future work, before concluding the book in Section 8.5.*

Reinforcement learning (RL) is a method for learning sequential decision making tasks from experience in the environment. RL could be used to make robots more useful in society by enabling them to learn and adapt to their tasks as they act. However, performing RL on robots raises four *RL for Robotics Challenges*:

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 select actions continually in real time.

In this book, I have presented TEXPLORE, the first RL algorithm to address all four of these challenges together in one algorithm. In addition, I have presented thorough empirical results demonstrating that TEXPLORE's approach to each of these challenges is at least as good as the alternatives, and that TEXPLORE's solutions to each challenge mesh together well.

In the next section, I summarize the TEXPLORE algorithm and the book itself. Then I summarize the contributions of this book in Section 8.2. I discuss the limitations of the TEXPLORE algorithm, its applicability and general issues with exploration in Section 8.3. Finally, I present directions for future work in Section 8.4 and conclude in Section 8.5.

8.1 Summary

The main focus of this book was to present the TEXPLORE algorithm. TEXPLORE is an RL algorithm intended for time-constrained domains where the agent has a very limited lifetime compared to the size of the domain. In addition, TEXPLORE addresses the four *RL for Robotics Challenges*: it is sample efficient, acts in

T. Hester, *TEXPLORE: Temporal Difference Reinforcement Learning for Robots and Time-Constrained Domains*, Studies in Computational Intelligence 503,
DOI: 10.1007/978-3-319-01168-4_8, © Springer International Publishing Switzerland 2013

continuous domains, handles sensor and actuator delays, and takes actions in real time.

TEXPLORE is a model-based RL method, meaning it learns a model of the transition and reward dynamics of the domain and then plans a policy on this learned model. In order to learn this model quickly, TEXPLORE uses decision trees to predict the next state and reward given the current state and action. Learning the model with these trees enables TEXPLORE to generalize the effects of actions across states, eliminating the need to explore every individual state-action in the domain.

However, in order to ensure that TEXPLORE learns an accurate enough model of the domain to plan a good policy, it must *explore* the domain. TEXPLORE explores by acting greedily with respect to a random forest model that is an aggregate of many decision trees. Each tree within the forest represents a different hypothesis of the true dynamics of the domain. By acting with respect to this aggregate model, TEXPLORE can naturally trade off between exploration and exploitation, exploring state-actions that look good under some tree models while avoiding others that look bad under other models.

While using a model-based method enables TEXPLORE to learn efficiently in few actions, model-based methods typically take considerable computation time to perform model learning and planning. With our desired goal of performing learning on robots in the world, we require that the algorithm be capable of selecting actions at a fast enough rate to control the robot. Therefore, we developed a real time model-based RL architecture (RTMBA) that parallelizes the model learning, planning, and acting such that the algorithm can select actions at the desired frequency without being constrained by the time taken to perform model updates or plan. In addition, rather than using a planning method such as value iteration, we use Monte Carlo Tree Search, which is an anytime method that focuses its value updates on the states the agent is likely to encounter next.

After presenting the TEXPLORE algorithm in Chapters 3 and 4, I evaluated it empirically in Chapter 5. For each of the *RL for Robotics Challenges*, I compared TEXPLORE's solution with other possible approaches on both a simulated vehicle velocity control task and a second task. In each case, I demonstrated that TEXPLORE's solution performs at least as well as alternative solutions. I also presented experiments demonstrating that TEXPLORE can learn a task that presents all four challenges: learning to control a physical robot in real time while running on-board the robot. TEXPLORE learns to control the velocity of an autonomous vehicle in just two minutes of driving time.

Following these experiments on the TEXPLORE algorithm, I looked deeper into the problem of exploration in Chapter 6. First, I presented three different classes of RL domains. Then, I examined *haystack* domains where states with unusual transition or reward function images are arbitrarily located. In these domains, the best the agent can do is to explore every state-action. I present an extension of TEXPLORE called TEXPLORE-EE for *haystack* domains and demonstrate its efficacy on both *Taxi* and on learning to score penalty kicks using an Aldebaran Nao robot.

Next, I looked at *informative* domains that have state features that predict the locations of unusual states. In these domains, the agent can utilize these more informative state features to perform more intelligent, targeted exploration. I present another extension of TEXPLORE, called TEXPLORE-VANIR, which uses two intrinsic motivations to drive exploration in such domains. In addition to speeding up learning in domains with more complex state features, TEXPLORE-VANIR can also be used to motivate a developing, curious agent in domains without external rewards.

As demonstrated by TEXPLORE-EE and TEXPLORE-VANIR, the best exploration strategy varies depending on the particular domain the agent is acting in. In Chapter 6, I also presented a method called LEO for learning the best exploration strategy from a given set of strategies on-line, while acting in the domain. I show that the combination of this method with TEXPLORE (the TEXPLORE-LEO algorithm) works well across a set of domains, while no single exploration strategy performs well across all four domains. Finally, at the end of Chapter 6, I empirically evaluate these three exploration extensions in comparison with TEXPLORE on a set of domains, showing that TEXPLORE and its extension, TEXPLORE-LEO, are the best algorithms for many domains.

After presenting the TEXPLORE algorithm, empirical evaluations, and exploration extensions, I discussed related work in Chapter 7. For each of the *RL for Robotics Challenges*, I present related work addressing that particular challenge. In addition, I looked at other RL algorithms focused on addressing robotics and other real world problems.

8.2 Contributions

This book provides the following six major contributions to the field:

1. TEXPLORE: The TEXPLORE algorithm, which is the first algorithm to address all four of the *RL for Robotics Challenges* together simultaneously in the same algorithm. In addition, TEXPLORE is effective at learning good policies and accruing high rewards on time-constrained domains. The TEXPLORE algorithm is not only presented in this book, but has been publicly released as an open-source ROS package at: `http://www.ros.org/wiki/rl-texplore-ros-pkg` . This algorithm provides a resource for others to use for their robotics problems, particularly if the problem presents the *RL for Robotics Challenges*.
2. *Generalized Models*: Methods for learning MDP models that: 1) generalize transition and reward dynamics across state-actions; 2) provide a measure of uncertainty in their predictions; 3) can model continuous domains; 4) can model domains with sensor or actuator delays; and 5) can learn accurate models of dependent feature transitions in factored domains.
3. *Targeted Exploration*: An examination of exploration methods for RL agents with models that generalize across state-actions. This examination includes methods to drive the agent to perform limited, targeted exploration, methods

to explore uncertain or novel states, and intrinsically motivated exploration for domains with little or no external rewards.

4. *Real Time Architecture*: A parallel real time model-based RL agent architecture that enables model-based RL agents to act in real time, without being constrained by the time required for model updates or planning. In addition, this architecture is capable of planning in both continuous domains and domains with sensor or actuator delays. This architecture is also part of the ROS package and can be used with other model learning and planning methods, making it useful to many RL researchers interested in combining sample efficient learning with real time action selection.
5. *ROS RL Interface*: We developed a RL interface for ROS (Robot Operating System) to make it easy to integrate RL with existing robots already using ROS. The interface defines messages for the agent to send and receive from the environment to perform learning. This interface is available as part of our ROS package at: `http://www.ros.org/wiki/rl_msgs`.
6. *Evaluation*: Empirical evaluation of TEXPLORE learning in a variety of time-constrained domains, and in particular, evaluation of TEXPLORE learning to control a physical robot while running in real time on-board the robot.

Many of these contributions can have a lasting impact on the field of RL. If later researchers find themselves working on robotics problems that present the *RL for Robotics Challenges*, they can use TEXPLORE to address their problem. If they do not want to use TEXPLORE completely, they can still take advantage of its real time architecture to combine sample efficiency with real time actions selection. If faced with only some of the *RL for Robotics Challenges*, the empirical evaluations in Chapter 5 provide insights on what solutions may be practical. Finally, even if using a completely different RL algorithm, our ROS RL interface makes it easy to apply other RL algorithms to robots already running ROS.

8.3 Discussion

The empirical evaluations of TEXPLORE presented in Chapters 5 and 6 demonstrated that TEXPLORE performs well across a wide range of tasks. However, it would be too much to expect TEXPLORE to out-perform other algorithms on *all* tasks. One aspect that can cause TEXPLORE to perform poorly is if its exploration is not suited to the task. One example of such a task was presented in Section 6.1.2: the *Taxi* domain. This task requires the agent to perform a "needle-in-a-haystack" search for arbitrarily located landmark states and penalizes the agent for trying to find these landmarks using the PICK-UP and DROP-OFF actions. Later, in Section 6.4, I demonstrated that TEXPLORE does perform well on the *Taxi* domain when given the landmark locations, and can perform well on "needle-in-a-haystack" domains when there is no extra penalty for exploratory actions. In general, such domains call for the agent to explore each individual state-action, rather than try to generalize the effects of actions across states at TEXPLORE does. While TEXPLORE's exploration is not well suited to such domains, it can still solve such tasks by exploring randomly until it finds some useful transitions or rewards.

Another way in which TEXPLORE could be poorly suited for a task is if its model is not well suited to the domain. TEXPLORE uses decision and regression trees to model the dynamics of the domain. These trees model many domains well, as they can split the state space into various regions that each have different dynamics. However, this model many not fit other domains as well, and could cause TEXPLORE to perform poorly. For example, a domain where the dynamics are different in each state-action allows for no generalization, and calls for an algorithm with a tabular model. While a tabular model would be preferable for such tasks, TEXPLORE's tree models can split the state space very finely so that each state is represented in its own leaf of the tree.

By addressing all four of the *RL for Robotics Challenges*, TEXPLORE is applicable to many domains. It is particularly effective in domains with large state spaces where the effects of actions are generalizable across states. In these domains, TEXPLORE's decision tree models work very well and its exploration enables it to learn a good policy in many fewer actions than other algorithms that would explore more thoroughly. As the opportunity for generalization in the model goes up, TEXPLORE performance gains compared to tabular model-based method increases as well. In addition, TEXPLORE is applicable to domains that require real time action selection, such as robot control, where other sample efficient methods would take too long for computation between each action selection.

TEXPLORE is focused particularly on *time-constrained* domains, where the agent does not have a long enough lifetime to guarantee that it can learn an optimal policy. *Time-constrained* domains are very common as many real-world problems have very expensive samples and large state-action spaces. To address these domains, TEXPLORE forgoes guarantees of optimality and instead focuses on learning a high-rewarding policy in a very small number of actions. Instead of exploring every state-action to guarantee it will learn an optimal policy, TEXPLORE must make some assumptions about the domain in order to learn a high-rewarding policy quickly. TEXPLORE assumes that the effects of actions will be similar across states with similar state features. This assumption enables TEXPLORE to learn a useful model of all the states quickly and thus to learn a high-rewarding policy within a short lifetime. We showed in Chapter 5 that TEXPLORE performs better empirically on *time-constrained* domains than other methods that guarantee they will eventually learn an optimal policy. We believe TEXPLORE's approach is the correct one to scale up RL to more real-world domains, since many of them are *time-constrained* domains.

One thing that our work on TEXPLORE brings to light is the comparison between the best exploration strategies for different types of domains. I argue that the typical gridworld goal-based domains used to test many RL algorithms (*haystack* domains) are not well suited for testing exploration as the best exploration for these tasks is simply to explore every state-action. While this is the best exploration strategy for these tasks, it is not going to scale up to larger and more complex tasks where it is impossible to explore every state-action. Rather than testing exploration on *haystack* domains, we need to develop test domains

that examine the ability of an RL agent to figure out the dynamics of its actions in the world to achieve its goal more efficiently (such as *informative* domains). Particularly on robots, these dynamics can be very complex, and exploring efficiently while learning enough about the dynamics to perform the task well can be very challenging.

Another aspect of RL that is very related to the performance of TEXPLORE and other RL algorithms is the state representation, or what state features are used. For tabular methods such as Q-LEARNING and R-MAX, there is no incentive to add any additional features beyond those necessary for the representation to be Markov. For RL methods that approximate the value function, it is useful to have features that provide information about state values and enable better approximations of the value function. In contrast, TEXPLORE performs function approximation on the *model* of the domain. Thus, for TEXPLORE, it makes the most sense to have features that are useful for generalizing transitions and rewards across states. In addition, in the TEXPLORE-VANIR algorithm, the state features are also used to drive exploration. Therefore, for TEXPLORE and TEXPLORE-VANIR it is beneficial to add more informative features to the state representation, even if it is already Markov, to help guide exploration and provide better features for approximating the transition and reward models.

8.4 Future Work

This book leads to multiple directions for future work. First, further work can be done to make RL applicable to more real-world problems, including more robotics tasks. Second, the work on exploration in this book could be extended to work on larger, more complex problems. Third, this work could be extended for modeling opponents when playing games. Fourth, work in this book could lead to advancements towards the problem of *lifelong learning*, where robots act and learn in their environments over their entire lifetime, continually improving their performance. I examine each of these four avenues of future work in more detail in this section.

8.4.1 Expanded Applicability of RL

While our development of TEXPLORE already makes RL more applicable to many real-world and robotics problems, it is still not applicable to *all* real-world problems. Applying RL to real-world problems, such as the autonomous vehicle velocity control problem addressed in this book, leads to many significant challenges that must be addressed and that bring the algorithm another step closer to being more broadly applicable. Working to apply RL to tasks such as video games, robotics, and environmental tasks will lead to the discovery of new challenges where new solutions need to be found to make RL applicable to such problems.

While I demonstrated TEXPLORE learning to drive an autonomous vehicle at different velocities and learning to score penalty kick goals on a humanoid robot, TEXPLORE could be extended to perform better on additional robotic tasks. One

area where TEXPLORE's performance on robots can be improved is handling continuous actions. While TEXPLORE selects from a set of discrete actions, robots typically take a vector of continuous commands. For example, controlling the Aldebaran Nao robot requires a continuous vector of either desired velocities or positions for each of the robot's 25 joints. TEXPLORE's tree models should already be able to handle multi-dimensional continuous actions as input in making predictions about the next state and reward. Thus, extending TEXPLORE to use multi-dimensional continuous actions mainly requires extensions to the UCT planning algorithm for sampling and selecting from a multi-dimensional continuous action space. One possible approach to this problem is to utilize recent work (Mansley et al., 2011; Weinstein and Littman, 2012) adapting the HOO algorithm for continuous bandit problems (Bubeck et al., 2011) to action selection at each level of the UCT tree.

Another approach to improving performance on robots is to make better use of data from simulation. Some of the trees in TEXPLORE's random forest model could be initialized with experiences from simulation. As TEXPLORE performs the task on the real robot, it could update both these models and new models, learning how to weight the real data with the simulated data appropriately to improve sample efficiency.

Another interesting set of domains for RL are environmental applications, such as the ones presented in Section 7.5. Example domains include deciding which parts of a forest to cut or re-plant (Crowley and Poole, 2011), or how to manage different animal populations (Chades et al., 2007). These domains present a different challenge than robotics domains. In many of these domains, actions are often taken every few years rather than seconds, so the ability to select actions quickly is not critical. Instead, the challenges with these domains are that there can be millions of state features and actions. Learning separate decision tree models of each state feature from millions of possible inputs is likely to require lots of samples. One solution to this problem would be to make TEXPLORE's model learning hierarchical, with it applying the same decision trees to predict many different state features. This problem could be addressed from another direction by making TEXPLORE massively parallel. To improve the computational efficiency of running the algorithm on such a large state and action space, the models predicting each feature could be learned on different cores and many parallel instantiations of UCT could be run at once.

In addition to robotics and environmental applications, video games present a good testbed for RL research. They have very large state spaces, complex dynamics, and allow for model generalization. They are easily run on a computer, not requiring real-world interactions as robots do. Importantly, since video games were created for humans to play rather than computers or RL agents, they are more realistic than many typical RL example domains. In particular, Atari games have already been used as a benchmark task for some learning algorithms (Bellemare et al., 2012). An Atari Learning Environment (ALE) framework (Naddaf, 2010) already exists, which could be connected with our ROS messages interface to enable RL agents to interact with many Atari games.

As a first step, one could connect ALE and our ROS RL messages interfaces together and test TEXPLORE on a few basic games. One of the most challenging aspects of performing RL on video games will be developing a good representation for learning. As the number of objects on the screen varies from moment to moment, there may not be a constant number of state features for the RL agent to use. Instead, it may be useful to adapt TEXPLORE to use an object-oriented approach (Diuk et al., 2008) to feature representation.

After a good state representation for TEXPLORE to learn these Atari games is developed, Atari games will provide a great testbed for exploration. These domains are large and complex, meaning that exploring every possible state-action is not feasible. In addition, many of the games include "hidden" features and paths that are difficult for even human players to discover. Therefore, such games will require more intelligent exploration mechanisms.

8.4.2 Exploration

More intelligent and targeted exploration mechanisms are important not just for video games, but any large and complex task. The TEXPLORE-VANIR and TEXPLORE-LEO algorithms presented in Chapter 6 represent steps towards exploration that will work in such domains.

There are multiple directions for future work on designing exploration mechanisms for larger and more complex domains. First, applying TEXPLORE-VANIR and TEXPLORE-LEO to domains like Atari games will demonstrate what aspects of the existing exploration methods work and do not work well in these domains. Next, more exploration strategies based on other properties of TEXPLORE's model can be developed. For example, one could develop an exploration strategy that rewards TEXPLORE for experiences that change its model. This strategy would encourage TEXPLORE to take actions that lead to outcomes that its model does not predict and cause model updates. Another approach is to examine the potential next splits in the leaves of TEXPLORE's tree models and reward experiences that would provide more information about whether these splits are useful or not. All of these strategies could be given to TEXPLORE-LEO and then it could select the best strategies from among this set.

Another issue with exploration in these domains is that external rewards are often not received for a long time. For example, in many games, the agent may not receive reward until it wins or loses a game after many time steps. In these cases, evaluating exploration strategies based on the on-line rewards they receive, as TEXPLORE-LEO does, may perform poorly, as no rewards are received until the end of the game. In such scenarios, it may be useful to develop alternative criteria for evaluating the different exploration strategies. The goal in this case would be to develop criteria that will reward strategies that lead the agent to learn the most accurate models. Since the true model accuracy cannot be evaluated without knowledge of the true dynamics of the domain, these criteria must evaluate model accuracy indirectly. Two possible criteria are to evaluate strategies on their ability to cause more updates to the model or to increase the size of the tree models. The combination of these evaluation criteria with new

exploration strategies might create a powerful learning algorithm that works in large, complex domains with sparse external rewards.

Another approach to driving exploration in TEXPLORE would be to modify its method of model learning. For example, TEXPLORE-VANIR could use the DECORATE algorithm (Melville and Mooney, 2003) instead of random forests for its model. The DECORATE algorithm explicitly tries to maximize diversity in the learners in its ensemble by creating artificial training examples that disagree with the predictions of the committee. Using this model with the VARIANCE-REWARD exploration presented in Section 6.2.1 might lead to better exploration. With the current random forest method, it is possible for the model to over-generalize and make bad predictions about unseen state-actions. The DECORATE algorithm would predict that such state-actions may have different outcomes than what the normal generalization would predict, resulting in disagreement in its models and leading to a higher VARIANCE-REWARD. This approach may have benefits over explicitly driving the agent to novel states using the NOVELTY-REWARD.

8.4.3 Opponent Modeling

TEXPLORE could also be extended to perform opponent modeling in games. In game playing, the rules of the game are known, and thus the challenge for the agent to win the game is to model the opponent's strategy quickly. TEXPLORE is well suited for this task for two reasons: 1) its models can be initialized with experience seeds; and 2) it explores to determine which of its models is correct.

In the application of TEXPLORE to typical RL domains, it learns a model of the domain using random forests of decision trees. In this model, each tree represents a possible hypothesis of the true dynamics of the domain. In addition, TEXPLORE's model can be pre-trained on a set of example transitions to initialize the model. For playing games against opponents, TEXPLORE could be given the rules of the game and only need to learn a model of the opponent. Each of TEXPLORE's tree models in each forest could be initialized with example transitions from *different* opponents. These transitions could come from actual opponents that the algorithm had played in the past or hypothetical possible opponents. TEXPLORE could then be adapted to adjust the weights of each model based on its accuracy in predicting the opponent's moves. Then TEXPLORE would plan on a weighted average of these possible opponent models.

Planning on this aggregate of possible opponent models would again lead TEXPLORE to balance the optimistic models with the pessimistic ones. In this case, that means that TEXPLORE would take moves that would be likely to be good moves against most of the opponent models while avoiding moves that might be bad against some of the opponents. As TEXPLORE learned to better model the opponent, it would consider the other models less and better be able to exploit its knowledge of the particular opponent it was facing.

8.4.4 Lifelong Learning

Finally, a long-term goal that TEXPLORE can help address is the problem of *lifelong learning* (Thrun and Mitchell, 1995). In lifelong learning, the objective is for robots to be able to act and learn in their environments over their entire lifetime, continually improving their performance while performing many different tasks. For agents and robots to be really useful in society, lifelong learning is important, as it will enable the agents to be persistent in the environment and learn multiple tasks. Lifelong learning raises a number of challenges. Lifelong learning agents face a set of challenges related to the *RL for Robotics Challenges* presented in this book. For lifelong learning, agents must 1) learn in an enormously large and complex state space that is rich enough to represent all the possible tasks the robot may learn; 2) handle an ever-growing collection of experiences over their entire lifetime; 3) be persistent in their environment while learning, and 4) generalize experience from other tasks to perform well on new tasks.

In typical RL domains, the agent is given a state and action space suitable for the task it is learning. In contrast, in lifelong learning, the goal is for the agent to learn *many* tasks over its lifetime. Therefore, the state and action space for lifelong learning must allow the agent to learn models and represent policies for many tasks. This state-action space will be significantly larger than ones used for agents learning a single task. TEXPLORE already makes significant progress towards handling a large and complex state space. Learning in such a large space will be a *time-constrained* domain, in that TEXPLORE is already capable of learning reasonable policies. TEXPLORE incorporates generalization into its model learning, which will enable it learn a useful model without attempting to visit every state. The TEXPLORE-VANIR and TEXPLORE-LEO algorithms presented in Chapter 6 will enable a robot to perform intelligent exploration in such a large domain, and enable it to perform more open-ended learning, not requiring the user to provide an external reward function. Future work towards this challenge includes developing more exploration strategies specifically for this type of task. Another avenue of work to address this challenge is to develop a massively parallel version of TEXPLORE, as was also suggested for addressing more real-world problems in Section 8.4.1. With this architecture, the model learning for each state feature would be performed in parallel on separate computer cores. In addition, UCT planning (Kocsis and Szepesvári, 2006) can parallelized as well, with many cores simulating trajectories in parallel (Gelly et al., 2008; Chaslot et al., 2008; Méhat and Cazenave, 2011). Handling an ever-growing set of experiences will require a different model learning method than TEXPLORE uses. Important future work will be to identify other supervised learning methods that do not grow with the size of the data set, but still have the desired generalization properties similar to decision trees.

TEXPLORE already addresses part of the problem of being persistent in the environment, as its real time architecture enables robots to learn while continuing to act in their environment in real time. The second part of this challenge will require work to enable the robot to charge itself. First, it must physically be

able to charge itself. Second, the robot should not travel so far from a charger or electrical outlet that it cannot re-charge. This aspect will require work on the reward structure and exploration of the robot so that it does not explore past its limits and is always able to return to an electrical outlet or charging station.

Generalizing knowledge from previous tasks to perform new tasks will also require some future work. Following the approach of TEXPLORE, work can be done on the state representation and the model approximation to best enable generalization across tasks. For example, the right representations within the agent's factored model will enable the robot to re-use its model of its physical dynamics for various tasks. Another approach to addressing this challenge could be to explicitly incorporate transfer learning (Taylor and Stone, 2009) into TEXPLORE. A different approach to this challenge would be to utilize work on multi-task learning (Wilson et al., 2007).

8.4.5 Summary

As this section shows, TEXPLORE presents plenty of opportunities for future work. I have presented four possible directions for future work. First, research can be done to continue making RL applicable to a broader range of tasks. Second, new exploration methods for larger and more complex domains can be researched. Third, TEXPLORE could be used to differentiate between different possible opponent models when playing games. Fourth, TEXPLORE can be extended and improved in multiple ways to address the lifelong learning problem.

8.5 Conclusion

To conclude, in this book, I have presented the TEXPLORE algorithm, which is the first algorithm to address all four *RL for Robotics Challenges* together. By addressing these four challenges, TEXPLORE is applicable to many real-world problems and especially many robot control problems. I demonstrated TEXPLORE's success in addressing each challenge on the problem of controlling the velocity of an autonomous vehicle by manipulating the throttle and brake of the vehicle. This work presents an important step towards making RL generally applicable to a wide range of such challenging robotics problems.

A TEXPLORE Pseudo-code

In this appendix, I present the full pseudo-code for the complete TEXPLORE *algorithm, including the extensions for handling dependent feature transitions, actuator and sensor delays and continuous state. Versions of these algorithms appeared earlier in Chapters 3 and 4, but this appendix represents a comprehensive collection of the pseudo-code for* TEXPLORE*.*

This appendix presents the pseudo-code for the complete TEXPLORE algorithm with all the extensions and modifications for handling dependent feature transitions, actuator and sensor delays and continuous state. In addition to the pseudo-code presented in this appendix, the *actual* code for TEXPLORE is available as an open-source ROS package at: `http://www.ros.org/wiki/rl-texplore-ros-pkg`. This appendix is intended to present the full TEXPLORE algorithm in a clear way, therefore the various extensions to TEXPLORE for exploration in different types of domains presented in Chapter 6 are not included here.

First, we present Algorithm A.1, which shows TEXPLORE's RTMBA architecture, complete with delay handling. This architecture splits the model learning, planning, and acting into three parallel threads so that the time required for action selection is not constrained by the time taken for model learning or planning. This algorithm merges the default architecture presented in Algorithm 3.4 with the extension for domains with delays presented in Algorithm 4.2.

Next, we present the UCT(λ) sample-based planning algorithm used within this real time architecture in Algorithm A.2. This algorithm is called on Lines 8, 18, and 24 of Algorithm A.1 to plan a policy on the model within the real time architecture. The UCT(λ) algorithm presented in Algorithm A.2 is a combination of the default UCT(λ) presented in Algorithm 3.2 and its extension for sensor and actuator delays presented in Algorithm 3.3.

Algorithm A.1. 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 ← s
5:     h ← ∅                                            ▷ Start with empty history h
6:     updateList ← ∅
7:     Initialize M to empty model
8:     UCT-INIT()                                    ▷ Initialize Planner (Alg A.2)
9: end procedure

10: procedure MODELLEARNINGTHREAD                         ▷ Model Learning Thread
11:     loop                                   ▷ Loop, adding experiences to model
12:         while updateList = ∅ do
13:             Wait for experiences to be added to list
14:         end while
15:         tmpModel ← M⇒COPY                      ▷ Make temporary copy of model
16:         tmpModel⇒UPDATE-MODEL(updateList)   ▷ Update tmpModel (Alg A.3)
17:         updateList ← ∅                                ▷ Clear the update list
18:         UCT-RESET()               ▷ Less confidence in current values (Alg A.2)
19:         M ← tmpModel                                    ▷ Swap model pointers
20:     end loop
21: end procedure

22: procedure PLANNINGTHREAD                                    ▷ Planning Thread
23:     loop                               ▷ Loop forever, performing rollouts
24:         UCT-SEARCH(M, agentState, h, 0)                        ▷ Algorithm A.2
25:     end loop
26: end procedure

27: procedure ACTIONTHREAD                              ▷ Action Selection Thread
28:     loop
29:         s_disc ← DISCRETIZE(s, nBins, minVals, maxVals)
30:         Choose a ← argmax_a Q(s_disc, h, a)       ▷ Values of state-history-actions
31:         Take action a, Observe r, s'
32:         augState ← ⟨s, h⟩                       ▷ Augment state with history
33:         updateList ← updateList ∪ ⟨augState, a, s', r⟩
34:         PUSH(h, a)                                         ▷ Keep last k actions
35:         if LENGTH(h) > k then
36:             POP(h)
37:         end if
38:         s ← s'
39:         agentState ← s              ▷ Set agent's state for planning rollouts
40:     end loop
41: end procedure
```

Algorithm A.2. PLAN: UCT(λ)

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

6: procedure PLAN-POLICY(M, s, h)    ▷ Approx. planning from state s and history h
7:     UCT-RESET()
8:     while time available do
9:         UCT-SEARCH(M, s, h, 0)
10:    end while
11: end procedure

12: procedure UCT-RESET()    ▷ Lower confidence in v.f. since model changed
13:    for all s_disc ∈ S_disc, h ∈ H_{s_disc} do    ▷ For all state-histories
14:        if c(s_disc, h) > resetCount · |A| then
15:            c(s_disc, h) ← resetCount · |A|    ▷ resetCount per action
16:        end if
17:        for all a ∈ A do
18:            if c(s_disc, h, a) > resetCount then
19:                c(s_disc, h, a) ← resetCount
20:            end if
21:        end for
22:    end for
23: end procedure

24: procedure UCT-SEARCH(M, s, h, d)    ▷ Rollout from state s with h
25:    if TERMINAL or d = maxDepth then
26:        return 0
27:    end if
28:    s_disc ← DISCRETIZE(s, nBins, minVals, maxVals)
29:    a ← argmax_a' ( Q(s_disc, h, a') + 2 · rmax/(1−γ) · sqrt( log c(s_disc, h) / c(s_disc, h, a') ) )
30:    (s', r) ← M⇒QUERY-MODEL(⟨s, h⟩, a)    ▷ Query model (Alg A.3)
31:    PUSH(h, a)    ▷ Keep last k actions
32:    if LENGTH(h) > k then
33:        POP(h)
34:    end if
35:    sampleReturn ← r + γUCT-SEARCH(M, s', h, d + 1)
36:    c(s_disc, h) ← c(s_disc, h) + 1    ▷ Update counts
37:    c(s_disc, h, a) ← c(s_disc, h, a) + 1
38:    Q(s_disc, h, a') ← α · sampleReturn + (1 − α) · Q(s_disc, h, a')
39:    return λ · sampleReturn + (1 − λ) · max_a' Q(s_disc, h, a')
40: end procedure
```

Next, we present TEXPLORE's model learning method in Algorithm A.3. This algorithm learns a separate prediction of each next state feature and reward using a random forest. This algorithm is called by the architecture on Line 16 of Algorithm A.1 to update the model with new experiences and is called by UCT(λ) on Line 30 of Algorithm A.2 to query the model for a prediction. This algorithm is a modified version of Algorithm 4.1 to incorporate the added synchronic arcs to make dependent feature predictions.

Algorithm A.3. MODEL

```
 1: procedure INIT-MODEL(n)                                ▷ n is the number of state variables
 2:     for i = 1 → n do
 3:         featModel_i⇒INIT()                     ▷ Init forest to predict feature i (Alg A.4)
 4:     end for
 5:     rewardModel⇒INIT()                         ▷ Init forest to predict reward (Alg A.4)
 6: end procedure

 7: procedure UPDATE-MODEL(list)                  ▷ Update model with list of experiences
 8:     for all ⟨s, a, s′, r⟩ ∈ list do
 9:         s^rel ← s′ − s                                     ▷ Calculate relative effect
10:         for all s_i^rel ∈ s^rel do
11:             depState ← ⟨s, s_0^rel, ..., s_{i−1}^rel⟩      ▷ Add dependent feature inputs
12:             featModel_i⇒UPDATE(⟨depState, a⟩, s_i^rel)          ▷ Train forest (Alg A.4)
13:         end for
14:         rewardModel⇒UPDATE(⟨s, a⟩, r)     ▷ Train forest to predict reward (Alg A.4)
15:     end for
16: end procedure

17: procedure QUERY-MODEL(s, a)                          ▷ Get prediction of ⟨s′, r⟩ for s, a
18:     for i = 1 → LENGTH(s) do
19:         depState ← ⟨s, s_0^rel, ..., s_{i−1}^rel⟩          ▷ Add dependent feature inputs
20:         s_i^rel ← featModel_i⇒QUERY(⟨depState, a⟩)       ▷ Sample prediction (Alg A.4)
21:     end for
22:     s′ ← s + ⟨s_1^rel, ..., s_n^rel⟩                          ▷ Get absolute next state
23:     r ← rewardModel⇒QUERY(⟨s, a⟩)          ▷ Sample r from distribution (Alg A.4)
24:     return ⟨s′, r⟩                             ▷ Return sampled next state and reward
25: end procedure
```

Finally, we present pseudo-code for the random forest models used to predict each state feature and reward in Algorithm A.4 (originally presented as Algorithm 4.3). This algorithm is used by TEXPLORE's model learning approach to learn separate random forest models of each feature and reward. It is called by TEXPLORE's model learning algorithm on Lines 12 and 14 of Algorithm A.3 to update the forest models with new experiences is called on Lines 20 and 23 to query the forest for predictions. This random forest model is made up of a set of m decision trees. These trees are initialized on Line 3, updated on Line 9 and queried on Line 15 of the Algorithm. For discrete domains, these calls are to

Algorithm A.4. MODEL: Random Forest

```
 1: procedure INIT(m)                                  ▷ Init forest of m trees
 2:     for i = 1 → m do
 3:         tree_i ⇒ INIT()                            ▷ Init tree i (either C4.5 or M5 tree)
 4:     end for
 5: end procedure

 6: procedure UPDATE(in, out)                          ▷ Update forest with (in, out) example
 7:     for i = 1 → m do                               ▷ For m trees in the random forest
 8:         if RAND() ≤ w then                         ▷ Update each tree with prob. w
 9:             tree_i ⇒ UPDATE(in, out)               ▷ Either C4.5 or M5 tree
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 ← tree_i ⇒ QUERY(in)   ▷ Get prediction from tree i (either C4.5 or M5 tree)
16:     return x                                       ▷ Return prediction
17: end procedure
```

C4.5 decision trees (Quinlan, 1986), and for continuous domains, these calls are to M5 regression trees (Quinlan, 1992).

These algorithms represent pseudo-code for the complete TEXPLORE algorithm, which is sample efficient, acts in real time, works with sensor and actuator delays, handles dependent feature transitions, and works in continuous domains. In addition to this pseudo-code, the actual code for TEXPLORE is freely available as a ROS package at: `http://www.ros.org/wiki/rl-texplore-ros-pkg`.

The goal of this appendix is to provide the complete TEXPLORE algorithm without the exploration extensions from Chapter 6, however here we will present some pointers to the pseudo-code for those extensions. The TEXPLORE-EE algorithm for *haystack* domains uses a single tree instead of a random forest, so the calls to the random forest model in Algorithm A.3 would be replaced with calls to a single decision tree. In addition, for acting and planning, TEXPLORE-EE uses Algorithm 6.1. TEXPLORE-VANIR calculates some extra properties of its model for use in driving exploration, which is accomplished by replacing the random forest model in Algorithm A.4 with Algorithm 6.2. Finally, the TEXPLORE-LEO algorithm also uses Algorithm 6.2 for its model learning and then utilizes Algorithms 6.3 and 6.4 for its planning and action selection.

B Evaluation Domains

In this appendix, I present a listing of each domain that was used in this book. For each domain, the state variables, actions, reward structure, total number of state-actions, time-constrained lifetime, and domain class are listed.

This appendix presents all of the domains that were used for empirical evaluations in this book. For each domain, I present the state variables, actions, reward structure, and total number of state-actions of the domain. In addition, I list the time-constrained lifetime for each domain, which is defined as $L < 2NA$ in Section 2.3. I also state whether each domain is a *haystack*, *prior information*, or *informative* domain, as defined in Chapter 6. In the caption of each table, I state which section in the book presented this domain. All of the domains in this book are available in our open-source ROS package at: `http://www.ros.org/wiki/rl-texplore-ros-pkg`.

Table B.1. Properties of the *Vehicle Velocity Control* task, introduced in Section 2.4. Note that each episode is 100 actions long, as it is 10 seconds of control of the car with actions taken at 10 Hz.

Vehicle Velocity Control

State	DES-VEL, CURR-VEL, BRAKE, ACCELERATOR
Actions	NO-OP, ACC-UP, ACC-DOWN, BRAKE-UP, BRAKE-DOWN
Reward	$-10.0 *$ |DES-VEL - CURR-VEL|
# State-Actions	218, 075
Time-Constrained Lifetime	436, 150 actions, 4, 361 episodes
Domain Class	*None*

Table B.2. Properties of the *Fuel World* task, introduced in Section 5.1.2

Fuel World

State	ROW, COL, FUEL
Actions	NORTH, EAST, SOUTH, WEST, NORTHEAST, SOUTHEAST,SOUTHWEST, NORTHWEST
Reward	Ranges from -400.0 to $+20.0$
# State-Actions	317, 688
Time-Constrained Lifetime	635, 376 actions
Domain Class	*Prior Information* with example transitions

Table B.3. Properties of the *Mountain Car* task (Moore, 1990; Sutton and Barto, 1998), presented in Section 5.2.2

Mountain Car	
State	POSITION, VELOCITY
Actions	LEFT, RIGHT, NONE
Reward	-1 each step, 0 upon reaching goal
# State-Actions	$30,000$
Time-Constrained Lifetime	$60,000$ actions
Domain Class	*Prior Information* with example transitions

Table B.4. Properties of the *Cart-Pole Balancing* task (Sutton and Barto, 1998), presented in Section 5.2.2

Cart-Pole Balancing	
State	CART-POS, CART-VEL, POLE-POS, POLE-VEL
Actions	LEFT, RIGHT
Reward	$+1$ each step until episode terminates
# State-Actions	$320,000$
Time-Constrained Lifetime	$640,000$ actions
Domain Class	*None*

Table B.5. Properties of the *Delayed Gridworld* task, introduced in Section 5.3.2

Delayed Gridworld	
State	ROW, COL
Actions	NORTH, EAST, SOUTH, WEST
Reward	-1 each step, 0 upon reaching goal
# State-History-Actions	$3,264$
Time-Constrained Lifetime	$6,528$ actions
Domain Class	*Prior Information* with example transitions

Table B.6. Properties of the *Trap Room* task, introduced in Section 5.5

Trap Room	
State	ROW, COL
Actions	NORTH, EAST, SOUTH, WEST
Reward	-1 each step, 0 on goal, -250 on trap
# State-Actions	252
Time-Constrained Lifetime	504 actions
Domain Class	*Prior Information* with example transitions

Table B.7. Properties of the *Vehicle Velocity Control* task with a *single* target velocity, introduced in Section 5.6. Note that each episode is 100 actions long, as it is 10 seconds of control of the car with actions taken at 10 Hz.

Vehicle Velocity Control (single target velocity)

State	DES-VEL, CURR-VEL, BRAKE, ACCELERATOR
Actions	NO-OP, ACC-UP, ACC-DOWN, BRAKE-UP, BRAKE-DOWN
Reward	$-10.0 * \lvert$DES-VEL - CURR-VEL$\rvert$
# State-Actions	$16,775$
Time-Constrained Lifetime	$33,550$ actions, 335 episodes
Domain Class	*None*

Table B.8. Properties of the *Taxi* task (Dietterich, 1998), presented in Section 6.1.2

Taxi

State Features	X, Y, PASSENGER, DESTINATION
Actions	EAST, WEST, NORTH, SOUTH, PICK-UP, DROP-OFF
Reward	-1 normally, $+20$ upon completion -10 for bad PICK-UP or DROP-OFF action
# State-Actions	$3,000$
Time-Constrained Lifetime	$6,000$ actions
Domain Class	*Haystack* normally, *Prior Information* with example transitions

Table B.9. Properties of the *Penalty Kick* task, introduced in Section 6.1.2

Penalty Kick

State Features	X, FOOT-SHIFT
Actions	MOVE-IN, MOVE-OUT, KICK
Reward	Ranges from -20 to $+20$
# State-Actions	$3,360$
Time-Constrained Lifetime	$6,720$ actions
Domain Class	*Haystack*

Table B.10. Properties of the *Light World* domain (Konidaris and Barto, 2007), presented in Section 6.2.2

Light World	
State Features	ID, X, Y, KEY, LOCKED, RED-E, RED-W, RED-N, RED-S, GREEN-E, GREEN-W, GREEN-N, GREEN-S, BLUE-E, BLUE-W, BLUE-N, BLUE-S,
Actions	EAST, WEST, NORTH, SOUTH, PRESS, PICKUP
Reward	0 each step, +10 when leaving room
# State-Actions	1, 464
Time-Constrained Lifetime	2, 928 actions
Domain Class	*Informative*

Table B.11. Properties of the *Sensor Goal* domain, introduced in Section 6.3.2

Sensor Goal	
State Features	X, Y, SENSE-N, SENSE-E, SENSE-S, SENSE-W
Actions	EAST, WEST, NORTH, SOUTH
Reward	−1 each step, +2 upon reaching goal
# State-Actions	58, 564
Time-Constrained Lifetime	117, 128 actions
Domain Class	*Informative*

Table B.12. Properties of the *Arbitrary Goal* domain, introduced in Section 6.3.2

Arbitrary Goal	
State Features	X, Y, GOAL-ID
Actions	EAST, WEST, NORTH, SOUTH
Reward	−1 each step, +2 upon reaching goal
# State-Actions	58, 564
Time-Constrained Lifetime	117, 128 actions
Domain Class	*Haystack*

Table B.13. Properties of the *Stock Trading* domain (Strehl et al., 2007), presented in Section 6.4

Stock Trading	
State Features	OWN-SEC-1, OWN-SEC-2, OWN-SEC-3, STOCK-1-1 STOCK-1-2, STOCK-2-1, STOCK-2-2, STOCK-3-1, STOCK-3-2
Actions	BUY-SEC-1, SELL-SEC-1, BUY-SEC-2 SELL-SEC-2, BUY-SEC-3, SELL-SEC-3
Reward	Ranges from −6 to +6
# State-Actions	3, 072
Time-Constrained Lifetime	6, 144 actions
Domain Class	*None*

References

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

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

Atherton, D.P., Majhi, S.: Limitations of pid controllers. In: Proceedings of the 1999 American Control Conference, pp. 3843–3847 (1999)

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

Auer, P., Cesa-Bianchi, N., Freund, Y., Schapire, R.E.: Gambling in a rigged casino: The adversarial multi-armed bandit problem. Electronic Colloquium on Computational Complexity (ECCC) 7(68) (2000)

Bagnell, J.A.D., Schneider, J.: Autonomous helicopter control using reinforcement learning policy search methods. In: Proceedings of the International Conference on Robotics and Automation 2001. IEEE (2001)

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

Barto, A.G., Bradtke, S.J., Singh, S.P.: Learning to act using real-time dynamic programming. Artificial Intelligence 72(12), 81–138 (1995)

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

Bellemare, M.G., Naddaf, Y., Veness, J., Bowling, M.: The arcade learning environment: An evaluation platform for general agents. ArXiv e-prints (2012)

Biau, G.: Analysis of a random forests model. Journal of Machine Learning Research 98888, 1063–1095 (2012)

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

Brafman, R., Tennenholtz, M.: 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 (2001)

Bratman, J., Singh, S.P., Sorg, J., Lewis, R.L.: Strong mitigation: nesting search for good policies within search for good reward. In: Proceedings of the Eleventh International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 407–414 (2012)

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

Bubeck, S., Munos, R., Stoltz, G.: Pure exploration in finitely-armed and continuous-armed bandits. Theoretical Computer Science 412(19), 1832–1852 (2011); Algorithmic Learning Theory (ALT 2009)

Chades, I., Martin, T.G., Curtis, J.M., Barreto, C.: Managing interacting species: A reinforcement learning decision theoretic approach. In: Proceedings of the 2007 International Congress on Modelling and Simulation, pp. 74–80 (2007)

Chakraborty, D., Stone, P.: 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 (2011)

Chapman, D., Kaelbling, L.P.: Input generalization in delayed reinforcement learning: an algorithm and performance comparisons. In: Proceedings of the 12th International Joint Conference on Artificial Intelligence, IJCAI 1991, vol. 2, pp. 726–731. Morgan Kaufmann Publishers Inc., San Francisco (1991)

Chaslot, G.M.J.-B., Winands, M.H.M., van den Herik, H.J.: Parallel Monte-Carlo tree search. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 60–71. Springer, Heidelberg (2008)

Chow, C., Tsitsiklis, J.: An optimal one-way multigrid algorithm for discrete-time stochastic control. IEEE Transactions on Automatic Control 36, 898–914 (1991)

Crowley, M., Poole, D.: Policy gradient planning for environmental decision making with existing simulators. In: Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence (2011)

Şimşek, O., Barto, A.G.: An intrinsic reward mechanism for efficient exploration. In: ICML, pp. 833–840 (2006)

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

Degris, T., Sigaud, O., Wuillemin, P.-H.: 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 (2006)

Deisenroth, M., Rasmussen, C.: Efficient reinforcement learning for motor control. In: 10th International PhD Workshop on Systems and Control, Hluboka nad Vltavou, Czech Republic (2009)

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

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

Diuk, C., Cohen, A., Littman, M.: An object-oriented representation for efficient reinforcement learning. In: Proceedings of the Twenty-Fifth International Conference on Machine Learning (ICML), pp. 240–247 (2008)

Diuk, C., Li, L., Leffler, B.: 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 (2009)

Duff, M.: Design for an optimal probe. In: Proceedings of the Twentieth International Conference on Machine Learning (ICML), pp. 131–138 (2003)

Ernst, D., Geurts, P., Wehenkel, L.: Iteratively extending time horizon reinforcement learning. In: Lavrač, N., Gamberger, D., Todorovski, L., Blockeel, H. (eds.) ECML 2003. LNCS (LNAI), vol. 2837, pp. 96–107. Springer, Heidelberg (2003)

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

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

Gelly, S., Hoock, J.-B., Rimmel, A., Teytaud, O., Kalemkarian, Y.: 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 (2008)
Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: Proceedings of the Twenty-Fourth International Conference on Machine Learning (ICML), pp. 273–280 (2007)
Gordon, G.: Stable function approximation in dynamic programming. In: Proceedings of the Twelfth International Conference on Machine Learning, ICML (1995)
Guestrin, C., Patrascu, R., Schuurmans, D.: 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 (2002)
Guez, A., Vincent, R.D., Avoli, M., Pineau, J.: Adaptive treatment of epilepsy via batch-mode reinforcement learning. In: Proceedings of the 20th National Conference on Innovative Applications of Artificial Intelligence, IAAI 2008, vol. 3, pp. 1671–1678. AAAI Press (2008)
Hester, T., Lopes, M., Stone, P.: Learning Exploration Strategies in Model-Based Reinforcement Learning. In: The Twelfth International Conference on Autonomous Agents and Multiagent Systems (AAMAS), St. Paul, Minnesota (May 2013)
Hester, T., Quinlan, M., Stone, P.: Generalized model learning for reinforcement learning on a humanoid robot. In: Proceedings of the 2010 IEEE International Conference on Robotics and Automation, ICRA (2010)
Hester, T., Quinlan, M., Stone, P.: 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 (2012)
Hester, T., Quinlan, M., Stone, P., Sridharan, M.: TT-UT Austin Villa 2009: Naos across Texas. Technical Report UT-AI-TR-09-08, The University of Texas at Austin, Department of Computer Science, AI Laboratory (2009)
Hester, T., Stone, P.: An empirical comparison of abstraction in models of Markov Decision Processes. In: Proceedings of the ICML/UAI/COLT Workshop on Abstraction in Reinforcement Learning (2009a)
Hester, T., Stone, P.: Generalized model learning for reinforcement learning in factored domains. In: Proceedings of the Eight International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS (2009b)
Hester, T., Stone, P.: Real time targeted exploration in large domains. In: Proceedings of the Ninth International Conference on Development and Learning, ICDL (2010)
Hester, T., Stone, P.: Learning and using models. In: Wiering, M., van Otterlo, M. (eds.) Reinforcement Learning. ALO, vol. 12, pp. 111–141. Springer, Heidelberg (2012)
Hester, T., Stone, P.: Intrinsically motivated model learning for a developing curious agent. In: Proceedings of the Eleventh International Conference on Development and Learning, ICDL (2012a)
Hester, T., Stone, P.: TEXPLORE: real-time sample-efficient reinforcement learning for robots. Machine Learning 90(3), 385–429 (2013), doi:10.1007/s10994-012-5322-7
Jong, N., Stone, P.: Model-based function approximation for reinforcement learning. In: Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS (2007)
Jonsson, A., Barto, A.G.: Active learning of dynamic bayesian networks in markov decision processes. In: Miguel, I., Ruml, W. (eds.) SARA 2007. LNCS (LNAI), vol. 4612, pp. 273–284. Springer, Heidelberg (2007)
Kakade, S.: On the Sample Complexity of Reinforcement Learning. PhD thesis, University College London (2003)

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

Kearns, M., Mansour, Y., Ng, A.: 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 (1999)

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

Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)

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

Kolobov, A., Mausam, Weld, D.: Lrtdp versus uct for online probabilistic planning (2012)

Kolter, J.Z., Jackowski, Z., Tedrake, R.: Design, analysis, and learning control of a fully actuated micro wind turbine. In: Proceedings of the American Control Conference (2012)

Kolter, J.Z., Plagemann, C., Jackson, D.T., Ng, A.Y., Thrun, S.: A probabilistic approach to mixed open-loop and closed-loop control, with application to extreme autonomous driving. In: Proceedings of the 2010 IEEE International Conference on Robotics and Automation (ICRA), pp. 839–845 (2010)

Konidaris, G., Barto, A.G.: Building portable options: Skill transfer in reinforcement learning. In: Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI), pp. 895–900 (2007)

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

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

Li, L., Littman, M., Walsh, T.: 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 (2008)

Lin, L.-J.: Reinforcement learning for robots using neural networks. PhD thesis, Pittsburgh, PA, USA (1992)

Lopes, M., Lang, T., Toussaint, M., Oudeyer, P.-Y.: Exploration in model-based reinforcement learning by empirically estimating learning progress. In: Neural Information Processing Systems (NIPS), Tahoe, USA (2012)

Mansley, C.R., Weinstein, A., Littman, M.L.: Sample-based planning for continuous action markov decision processes. In: ICAPS (2011)

McCallum, A.: 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 (1996)

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

Melo, F.S., Meyn, S.P., Ribeiro, M.I.: An analysis of reinforcement learning with function approximation. In: Cohen, W.W., McCallum, A., Roweis, S.T. (eds.) ICML. ACM International Conference Proceeding Series, vol. 307, pp. 664–671. ACM (2008)

Melville, P., Mooney, R.J.: Constructing diverse classifier ensembles using artificial training examples. In: Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003), Acapulco, Mexico, pp. 505–510 (2003)

Moore, A.: Efficient Memory-based Learning for Robot Control. PhD thesis, Computer Laboratory, University of Cambridge, Cambridge, UK (1990)

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

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

Naddaf, Y.: Game-Independent AI Agents for Playing Atari 2600 Console Games. Masters, University of Alberta (2010)

Ng, A., Harada, D., Russell, S.: Policy invariance under reward transformations: Theory and application to reward shaping (1999)

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

Ng, A.Y., Jordan, M.I.: PEGASUS: A policy search method for large mdps and pomdps. In: Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence (UAI), pp. 406–415 (2000)

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

Ohyama, T., Nores, W.L., Murphy, M., Mauk, M.D.: What the cerebellum computes. Trends in Neurosciences 26(4), 222–227 (2003)

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

Peters, J., Schaal, S.: Natural actor-critic. Neurocomput. 71(7-9), 1180–1190 (2008)

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

Precup, D., Sutton, R.S., Singh, S.P.: Eligibility traces for off-policy policy evaluation. In: Proceedings of the Seventeenth International Conference on Machine Learning (ICML), pp. 759–766 (2000)

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

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

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

Randløv, J., Alstrøm, P.: Learning to drive a bicycle using reinforcement learning and shaping. In: Proceedings of the Fifteenth International Conference on Machine Learning, ICML (1998)

Riedmiller, M.: Neural fitted Q iteration - first experiences with a data efficient neural reinforcement learning method. In: Gama, J., Camacho, R., Brazdil, P.B., Jorge, A.M., Torgo, L. (eds.) ECML 2005. LNCS (LNAI), vol. 3720, pp. 317–328. Springer, Heidelberg (2005)

Röfer, T., Hester, T.: Meriçli, T., Middleton, R., Quinlan, M.: Robocup standard platform league rule book (2009)

Rummery, G., Niranjan, M.: On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Department (1994)

Sam Devlin, M.G., Kudenko, D.: An empirical study of potential-based reward shaping and advice in complex, multi-agent systems. Advances in Complex Systems 14(2), 251–278 (2011)

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

Schuitema, E., Busoniu, L., Babuska, R., Jonker, P.: 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 (2010)

Seung, H.S., Opper, M., Sompolinsky, H.: Query by committee. In: Proceedings of the Fifth Annual Workshop on Computational Learning Theory, COLT 1992, pp. 287–294. ACM, New York (1992)

Shortreed, S.M., Laber, E.B., Lizotte, D.J., Stroup, T.S., Pineau, J., Murphy, S.A.: Informing sequential clinical decision-making through reinforcement learning: an empirical study. Machine Learning 84(1-2), 109–136 (2011)

Silver, D., Sutton, R., Müller, M.: 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 (2008)

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

Singh, S., Barto, A.G., Chentanez, N.: Intrinsically motivated reinforcement learning. In: Advances in Neural Information Processing Systems (NIPS) 17 (2005)

Singh, S.P., Lewis, R.L., Barto, A.G., Sorg, J.: Intrinsically motivated reinforcement learning: An evolutionary perspective. IEEE Transactions on Autonomous Mental Development (TAMD) 2(2), 70–82 (2010)

Smart, W., Kaelbling, L.P.: Effective reinforcement learning for mobile robots. In: Proceedings of the 2002 IEEE International Conference on Robotics and Automation (ICRA), vol. 4, pp. 3404–3410 (2002)

Sorg, J., Singh, S.P., Lewis, R.L.: Optimal rewards versus leaf-evaluation heuristics in planning agents. In: Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence (2011)

Stout, A., Barto, A.: Competence progress intrinsic motivation. In: Proceedings of the Ninth International Conference on Development and Learning (ICDL), pp. 257–262 (2010)

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

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

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

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

Sung, S.W., Lee, I.-B.: Limitations and countermeasures of pid controllers. Industrial and Engineering Chemistry Research, 2596–2610 (1996)

Sutton, R.: 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 (1990)

Sutton, R.: Generalization in reinforcement learning: Successful examples using sparse coarse coding. In: Advances in Neural Information Processing Systems (NIPS) 8 (1996)

Sutton, R., Barto, A.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998)
Sutton, R., Modayil, J., Delp, M., Degris, T., Pilarski, P., White, A., Precup, D.: 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 (2011)
Tanner, B., White, A.: RL-Glue: Language-independent software for reinforcement-learning experiments. Journal of Machine Learning Research 10, 2133–2136 (2009)
Taylor, M., Stone, P.: Transfer learning for reinforcement learning domains: A survey. The Journal of Machine Learning Research 10, 1633–1685 (2009)
Tesauro, G.: Temporal difference learning and TD-Gammon. Communications of the ACM 38(3) (1995)
Thrun, S., Mitchell, T.M.: Lifelong robot learning. Robotics and Autonomous Systems 15(12), 25–46 (1995); The Biology and Technology of Intelligent Autonomous Agents
Veness, J., Ng, K.S., Hutter, M., Uther, W.T.B., Silver, D.: A Monte-Carlo AIXI approximation. Journal of Artificial Intelligence Research 40, 95–142 (2011)
Vigorito, C.M., Barto, A.G.: Intrinsically motivated hierarchical skill learning in structured environments. IEEE Transactions on Autonomous Mental Development (TAMD) 2(2) (2010)
Walsh, T., Goschin, S., Littman, M.: Integrating sample-based planning and model-based reinforcement learning. In: Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (2010)
Walsh, T., Nouri, A., Li, L., Littman, M.: Learning and planning in environments with delayed feedback. Autonomous Agents and Multi-Agent Systems 18, 83–105 (2009a)
Walsh, T., Szita, I., Diuk, C., Littman, M.: Exploring compact reinforcement-learning representations with linear regression. In: Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence, UAI (2009b)
Wang, T., Lizotte, D., Bowling, M., Schuurmans, D.: Bayesian sparse sampling for on-line reward optimization. In: Proceedings of the Twenty-Second International Conference on Machine Learning (ICML), pp. 956–963 (2005)
Wang, Y., Gelly, S.: Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In: IEEE Symposium on Computational Intelligence and Games (2007)
Watkins, C.: Learning From Delayed Rewards. PhD thesis, University of Cambridge (1989)
Weinstein, A., Littman, M.L.: Bandit-based planning and learning in continuous-action markov decision processes. In: Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS (2012)
Whiteson, S., Taylor, M.E., Stone, P.: Adaptive tile coding for value function approximation. Technical Report AI-TR-07-339, University of Texas at Austin (2007)
Wiering, M., Schmidhuber, J.: Efficient model-based exploration. In: From Animals to Animats 5: Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior, pp. 223–228. MIT Press, Cambridge (1998)
Willems, F.M.J., Shtarkov, Y.M., Tjalkens, T.J.: The context tree weighting method: Basic properties. IEEE Transactions on Information Theory 41, 653–664 (1995)
Wilson, A., Fern, A., Ray, S., Tadepalli, P.: Multi-task reinforcement learning: a hierarchical bayesian approach. In: Proceedings of the 24th International Conference on Machine Learning, pp. 1015–1022. ACM (2007)
Ziegler, J.G., Nichols, N.B.: Optimum settings for automatic controllers. Transactions of the ASME 64, 759–768 (1942)