APPROVAL SHEET

Title of Thesis: Multi-Source Option-Based Policy Transfer

Name of Candidate: James Alexander MacGlashan
Ph.D., 2013

Thesis and Abstract Approved: _____

Marie desJardins
Professor
Department of Computer Science and
Electrical Engineering

Date Approved: _____

Curriculum Vitae

Name: James Alexander MacGlashan.

Degree and date to be conferred: Ph.D., May 2013.

Secondary Education: Dulaney High School, Timonium, Maryland.

Collegiate institutions attended:

University of Maryland Baltimore County, B.S. Computer Science, 2007.

Professional publications:

MacGlashan, J., Babes-Vroman, M., Winner, K., Gao, R., Adjogah, R., desJardins, M., Littman, M., and Muresan, S., in Proceedings of the AAAI Workshop on Grounding Language for Physical Systems, 2012.

MacGlashan, J., Miner, D., and desJardins, M., "A Game Playing System for Use in Computer Science Education", in Proceedings of the 23rd Florida Artificial Intelligence Research Society Conference, May 2010.

desJardins, M., MacGlashan, J., and Wagstaff, K., "Confidence-Based Feature Acquisition to Minimize Training and Test Costs," in Proceedings of the 2010 SIAM International Conference on Data Mining, April 2010.

MacGlashan, J. and desJardins, M., "Hierarchical Skill Learning for High-level Planning," in Proceedings of the ICML/UAI/COLT Workshop on Abstraction in Reinforcement Learning, Montreal, Canada, 2009.

desJardins, M. and MacGlashan, J., Ferraioli, J., "Interactive visual clustering for relational data," in Constrained Clustering Advances in Algorithms Theory and Applications, Kiri Wagstaff, Sugato Basu, Ian Davidson, Ed., chapter 14, pp. 329-355. Chapman & Hall/CRC Press, Boca Raton, 2008.

desJardins, M., MacGlashan, J., and Ferraioli, J., "Interactive visual clustering," Proceedings of the 2007 International Conference on Intelligent User Interfaces, January 2007

Professional positions held:

Instructor. University of Maryland, Baltimore County (2009 – 2011).

Graduate Assistant. University of Maryland, Baltimore County (2007 – 2013).

Webmaster. Johns Hopkins Clinical Immunology Division (2003 – 2012).

Webmaster. Main St. Gallery (2003 – 2007).

Quality Assurance Tester and Technical Support Representative. Absolute Quality (2000 – 2002).

Sold Software:

Major League Kickball 2010. iOS App Store (2010).

Berserker. iOS App Store (2009).

QuickStat. iOS App Store (2009).

Primer Match. Johns Hopkins Clinical Immunology Division.

IRB/Regulatory Deadline Database. Johns Hopkins Clinical Immunology Division.

ABSTRACT

Title of Thesis: Multi-Source Option-Based Policy Transfer

James Alexander MacGlashan, Ph.D., 2013

Thesis directed by: Marie desJardins, Professor
Department of Computer Science and
Electrical Engineering

Reinforcement learning algorithms are very effective at learning policies (mappings from states to actions) for specific well defined tasks, thereby allowing an agent to learn how to behave without extensive deliberation. However, if an agent must complete a novel variant of a task that is similar to, but not exactly the same as, a previous version for which it has already learned a policy, learning must begin anew and there is no benefit to having previously learned anything. To address this challenge, I introduce novel approaches for *policy transfer*. Policy transfer allows the agent to follow the policy of a previously solved, but different, task (called a source task) while it is learning a new task (called a target task). Specifically, I introduce *option-based policy transfer* (OPT). OPT enables policy transfer by encapsulating the policy for a source task in an *option* (Sutton, Precup, & Singh 1999), which allows the agent to treat the policy of a source task as if it were a primitive action. A significant advantage of this approach is that if there are multiple source tasks, an option can be created for each of them, thereby enabling the agent to transfer knowledge from multiple sources and to combine their knowledge in useful ways. Moreover, this approach allows the agent to learn in which states of the world each source task is most applicable. OPT's approach to constructing and learning with options that represent source tasks allows OPT to greatly outperform existing policy transfer approaches. Additionally, OPT can utilize source tasks that other forms of transfer learning for reinforcement learning cannot.

Challenges for policy transfer include identifying sets of source tasks that would be useful for a target task and providing mappings between the state and action spaces of

source and target tasks. That is, it may not be useful to transfer from all previously solved source tasks. If a source task has a different state or action space than the target task, then a mapping between these spaces must be provided. To address these challenges, I introduce object-oriented OPT (OO-OPT), which leverages object-oriented MDP (OO-MDP) (Diuk, Cohen, & Littman 2008) state representations to automatically detect related tasks and redundant source tasks, and to provide multiple useful state and action space mappings between tasks. I also introduce methods to adapt value function approximation techniques (which are useful when the state space of a task is very large or continuous) to the unique state representation of OO-MDPs.

Multi-Source Option-Based Policy Transfer

by

James MacGlashan

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Ph.D. in Computer Science
2013

For my wife, Lissa

ACKNOWLEDGMENTS

Numerous people through out my academic career have been instrumental in getting me to this stage of my life; I would not have been able to produce this work without the role they played and I would like to thank them for that.

I would first like to thank Marie desJardins, my doctoral advisor. Before I entered the doctoral program at UMBC, I was an undergraduate at UMBC and Marie gave me the rare opportunity to perform undergraduate research and gain a publication before I graduated. Soon after entering the doctoral program at UMBC, I asked Marie to be my advisor and she has since been critical in shaping me into an academic and making me more organized. Marie provided me the freedom and opportunity to explore many different interesting ideas in AI. As a result, I was able to make informed decisions about what I wanted to research. In doctoral programs, advisor horror stories are not altogether uncommon; however, I can happily say that I have never had a bad experience with Marie and she has always provided useful feedback in friendly and constructive ways.

I would also like to thank everyone in my doctoral committee; each of whom have played a role in my academic development. I would like to thank Tim Oates for the various discussions I've had with him over the years, both academic and casual. He has always been a source of knowledge and enthusiasm for AI and I am grateful that MAPLE and CORAL shared lab space for most of my years at UMBC. I would like to thank Michael Littman for his feedback on my proposal and the subsequent projects on which we've collaborated. Working with him has been informative and fun, and I look forward to continuing to work with him in the future. I would like to thank Tim Finin for his feedback on my proposal and work and over the years, as well as his efforts to increase the community and involvement of graduate students in the CS department. Yun Peng taught a number of the AI courses

that I've taken at UMBC I would like to thank him for all of the knowledge that he has provided and the discussions that we've had that has encouraged me to continue with the field.

Outside of my committee, numerous other people in the UMBC CS department have been a source of feedback, academic commiseration, and friendship and I'd like to thank all of them for that. In particular, I'd like to thank: Richard Adjogah, Adam Anthony, Blazej Bulka, Rebecca Chhay, Niyati Chhaya, Shelby Clarke, Eric Eaton, Rémi Eyraud, Wesley Griffin, Yasaman Haghpanah, Peter Hamilton, Genaro Hernandez, Niels Kasch, Zachary Kurtz, Don Miner, Max Morawski, Alex Morrow, Sourav Mukherjee, Patti Ordóñez, Marc Pickett, Soumi Ray, Matt Schmill, Shawn Squire, Abigail Williams, and Kevin Winner.

I also could not have done this without the support of my friends and family. My parents, Donald and Sherry MacGlashan, have always nurtured my interest in computers and science and I would not have started a Ph.D. program without their life long encouragement. I also must give enormous thanks to my loving wife, Lissa MacGlashan. She has been an amazing source of encouragement through these many years of education and she has always offered unconditional support when things were difficult.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

If an agent has to complete the same task many times, it is often beneficial for the agent to learn a solution, or *policy*, for the task. Once a policy to perform a task is learned, an agent can complete the task without extensive deliberation about how to behave. Traditional learning algorithms, such as reinforcement learning algorithms, are very effective at learning policies to specific well defined tasks. However, if an agent must complete a novel variant of the task that is similar to, but not exactly the same as, a previous version for which it has already learned a policy, learning must begin anew and there is no benefit to having previously learned anything. Unfortunately, because the world is so dynamic, facing slightly different versions of a task is common. For instance, consider a problem as seemingly static as typing on a new keyboard. The keys between keyboards are basically the same with the same arrangement, but slight changes in the keys' position, shape, depression distance, etc. result in a slightly different task. These differences, though seemingly trivial, have real consequences, as evidenced by the fact that when a person switches to a new keyboard, they may need to spend some time adjusting before they are typing at their maximum speed again. However, a person also does not have to relearn how to type on a new keyboard as though it was the first time they started typing. Instead, they will type reasonably well on a new keyboard and quickly adjust back to their best speed and

accuracy.

Tasks may also differ in larger ways, but still be strongly related such that learning should not begin anew on each variant. For instance, consider a game of "keep away" in which there are two teams: keepers and takers. The goal of the keepers is to keep possession of a ball; the goal of the takers is to gain possession of the ball. The takers will move into positions to try to take the ball and the keepers pass the ball between each other and move around the field in order to prevent the takers from doing so. The keeper in possession of the ball must decide whether to hold onto the ball or pass it to one of his teammates. This task may vary depending on how many keepers and how many takers there are. For instance, in one task, there might be a total of three keepers and four takers; in another, there might be four keepers and three takers. Although these two tasks would require the keeper in possession of the ball to consider the position of a different number of other agents and to consider a different number of passing opportunities, ideally, the keeper should not have to learn from scratch how to play four vs. three keep away if they had already learned how to play three vs. two keep away.

Given how dynamic the world is, the ability to adapt and generalize knowledge to new situations is critical for the development of autonomous agents. Recent research in reinforcement learning has explored various different forms of knowledge reuse (see Chapter 6). One such method of knowledge reuse is *policy transfer*, which allows the agent to reuse the policy from previously learned tasks while learning the policy for a different current task. Policy transfer algorithms enable an agent to have better initial performance on a new task as well as accelerating the learning process. However, existing policy transfer approaches have shortcomings, such as only being able to provide transfer from one previous task at a time, or limiting the ways in which the agent uses an existing policy. The contribution of my dissertation research is a set of novel approaches and algorithms for policy transfer that address the shortcomings of existing policy transfer algorithms, thereby

further improving an agent's initial performance on a new task and further accelerating the learning process. In particular, these contributions enable an agent to (1) learn in which states to use policy transfer and select from among multiple tasks when transferring knowledge, (2) transfer policies from tasks with different state representations, and (3) integrate these policy transfer techniques with function approximation, which is often necessary for learning tasks with continuous state spaces.

1.1 Option-based Policy Transfer

In this work, I present a novel approach to policy transfer called *Option-based Policy Transfer* (OPT), which enables policy transfer by casting the problem as an *option learning problem* (Sutton, Precup, & Singh 1999). Options are temporally extended actions, or policies, that lead the agent to some option-defined termination state and that can be used much like a typical action. Although options are typically used as macro-actions that allow the agent to complete important subgoals of a task and thus more rapidly reach the ultimate goal, an option need not be limited to achieving the subgoals of a task. In fact, if the policy for an option brought an agent very close to its ultimate goal—as would be expected if the policy of a similar task were followed—the option would be even more useful than an option that achieves a more distant subgoal. This property motivates the choice to cast policy transfer as a option learning problem. Specifically, in OPT, when an agent is presented with a new task, it creates an option for each similar task that it has previously solved and then applies modified option learning techniques. I refer to these kinds of options *Transfer Options* (TOPs). Because the option framework can perform learning with any number of options, transfer learning can be performed with as many source tasks as desired, thereby providing multi-task transfer. Constructing options from previous tasks for use in new tasks requires only a state space and action space mapping between the tasks (if the state and ac-

tion spaces between the tasks are different), and a few parameters that do not require much domain knowledge.

Although there is existing work that uses options with state and action mappings between tasks to enable policy transfer (Soni & Singh 2006), this previous work used an option learning algorithm and option construction approach that performs poorly in a large range of transfer scenarios. As a result, this previous work was only ever effectively applied to transfer from a single previously solved task that could be related to the current target task being learned in multiple ways. OPT's contribution is an option learning algorithm and method for constructing options from source tasks that enables an agent to greatly accelerate its learning performance across a range of transfer problems. OPT's approach to learning also enables the agent to effectively combine the policies from multiple source tasks, thereby providing even greater performance gains.

The key to OPT's performance gains is in how it balances the agent's reliance on TOPs. If there is an underreliance on TOPs, then there is little chance for the agent to learn the ways in which the source task is useful for transfer. If there is an overreliance on TOPs, then this may be equally bad, because while the agent should benefit from transferring the policy of a source task, it is not expected that the transferred policy will be useful in all states. Therefore, if there is an overreliance, the beneficial applications of a TOP would be canceled out by the detrimental applications. Soni and Singh's policy transfer work suffers from an underreliance because the learning algorithm only enables an agent to execute a maximum of one step of a TOP, after which the agent might switch to a different TOP or a different action. OPT addresses this problem of underreliance by using a learning algorithm that executes a TOP to its natural termination conditions, while simultaneously learning about the current task's policy. To prevent overreliance, OPT limits the conditions under which a TOP can be executed to states in which the source task's policy is well explored (and therefore will have a meaningful policy) and also requires that a TOP always terminate

in any state with some probability, thus preventing very long or indefinite execution of a TOP.

1.2 Object-Oriented OPT

Although OPT is a very flexible approach for incorporating multi-task policy transfer into RL algorithms, it requires that a state and action space mapping be provided between tasks and requires the user to specify the set of previous source tasks that should be used for transferring knowledge to each new task. In Chapter 4, I present Object-Oriented OPT (OO-OPT), which automates the generation of mappings between tasks and, given a library of tasks, determines related subsets of these tasks between which transfer is relevant, and provides heuristics to further filter the the set of tasks that any new task should use for transfer.

Although a state and action mapping is trivial when two tasks have identical state and action spaces (but differ in how actions affect the world or in how the reward function is defined), such a mapping is more difficult to provide when two tasks have different state and action spaces. I present a method to automatically provide this mapping by leveraging the Object-oriented MDP (OO-MDP) state representation (Diuk, Cohen, & Littman 2008). OO-MDP is a factored representation that partitions a state into a collection of objects, with each object belonging to a class that defines a feature vector for the object. Tasks with different state representations of the same OO-MDP domain are tasks whose states contain a different number of objects of each possible class. By leveraging the object representation, multiple valid state mappings between two tasks can be generated by finding all of the ways in which the sets of objects of the same class in each task could be mapped to each other. Because OPT supports multi-task policy transfer, OPT can use *all* of these possible mappings for each task by creating a parametric option and letting the agent choose

from among the possible mappings. If the action space of an OO-MDP task is dependent on the objects present, then tasks with different objects will have not just different state spaces, but different action spaces. However, actions may also be parameterized by the objects on which they act and therefore an action space mapping is provided by the same object-wise mapping used to create the state space mapping.

In addition to this object-oriented state representation, OO-MDPs also define a set of propositional functions that operate on the objects present in a state. If a task's goal and reward function are defined as a first-order logic (FOL) expression using these propositional functions, then this goal and reward function can be mapped to tasks with different objects and object values. Therefore, a goal/reward function defined by a FOL expression of OO-MDP propositional functions defines a set of related tasks between which transfer can be used. Among the tasks in the set defined by a goal/reward FOL expression, some will be more similar to the current task than others depending how similar their state representation is. This similarity measure in turn allows an agent to automatically determine from which tasks it would be most useful to transfer.

1.3 Object-Oriented Value Function Approximation

While policy transfer provides a crucial form of learning generalization, it does not provide the kind of learning generalization that is necessary for tasks with extremely large or continuous state spaces. In such domains, an agent is unlikely to visit the same state many times, and may often encounter previously unvisited states. Using standard reinforcement learning algorithms, the agent will never be able to learn the appropriate policy for each state. This problem is typically resolved by using *value function approximation* (VFA), which allows an agent to estimate the appropriate policy from similar states. VFA RL algorithms typically require a single feature vector to be used to represent each state.

However, OO-MDPs, which are leveraged to provide automatic state mappings and task similarity measures, are represented by an unordered collection of objects, each defined by its own feature vector, which prevents the use of typical VFA algorithms. In order to benefit from OO-MDP's advantages with policy transfer and to enable the agent to solve complex problems with very large or continuous state spaces, I introduce a class of techniques that can be applied to any discretization based VFA—such as *cerebellar model articulator controllers* (CMACs) (Albus 1971)—or any instance-based VFA—such as *sparse distributed memories* (SDMs) (Ratitch & Precup 2004). For discretization approaches, efficient VFA can be performed by discretizing at the object level, computing hash functions on each object in a state, and then combing the hashes of each object in an ordered way. In instance-based VFA, a database of states and associated weights that spans the entire state space is maintained. To predict the value for a new query state, the result is some distance-based function from the query state to the stored states and their associated weights. I generalize such instance-based approaches by defining a distance metric that can be computed between sets of objects that are each defined by a feature vector. I discuss the implementation details for both of these approaches and present empirical results in Chapter 5.

1.4 Contributions

This work provides a number of important contributions to learning agent control:

- An approach to policy transfer (Option-based Policy transfer, or OPT) that enables an agent to learn when and from which source task to transfer in each state that is robust to various differences between tasks, including their termination states, reward functions, transition dynamics, and state representations. OPT improves an agent's initial performance on a new task while accelerating its learning process on new tasks. Although using options to perform policy transfer is a concept used previously

by Soni and Singh (2006), OPT's approach enables the agent to benefit from transfer learning in scenarios that Soni and Singh's approach cannot handle, significantly improving performance.

- An algorithm called Object-Oriented OPT (OO-OPT), which leverages the Object-Oriented MDP (OO-MDP) state representation to define sets of related tasks between which transfer is beneficial, determine which of those tasks should be used for transfer in a new task, and automatically provide multiple state and action space mappings between which an OPT agent learns to select.

- A generalization of discretization-based VFA and instance-based VFA approaches, which enables function approximation with tasks using the OO-MDP state representation, thereby extending the automatic transfer capabilities of OO-OPT to continuous or very large state space domains.

The value of these contributions is extremely important in the quest to make fully autonomous robots that learn how to solve tasks on their own. A critical challenge in achieving this goal is the ability for robots to be able to generalize what they learn to the extremely dynamic nature of the world. Virtual agents may also benefit from this work, since they too may need to deal with many different versions of a problem. For instance, non-player controllable agents in video games can benefit from this work, since a game environment can vary in many ways while still maintaining the same basic goal.

Chapter 2

BACKGROUND

The contributions of this dissertation leverage and extend reinforcement learning techniques and related representations and methods including Markov Decision Processes (MDPs) and Object-oriented Markov Decision Processes (OO-MDPs) (Diuk, Cohen, & Littman 2008), State-Action-Reward-State-Action (SARSA) learning (Rummery & Niranjan 1994), Q-learning (Watkins & Dayan 1992), value function approximation (Rummery & Niranjan 1994; Sutton 1988), and options (Sutton, Precup, & Singh 1999). In this chapter, I review all of these topics.

2.1 Markov Decision Processes

A Markov Decision Process (MDP) is a specific formulation of a decision-making problem that is defined by a four-tuple: $\{\mathcal{S}, \mathcal{A}, \mathcal{P}_\cdot(\cdot,\cdot)\mathcal{R}_\cdot(\cdot,\cdot)\}$, where $\mathcal{S}$ is the set of possible states in which the agent can find itself; $\mathcal{A}$ is the set of actions the agent can take, with $\mathcal{A}_s \subseteq \mathcal{A}$ being the set of actions an agent can take in state $s \in \mathcal{S}$; $\mathcal{P}_a(s, s')$ is the probability of the agent transitioning from state $s \in \mathcal{S}$ to $s' \in \mathcal{S}$ after applying action $a \in \mathcal{A}_s$—these probabilities are called the *transition dynamics*; and $\mathcal{R}_a(s, s')$ is the reward received by the agent for transitioning from state $s \in \mathcal{S}$ to $s' \in \mathcal{S}$ after executing action $a \in \mathcal{A}_s$. An agent interacts with an MDP over a sequence of discrete time steps ($t = 0, 1, 2, \cdots$). At each

time step (t), an agent perceives its current state ($s_t \in \mathcal{S}$) and chooses an action ($a_t \in \mathcal{A}_s$). Applying a_t in s_t produces the next state $s_{t+1} \in \mathcal{S}$, according to the probability function $\mathcal{P}_{a_t}(s_t, \cdot)$, and yields the reward $r_{t+1} = \mathcal{R}_{a_t}(s_t, s_{t+1})$.

Given an MDP definition, the goal of an agent is to find a *policy* that maximizes the expected discounted future reward. A policy is a mapping from states to probabilities of actions: $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$. The notation $\pi(s, a)$ refers to the probability of taking action a in state s under policy π; $\pi(s_t)$ refers to the action selected by policy π in state s at time t. The expected discounted future reward from a given state under a certain policy can expressed as the *state value* function:

$$\begin{aligned} V^{\pi}(s) &= E\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} \cdots | s_t = s, \pi\} \\ &= E\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) | s_t = s, \pi\} \\ &= \sum_{a \in \mathcal{A}_s} \pi(s, a) \sum_{s' \in \mathcal{S}} \mathcal{P}_a(s, s') \left[\mathcal{R}_a(s, s') + \gamma V^{\pi}(s')\right], \end{aligned} \tag{2.1}$$

where $\gamma \in [0, 1]$ is a *discount* parameter that affects how much an agent is concerned about future rewards in comparison to more immediate rewards. When $\gamma = 0$, the agent only cares only about immediate rewards and when $\gamma = 1$, an agent cares equally about immediate and future rewards. If an MDP can last for an infinite amount of time, then γ should be set to less than one, because for a γ value of one, the total return over an infinite time may be infinite, but for a value less than one, it is bounded.

The optimal state value function is the value function for an optimal policy; that is, it is the value function under a policy that maximizes expected discounted future reward.

Formally, the optimal value function is defined as:

$$\begin{aligned} V^*(s) &= \max_{\pi} V^{\pi}(s) \\ &= \max_{a \in \mathcal{A}_s} E\{r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a\} \\ &= \max_{a \in \mathcal{A}_s} \sum_{s' \in \mathcal{S}} \mathcal{P}_a(s, s') \left[\mathcal{R}_a(s, s') + \gamma V^*(s')\right]. \end{aligned} \tag{2.2}$$

Therefore, if an agents finds the optimal state value function, the optimal policy can be derived by choosing an action that maximizes Equation 2.2. That is, the optimal policy π^* is derived as:

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}_s} \sum_{s' \in \mathcal{S}} \mathcal{P}_a(s, s') \left[\mathcal{R}_a(s, s') + \gamma V^*(s')\right] \tag{2.3}$$

A corollary to the state value function is the *state-action value* function (or *Q-value*), which represents the expected discounted future reward for taking a given action in a given state and then following the given policy thereafter. Formally, the state-action value function is defined as:

$$\begin{aligned} Q^{\pi}(s, a) &= \sum_{s' \in \mathcal{S}} \mathcal{P}_a(s, s') \left[\mathcal{R}_a(s, s') + \gamma V^{\pi}(s')\right] \\ &= \sum_{s' \in \mathcal{S}} \mathcal{P}_a(s, s') \left[\mathcal{R}_a(s, s') + \gamma \sum_{a' \in \mathcal{A}_{s'}} \pi(s', a') Q^{\pi}(s', a')\right]. \end{aligned} \tag{2.4}$$

The optimal state-action value function would therefore be the state-action value function for an optimal policy:

$$\begin{aligned} Q^*(s, a) &= \max_{\pi} Q^{\pi}(s, a) \\ &= \sum_{s' \in \mathcal{S}} \mathcal{P}_a(s, s') \left[\mathcal{R}_a(s, s') + \gamma \max_{a' \in \mathcal{A}_{s'}} Q^*(s', a')\right]. \end{aligned} \tag{2.5}$$

As with the optimal state value function, if an agent finds the the optimal state-action value function, the optimal policy can be derived from it. However, a useful property of the state-action value function is that this policy can be derived without knowing the transition dynamics. That is, given the optimal state-action values, the optimal policy can be derived from each state by taking any action with the maximum Q-value for that state:

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}_s} Q^*(s, a). \tag{2.6}$$

Many decision problems contain terminal states which, once reached, terminate the actions of an agent. Formally, a terminal state in an MDP is any state (s) for which $\mathcal{A}_s = \emptyset$. Terminal states are common in goal-directed tasks, in which there is some goal condition that defines a set of goal states that an agent is trying to reach. Once the agent reaches any of these goal states, the task is complete and there is no more reason for the agent to act. For instance, if an agent is trying to escape a maze, any exit point of the maze would be a goal state. Such goal states are therefore terminal states, because once they are reached, the agent will not continue acting (or at least the agent will not continue acting in the context of the defined MDP). Terminal states may also be defined for failure conditions. For instance, if an agent must drive a car somewhere runs out of gas, then it will no longer be able to act.

The existence of terminal states makes an MDP *episodic*, where an episode consists of all of the visited states and actions taken by an agent from some initial state of an MDP to a terminal state. Typically, when an episode is completed by reaching a terminal state, the agent is reset to an initial state of the MDP. When dealing with such episodic MDPs, value functions are typically defined in terms of the cumulative discounted reward until a terminal state is reached, rather than including experiences in subsequent episodes.

2.1.1 Object-Oriented Markov Decision Processes

An Object-oriented Markov Decision Process (OO-MDP) (Diuk, Cohen, & Littman 2008) is an extension of the MDP formalism to include additional information. Specifically, OO-MDPs add a set of classes $\mathcal{C}$, where each class ($c \in \mathcal{C}$) has an associated set of attributes ($\mathcal{T}_c$). Each object (o) belongs to a class ($c \in \mathcal{C}$) and has instantiated values for each attribute in $\mathcal{T}_c$, which can be represented as a feature vector of o: $o.\vec{v}$. If two objects (o_1 and o_2) belong to the same class and $o_1.\vec{v} = o_2.\vec{v}$, then $o_1 = o_2$. A state of an OO-MDP ($s \in \mathcal{S}$) is defined as a non-empty set of objects: $s = \{o_1, o_2, \cdots\}$. If there exists a bijection between the objects of two states: $f : s_1 \rightarrow s_2$ such that $o = f(o)$ and $o' = f^{-1}(o')$ for all $o \in s_1$ and $o' \in s_2$, then $s_1 = s_2$. In other words, two states are equal if they have a set of equivalent object instantiations.

In addition to these object extensions to MDPs, an OO-MDP also defines a set of propositional functions ($\mathcal{F}$). The propositional functions are defined for objects of specific classes and are evaluated on objects that belong to the same state. For instance, in a blocks world OO-MDP with a "block" class, a propositional function "on" might be defined for block classes: $\texttt{on}(block, block)$, which returns true when the attribute values of the block object parameters indicate that the first block is on top of the second (or false otherwise). In the original OO-MDP work, these propositional functions were used to facilitate the learning of the MDP transition dynamics (Diuk, Cohen, & Littman 2008).

2.2 SARSA and Q-learning

In this section, I discuss how an agent can learn an optimal policy for an MDP using State-Action-State-Reward-Action (SARSA) learning, Q-learning, and SARSA(λ).

2.2.1 SARSA

Algorithm 1 SARSA

Initialize $Q(s, a)\ \forall\ s$ and a
repeat forever
  Initialize s
  Choose a from s using a policy derived from Q
  while s is not terminal
    Take a, observe s' and r
    Choose a' from s' using a policy derived from Q
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$
    $a \leftarrow a'$
  end while

Given that an optimal policy can be derived from the optimal Q-values, if an agent can learn the optimal Q-values of an MDP, it can learn the optimal policy. This observation motivates State-Acton-Reward-State-Action (SARSA) learning (Rummery & Niranjan 1994), which is a form of *temporal difference* (TD) learning (Sutton 1988). SARSA learning is a form of "model-free" learning, which means the agent does not need to know the transition dynamics of an MDP. Instead, the agent only needs to know the state they are in at any given time and to receive a reward for each step.

The SARSA algorithm (shown in Algorithm 1) starts by initializing every Q-value to an arbitrary value. From an initial state s, the agent chooses an action a that is derived from its current Q-value estimates for s. After applying action a, the agent observes the reward received (r) and the resulting state (s'). Using the current estimated Q-values for s', the agent again determines what its next action a' will be, then updates its estimate for the previous state-action's Q-value ($Q(s, a)$) using the following equation:

$$Q(s, a) := Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)], \tag{2.7}$$

where α is a learning rate between zero and one and γ is the discount parameter defined for

the value function (see Equations 2.1 and 2.4). The agent repeats this process for the next state. If the agent finds itself in a terminating state, then the process repeats itself from an initial state of the problem. SARSA converges to the optimal Q-values (and optimal policy) if every state-action pair is taken infinitely many times and if the policy derived from the Q-values converges in the limit to a greedy policy (Rummery & Niranjan 1994) (i.e., one that greedily selects any action with the maximum Q-value).

There are two important choices to be made in the SARSA algorithm: (1) how to initialize the Q-values and (2) how to define a learning policy that is derived from the current Q-values. The convergence properties of SARSA hold regardless of the initialization value, but in practice, the choice of initial values can have large effects on the performance and learning speed. This behavior is trivially demonstrated by considering the learning time of an agent whose Q-values are initialized to the Q^* values. In this case, there would be nothing to learn and the learning time would be zero. Similarly, as the initial values are moved away from the true values, it takes more updates to converge to the true values. Initial values may therefore be used to reflect prior knowledge about the domain. If no prior knowledge is available, the choice of Q-value initialization may also still be selected in informed ways that alter behavior. For instance, if the Q-values are initialized optimistically, such as being set to the maximum reward, then exploration is encouraged initially and is decreased as the values are learned (Sutton & Barto 1998). The result of greater initial exploration can mean worse initial performance since the agent is randomly taking actions, but then the agent will quickly converge to a near optimal policy. In contrast, an agent with pessimistically initialized Q-values will exploit useful actions earlier, resulting in better initial performance, but it may converge to the optimal policy more slowly.

The learning policy derived from the current Q-values used for SARSA is another important choice that the designer must make. One obvious choice is to use a greedy policy. However, such an algorithm will not guarantee convergence of SARSA to the optimal pol-

icy, because convergence to the optimal policy requires that each state-action pair is taken an infinite number of times. Instead, some level of constant exploration is required. Using optimistically initialized Q-values will not resolve this difficulty, because the exploration provided by doing so is temporary. To resolve these issues, one common choice for the learning policy is an ϵ-*greedy* policy, which selects an action at random with probability ϵ and greedily selects an action with probability $1-\epsilon$. Although this satisfies the requirement that every state-action pair is taken an unbounded number of times, SARSA will not converge to the optimal Q-values, but will instead converge to the ϵ-greedy policy's Q-values. In order for SARSA to converge to the optimal Q-values, the learning policy must also converge to the greedy policy in the limit. One way to achieve such convergence is to slowly decrease ϵ in such a way that it converges to zero in the limit. This approach will guarantee convergence to the optimal Q-values (and policy); however, as with choosing optimistic or pessimistic initial values, there is a tradeoff in learning speed and initial performance that is determined by the degree of exploration and by how fast ϵ is decreased (and by the value at which it starts). This recurring dilemma is known as *exploration versus exploitation*. There are other learning policies in the literature that satisfy the requirements for SARSA to converge to the optimal Q-values and policy, but they will not be discussed at length in this work, because they are and tradeoffs and benefits for each that are not important to the underlining learning theory.

2.2.2 Q-learning

An algorithm similar in nature to SARSA is *Q-learning* (Watkins & Dayan 1992), shown in Algorithm 2. Q-learning proceeds in much the same way as SARSA: Q-values are initialized to an arbitrary value, and the agent takes actions for each state based on a policy derived from the current Q-values. The subtle, but critical, difference between SARSA and Q-learning is the Q-value update step. Instead of using the Q-value of the

Algorithm 2 Q-learning

Initialize $Q(s, a)\ \forall\ s$ and a
repeat forever
  Initialize s
  while s is not terminal
    Choose a from s using a policy derived from Q
    Take a, observe s' and r
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$
  end while

action taken by the agent in the next state to update the current Q-value, Q-learning uses the maximum Q-value of the next state. Formally, the Q-learning Q-update rule is:

$$Q(s, a) := Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]. \tag{2.8}$$

The SARSA algorithm is considered "on-policy" because it learns the Q-values of the policy it is following. In contrast, Q-learning is considered "off-policy" because regardless of the policy the agent is actually following, as long as every state-action pair is taken infinitely many times, the Q-values learned are the Q-values of the optimal policy. The consequence of this difference is that Q-learning does not require its learning policy to converge to the greedy policy in order to learn the optimal Q-values. However, if the agent wanted to follow the optimal policy, it would of course have to use a greedy policy, assuming that the optimal Q-values were sufficiently approximated.

Whether Q-learning or SARSA is preferred depends on what the designer wants to accomplish. It may seem that Q-learning is always better, since Q-learning will directly learn the optimal Q-values rather than converging to them as its learning policy converges to the greedy policy. However, there may be performance advantages with SARSA while the agent is learning and following a non-greedy policy. For example, consider a grid

world in which an agent must traverse a cliff that the agent does not want to walk off. If the optimal policy of this problem is for the agent to walk along the edge of the cliff, then the Q-values learned by Q-learning will reflect this policy. However, if the agent is actually following an ϵ-greedy policy, rather than a greedy policy, then walking along the edge of the cliff, as indicated by the learned Q-values, is not appropriate because the random action selection of an ϵ-greedy policy will result in the agent stepping off the ledge with some probability. Instead, for an ϵ-greedy policy, the optimal behavior would be to walk further from the edge so that when a random action is taken, it will not result in the agent falling off. SARSA, unlike Q-learning, will learn this safer policy; therefore, even though Q-learning would directly learn the optimal Q-values, performance with SARSA would be better since in practice an ϵ-greedy policy is being followed rather than a greedy one. This example, and empirical results supporting it, were provided by Sutton and Barto (1998).

2.2.3 SARSA(λ)

Because SARSA is a form of TD-learning, it can be generalized to a class of algorithms, called SARSA(λ) (Rummery & Niranjan 1994), that make use of *eligibility traces* (Sutton 1988). An eligibility trace can be thought of as a record of all experiences within an episode. Given this record, the Q-value of a state visited earlier in an episode may be updated not just by the immediately following experience (the immediate reward received and the Q-value of the next taken action), but also by every subsequent experience.

SARSA(λ) is shown in Algorithm 3 and is very similar to typical SARSA. The critical difference is the addition of a set of state-action eligibility values ($e(s, a)$) that are initialized to zero at the start of every episode (line 4). Every time an action is taken, the difference between the reward and discounted Q-value from the previous Q-value estimate is computed and stored in the value δ (line 9) and the eligibility value for this state-action pair is increased by one (line 10). Note that δ is set to the same difference used to update

Algorithm 3 SARSA(λ)

1: Initialize $Q(s, a)\ \forall\ s$ and a
2: **repeat forever**
3: Initialize s
4: Initialize $e(s, a) \leftarrow 0\ \forall\ s$ and a
5: Choose a from s using a policy derived from Q
6: **while** s is not terminal
7: Take a, observe s' and r
8: Choose a' from s' using a policy derived from Q
9: $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
10: $e(s, a) \leftarrow e(s, a) + 1$
11: For all $s\ a$:
12: $Q(s, a) \leftarrow Q(s, a) + \alpha\delta e(s, a)$
13: $e(s, a) \leftarrow \gamma\lambda e(s, a)$
14: $s \leftarrow s'$
15: $a \leftarrow a'$
16: **end while**

Q-values in typical SARSA. For every state-action pair in the MDP, the Q-value is updated with respect to the learning rate (α), the immediate difference (δ), and the eligibility value ($e(s, a)$) for that state-action pair (line 12). Following that, every state-action eligibility value is decayed by the discount parameter (γ) and the parameter λ (line 13). The consequence of these changes is that every step in an episode has its Q-value updated to reflect all future experiences instead of just the immediate experience. The degree to which they are affected by future experiences is controlled by the parameter λ. If $\lambda = 1$, then the effect of future experiences is fully considered and used to update every Q-value that was previously visited in the episode. As λ is reduced from 1, the effect of future experiences is scaled down; at $\lambda = 0$, SARSA(λ) behaves identically to typical SARSA, which only updates Q-values with respect to the immediate subsequent experience. For this reason, typical SARSA is often called SARSA(0) or one-step SARSA.

Because Q-learning uses off-policy learning, modifying it to use eligibility traces is not as straightforward. However, with some limitations, different eligibility trace versions

Algorithm 4 Linear Gradient Descent SARSA(λ)

Initialize $\vec{\theta}_a$ arbitrarily $\forall\ a$
repeat forever
  Initialize s
  For all $a \in \mathcal{A}_s$
    $Q_a \leftarrow \sum_{i=0}^{n} \theta_a(i)\phi_s(i)$
    Initialize $\vec{e}_a \leftarrow \vec{0}$
  Choose a using a policy derived from Q
  while s is not terminal
    Take a, observe s' and r
    For all $b \in \mathcal{A}$
      $Q'_b \leftarrow \sum_{i=0}^{n} \theta_b(i)\phi_{s'}(i)$
      $\vec{e}_b \leftarrow \gamma\lambda\vec{e}_b$
    Choose a' using a policy derived from Q'
    $\delta \leftarrow r + \gamma Q'_{a'} - Q_a$
    $\vec{e}_a \leftarrow \vec{e}_a + \vec{\phi}_s$
    $\vec{\theta}_a \leftarrow \vec{\theta}_a + \alpha\delta\vec{e}_a$
    $s \leftarrow s'$
    $a \leftarrow a'$
  end while

of Q-learning have been suggested (Watkins 1989; Peng 1993), though I do not cover them here.

2.3 Value Function Approximation

SARSA and Q-learning are effective approaches to learn the optimal policy for an MDP when the state space of an MDP is tractable. However, if the state space is extremely large or infinite (as would be the case in a continuous state space), then these approaches are not effective, because they require an agent to be able to visit a state many times to update its Q-values. If the state space is extremely large or infinite, it may be exceedingly unlikely for an agent to even visit any state more than once, in which case the Q-values for each state are always poorly estimated, resulting in a poor policy.

To address these challenges, experience generalization can be used so that experience in one state can be generalized to many states. *Value Function Approximation* (VFA) is one of the most common ways to provide experience generalization. Instead of learning individual Q-values for each state-action pair, VFA learns a Q-value function of state features for each action. States with similar state features will therefore produce similar output from the learned function. In effect, the Q-update step becomes a training example for a regression problem.

The most common way to solve the regression problem in the context of SARSA(λ) is to use a linear function of the state features. That is, the Q-function being learned is defined as:

$$Q(s,a) = \sum_{i=0}^{n} \theta_a(i)\phi_s(i), \tag{2.9}$$

where $\theta_a(i)$ is the ith scalar parameter (or weight) for action a and $\phi_s(i)$ is the ith state feature value for state s. Given such a function definition, gradient descent methods can be incorporated into SARSA(λ) to learn this linear function, instead of the explicit Q-values for each state-action pair. Linear gradient descent SARSA(λ) is shown in Algorithm 4. The primary differences between linear gradient descent SARSA(λ) and typical SARSA(λ), are that (1) instead of a tabular list of Q-values and eligibility values, a weight vector and eligibility vector for each action are maintained and updated; (2) Q-values are derived as a linear combination of the action weight vector and the state feature vector (lines 5 and 11); and (3) the eligibility vector is incremented by an amount equal to the state feature vector (line 15) rather than being incremented by one.

Although linear gradient descent SARSA(λ) can be generalized to nonlinear functions, linear functions are typically chosen because they have good convergence guarantees. Specifically, linear gradient descent SARSA(λ) is guaranteed to converge to a weight

vector with error bounded by:

$$MSE(\vec{\theta}_\infty) \leq \frac{1-\gamma\lambda}{1-\gamma} MSE(\vec{\theta}^*), \tag{2.10}$$

where $MSE(\vec{\theta})$ is the mean squared error with respect to the true Q-values using a function with the weight vector $\vec{\theta}$, $\vec{\theta}_\infty$ is the weight vector to which linear gradient descent SARSA(λ) converges in the limit, and $\vec{\theta}^*$ is the optimal weight vector that produces the minimal mean squared error (Tsitsiklis & Van Roy 1997). Note that if $\lambda = 1$, then linear gradient descent is guaranteed to converge to the optimal weight vector. This property seems to suggest that one should always set $\lambda = 1$; however, empirical studies have shown that convergence is often faster with lower values of λ (Sutton & Barto 1998).

It may seem that using linear function approximation is overly restrictive in the kinds of problems that can be solved. If an agent needs to navigate a grid world, for instance, a linear function over the agent's x-y coordinates will be insufficient to closely approximate the true value function. However, this is typically avoided by constructing state features that are nonlinear with respect to the actual state space variables and then learning a linear function of these nonlinear features. For instance, the set of state features may be a set of radial basis functions of the state variables. A review of common feature construction techniques for use with linear function approximation is provided by Sutton and Barto (1998).

Q-learning may also be adapted to use gradient descent function approximation, but because it is an off-policy learner, it does not have the same convergence guarantees as SARSA(λ) and may result in the mean squared error diverging to infinity (Baird 1995; Bradtke 1993). For this reason, SARSA(λ) is typically preferred for function approximation.

2.4 Options

VFA serves to make learning possible in domains with very large or infinite state spaces by generalizing the experience from one state to other similar states. However, some problems may still require longer than desired learning times if the MDP is goal-directed and the number of actions required to reach the goal is very large, because it will take a long time for the agent to both discover the goal and back up the reward to earlier states. The option framework (Sutton, Precup, & Singh 1999) addresses this problem by adding temporally extended actions, called options, to the action set, which turns the decision making problem into a special case of semi-MDPs (an MDP for which actions have durations). An option can be thought of as a temporally extended action, because the agent views an option as if it were a normal action, except that upon taking it, the option will result in the agent taking numerous steps, typically to some other specific state or specific set of states. By using options, an agent is able to efficiently navigate large regions of the state space. If the agent can reach a goal state in fewer options steps than action steps, then the agent will reach the goal sooner with options than without options, resulting in faster learning.

This section will first cover how to formalize options and reformulate MDPs to accommodate them, and then show how learning can be performed with options. The formalisms and techniques reviewed in these sections are derived from the original options work by Sutton, Precup, and Singh (1999).

2.4.1 Option Formalism

Formally, an option is a three-tuple: $\{\mathcal{I}, \pi, \beta\}$, where $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ is a policy, $\mathcal{I} \subseteq \mathcal{S}$ is a set of initiation states, and $\beta : \mathcal{S} \rightarrow [0, 1]$ is a set of termination conditions. For a given option (o), each of its three components are denoted by $o.\mathcal{I}$, $o.\pi$, and $o.\beta$, respectively.

An agent can only take an option (o) in a state (s_t) if $s_t \in o.\mathcal{I}$. If the agent takes o, then the agent will probabilistically take an action (a_t) according to $o.\pi(s_t, \cdot)$, which will result in state s_{t+1}. Option o then either terminates in s_{t+1} with probability $\beta(s_{t+1})$, releasing control back to the agent, or continues (with probability $1 - \beta(s_{t+1})$) and selects another action (a_{t+1}) according to $o.\pi(s_{t+1}, \cdot)$. This process continues until the option terminates.

In the above formulation, a *Markov* option is discussed. It is considered a Markov option because the policy and termination conditions are only dependent on the current state of the actual MDP of the problem. However, an option may also be *semi-Markov*. A semi-Markov option is one in which the policy or termination conditions are dependent on factors beyond the current state of the actual MDP. Examples of variations that make an option semi-Markov include: (1) an option that terminates after a certain number of steps past the execution of the option, regardless of the current state; (2) options whose policies or termination conditions are dependent on a history of previous states and actions; (3) options that use a state representation that is more detailed than the state representation of the policy that selects the option; and (4) options that execute lower-level options themselves. Although an agent can still use, reason, and learn with semi-Markov options, there are greater restrictions on what can be done with them than there are for Markov options, as will be noted later.

To include options in an MDP, an MDP is reformulated to have a set of options ($\mathcal{O}$) rather than a set of actions ($\mathcal{A}$). Just as there may be some subset of actions that are applicable in each state ($\mathcal{A}_s \subseteq \mathcal{A}$) there may also be a subset of options that are applicable in each state ($\mathcal{O}_s \subseteq \mathcal{O}$). This set of applicable options is explicitly defined as $\mathcal{O}_s = \{o \in \mathcal{O} : s \in o.\mathcal{I}\}$. It may be desirable for an agent to consider both the options available to it as well as the actions of the MDP. This can be naturally accomplished by recasting actions as a special case of options, called *primitive options*. A primitive option (o_a) for action a is

defined as:

$$\begin{aligned} o_a.\mathcal{I} &= \{s \in \mathcal{S} : a \in \mathcal{A}_s\} \\ o_a.\pi(s,a) &= 1 \; \forall s \in \mathcal{S} \\ o_a.\pi(s,a') &= 0 \; \forall s \in \mathcal{S} \wedge \forall a' \in \mathcal{A} : a' \neq a \\ o_a.\beta(s) &= 1 \; \forall s \in \mathcal{S}. \end{aligned}$$

In other words, a primitive option for an action (a) is defined to have its initiation states be the set of states in which a is applicable; the option's policy always takes action a in all states and never takes any other action; and the option always terminates in every state (so that the option can only execute the action once each time the agent chooses it).

Given that actions are replaced with options, the MDP is further reformulated to adjust the transition dynamics and reward function to respect these options. The transition dynamics can be reformulated as:

$$\mathcal{P}_o(s,s') = \sum_{k=1}^{\infty} p(s',k)\gamma^k, \tag{2.11}$$

where o is the option, k is the number of steps taken by the option, $p(s',k)$ is the probability of the option terminating in state s' after k steps, and γ is the typical discount parameter. In this formalism, $\mathcal{P}_o(s,s')$ represents the probability of the option being initiated in state s and terminating in state s' and is adjusted by how delayed the outcome will be with respect to γ. It should be noted that this value can be computed using the MDP's action transition dynamics coupled with the option's policy and termination conditions.

While reward functions for MDPs are defined in terms of the reward received for taking a certain action in a certain state and winding up in another certain state ($\mathcal{R}_a(s,s')$), when adapting a reward model for options, we can restrict the focus to the expected future

discounted reward received for taking a certain option in a certain state:

$$\mathcal{R}_o(s) = E\{r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{k-1} r_{t+k} \mid \mathcal{E}(o, s, t)\}, \tag{2.12}$$

where o is the option being considered, r_t is the reward received at time t, k is the number of steps taken by the option, γ is the standard discount parameter, and $\mathcal{E}(o, s, t)$ is the event that option o is taken in state s at time t. Similar to the option transition dynamics, this expected reward can be computed using the MDP's action transition dynamics, the MDP's reward function, and the option's policy and termination conditions.

Given these new formalisms, the state value function and action value function can also be reformulated to account for options. The state value function becomes:

$$\begin{aligned} V^{\mu}(s) &= E\{r_{t+1} + \cdots + \gamma^{k-1} r_{t+k} + \gamma^k V^{\mu}(s_{t+k}) \mid \mathcal{E}(\mu, s, t)\} \\ &= \mu(s, o) \left[\mathcal{R}_o(s) + \sum_{s' \in \mathcal{S}} \mathcal{P}_o(s, s') V^{\mu}(s') \right], \end{aligned} \tag{2.13}$$

where $\mu(s, o)$ is the probability that option o will be taken in state s under the option-aware policy μ. The state-action value function is similarly modified:

$$\begin{aligned} Q^{\mu}(s, o) &= E\{r_{t+1} + \cdots + \gamma^{k-1} r_{t+k} + \gamma^k V^{\mu}(s_{t+k}) \mid \mathcal{E}(o, s, t)\} \\ &= E\{r_{t+1} + \cdots + \gamma^{k-1} r_{t+k} + \gamma^k \sum_{o' \in \mathcal{O}_{s_{t+k}}} \mu(s_{t+k}, o')\, Q^{\mu}(s_{t+k}, o') \mid \mathcal{E}(o, s, t)\} \\ &= \mathcal{R}_o(s) + \sum_{s' \in \mathcal{S}} \mathcal{P}_o(s, s') \sum_{o' \in \mathcal{O}_{s'}} \mu(s', o')\, Q^{\mu}(s', o'). \end{aligned} \tag{2.14}$$

2.4.2 Learning with Options

Given the reformulated definitions for state value functions and state-action value functions for options, the previous MDP policy learning algorithms can be easily adjusted

to work with options. For Q-learning, the Q-update step (line 7 of Algorithm 2) is modified to:

$$Q(s,o) \leftarrow Q(s,o) + \alpha \left[r + \gamma^k \max_{o' \in O_{s'}} Q(s',o') - Q(s,o) \right], \tag{2.15}$$

where k is the number of steps taken by option o, and s' is the state in which option o terminated. For an on-policy paradigm like SARSA, the difference (in line 9 of Algorithm 3 and line 14 of Algorithm 4) would be modified to:

$$\delta \leftarrow r + \gamma^k Q(s',o') - Q(s,o). \tag{2.16}$$

Although the modifications to Q-learning and SARSA are sufficient for either of them to learn a policy over options, additional option learning opportunities include *intra-option value learning* and *option interruption*. Intra-option value learning is a different Q-value update method than the one used by Q-learning or SARSA that allows the Q-values for an option to be updated while another option or policy is being followed, if the option to be updated is a Markov option. For instance, if option o is being followed and takes the action a_t in state s_t at time t, then an option o' may update the value $Q(s,o')$ if a_t is an action selection consistent with the policy of o' ($o'.\pi$) and o' is a Markov option. While intra-option value learning may be used whenever those conditions hold, there are two special cases where these conditions will always hold. One case in which that condition always holds is when when $o' = o$; that is, intra-option value learning may be used to update the Q-values of an option for each step of its execution, rather than only for the first state in which it was initiated, as would be the case with only Q-learning or SARSA(λ). Another special case is intra-option value learning for primitive options. If a non-primitive option is being followed, intra-option value learning can be used on primitive options for every action taken by the non-primitive option.

To enable intra-option value learning, a *U-value* for state option pairs is introduced.

Algorithm 5 One-step Intra-option Value Learning

Initialize $Q(s, a)\ \forall\ s$ and a
repeat forever
  Initialize s
  while s is not terminal
    Choose action a according to policy $\pi(s)$
    Take a, observe s' and r
    for all $o' \in \mathcal{O} : s \in o'.\mathcal{I}$ and $a = o'.\pi(s)$
      $Q(s, o') \leftarrow Q(s, o') + \alpha[r + \gamma U(s', o') - Q(s, a)]$
    $s \leftarrow s'$
  end while

The U-value $U(s, o)$ represents the value of a state-option pair, given that the option o is executing before arrival in state s. The optimal U-values for a state-option are defined as:

$$U^*(s, o) = (1 - o.\beta(s))Q^*(s, o) + o.\beta(s) \max_{o' \in \mathcal{O}_s} Q^*(s, o'). \tag{2.17}$$

U-values can be used in off-policy and on-policy one-step paradigms to update the Q-values of the states visited during the execution of an option. In either the off-policy or the on-policy paradigm, the Q-values for states visited from some policy would be updated with the equation:

$$Q(s_t, o) \leftarrow Q(s_t, o) + \alpha \left[r_{t+1} + \gamma U(s_{t+1}, o) - Q(s, o)\right], \tag{2.18}$$

where for the off-policy paradigm, the U-value is defined as:

$$U(s_t, o) = (1 - o.\beta(s_t))Q(s_t, o) + o.\beta(s) \max_{o' \in \mathcal{O}_{s_t}} Q(s_t, o') \tag{2.19}$$

and for the on-policy paradigm, it is defined as:

$$U(s_t, o) = (1 - o.\beta(s_t))Q(s_t, o) + o.\beta(s)Q(s_t, o_t). \tag{2.20}$$

Given these one-step update rules, an agent could follow any arbitrary policy and update the Q-values (in either an off-policy or on-policy fashion) for all options consistent with the action selection after each step of the policy, as shown in Algorithm 5. Note that if an on-policy paradigm was being followed, the Q-values in Algorithm 5 would have to be updated after the next action selection was known.

Another learning opportunity with options is option interruption. Option interruption occurs when an agent is executing an option but prematurely terminates the option (before the option terminates on its own naturally), because the agent determines that it would be better to take another option from that state instead. More formally, if an agent following policy μ is executing option o at time t, and the current Q-value for the option ($Q^{\mu}(s_t, o)$) is less than the state-value ($V^{\mu}(s_t)$), then the agent may interrupt o and select a better option o' such that $Q(s_t, o') > Q(s_t, o)$. An intuitive way to conceptualize why option interruption is permissible is to imagine creating a new option set ($\mathcal{O}'$) that contains the same options as in $\mathcal{O}$, except that they have their termination conditions modified to be $\beta(s) = 1 \; \forall s : Q(s, o) < V(s)$; that is, they terminate when the Q-value of the option is less than the state value. A complete proof of this result can be found in the original options research (Sutton, Precup, & Singh 1999). Note that if the stored Q-values are very inaccurate, this approach will not be useful; moreover, for the approach to be most effective, the optimal Q-values need to be known. For a learning environment, this means option interruption should not be used from the start of learning, but only after a reasonable approximation of the Q-values has been learned.

Chapter 3

OPTION-BASED POLICY TRANSFER

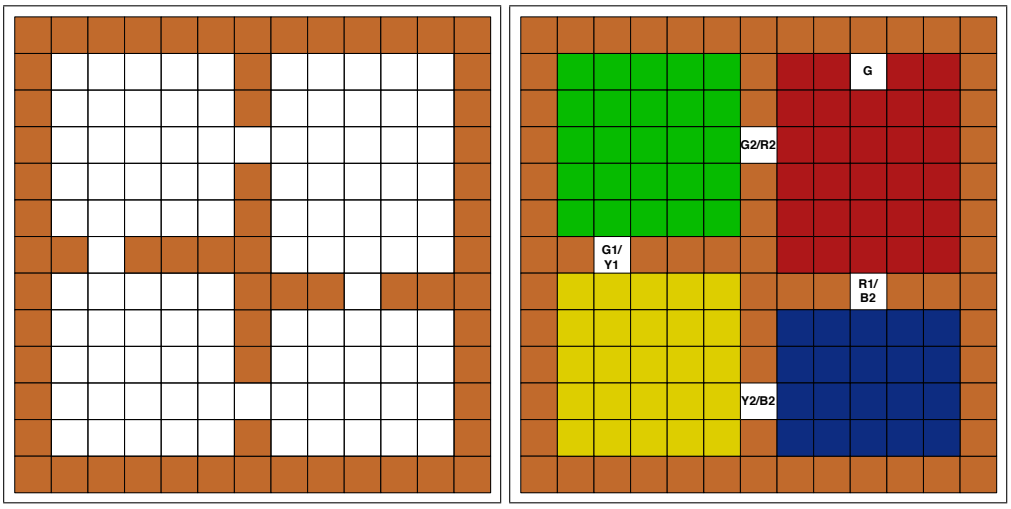
Reinforcement learning (RL) methods are effective at learning policies—mappings from states to actions—for decision making problems. One drawback to these approaches is that if an agent is faced with a new task that is even slightly different from a previous task it has solved, it must begin learning anew. To address this issue, transfer learning techniques can enable the agent to exploit previously learned knowledge. One form of transfer learning is policy transfer, which allows an agent to follow the policy from a different but similar task, called a *source task*, while learning a policy for the current task, called the *target task*. In this chapter, I present a novel approach to policy transfer called *Option-based Policy Transfer* (OPT). OPT has a number of advantages over existing policy transfer techniques, such as enabling an agent to learn in which states to use policy transfer and from which of multiple source tasks to transfer, all in an online fashion.

OPT provides online multi-source policy transfer by treating the policy transfer problem as an option learning problem (Sutton, Precup, & Singh 1999). Specifically, in addition to all primitive options (options that execute a specific action of the MDP), OPT creates a special option for each source task that follows that task's learned policy in the target task. I call these kinds of options *transfer options* (TOPs). Given a set of TOPs, standard temporal difference (TD) methods such as option-based Q-learning (Watkins & Dayan 1992)

or SARSA(λ) (Rummery & Niranjan 1994) can be used to learn a policy for the target task that incorporates transfer knowledge as well as knowledge about the current task. The *transfer policy* refers to the learned policy that is derived from and uses both primitive options and TOPs; the *target policy* uses only the primitive options and ignores TOPs. While TD methods allow a transfer policy to be learned, incorporating intra-option value learning methods accelerates the learning of both the transfer policy and target policy (by updating the Q-values of the MDP actions while a TOP is being followed) and option interruption can be used to allow the agent to terminate transfer from a source task in states for which that task is not useful.

Using options to as a means to provide policy transfer is not a unique concept itself and has been used previously by Soni and Singh (2006). However, the previous work used a learning algorithm and method of option construction that resulted in an underreliance on TOPs, which greatly limited the benefit that an agent could receive from TOPs. Moreover, in the previous work, the only constraints on the initiation conditions were that the state mapping from the target task to the source task resulted in a valid source task state and that the termination conditions were limited to states that mapped to terminal states in the source task. As a result, Soni and Singh's method was limited to transfer from tasks that had structurally similar state spaces with homomorphic transition dynamics that guaranteed termination. In contrast, OPT uses a learning algorithm and construction of TOPs that balances the agent's reliance on TOPs during while learning the target task, allowing an agent to more greatly benefit from policy transfer, apply it in a larger number of transfer scenarios, and combine the policies of multiple tasks in useful ways.

This chapter will first discuss the difference between TOPs and how options have typically been used to solve subgoals, then explain OPT's learning algorithm and the TOP construction methodology. Finally, numerous experiments are presented to demonstrate the effectiveness of OPT and to compare its performance to that of Soni and Singh's method



(a) Rooms Domain (b) Subgoal Options

FIG. 3.1. A rooms domain from the original options framework work with two options defined for each room, for a total of eight options. Colored cells represents the initiation states for two options available in that room.

and to another existing policy transfer approach called *Policy Reuse with Q-learning* (Fernández & Veloso 2005; 2006; Fernández, García, & Veloso 2010).

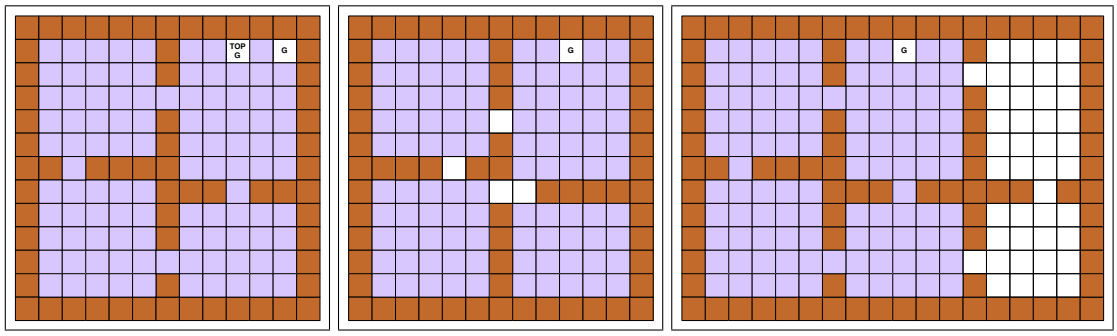
3.1 Transfer Options versus Subgoal Options

The core contribution of OPT is how policy transfer can be solved as an option learning problem. This section discusses the differences between the transfer options (TOPs) that OPT introduces, and options as they have typically be used in the past, which are henceforth referred to as *subgoal options* (SOPs).

TOPs and SOPs differ both in terms of the purpose of the option as well as how it is mechanically designed. When the option framework was first introduced (Sutton, Precup, & Singh 1999), the motivation behind options was to act as a sort of "macro-action" or "temporally extended action" of an MDP. That is, options were meant to be treated like

a typical action of an MDP, except that they would execute for multiple steps, thereby allowing the agent to cross a larger portion of the state space. If terminal goal states could be reached by executing fewer options than the number of actions it would need to take in the MDP without options, then the options would allow the agent to accelerate its learning process. To this end, options were motivated by the idea of solving subgoals of a task. Typically, this also meant that options specified a limited set of initial states from which it could be useful to achieve a subgoal, so that an agent only considered options when they were most relevant, making the options very local. To illustrate this purpose of using local options to solve subgoals, Figure 3.1 (a) shows an example grid world domain presented in the original options work (Sutton, Precup, & Singh 1999). In Figure 3.1 (b), the options constructed for this domain are shown, where each color represents a set of initiation states for two different options that take the agent to, and terminate in, one of the doorways in the room. For instance, the yellow room in the bottom left has two options, each with initiation states in all of the shaded cells of the room. One of the options will take the agent to the doorway in the cell labeled $Y1$ and another will take the agent to the doorway in cell $Y2$. This is similarly true for each of the four rooms, for a total of eight options. When the goal location is in the red shaded room (at the top right) at the cell labeled G, these options allow the agent to quickly navigate into the red room by moving to each door way, at which point the agent can use primitive options to reach the final cell.

As this above example illustrates, the core motivation for options in the original options work was to create local options defined for the given MDP that allow an agent to quickly move across the state space by achieving subgoals of an MDP. Subsequent work with the options framework retained the concept of using options to solve local subgoals and was oriented around identifying such options so that they could be used to accelerate the learning of a larger task (McGovern & Barto 2001a; 2001b; Menache, Mannor, & Shimkin 2002; Simsek & Barto 2004; 2007; Konidaris & Barto 2007;



(a) Different goal location (b) Different transition dynamics (c) Different state space

FIG. 3.2. Three different tasks in which a TOP derived from a previously solved task can be used. The purple shaded cells represent states that are initiation states for the TOP.

2009). Related work with the MaxQ framework (Dietterich 1998), which can be thought of a special case of the options framework with a deep option hierarchy and a specific form of learning with the hierarchy, also revolved around the idea of identifying ways to break up a task into a hierarchy of subgoals (Jonsson & Barto 2005; Marthi, Kaelbling, & Lozano-Perez 2007; Mehta *et al.* 2008). TOPs, however, depart from the idea of using an option to represent local subgoals of a task. Instead, a TOP is designed to work more globally and is meant to approximate the final goal of a task, rather than local subgoals of the task. Further, TOPs are derived from the polices of different MDP definitions that can differ in a variety of ways.

To illustrate how TOPs might be defined in contrast to SOPs, consider the previous Rooms example. Suppose a policy had been learned for the domain for the previous goal location in the red room. Figure 3.2 shows how a TOP using the previously learned policy might be constructed and used in three different tasks that are similar to the rooms task in Figure 3.1. The purple shaded cells in all subfigures represent the initiation states of the TOP. Figure 3.1 (a), shows how the TOP derived from the original task might be used

in a new task that differs in the location of the goal state (and therefore has a different reward function). Figure 3.1 (b) shows how the TOP might be used in a task that has different transition dynamics by having a different arrangement of walls and doorways. Figure 3.1 (c) shows how the TOP can be used in a task that has different state spaces. Other possibilities not illustrated include differences in the action set or differences in state variables, or some combination of these possibilities. Deriving TOPs from any number of these very different situations as well as the fact it is more globally applied in a target task and meant to approximate the final goal of a task, rather than local subgoals, are the key differences between TOPs and more typical SOPs. These differences between the problems subgoal options were meant to solve and the problems transfer learning algorithms were meant to solve are possibly why with the exception of work by Soni and Singh (2006), options have not been more greatly explored as a means to solve transfer learning problems. As this work will show, however, casting the transfer problem as a option learning problem is a very effective approach in many scenarios that has a number of advantages over existing transfer work.

3.2 Option-based Policy Transfer

When an agent faces a new target task to solve, it should ideally reuse its knowledge from multiple previously solved source tasks. OPT accomplishes this by using policy transfer and by treating the policy transfer problem as an option learning problem. Given a set of source tasks to use for transfer in a target task, OPT solves two types of challenges to transform the transfer learning problem into an option learning problem: (1) how to implement option learning in a way that facilitates the transfer of knowledge and allows an agent to learn when and from which source tasks to transfer; and (2) how options for source tasks should be constructed.

3.2.1 Learning with TOPs

Option learning that facilitates transfer learning and allows an agent to learn when and from which source tasks to transfer is presented in OPT as follows. Given a set of TOPs, conventional temporal difference (TD) option learning paradigms, such as option-based Q-learning or SARSA, are used to learn both in which states to use each source task for transfer and in which states the agent should use its currently learned policy without transfer from a source task's policy. TD methods allow the agent to learn which tasks to use for transfer on a state-by-state basis, because if one source task has a more useful policy to follow from a given state than another source task, then this difference will be captured in the Q-value distribution of the target task's TOPs. Conversely, when the target policy becomes better than using any source task, the Q-value of a primitive option will be greater than the Q-value of any TOP. As a result, TD methods enable the agent to learn when to rely on the target policy.

To further facilitate transfer from source tasks to the target task, intra-option value learning updating for the primitive options is incorporated into the normal TD algorithm: while an agent is following the policy of a TOP, it also updates the Q-values of the primitive options (which represent the target task's MDP action set), thereby improving both the transfer policy and target policy. Intra-option value updating is also used to update the Q-values for states visited by the TOP, instead of just the state in which the TOP was initiated. Henceforth, I refer to this specific application of intra-value option learning as *intra-option history learning*. Intra-option history learning allows the agent to accelerate the learning of the transfer policy because it increases the frequency with which the Q-values for TOPs are updated. However, the standard one-step intra-option value learning update (discussed in Section 2.4.2) is not used for intra-option history learning. In typical one-step intra-option value learning, an option's Q-value is updated with respect to the U-

value for the next state visited and the immediate reward received. In contrast, typical TD methods update the Q-value of an option for the state in which it is initiated with respect to the Q-values of the terminal state and the *total* discounted reward received from the entire execution of the option, rather than the immediate reward. I refer to the typical TD updates that use the total discounted return achieved during execution as the *complete-step* value update. Preliminary experiments indicated that combining complete-step updates for the states in which a TOP is initially executed with the one-step U-value updates for the states visited during the execution of the TOP resulted in slower learning than not including the one-step updating for states visited during a TOP's execution. To remedy this conflict, OPT implements a complete-step update for all states visited during execution of a TOP. A complete-step value update for each visited state is performed by first recording the reward received at each step of the execution of a TOP. Once execution of the TOP is complete, the Q-value for each state visited by the TOP is updated with respect to Q-values of the terminal state and the total discounted reward from that state to the terminal state, as if the TOP had been initiated in each state it visited and used standard TD methods to update the Q-values. Formally, if TOP o is initiated at time t in state s_t and executes for k steps, then in an off-policy paradigm, each of the $i = 1$ to $i = k - 1$ states visited after state s_t in which the TOP o continued to execute will have its Q-value for o updated according to:

$$Q(s_{t+i}, o) = Q(s_{t+i}, o) + \alpha \left[\sum_{j=i}^{k-1} \gamma^{j-i} r_{t+j} + \gamma^k \max_{o' \in O_{s_{t+k}}} Q(s_{t+k}, o') - Q(s_{t+i}, o) \right], \quad (3.1)$$

where r_t is the reward received for the action taken in state s_t and α and γ are the usual Q-learning parameters. This equation would be revised as expected for SARSA, where the discounted Q-value is the Q-value of the action actually taken in the state in which the option terminated, rather than the max Q-value of the state in which the option terminated.

It could also be useful to start transferring from a source task in some state, but then

stop transfer before the TOP would naturally terminate, using some other source task for transfer or the target policy. This can be accomplished by using option-interruption, which allows the agent to terminate an option prematurely when the Q-value of the current executing option in the current state is less than the state-value for that state. However, in order for option interruption to work effectively, the Q-values of the task must be well approximated. To determine when sufficient approximation is achieved, OPT uses a threshold parameter (ι): when the Q-values for a state have been updated in more episodes than the the threshold, option interruption is allowed. I call this measure of updates the *episodic update frequency* (EUF). More specifically, an EUF is kept for each state-option pair. When the Q-value for the state-option pair is updated for the first time in an episode, its state-option EUF is increased by one. The EUF for the state is then calculated as the sum of the EUFs for all options applicable in the state. As a result, the EUF for a state can increase by at most $|\mathcal{O}|$ in a single episode. The motivation behind using an update frequency measure is that the more episodes in which a Q-value is updated, the more likely it is to have a better Q-value estimate. Further, since the most important rewards in goal-oriented tasks are backed up from goal terminal states, using the *episodic* frequency of updates, rather than the total number of updates, is a better indicator of how many useful updates there were.

3.2.2 Constructing TOPs from Source Tasks

Constructing a TOP from a source task requires mapping the policy from the source task to the current task, creating a meaningful initiation set for the TOP, and creating termination conditions so that the agent knows when the source policy is no longer applicable and does not indefinitely execute the TOP. To satisfy these requirements, OPT assumes that for each source task, a mapping from states of the target task to states of the source task is provided as well as a mapping from actions of the source task to actions of the target task. It may also be possible that multiple state and action mappings between a single source

task and the target task exist. In this case, OPT can create a separate TOP for each mapping pair, thereby allowing the agent to learn which mappings are best in each state. (In Chapter 4, I provide a way to supply sets of mappings automatically for object-oriented MDPs.) Similar task mapping ideas have been used in previous RL transfer learning work (Taylor & Stone 2005; Taylor, Stone, & Liu 2007; Taylor, Whiteson, & Stone 2007; Fernández, García, & Veloso 2010), but were used for different methods of transfer learning. A state mapping between two tasks (t_1, t_2) is defined as a function $\phi^s_{t_1,t_2} : \mathcal{S}_{t_1} \rightarrow \mathcal{S}_{t_2}$, where $\mathcal{S}_t$ is the state space for task t. An action mapping between two tasks is defined as a function $\phi^a_{t_1,t_2} : \mathcal{A}_{t_1} \rightarrow \mathcal{A}_{t_2}$, where $\mathcal{A}_t$ is the action set for task t. Using these mappings, the policy for a TOP can be defined by mapping the current target task state to the TOP's source task state, finding the action in the source task with the highest Q-value, and mapping that resulting action to the corresponding target task action. Formally, for a TOP o representing source task t_2 that is invoked in target task t_1, the policy $(o.\pi)$ for all states $s \in \mathcal{S}_{t_1}$ is defined as:

$$o.\pi(s) = \phi^a_{t_2,t_1}\left(\arg\max_{a \in \mathcal{A}_{t_2}} Q(\phi^s_{t_1,t_2}(s), a)\right). \tag{3.2}$$

The initiation states for a TOP are defined as the states for which the corresponding state in the source task has an EUF greater than a threshold parameter (v):

$$o.\mathcal{I} = \left\{s \in \mathcal{S}_{t_1} : EUF(\phi^s_{t_1,t_2}(s)) \geq v\right\}. \tag{3.3}$$

The decision to use an EUF threshold is justified on the assumption that if the agent has not frequently visited a state, it has not adequately learned the policy for the state, in which case transfer would not be beneficial.

The termination conditions for a TOP are determined by three factors: (1) whether the

TOP's corresponding source task state is a termination state in the source task, (2) whether the corresponding source task state has an EUF less than v (that is, the state is not an initiation state), and (3) a minimum probability of termination (τ). If either of the first two cases are are true, then the TOP will terminate with probability 1.0. Otherwise, the TOP will terminate with probability τ, which is a parameter of OPT. Formally:

$$o.\beta(s) = \begin{cases} 1.0 & \text{if } T_{t_2}(\phi^s_{t_1,t_2}(s)), \\ 1.0 & \text{if } EUF(\phi^s_{t_1,t_2}(s)) < v \text{ and} \\ \tau & \text{otherwise,} \end{cases} \tag{3.4}$$

where $T_t(s)$ is a Boolean function that returns true when s is a terminal state of task t and false otherwise. Note that in a tabular MDP learning approach, setting v to a positive value alway entails the TOP's termination in the TOP's corresponding source task termination states. That is, for a TOP representing task t_2 in a tabular learning environment, if $T_{t_2}(s)$ is true, then $EUF(s) = 0$, for all $s \in \mathcal{S}_{t_2}$. This entailment will always be true in a tabular environment because an agent can never update its Q-values for termination states (since it stops acting when it reaches them, preventing the EUF values for termination states from ever increasing). Therefore, in a tabular learning environment, the termination conditions of the source task never have to be known because the EUF condition will always entail the termination state condition if $v > 0$. However, in a value function approximation (VFA) learning approach, this entailment is not guaranteed because EUFs would have to be estimated and generalized from experience in other (possibly non-terminal) states such that termination states may have positive EUF estimates that are greater than v. Therefore, condition (1) is useful in VFA learning paradigms or if v is set to zero. However, if the termination states of tasks cannot be provided to the agent, then the TOP will only terminate under conditions (2) or (3).

It may seem unnecessary to include a τ parameter and only let a TOP terminate in conditions (1) or (2). However, there are three reasons to set τ to a positive value. (1) A positive value helps to prevent overreliance on a TOP (executing the TOP for long periods of time), which would be detrimental to learning, because a TOP's policy is not expected to be useful in all states and always following a TOP to the source task's termination states would result in the agent being unable to separate the beneficial parts of the policy from the non-beneficial parts. (2) If termination states of the source task are not provided to the agent and the agent is using a VFA learning approach, a TOP could execute indefinitely without a positive τ. (3) If the transition dynamics between two tasks differ in such a way that it is unlikely for a TOP to reach its natural termination conditions, this could also result in a TOP executing indefinitely without a positive τ.

3.2.3 Choosing OPT Parameter Values

OPT requires the user to set three parameters: υ, ι, and τ. While different choices will result in varying degrees of transfer performance, in general, effective parameter choices can be made without much effort. For instance, none of the parameter choices in this chapter or subsequent ones required exhaustive optimization of the parameters. In particular, effective choices can typically be made by analyzing agent performance in source tasks which were learned without transfer. For instance, the υ parameter represents the minimum EUF required of a source task state to be an initiation state of the task's TOP. A heuristic to choose a value for υ is to look at the mean (m) and standard deviation (σ) of the EUF of all states visited in the last n episodes of learning in the source task, and set $\upsilon = m - 2\sigma$. This general approach was often used in this work, except the mean and standard deviation were not even formally measured and instead guessed from visually scanning the average EUF of states visited in each episode. The informal nature of choosing values for υ indicates that good performance can be provided without exhaustive optimization or background

knowledge. A value for ι can similarly be chosen by setting $\iota = cv$, where c is some small constant greater than 1. The τ parameter represents the minimum probability that a TOP will terminate in every state. One of the primary reasons for setting $\tau > 0$ is to prevent an overreliance on the TOP policy. That is, if the agent follows a TOP for too many steps before terminating, it prevents the agent from separating the beneficial parts of a source task's policy from detrimental parts of the source task policy. As is such, a heuristic for choosing a value for τ is to decide what fraction of source task policy the agent should follow on average and set τ to a value that achieves that. Specifically, if a designer would like the agent to on average follow a fraction p of a source task's policy and n is the average number of steps taken per episode in the source task (which again can be measured during learning of the source task), then by the geometric distribution, τ can be set to $\tau = \frac{1}{pn}$. Choosing τ as a function of the average number of steps taken in a source task episode and the fraction of the policy for which the designer would like the TOP to execute allows choices of τ to be more domain invariant. As with v and ι, however, no formal calculations were actually made when choosing a value for τ in this work. For results in this chapter, for instance, τ is always set to 0.1 and results may only be especially sensitive to choices of τ when the number of steps to complete a task are very small.

3.3 Experimental Methodology

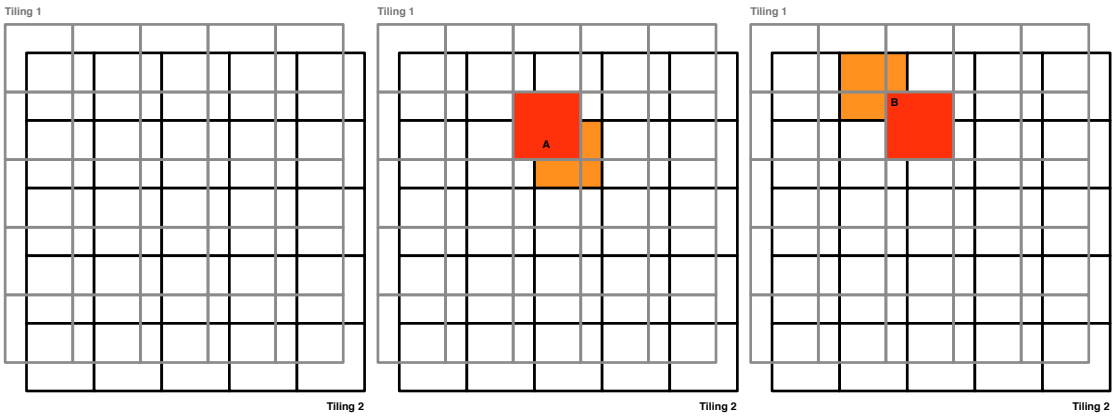
The performance for OPT is shown in two domains under a number of different transfer circumstances: (1) a discrete space office navigation domain and (2) a continuous lunar lander domain. In the office domain, tasks differ in terms of the goal location the agent is trying to reach. In the lunar lander domain, the tasks differ in terms of the action set available to the agent and the goal landing location. Because the lunar lander domain is continuous, it also requires function approximation, which is implemented using a *cerebel-*

lar model articulator controller (CMAC) (Albus 1971). These domains are described in more detail in Section 3.4.

Performance on these domains is analyzed using either the cumulative reward received or cumulative steps taken by the agent after each episode, averaged over multiple trials. As a baseline, OPT is compared to learning the task without any transfer learning. Additionally, OPT is compared to two existing policy transfer algorithms: a similar options-based method by Soni and Singh (2006) (henceforth referred to as SSO), and *Policy Reuse in Q-learning* (PRQL) (Fernández & Veloso 2005; 2006; Fernández, García, & Veloso 2010), which learns a target task with transfer learning from multiple source tasks. For comparison on these domains and to provide comparative learning settings, I implemented SSO and PRQL myself; that is, results for neither SSO nor PRQL use existing code from the original authors.

3.3.1 CMAC Value Function Approximation

A common approach to implementing value function approximation is to use a CMAC to generate the state features and then use the linear gradient descent TD learning methods discussed in Section 2.3 on those features. In particular, this approach to VFA was used for the original work on the mountain car domain (Singh & Sutton 1996). CMACs generate a set of binary state features by creating one or more tilings over the continuous state variables of the domain. A tiling is a multivariable discretization of the state variables into a set of tiles that are typically grid-like, with each tile covering a rectangular section of the state space (though other tiling shapes can be used as well). Each tile represents a binary state feature; for any set of state variable values, only one tile of a tiling (which is considered an "active" tile) has a value of one: the tile that contains the specific state variable values. The rest of the tiles that do not contain the variable values have a value of zero (and are considered "inactive"). If there is only one tiling, a CMAC effectively is



(a) Two Tilings of a 2D Space (b) Activation for State 'A' (c) Activation for State 'B'

FIG. 3.3. Generalization and discrimination of CMAC state features: (a) A two-variable state space with two offset grid tilings of the space; (b) a state with variable values located at 'A' activates a tile in tiling 1 and tiling 2; (c) a different state located at 'B' activates the same tile as state 'A' in tiling 1, but a different tile in tiling 2.

just a discretization of the state space and learning is equivalent to tabular learning over the discretized space, which might be overly general since all states within a tile are treated the same. However, when two or more partially overlapping tilings are used, multiple tiles will be active and more discrimination is provided by the fact that tiles of different tilings will activate together under different circumstances. For instance, consider a domain with two state variables. Two partially overlapping grid-like tilings of the state variables can be provided as shown in Figure 3.3. For a state located at position 'A' shown in Figure 3.3 (b), the tile in the second row from the top and third column from the right for the first tiling and the second tiling will be active. Another state, such as state 'B' shown in Figure 3.3 (c), will also activate the same tile in the first tiling as state 'A'; however, it will activate a different tile of the second tiling than state 'A'. This difference in activation pairing allows CMAC state features to generalize the state space by having tiles cover intervals of values as well as discriminate between states within those intervals by having different tiles co-activate

depending on the specific state variable values.

When learning with CMAC VFA, a learning rate of $\frac{1}{n}$, where n is the number of tilings, will result in a one-step update to the new Q-value (Sutton & Barto 1998). Tabular learning environments, on the other hand, achieve one step learning when $\alpha = 1$. Therefore, learning rates with CMAC VFA should typically be smaller, since more gradual changes are usually beneficial. A good heuristic for choosing a learning rate for CMAC VFA would be to decide what the tabular learning rate would be and multiply it by $\frac{1}{n}$, where n is again the number of tilings used.

Since OPT requires the agent to maintain the EUF of states, CMACs must be modified to *estimate* the EUF of states since the agent cannot store statistics for every unique state in the MDP. To do so, CMACs are modified to also keep an episodic update frequency for each tile-option pair. The EUF for a state-option pair is estimated as the sum of the EUFs for each of the tile-options pairs activated by the state. The EUF of the state is estimated as the sum of the estimated state-option EUFs over all options. When the Q-value weight for a tile-option pair is updated for the first time in an episode, the episodic update frequency of the tile-option pair is increased by $\frac{1}{n}$, where n is the number of tilings. Increasing the tile-option EUF by a fraction proportional to the number of tilings has the effect that if the same state was queried again, it would produce an EUF with a value exactly one greater than the value it was before the update, assuming that the previous update was the first update of the episode for all the tiles activated by the state.

3.3.2 Previous Policy Transfer Work with Options

Existing work by Soni and Singh (2006) (SSO) is conceptually similar to OPT in that it also uses options as a means to provide policy transfer. SSO builds on the concepts of SMDP homomorphisms (Ravindran & Barto 2003). The earlier work on SMDP homomorphisms was meant to make more general subgoal options that enabled the same sub goal

option to be applied in different state spaces of the MDP by exploiting homomorphisms between transition dynamics and the reward function of the different spaces. This is typically too strong a requirement for transfer learning scenarios in which source and target tasks can differ in a number of ways. SSO, relaxes these constraints for use in transfer learning by only requiring that there is a homomorphism between the transition dynamics of the source and target task (and *not* the reward function). Under this relaxation, transfer from a source task may not be sufficient to solve a target task, but could provide useful behavior for solving the task. Like OPT and other former work, SSO requires that a state and action space mapping between a target and source task exist. Specifically, SSO was designed for factored domains, in which the state space mapping from the target to the source is provided as a mapping between state variables in the target to state variables in the source. When multiple state variable mappings exist, SSO creates a TOP for each mapping. Unlike OPT, the initiation set for a TOP is any state in the target task that maps to a valid state of the source task. (Note that since SSO maps state *variables* between tasks rather than explicitly mapping states between tasks, it would be possible for a target task state's variables to map a set of variable values that define a non-permissible state in the source task and this TOP initiation set requirement prevents the TOP from being used in such situations.) Termination conditions for the TOPs are any states that are no initiation states or states that are termination states of the source task. If there is only one mapping from source actions to target actions, then the policy of a TOP in SSO is the same as the policy defined by OPT (see Equation 3.2). However, if multiple action mappings exist, SSO defines the policy of the TOP to randomly select from each of the possible mapped actions with uniform probability. This differs from OPT, which creates a separate TOP for each action mapping.

The SSO definition of TOPs risks overreliance on TOPs, since the initiation states and termination conditions are so unrestrictive. However, the learning algorithm that SSO

uses not only prevents this from happening but can actually result in an underreliance on the TOPs. Specifically, SSO uses a version of the one-step intra-option value learning algorithm shown in Algorithm 5, called *Intra-option Q-learning* (IOQ). In IOQ, the policy used to select actions is determined from an ϵ-greedy policy of all *options* (TOPs and primitive options in this case, since this is designed for transfer learning). For the given option selected, IOQ then follows the option for one step before updating the Q-values for all permissible options in accordance with the intra-option update rule and then selects a new option. As a result, selecting an option does not entail that it is executed to termination and instead a new option may be selected before the option would naturally terminate. Soni and Singh also make the usual adjustments to the Q-value update rules to enable the use of eligibility traces and function approximation.

While IOQ does prevent an overreliance on TOPs by making a new option decision after each step, it also results in a severe underreliance. The primary benefit of policy transfer is that it biases the space explored by the agent to more useful states. If an option is only taken once and then another action or option is considered, a TOP has little impact on the exploration unless the agent continues to decide to execute that TOP in many successive states. However, because the Q-values of all options will be uninformative in the initial learning period (when TOPs are most useful), the agent will have no reason to continue to select the same TOP and so IOQ will not result in much bias in the exploration. In contrast, OPT has a more balanced reliance on TOPs by using a learning algorithm that more greatly relies on TOPs, while constructing TOPs with more restrictive initiation states and termination conditions.

It should be noted that the domains and transfer scenarios in which SSO is tested here are not the same scenarios under which SSO was tested or designed to be used. Specifically, SSO was only tested for use in transfer from a single factored source task that had multiple possible mappings between state variables and action sets, had homomorphic source and

target task transition dynamics, and had fairly similar reward functions between source and target task. OPT's ability to outperform SSO in the more challenging transfer scenarios presented here demonstrates a significant strength of OPT.

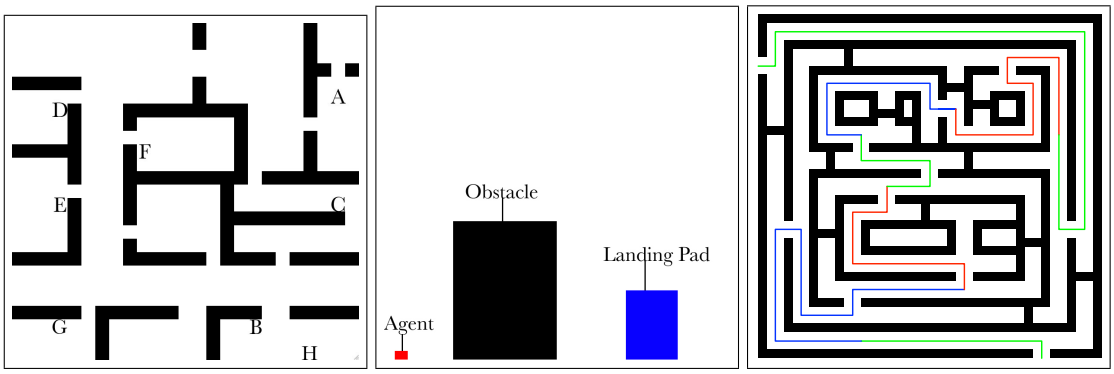
3.3.3 Policy Reuse in Q-learning

In addition to comparing performance to learning without transfer, I compare OPTs results to another policy transfer algorithm called *Policy Reuse in Q-learning* (PRQL) (Fernández & Veloso 2005; 2006; Fernández, García, & Veloso 2010). PRQL is an approach to policy transfer in which an agent learns which source tasks are the best to use for transfer and probabilistically chooses—on an episode-by-episode basis—whether to use a source task for transfer (and if so, which one) or whether to use only the currently learned target policy. To choose whether to use transfer and which source task to use, PRQL learns the expected discounted return of using each source task for transfer, as well as the expected discounted return of using only the target policy, and chooses a source task for transfer (or not to use one) probabilistically according a Boltzmann distribution of the expected discounted returns. For source task Π_j, the probability that PRQL selects the task for transfer for an episode is $p(\Pi_j) = \frac{e^{\tau W_j}}{\sum_{p=0}^{n} e^{\tau W_p}}$, where n is the number of source tasks, W_i is the expected discounted return of using a task for transfer, and τ is a cooling parameter increased after each episode so that the selection becomes more greedy with time. Note that the target task is included in this distribution; as a result, as the agent learns the optimal target policy, the agent should be selecting to use the target policy without transfer more often.

If PRQL chooses to use a source task for transfer for an episode, the agent follows a transfer policy called π-reuse, which probabilistically chooses at each state of the episode whether to follow the chosen source task's policy or whether to follow the currently learned target task policy. Like OPT, π-reuse determines what the source task's policy is in the target task by using a mapping of states and actions between the target task and source

task. Specifically, at each state, π-reuse chooses to use a greedy policy of the source task with probability ψ or an ϵ-greedy policy of the target task with probability $1 - \psi$, where ψ is initialized to a base value (typically one) at the start of the episode and is decayed after each action with the formula $\psi \leftarrow \psi\upsilon$, where υ is a parameter between zero and one. This decaying has the effect that the agent uses the source task's policy frequently in the beginning of the episode and gradually transitions to using the target task policy more frequently as the agent acts in the episode. If the agent probabilistically chooses to follow the ϵ-greedy target task policy (with probability $1 - \psi$) for a given state of the episode, then ϵ is set to ψ, which has the effect of making the agent more greedy as the episode progresses.

There are a few significant differences between OPT and PRQL. The first major difference is that although PRQL learns which source tasks are best overall and performs policy transfer from multiple source tasks, it only uses at most one source task for transfer in each episode. Additionally, while using a task for transfer, π-reuse does not learn in which states it should use the source task, but randomly chooses whether to follow the source task policy or not and decreases the likelihood of following a source task's policy with time. In contrast, OPT can alternate between multiple source tasks on a state-by-state basis and learns which source tasks are best for each state. The approach of decaying the use of a source task within an episode as time goes on seems valid only under the assumption that transfer is most beneficial in the earlier steps of an episode. However, if a task would not benefit from transfer in the earlier steps of an episode, but would in the latter steps, this approach would be detrimental because the agent would end up using transfer in the least applicable states and ignoring it in the most applicable states. Again, OPT does not suffer from this problem because it learns in which states transfer from a task is most useful. Another significant difference is that π-reuse can select the source policy in any target state, regardless of whether the policy for the corresponding source task state was



(a) Office Navigation (b) Lunar Lander (c) Maze Navigation

FIG. 3.4. A representation of each of the domains used: (a) the office navigation domain and the goal locations of the different tasks (A-F); (b) the lunar lander domain in which the agent starts on the left, must take off and avoid an obstacle, and must land on the top of the landing pad; and (c) the maze navigation domain's target task maze with the solution path for the maze (green portions of the path indicate when both source tasks have the same optimal policy, red when only the first has the same optimal policy, and blue when only the second source task has the same optimal policy).

learned. If there are regions of the state space in the source task that are underexplored, the resulting policy in those regions may be random and transfer may be detrimental, but will occur nevertheless in π-reuse. In OPT, TOPs have initiation states that ensure that when the policy of a source task is followed for the state, the corresponding source task state has had a minimum number of learning updates to provide a non-random policy.

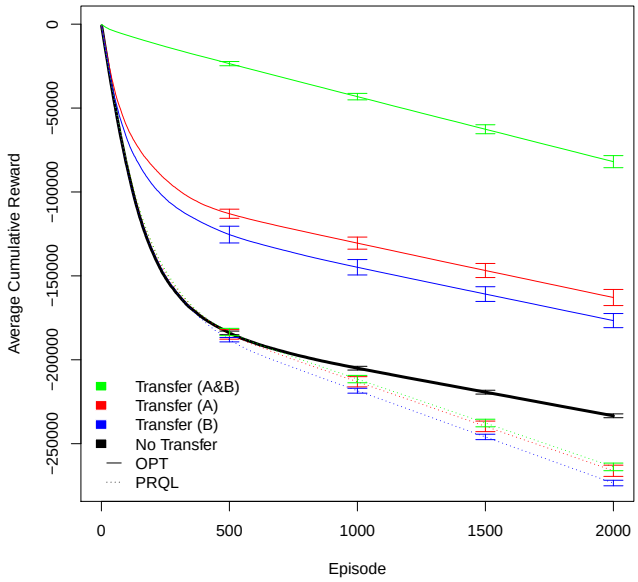
3.4 Results

This section describes the details of the three domains tested (office navigation, lunar lander, and maze navigation), the domain tasks that were used, the learning algorithms that were used for Q-value learning, and empirical results for each of the domains.

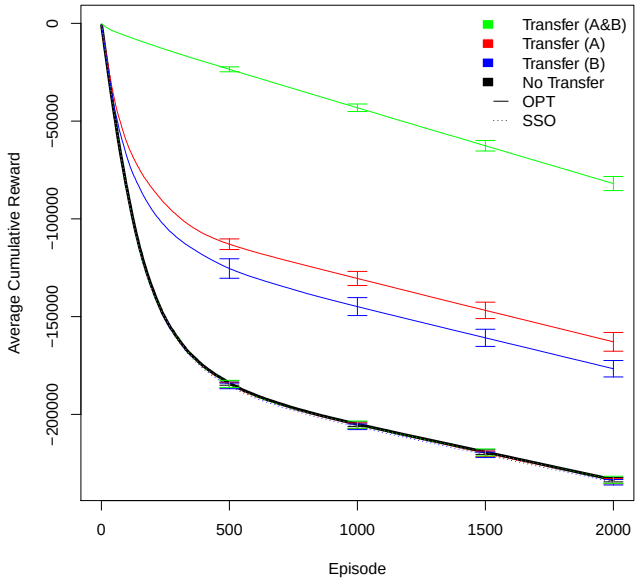
3.4.1 Office Navigation

The office navigation domain is a 25x25 grid world with walls creating separate rooms (similar to one of the domains in which PRQL was analyzed (Fernández & Veloso 2005)). The agent has actions for moving to the north, south, east, or west, except when doing so would move the agent into a cell occupied by a wall, in which case the action does not change the state. Each task corresponds to a unique location that the agent needs to reach. Performance in this domain is assessed with a reward function that returns 1 when the agent reaches the goal location and -1 everywhere else. For a given goal location, the initial state of an agent in each episode is a random location that is at least a manhattan distance of 20 away from the goal. If an episode lasts longer than 1000 steps, it is automatically terminated.

Figure 3.4 (a) shows the layout of the office navigation domain, with eight different goal locations (A-H), each defining a different task. Tasks A, B, D, and E are source tasks in the experiments; C, F, and G are target tasks. Note that that state space and action space between tasks is identical in this domain and therefore no complex state and action space mappings are required. Learning performance is first compared on target task C under four learning conditions: no transfer; transfer from task A; transfer from task B; and transfer from tasks A and B (multi-source policy transfer). In this first scenario, the agent is likely to encounter location C when moving between A and B, which gives multi-source policy transfer a distinct advantage. To test a more difficult scenario, in which the gains of multi-source transfer are less obvious, a similar set of experiments with transfer from source tasks D and/or E to target task F is run. Finally, to test how robust OPT is to malicious transfer, an additional experiment is run on target task G with transfer from source A. This transfer scenario is considered malicious transfer, because following the policy of A will almost always take the agent away from G, since A and G are on opposite corners of the domain

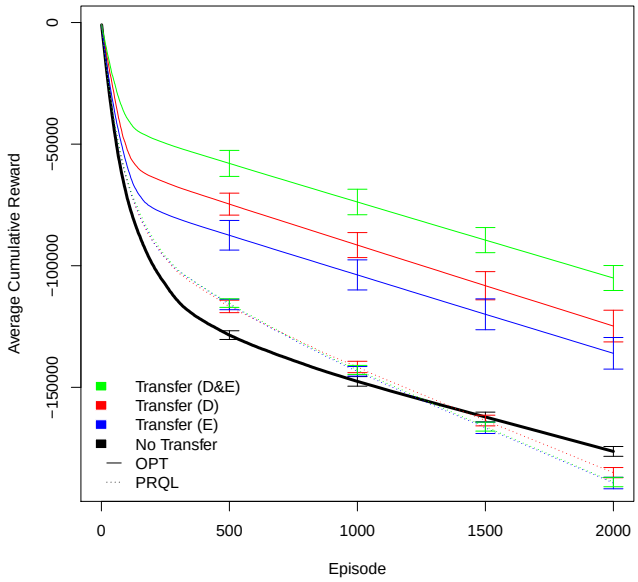


(a) OPT compared to PRQL

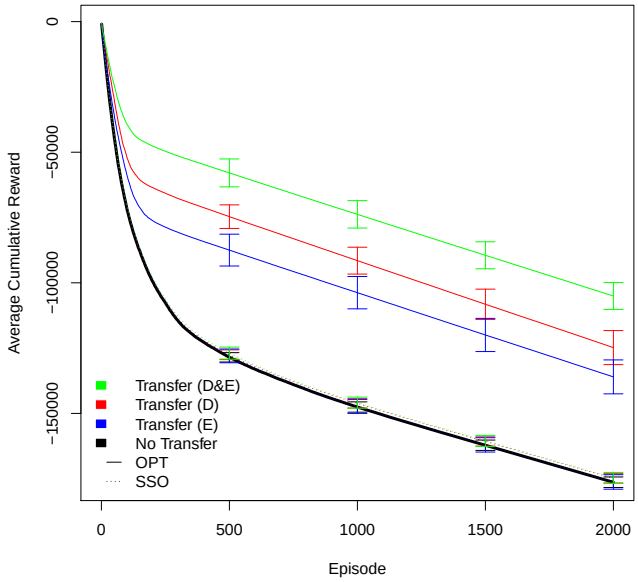


(b) OPT compared to SSO

FIG. 3.5. Average cumulative reward over ten trials for the office domain with transfer to target task C from sources A and B. OPT has significant performance gains and performs best with multi-source transfer. Figure (a) compares OPT to PRQL where PRQL provides no transfer benefit and ultimately converges to a worse policy. Figure (b) compares OPT to SSO, which provides no transfer benefit.



(a) OPT compared to PRQL



(b) OPT compared to SSO

FIG. 3.6. Average cumulative reward over ten trials for the office domain with transfer to target task F from sources D and E. OPT has significant performance gains and performs best with multi-source transfer. Figure (a) compares OPT to PRQL: PRQL provides some acceleration to learning, ultimately converging to a worse policy. Figure (b) compares OPT to SSO, which provides no transfer benefit.

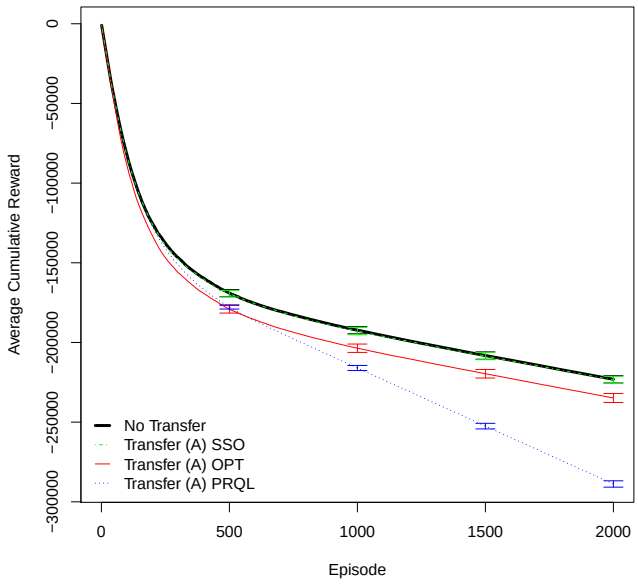


FIG. 3.7. Average cumulative reward over ten trials for the office domain with malicious transfer to target task G from source A. OPT suffers slightly slower learning speed from the malicious transfer. PRQL also suffers from a slightly slower learning speed and ultimately converges to a worse policy. SSO suffers no cost from the malicious transfer.

with a manhattan distance of 37 between them.

Since this domain is a grid world, tabular Q-learning is used to learn the Q-values. The Q-learning parameters are set as follows: $\alpha = 0.1$, $\gamma = 0.99$. The agent follows an ϵ-greedy policy with ϵ initialized to 0.1 for all states and is decayed to a minimum value of 0.01 based on the EUF of the state according to $\epsilon(s) = \max(0.01, 0.1 \cdot 0.95^{EUF(s)})$. The Q-values are initialized to the semi-pessimistic value of -90 (note that the most pessimistic value would be -100, which would be the true Q-values for a policy that always avoided the goal state and was evaluated with a discount factor of 0.99). This allows the agent both to quickly learn to favor actions that transition to the goal state and to explore the state space when it is has not encountered the goal from a specific state. The OPT parameters were set as follows: $\upsilon = 200$, $\iota = 500$, and $\tau = 0.1$. All tasks are learned for 2000 episodes. PRQL, which is used for comparison, has its parameters set as follows: $\upsilon = 0.95$, ψ is initialized to 1.0 at the start of each episode, τ is initialized to 0, and τ is incremented by 0.05 after each episode. These values were chosen because they are the same values used in the previous PRQL work that was evaluated on a similar domain (Fernández & Veloso 2005).

The results showing the average cumulative reward from ten separate trials for targets C, F, and G are shown in Figures 3.5, 3.6, and 3.7, respectively; the confidence intervals shown are at the 95% threshold. Note that cumulative reward is negative, so a flatter slope corresponds to better performance. For targets C and F, all OPT transfer variants learn much faster than without transfer. In the first experiment (target task C), we find a significant advantage to using multi-source policy transfer with OPT with extremely good initial performance. All variants of OPT perform much better than all variants of PRQL and all variants of SSO.

In the second experiment (target task F, Figure 3.6), OPT's multi-source transfer is not as beneficial as in the first experiment, since this was designed as a less useful scenario for

multi-source transfer. Nevertheless, OPT's multi-source transfer still outperforms either of the single-source transfer experiments with statistical significance and performs significantly better than learning without transfer and better than all variants of PRQL and SSO. However, it should be noted that in this specific task configuration, transfer learning performance for OPT is better if a constant ϵ value for the learning policy is used, Moreover, when a constant ϵ is used, multi-source transfer does not perform better than single-source transfer from D in a statistically significant way but does perform better than single-source transfer from E . I present results for this variation of the experiment and hypothesize why there is a difference in multi-source transfer benefit in Appendix A.1.2. Results for a constant ϵ are not provided for the other office navigation transfer configurations because they do not benefit in any way from a constant ϵ.

In the malicious transfer experiment (target G, Figure 3.7), OPT does learn more slowly than without transfer (and the difference is statistically significant); however, it only performs marginally worse, demonstrating that OPT is reasonably robust to malicious transfer in this domain. OPT also performs better than PRQL under this malicious transfer scenario. SSO does not appear to have any degradation in performance, but since SSO did not provide any transfer benefit in the other transfer scenarios, this is not surprising.

Of interest is how badly PRQL performs in all of these experiments, since PRQL was originally tested in a very similar domain (Fernández & Veloso 2005). For target C, PRQL not only performs significantly worse than all OPT variants, but also learns no faster than learning without transfer and results in a poorer policy being followed (due to PRQL always selecting a source task for transfer, when it should eventually just use the target policy without transfer). For target F, PRQL again performs significantly worse than all OPT variants. While PQRL does learn slightly faster than learning without transfer, PRQL does not benefit from using multiple source tasks and it ultimately does not result in following a policy that is as good as the policy that learning without transfer would

provide (again due to PRQL always selecting a source task for transfer). For the malicious experiment (target G), PRQL also performs badly; it learns just as slowly as OPT in this condition and ends up following a policy that is much worse.

The degree of PRQL's failures here may seem unexpected since PRQL is an established policy transfer algorithm. However, there are a few key differences in the tasks in this domain from the previous domains on which it was tested. One major difference is that in the original work of PRQL with a similar office domain, PRQL seemed to benefit from having source tasks from which to transfer in each room. This preference indicates that transfer in PRQL worked best when there was at least one source task that was much more similar to the the target task than the source and target tasks are in these experiments. In this context, OPT has a significant advantage, because it can benefit from transfer from less similar tasks. Another possible difference is that in the original PRQL work (Fernández & Veloso 2005), the reward function was slightly different: transitions to the goal returned a value of 1 and 0 everywhere else (rather than -1 everywhere else). Further, Q-values were initialized very pessimistically to 0. To further explore the the effect of these differences, Appendix A.1.3 has a set of additional experiments that explore using more similar source and target tasks for transfer and using a reward function that returns rewards of 1 and 0 and is initialized at different Q-values. For task similarity, the findings show that PRQL can be more beneficial than it appeared to be in the experiments presented in this chapter, but still not as beneficial as OPT. For the reward function and Q-value initialization, the findings show that the reward function is not a significant factor, but that the Q-value initialization can drastically affect performance in such a way that PRQL is more beneficial compared to learning without transfer with highly pessimistically initialized Q-values than it appears in this chapter's experiments. However, it was also found that using a highly pessimistic Q-value initialization results in slower and more random learning in the baseline learning without transfer, making the gains in this context somewhat artificial since the baseline

performs better if less pessimistic initialization values are used. When Q-values were initialized less pessimistically, results mimicked those found in this chapter where PRQL does not offer much, if any, benefit and OPT performs significantly better.

The motivating reason for OPT using SMDP Q-learning methods with more restrictive initiation and termination conditions in the TOP construction was to provide a balanced reliance on the TOPs. SSO, in contrast, suffers from an extreme underreliance, because it uses a form of one-step intra-option value learning (rather than SMDP methods) in which only one step of a TOP is taken before another TOP or action is considered. The problem with this approach is reflected in the results for this domain, where SSO produces no no benefit from transfer learning at all. To show that the restrictive initiation and termination conditions that OPT imposes on TOPs is necessary to prevent the SMDP Q-learning methods from resulting in an overreliance on TOPs, Appendix A.1.1 shows results in each of the domains when a TOP can be initiated anywhere and only terminates when it reaches a state that maps to a termination state in the source task. As expected, performance is very bad due to the overreliance, justifying the need for these additional restrictions on the initiation and termination conditions of TOPs.

3.4.2 Lunar Lander

The lunar lander domain is a continuous domain based on the classic Atari game from 1979. An example task of this domain is visualized in Figure 3.4 (b). In this domain, the agent controls a lunar lander that starts on the ground and the agent's goal is to pilot the lander to the surface of a landing pad while avoiding obstacles. To do so, the agent is provided actions for rotating clockwise or counterclockwise by 9° increments for a maximum of 45° from the vertical axis in either direction, idling (in which case the movement of the lander is subject to its current velocity and gravity), and exerting thrust from the bottom of the lander. The state representation is defined by five continuous variables: the x-y position

of the lander, the velocity vector of the lander ($\dot{x}, \dot{y}$), and the orientation of the lander, specified as the angle rotated from the vertical axis (r). The motion of the lander is governed by simplified physics where the position and velocity variables in the next time step are updated according to:

$$\begin{aligned} x_{t+1} &:= x_t + \dot{x}_t + \frac{1}{2}\cos(\theta_t)T_t \\ y_{t+1} &:= y_t + \dot{y}_t + \frac{1}{2}(\sin(\theta_t)T_t + g) \\ \dot{x}_{t+1} &:= \dot{x}_t + \cos(\theta_t)T_t \\ \dot{y}_{t+1} &:= \dot{y}_t + \cos(\theta_t)T_t + g \end{aligned} \quad ,$$

where $\theta_t = 180^\circ - r_t$, $g = -0.2$ is the gravity, and T_t is the thrust exerted at time t. Note that the thrust exerted is always zero except when the action selected at time t was a thrust action, in which case the amount of thrust is dictated by the action. The movement of the lander is confined to a 100x50 space and if the lander reaches an edge of this space, its velocity in that direction is set to zero. Collisions with obstacles or the side of the landing pad do not result in termination of the episode (to make learning easier); rather, a collision zeros out the velocity in that direction, similar to hitting the edge of the space. However, hitting the edge of the space, landing on the ground, colliding with an obstacle, or hitting the side of the landing pad all result in a reward of -100. Landing on the top surface of the landing pad results in a reward of 1000 and any other state transition results in a reward of -1. The episode terminates when the agent lands on the landing pad.

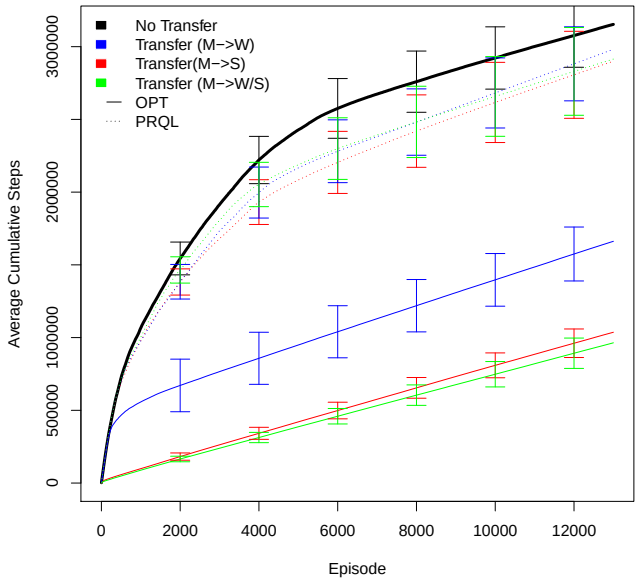
For this domain, two different transfer experiments were performed. In both experiments, the agent starts on the ground on the left side of a tall rectangular obstacle that it has to clear before it can land on a shorter landing pad to the right of the obstacle. In both experiments, the source and target tasks differ in the action set available to the agent. In

the source task, the agent has a single medium thrust action that provides an acceleration of 0.26. In the target task, the agent does not possess this medium thrust action, but instead possesses two other thrust actions: a weak thrust (exerting an acceleration of 0.2, which is proportional to gravity) and a strong thrust (exerting an acceleration of 0.32). In order to provide transfer from the source to target task, a mapping from the source action set to the target action set must be provided. Note that the medium thrust exerts an acceleration that is the midpoint of the weak and strong thrust action accelerations; therefore, the medium thrust is no more numerically similar to the weak thrust action than it is to the strong thrust action. As a result, there are two possible action set mappings that make sense: medium thrust $\rightarrow$ weak thrust and medium thrust $\rightarrow$ strong thrust. In both experiments, the performance of OPT when using either of these two action mappings is shown. However, in the same way OPT supports multi-source transfer, both TOPs can be provided to the agent (providing multi-mapping transfer), allowing the agent to learn when to use each mapping; therefore, the results when the agent is provided both TOPs are also shown. For the first transfer experiment, the source and target tasks are identical in all respects except the previously mentioned action set. Specifically, the agent always starts at position (5,0), the obstacle spans a horizontal space from position 20 to 50 with a height of 20, and the landing pad space a horizontal space from 80 to 95 with a height of 10. In the second experiment, the source task has the same world configuration as the previous experiment, except that in the target task, the landing pad is translated to the left to range from 70 to 85 (therefore only partially overlapping the space it spanned int the source task). This additional difference between the source and target tasks for the second experiment is meant to make the transfer even harder. Specifically, the medium $\rightarrow$ strong action mapping will be less useful because with a stronger thrust, the agent can be expected to end up overshooting when following the policy of the source task; this overshooting will be more problematic if the landing pad is translated to the left. To map between the state spaces of each task, the state space is

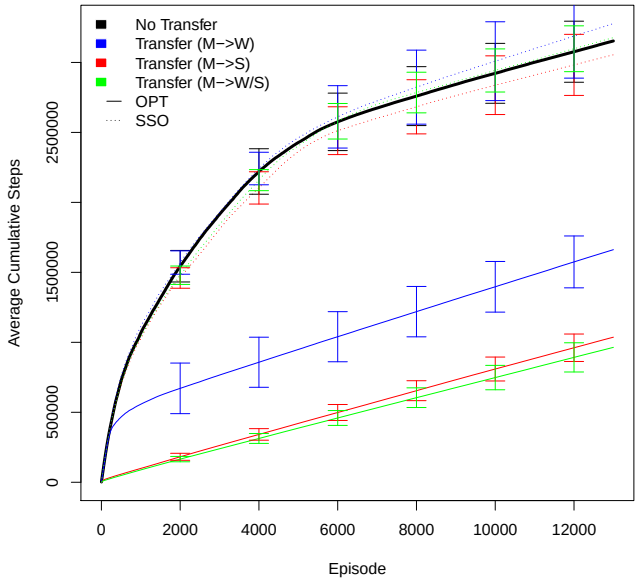
represented with only the agent position, velocity, and orientation variables (ignoring the obstacle and landing pad position information), thereby allowing a direct mapping of these variables from the target task to a source task. Because the landing pad and obstacle information is constant for all states of a task, this representation preserves the Markov property. Each task is learned for 13000 episodes and if an episode lasts longer than 2000 steps, it is prematurely terminated.

Because this domain is continuous, CMAC VFA with SARSA(0) is used. The widths of the CMAC tiles along each of the state variables ($x, y, \dot{x}, \dot{y}, r$) are 10, 5, 0.4, 0.4, and 9 respectively. The CMAC also uses five tilings that are randomly jittered. The SARSA parameters are as follows: $\alpha = 0.02$ (which with 5 tilings is similar to $\alpha = 0.1$ for a tabular learning environment) and $\gamma = 0.99$. The agent follows a constant ϵ-greedy policy with $\epsilon = 0.05$. The weights of the CMAC are optimistically initialized to 0.5, which with five tilings results in initial Q-value estimates of 2.5. The OPT parameters were set without any fine tuning to this domain, using the same values as those used in the office navigation domain: $\upsilon = 200$, $\iota = 500$, and $\tau = 0.1$. PRQL was used for comparison with the same action mappings and had its parameters set as follows: $\upsilon = 0.95$, ψ is initialized to 1.0 at the start of each episode, τ is initialized to 0, and τ is incremented by 0.05 after each episode. Note that these are also the same parameter values as those used in the office navigation, although these values have been used by PRQL in other domains with success (Fernández, García, & Veloso 2010), indicating that they should be fairly general choices. However, tests with PRQL set to different values ($\psi = 0.98$ and τ incremented by 0.001) were also performed, but the results were similar without statistically significant differences and therefore are not presented.

Results for the lunar lander domain experiments are shown in Figures 3.8 and 3.9. Because of the extreme values used in the reward function used for this domain (+1000 for the goal, -100 for collision, and -1 everywhere else), results are presented in terms of average

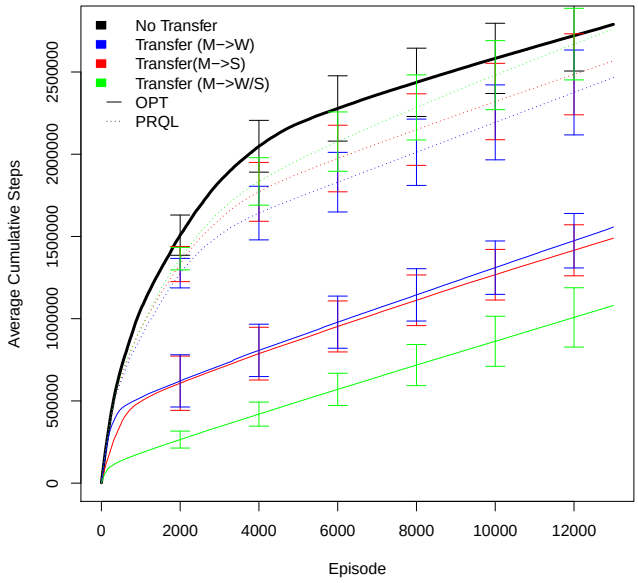


(a) OPT compared to PRQL

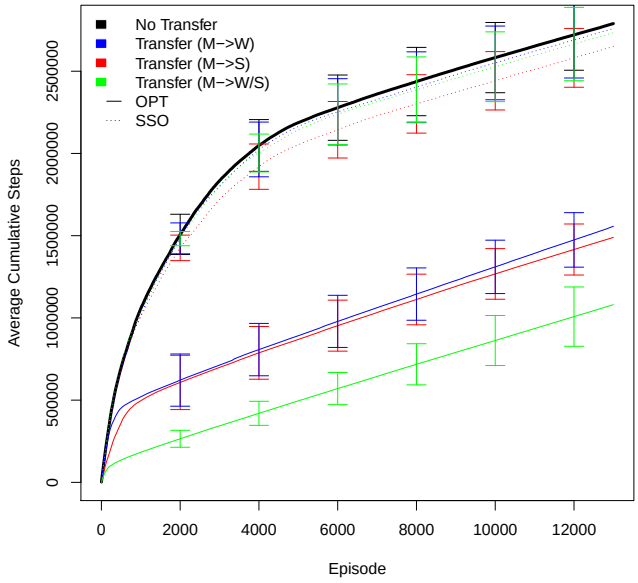


(b) OPT compared to SSO

FIG. 3.8. Average cumulative steps over ten trials for the lunar lander domain when the only difference between the source task and target task is the action set. Mapping the medium thrust to the strong thrust results in much better performance than mapping the medium thrust to the weak thrust; providing OPT both mappings performs just as well as mapping the medium thrust to strong thrust. Subfigure (a) compares OPT to PRQL: OPT performs much better and PRQL offers minimal, if any, benefit over the baseline. Subfigure (b) compares OPT to SSO, which does not appear to offer any benefit over the baseline.



(a) OPT compared to PRQL



(b) OPT compared to SSO

FIG. 3.9. Average cumulative steps over ten trials for the lunar lander domain when the source task and target task differ in action set and the landing pad is translated to the left in the target task. Both forms of single-mapping transfer provide substantial benefits over the baseline; multi-mapping transfer provides the greatest benefit. Subfigure (a) compares OPT to PRQL; OPT performs much better and PRQL offers minimal, if any, benefit over the baseline. Subfigure (b) compares OPT to SSO, which does not appear to offer any benefit over the baseline.

cumulative number of steps (over ten trials), so a lower and flatter curve indicates better performance. The error bars are shown for the 95% confidence threshold. Figure 3.8 shows results for the first experiment, in which the landing pad is in the identical position in both the source and target task and the tasks only differ in terms of the thrust action set. In this case, the mapping from the medium to strong thrust action is extremely beneficial, as little modification to the policy is needed to land on the landing pad. This transfer yields extremely good performance compared to the baseline learning. The mapping from medium to weak thrust also performs much better than the baseline learning, but is much less beneficial than the medium to strong mapping. One critical reason for this may be because the policy in the beginning of the episode is never very useful, since the weak thrust is not able to lift the lander off the ground (since it is proportional to gravity), whereas the mapping from medium to strong is useful in many regions of the state space. Because the mapping from medium to strong seems to be superior to or as good as the medium to weak mapping in all cases, there would not be much benefit to multi-mapping transfer in this scenario. This expectation is reflected in the results: multi-mapping transfer performs no better (in a statistically significant way) than the single-mapping medium to strong mapping. However, multi-mapping transfer also performs just as well as single-mapping transfer with the medium to strong thrust mapping, which has value from a designer's perspective because it means that if a designer does not know which mapping might be the best to use for a domain, then OPT can be provided multiple mappings and learn which to use without serious cost to performance.

The results for the second experiment, in which the landing pad in the target task is shifted to the left of its position in the source task (as well as having the differences in the action set), are shown in Figure 3.9. In this case, OPT single-mapping transfer for both mappings performs better than learning without transfer. However, in this experiment, the action mapping from medium to strong does not have nearly the same advantages be-

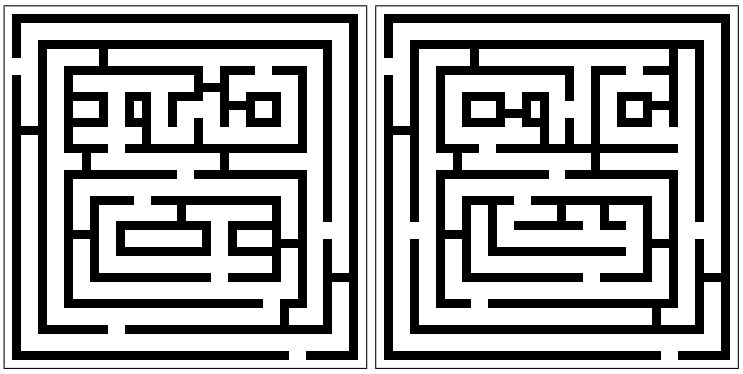
cause the shifted landing pad exacerbates the fact that this action mapping will result in a policy that overshoots what would be expected in the source task. As a result, the mapping from medium to strong thrust performs about as well as the mapping from medium to weak thrust. However, when both options are included in the multi-mapping transfer, OPT learns to combine them in useful ways; as a result, multi-mapping transfer has the best performance.

PRQL performance in both experiments is not especially beneficial or detrimental. PRQL performance does not appear to be worse than the baseline and appears as though it may increase the learning rate, but only the mapping from medium to weak in the second experiment results in a performance gain that is better than the baseline without transfer in a statistically significant way. SSO does not appear to offer improvement over the baseline learning in a statistically significant way in either experiment.

To test whether OPT's initiation and termination constraints were important to preventing an overreliance on TOPs on this domain, additional experiments were run where TOPs could be initiated in any state and only terminated in states that mapped to termination states in the source task. These results are presented in Appendix A.2. As with the office domain, in general this variant performed much worse than when OPTs initiation and termination constraints were used, justifying that these are necessary to prevent an overreliance on TOPs.

3.4.3 Maze Navigation

The maze navigation domain is a 40x40 grid world in which the agent can move north, south, east, or west. The goal of the agent is to enter the maze from a specified point and traverse the maze to the exit. Although similar to the office navigation domain, the maze navigation domain is designed to test a different kind of transfer: instead of different goal locations between tasks, the transition dynamics change between tasks. Specifically, for



(a) Maze Source Task A (b) Maze Source Task B

FIG. 3.10. The two source task mazes used in the maze navigation experiments.

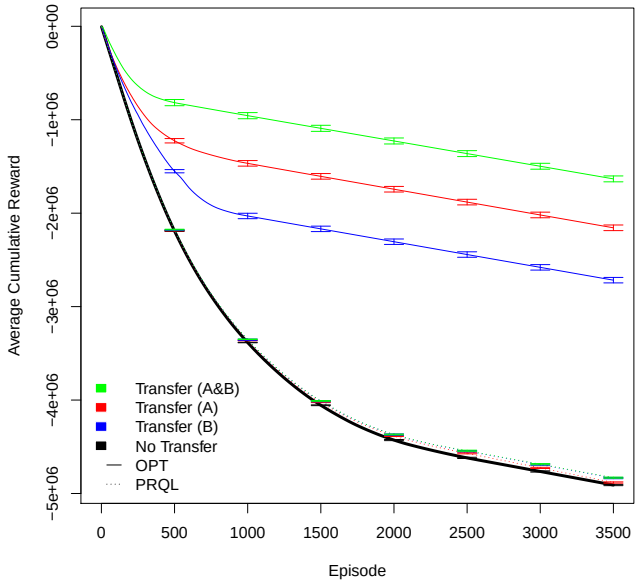
any given task, there is only one path from the entry point of the maze to the goal exit point. Different tasks have different maze designs requiring different paths to be followed. Experiments are performed with two source mazes (A and B), shown in Figure 3.10 (a) and (b), respectively, and one target maze, shown in Figure 3.4 (c), which also shows the solution path to the target maze and which parts of the path are shared with the source task maze solutions. Specifically, green portions of the solution path indicate that the solution is the same in both source tasks; red portions of the path indicate that the solution is only the same in source A; and blue portions of the task indicate that the solution is only the same in source B.

As with the office navigation domain, a reward function that returns 1 when the agent reaches the goal and -1 everywhere else is used. The optimal solution for each task takes about 260 steps; if an episode lasts for more than 5000 steps, it is prematurely terminated. Since this domain is a grid world, tabular Q-learning is used to learn the Q-values. The Q-learning parameters are set as follows: $\alpha = 0.1$, $\gamma = 0.99$. The agent follows an ϵ-greedy policy with ϵ initialized to 0.1, which is not decayed and constant for all episodes. Like the office navigation, the Q-values are initialized to the semi-pessimistic value of -90. The

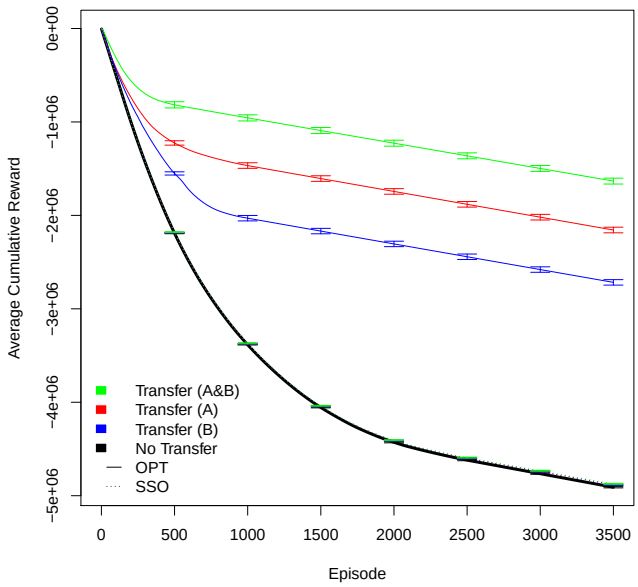
OPT parameters were set as follows: $\upsilon = 500$, $\iota = 2000$, and $\tau = 0.1$. Note that the υ and ι values are larger than they were in the office domain, since learning takes much longer in this domain (due to the much larger state space). All tasks are learned for 3500 episodes. PRQL, which is used for comparison, has its parameters set as follows: $\upsilon = 0.98$, ψ is initialized to 1.0 at the start of each episode, τ is initialized to 0, and τ is incremented by 0.01 after each episode. υ was set to a higher value than in the office domain since the optimal solution requires many more steps and a higher value of υ results in the agent using transfer for longer. Similarly, τ is incremented by a lower value than in the office domain, since it takes longer to learn a solution to this task and a smaller increment of τ will result in the agent exploring the various source tasks for more episodes.

Results showing the average cumulative reward over ten trials are shown in Figure 3.11. All transfer variants of OPT perform significantly better than the baseline learning without transfer; the multi-source transfer works the best with a statistically significant difference compared to the single-source variants. PRQL does perform better than the baseline in a statistically significant way, however, improvements are very minor. SSO also seems to provide a statistically significant benefit to learning over the baseline, but the improvement is very small. It should be noted that of all the domains tested in this chapter, this maze domain is the least representative of the kinds of transfer scenarios SSO was meant to solve, since the transition dynamics between the source and target tasks are not homomorphic.

Experiments were also run to test whether the initiation and termination conditions that OPT uses for TOPs are necessary to prevent an overreliance on TOPs in this domain. Specifically, these experiments removed the EUF constraints and had no probability of termination in every sate. However, the results are not presented, because results were so poor that in no situation did the agent ever reach the goal exit in all 3500 episodes. This should be expected, because if an agent initiated and followed a TOP that resulted in the



(a) OPT compared to PRQL



(b) OPT compared to SSO

FIG. 3.11. Average cumulative reward over ten trials for the maze domain. Multi-source OPT performs the best with statistical significance. Single-source OPT variants also perform very well. Subfigure (a) compares OPT to PRQL; PRQL provides a statistically significant transfer benefit over the baseline, however, the benefit is extremely small. Subfigure (b) compares OPT to SSO, which provides a statistically significant benefit over OPT, but the gain is even smaller than it is from PRQL.

agent walking into a wall that was not present in the source task, then the agent would repeatedly walk into the wall and the TOP would never terminate.

3.5 Conclusions

In this chapter, Option-based Policy Transfer (OPT), which takes transfer learning problems and turns them into an option learning problems, was introduced. OPT allows an agent to transfer from multiple source tasks on a state-by-state basis so that the agent learns in which states it would be useful to transfer an option and in which states to use the currently learned target policy.

Empirical results for OPT were shown on three different domains: office navigation, lunar lander, and maze navigation, each representing a different kind of transfer and learning context. Results on the office navigation domain showed that learning performance is significantly improved with transfer from OPT when reward functions differ, even with fairly dissimilar source tasks, and can greatly benefit from multi-source transfer. These results also showed that OPT is resistant to malicious transfer (when the source policy is detrimental to the target task) and yields only a slight decrease in learning performance.

The lunar lander domain results showed that OPT is also effective with on-policy learning algorithms like SARSA and in continuous domains that utilize value function approximation. Additionally, the lunar lander domain results demonstrated that OPT is effective even when transfer is across different action spaces; due to the multi-task nature of OPT, it can even use multiple action mappings in its transfer learning. If one of the multiple action mappings were universally superior to the other, then OPT would learn to focus on that one mapping without a noticeable degradation in performance compared to only having the superior one from the start. If each mapping had strengths in different situations, OPT would learn to combine them for better learning performance than if it was

only provided either one.

The maze navigation domain results showed how OPT performed when the transition dynamics differed between source and target tasks. OPT again resulted in significant gains to learning performance and OPT was also able to benefit from multiple source tasks when each were similar to the target in different ways.

Finally, OPT was also compared to the existing multi-task policy transfer algorithm, Policy Reuse in Q-leanring (PRQL), and the conceptually similar option transfer work by Soni and Singh. OPT performed significantly better than PRQL and SSO in all experiments and had much larger performance gains when using multiple source tasks compared to the performance of PRQL and SSO.

One of the critical requirements of OPT is that state space and action space mappings are provided between source and target tasks. In Chapter 4, I present a method to automatically generate these mappings for domains represented with the Object-oriented MDP (OO-MDP) framework (Diuk, Cohen, & Littman 2008). I also show how OO-MDPs can be leveraged to help automate the process of detecting different tasks that can be used for transfer and help determine which subset of tasks should be provided as source tasks for a new target task.

Chapter 4

OBJECT-ORIENTED OPT

One way in which tasks can differ is in the state space or action space of the tasks. In these cases, providing the agent state and action space mappings between tasks is required for OPT to work. Without such mappings, neither a transfer option's (TOP's) policy, its initiation conditions, nor its termination conditions can be defined. One way to obtain the mappings is to have an expert provide a useful mapping between each pair of tasks to the agent. Requiring an expert to provide such mappings, however, reduces the autonomy of the agent and may not be possible in all situations. Some transfer learning work address this problem by only requiring an expert to provide a set of permissible mappings (rather than providing a single good mapping between each source and target task pair) and using a transfer learning algorithm that can learn from multiple mappings (Soni & Singh 2006; Fachantidis *et al.* 2012). OPT can also use this approach, as was shown by the lunar lander results in Chapter 3. However, requiring an expert to provide a set of mappings is still less desirable than having the agent be able to determine the mappings on its own. Existing work has tackled the challenge of automatically determining mappings in the past. For instance, Kuhlmann and Stone (2007) provide a way to generate a task mapping when a graph-based representation of the system dynamics is provided; Liu and Stone (2006) describe a method for generating a mapping when a qualitative dynamic Bayesian network is

provided; Taylor and Stone (2007) provide an approach (but not a formal process) for finding a mapping; and the AtEast algorithm (Talvitie & Singh 2007) takes a set of candidate mappings and returns the best one after testing each. In general, the graph-based system dynamic approach and the qualitative dynamic Bayesian network approach could be used to create a mapping and then used with OPT; however, these approaches do require more information about system dynamics to be provided. The AtEast algorithm could also be used for selecting a mapping for OPT, however, since OPT can learn the target task while using multiple mappings, it is probably better to simply input the set of candidate mappings to OPT. In this chapter, I introduce *object-oriented OPT* (OO-OPT), which leverages the state representation of object-oriented MDPs (OO-MDPs) (Diuk, Cohen, & Littman 2008) to generate useful state and action space mappings between tasks in the same OO-MDP domain and does not require system dynamic information. Although this creates multiple task mappings, OPT is able to use all of them by creating a separate TOP for each and actually benefits from using all of them because the utility of each mapping can vary depending on the specific target task state. Additionally, the OO-MDP representation assists in defining classes of related tasks between which transfer would be useful and in selecting which source tasks to use for transfer.

This chapter discusses how domains and tasks can be formalized in OO-MDPs, how to generate state and action space mappings between OO-MDP tasks with different state and action spaces, and how these formalisms, along with reward functions defined in terms of OO-MDP propositional functions, can be used to find a candidate set of source tasks for a target task. Finally, empirical results are presented to demonstrate the effectiveness of these mappings when paired with OPT.

4.1 Using OO-MDPs to Facilitate Transfer Learning

OO-MDPs extend the normal MDP definition to provide a factored representation of states. Specifically, an OO-MDP state consists of a set of objects that belong to object classes and are described by a feature vector. OO-MDPs also define a set of propositional functions that operate on objects in a state. (See Section 2.1.1 for more information on OO-MDPs.) For the purposes of discussing transfer learning with OO-MDPs, a domain, in which many possible tasks may be applicable, can be formalized. An OO-MDP domain is a set of object classes ($\mathcal{C}$) with their associated set of attributes ($\mathcal{A}$), a set of propositional functions ($\mathcal{F}$), and a set of actions ($\mathcal{A}$). Further, these actions may be parameterized to object classes. For instance, in a blocks world, a "pickup(block b)" action may be defined that dynamically creates a "pickup" action for each block object in a state. For a given OO-MDP domain, tasks could vary in a variety of ways including (1) different transition dynamics, (2) different reward functions, and (3) different sets of objects present in the state space. Note that the latter case—having different sets of objects present between tasks—induces different state spaces between tasks and potentially different action sets (if actions are parameterized to object classes), and these kinds of tasks will need mappings generated between them. Before discussing how such mappings are generated, it is useful to first consider how different tasks can be identified. Although perhaps not able to distinguish all possible ways in which tasks of the same OO-MDP domain may vary, two ways to differentiate tasks are by *task object classes* and by *object signatures*.

A task object class is a user designated class in the set of OO-MDP classes. Any object in state the belongs to a task object class is a *task object*. If the set of task objects between two states is not equal (that is, there is no bijection between two states' tasks objects that matches equal objects), then the states belong to different tasks. For instance, consider a navigation domain such as the office navigation domain discussed in Section 3.4.1. In

such a navigation domain, the agent's goal is to reach some goal location; different tasks within this domain have different goal locations. In an OO-MDP, these task differences might be captured by a goal location object whose attributes specified its position (and perhaps its size). If the goal location object class is marked as a task object class, then any states that had different values in their goal location object would indicate that the states belonged to different tasks of the same OO-MDP domain. Further, if two states had different numbers of objects of the same task object class, this would also indicate different tasks. For instance, consider the lunar lander domain discussed in Section 3.4.2; one way in which tasks differed was in the thrust action set. In an OO-MDP, the lunar lander domain could be defined with thrust objects that had an attribute specifying a degree of thrust. The action set could then use a parameterized thrust action that operated on the possible thrust objects available (thereby allowing the agent to choose between different thrust actions). If the thrust object class was marked as a task object class, then tasks would be differentiated and identified by different thrust values and by the number of thrust options available to the agent. It may also be possible for multiple object classes to be marked as task object classes, in which case two states would belong to different tasks if at least one of the task objects between them was different. The lunar lander domain, for instance, could mark the thrust object class and a landing pad object class as task classes.

The *object signature* of a state is the number of object instances of each class, excluding task classes, that are present in the state. For instance, if a state representation contains a single agent object and two obstacle objects (neither of which are task classes), its object signature would be: $\langle \texttt{agent}(1), \texttt{obstacle}(2) \rangle$. Because a differing number of objects between two states can result in different reward functions, state spaces and features, and transition dynamics, the object signature is also useful for distinguishing tasks of an OO-MDP domain.

If the only difference between a target task and a source task is their task class objects,

then finding a mapping from the target task state space to the source task state space is trivially performed by replacing the task objects of the target task with the task objects of the source task and leaving all else the same. However, there is not an obvious mapping when the object signatures of two tasks differ. This ambiguity can be resolved under the assumption that a target task's object signature *dominates* the object signature of a source task.[1] Object signature A is dominant over object signature B when there are a greater or equal number of object instances of each class in A as in B. For instance, $\langle \mathtt{x}(2), \mathtt{y}(3) \rangle$ dominates $\langle \mathtt{x}(2), \mathtt{y}(2) \rangle$, but it does not dominate $\langle \mathtt{x}(3), \mathtt{y}(2) \rangle$. Note that under this definition two equal object signatures both dominate each other. If signature A dominates B and there are a greater number of object instances in A than in B for at least one non-task object class, then A *strictly dominates* B. If a target task strictly dominates a source task, there exists a one-to-many mapping from states of the target task to states of the source task. Specifically, for an object class c for which the target task has n instances of c and the source task has $k < n$ instance of c, there are $\binom{n}{k}$ possible mappings between the objects belonging to class c. The set of possible mappings between a target task and source task for objects belonging to a specific class is called a *class-wise mapping*. The set of possible state mappings from the target to the source is the cross product of all class-wise mappings between the tasks. Each of these mappings induces a different policy, but rather than restrict the agent to defining a TOP by only one of these policies, we allow the agent to select from all of them dynamically, by creating a TOP for each. To compactly represent each of these, a single *parameterized TOP* (PTOP) is created for the agent to use.

A PTOP is a special form of a *parameterized option* (POP) that is used for transfer. A POP is an option that requires the agent to specify parameters in order to execute the

[1] The restriction that a target task's object signature dominates a source task's object signature is not unreasonable for the typical situations in which transfer is used: namely, to accelerate the learning of target tasks that are equally or more difficult than the source tasks (rather than than trying to accelerate the learning of an easier task).

option; the parameters chosen affect how the option is executed, much like a parameterized action. (Note that although we introduce the term POP here, this concept has been used in other contexts in existing work, such as MaxQ (Dietterich 2000) and in relational domains that define states as sets of objects similar to OO-MDPs (Croonenborghs, Driessens, & Bruynooghe 2008).) For an OO-MDP, the parameters are typed to object classes and the set of possible parameter choices is the set of object combinations from the current state that would satisfy those parameters. For instance, consider an OO-MDP that defines an object class c by two numeric attributes and a POP (p) that takes one parameter typed to class c. If the current state has three instances of c (specifically: $\{c(2,5), c(4,1), c(3,6)\}$), then the POP could be applied in three different ways: to object $c(2,5)$, $c(4,1)$, or $c(3,6)$. More generally, there would be as many possible applications of a POP to a state as there are combinations of objects that satisfy the parameters. In order for the agent to be able to choose between possible POP applications, the agent needs to store unique Q-values, not just for the POP, but each possible application of the POP; therefore, the object parameter's feature vectors should be part of the namespace of the option in the Q-value option-state pair. For instance, in the previous example, the state ($\{c(2,5), c(4,1), c(3,6)\}$) would have a Q-value for $p(c(2,5))$, $p(c(4,1))$, and $p(c(3,6))$.

For a PTOP, the goal is to create parameters that allow the agent to choose between the different state mappings from the target task to the source task. Consider an OO-MDP domain with one object class (c), a target task with n instances of c, and a source task with k instances of c (with $0 < k < n$). In this case, the PTOP would have k parameters typed to class c that indicate which objects of the target task should be used for representing the state. Because the parameters of a PTOP represent which objects to use in the state representation, the order of the parameters does not matter. For instance, if a PTOP is named p and has two parameters for class c, then the applications $p(c(2,5), c(4,1))$ and $p(c(4,1), c(2,5))$ are the same and return the same Q-value for any state in which they can

be applied. If a target task and source task differed in the number of instances for multiple classes, then parameters would be associated with each of those classes. If a source task contained no instances of an object class present in the target task, then objects of that class would simply be removed from the mapped source task state.

This object-based state mapping also implicitly creates a mapping from the action space of the source task to the action space of the target task, provided that the differences between action spaces are due to differences in the sets of objects between tasks to which the parameterized actions can be applied. Specifically, if the action returned by the policy of a source task is action a and is parameterized to object o in the source task state ($a(o)$), then $a(o)$ maps to action $a(o')$ in the target task, where o' is the object in the target task state that maps to o. Since the PTOP will produce all possible object mappings, all possible action parameterizations in the target task could be selected by the source task. It is important to note that this multi-mapping approach can only work with transfer algorithms like OPT that enable multi-task transfer, because different mappings can be supported by treating them like different source tasks, even though they refer to the same source task.

A final useful property of using OO-MDPs is how reward functions can be defined in general and compact ways by using the propositional functions of the OO-MDP and first-order logic (FOL). For instance, consider an OO-MDP navigation domain in which the goal is for the agent (which is represented by an object) to reach a goal object location. This can be compactly defined with the reward function:

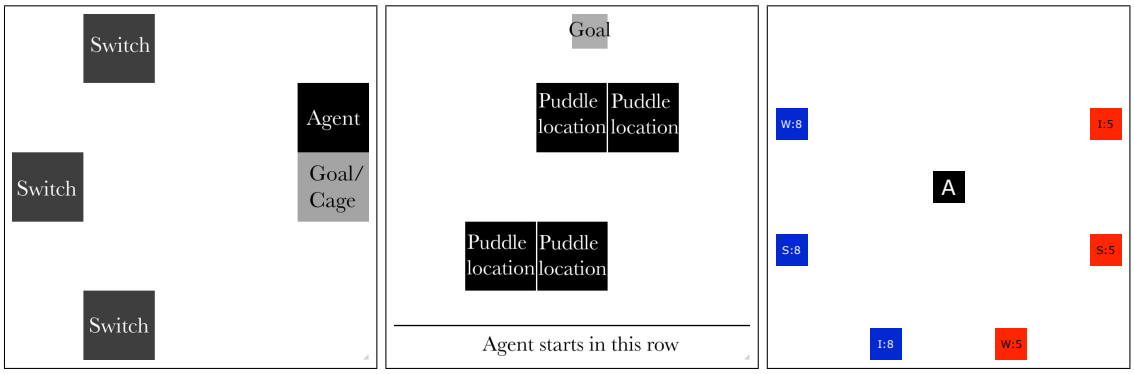
$$\mathcal{R}_{\mathcal{A}}(s, s') = \begin{cases} 1 & \text{if } \exists a \in s', g \in s' \ \texttt{inGoal}(a, g) \\ -1 & \text{otherwise} \end{cases},$$

where $\texttt{inGoal}(a, g)$ is a propositional function of the OO-MDP that returns true when agent object a is in same location defined by goal object g. Note that this reward function

is actually applicable for many different tasks that return different rewards in different states and can be differentiated by setting the goal object class to be a task object class. Further, the reward function is also applicable across tasks with different object signatures and therefore different state and action spaces. The ability to compactly define such general reward functions allows sets of related tasks of an OO-MDP domain to be defined, thereby providing a set of tasks that are good candidates to use for transfer. The object signature of the tasks also lets this set of candidates be further reduced. For instance, if two possible source tasks (A and B) have the same task objects, but A strictly dominates B, then only A should be considered for transfer to a new target task that dominates both, since B would utilize less information from the target than A and would utilize no information that A would not.

4.2 Results

In this section, results are presented for three different domains: a remote switches domain, a puddles domain, and city trading domain. In all domains, different tasks have a different number of objects present, thereby creating a different state space that requires a mapping. In the case of the remote switches domain, the different number of objects induces different transition dynamics. In the case of the puddles domain, the different number of objects creates different reward signals. In the city trading domain, actions are parameterized to objects in the world; therefore, the action spaces between tasks are different as well as the state space. Despite the differences in the number of objects between tasks in all domains, they all use the same FOL-expressed reward functions, which indicates that all of the tasks are related.

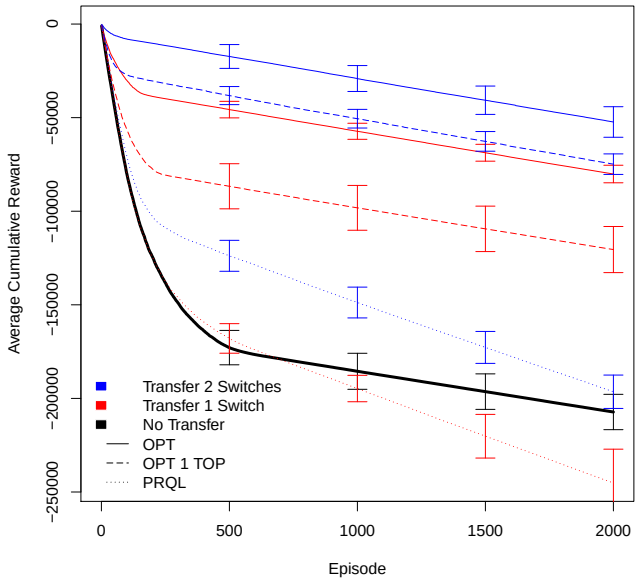


(a) Remote Switches (b) Puddles (c) Trading

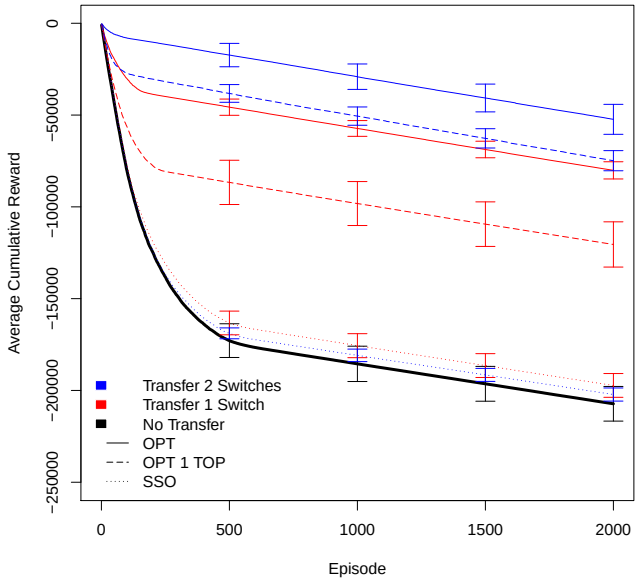
FIG. 4.1. A representation of each of the domains used: (a) the remote switches domain, showing the goal and possible switch locations; (b) the puddles domain, showing the goal and possible puddle locations; and (c) the trading domain, showing the the possible locations of cities, their supply or demand for goods, and the initial position of the agent (the 'A' represents the agent; blue and red squares represent supply and demand cities, respectively; the letter and number on a city square represents the good supplied/demanded and its quantity).

4.2.1 Remote Switches

The remote switches domain is a 5x5 grid world with north, south, east, and west movement actions. The objective is for the agent to reach the goal location. However, the goal is surrounded by a cage that prevents the agent from moving into the goal cell. To lower the cage (and allow access to the goal location), the agent must turn every switch in the world from an "off" channel to the same "on" channel from three possible "on" channels. To turn a switch on and change its channel, the agent must move to the same location as the switch and apply a switch action. Every time the agent applies a switch action to a switch, the switch changes to a random channel other than the current channel (making this a stochastic domain). This stochastic nature requires the agent to be aware of the current channel for every other switch in the world in order to sync them all to the same



(a) OPT compared to PRQL



(b) OPT compared to SSO

FIG. 4.2. Average cumulative reward over ten trials for the remote switches domain with transfer to a three-switch task from sources with either two or one switches present. OPT performs the best and benefits the most from the two switches source task. Moreover, OPT performs better when provided multiple mappings to the source task. Figure (a) compares OPT to PRQL; PRQL provides accelerated learning from the two switches task, but ultimately converges to a less optimal policy. Figure (b) compares OPT to SSO, which provides no statistically significant transfer benefit.

channel and lower the cage. The domain defines four OO-MDP object classes: an agent class, a goal class, a switch class, and a cage class. The agent, goal, and switch class all have two attributes for their x-y position in the world and the switch class has a third attribute with four possible values to represent the "off" channel and the three "on" channels. The cage class consists of one attribute with two possible values indicating whether the cage is up or down. The agent receives a reward of 1 when it reaches the goal object, at which point the episode ends, and -1 for every other state transition.

The goal/cage position, initial agent position, and possible valid switch positions are shown in Figure 4.1 (a). If a task has fewer than three switches in it, then in the initial state of each episode, a random location from the three possible switch locations is selected for each switch. All switches are initialized to "off." Learning performance on the three-switch task is evaluated under five learning conditions: no transfer, transfer from a one-switch task, transfer from a two-switch task, and transfer from each of those source tasks but limiting the number of mappings the agent is provided to one (which means that the agent is only provided one TOP). Comparing performance of OPT when all or only one mapping is provided is meant to determine whether it is useful to use multiple mappings (which requires a multi-source transfer algorithm like OPT), or whether a single mapping provides just as much transfer benefit. Note that different number of switches in each task causes a difference in the transition dynamics between tasks, because all switches must be switched to the same state in order to lower the cage. OPT is also compared against PRQL and SSO, as in Chapter 3. In the case of PRQL, a different policy for each possible mapping was provided to the agent in an effort to make the comparison more advantageous to PRQL, although PRQL hasn't been used to transfer from multiple mappings in the past.

Q-learning was used for all learning and all tasks are learned for 2000 episodes. If an episode lasted for longer than 1000 steps, it was prematurely terminated. The Q-learning parameters were set as follows: $\alpha = 0.1$, $\gamma = 0.99$. The agent followed an ϵ-greedy policy

with ϵ initialized to 0.1 for all states and decayed to a minimum value of 0.01 based on the EUF of the state, according to $\epsilon(s) = \max(0.01, 0.1 \cdot 0.95^{EUF(s)})$. The Q-values were initialized to the semi-pessimistic value of -90. The OPT parameters were set as follows: $\upsilon = 25$, $\iota = 10$, and $\tau = 0.1$. The υ and ι values are smaller than those used in the domains in Chapter 3, because while there is a large state space for this problem, the optimal number of steps required to reach the goal is smaller, therefore requiring fewer updates per state to reach a good policy. PRQL parameters were $\upsilon = 0.97$, ψ is initialized to 1.0 at the start of each episode, τ is initialized to 0, and τ is incremented by 0.05 after each episode. The υ parameter was chosen empirically to be 0.97 due to its balance of learning acceleration and convergence properties. For SSO, TOPs may terminate before reaching a goal terminal state of the source task because not all mappings produced possible states in the source task. Specifically, when the mapped switches are synchronized, but the cage is still down (because of an additional unsynchronized switch in the target task not mapped into the source task), this is not a permissible state in the source task and therefore TOPs in SSO will terminate in these conditions.

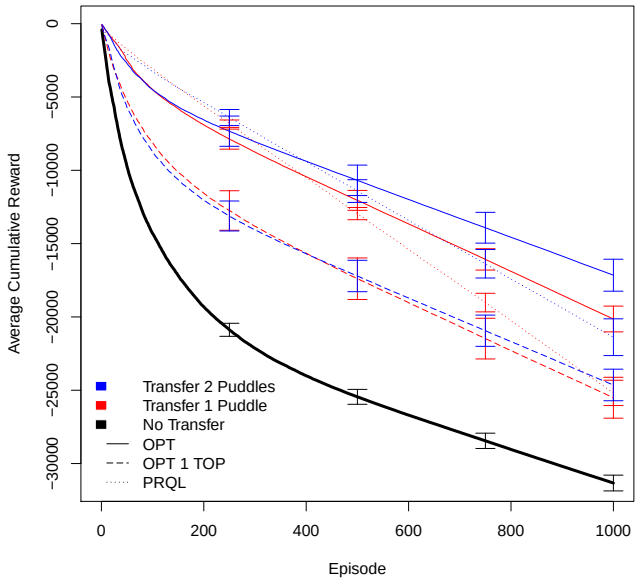
The results showing the average cumulative reward from ten separate trials are shown in Figure 4.2. (Note that results for PRQL and SSO on the one mapping variants are not presented here because results were not sufficiently different.) Performance of all OPT transfer variants are much better than learning without transfer. Transfer from the two-switch task results in the fastest learning, demonstrating that using tasks with as many common features as possible is beneficial. Transfer performance from either the two-switch task or the one-switch task is better when the agent is supplied with a TOP for each possible mapping, rather than just one, demonstrating that it is beneficial to provide the agent with multiple valid mappings. For PRQL (shown in Figure 4.2 (a)), there is a substantial amount of learning acceleration from the two-switch task (though not as much as any OPT variant); however, the policy ultimately converges to a less optimal policy than the baseline, because

the agent always decides to use transfer from a source task, rather than using the target task. It was found in other experiments (not shown here) that lowering the υ value would result in a more optimal policy (since lowering υ results in the agent following a selected source task less frequently during an episode), but at the cost of learning acceleration. There does not seem to be a statistically significant acceleration in learning when PRQL transfers from the one-switch task; it also converges to a less optimal policy than the baseline for the same reason that the agent always selects a source task for transfer. SSO results appear to have slight improvements over the baseline learning, but results are not statistically significant.

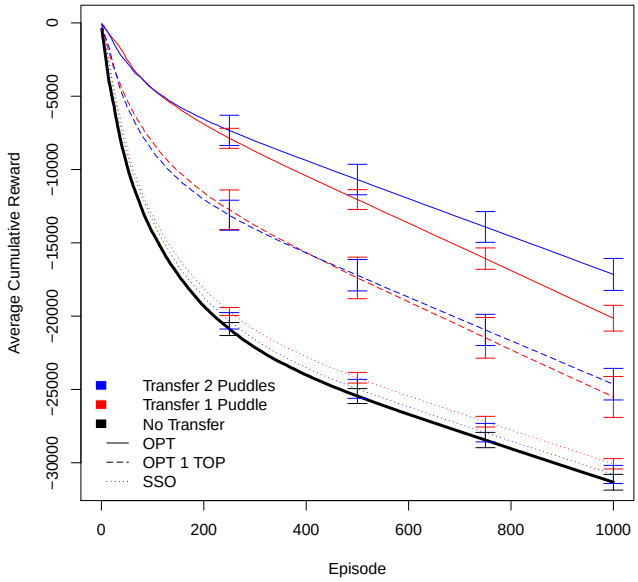
4.2.2 Puddles

The puddles domain is a 10x10 grid world with north, south, east, and west movement actions. The objective is for the agent to reach the goal location while avoiding puddles in the world, which are passable, but undesirable to cross. A single puddle in the world covers 2x2 cells of the world. The domain defines three OO-MDP object classes: an agent class, a goal class, and a puddle class. All classes are defined by two attributes indicating the object's x-y position. The agent receives a reward of 1 when it reaches the goal object, at which point the episode ends, a reward of -5 for entering a cell that a puddle covers, and a reward of -1 for every other state transition. Note that because a puddle covers a 2x2 space, the cost of avoiding a single puddle is at most two actions more than walking straight through it. As a result, the cost of walking completely across a puddle (two cells in diameter) is five times worse than avoiding it.

The initial agent positions, goal location, and possible puddle locations are shown in Figure 4.1 (b). The target task for the experiments performed is a three-puddle task; transfer performance is compared on two source tasks: a one-puddle task and a two-puddle task. Note that because crossing a puddle incurs a greater reward cost than moving through an empty cell (-5 vs. -1), the puddle differences between tasks results in a different reward



(a) OPT compared to PRQL



(b) OPT compared to SSO

FIG. 4.3. Average cumulative reward over thirty trials for the puddles domain with transfer to a three-puddle task from sources with either two or one puddles present. OPT performs very well, with equal learning acceleration from either task, but converges to a better policy from the two-puddle source task. Moreover, OPT performs better when provided multiple mappings to the source task. Figure (a) compares OPT to PRQL; PRQL accelerates learning from both source tasks even more than OPT does, but converges to a less optimal policy. Figure (b) compares OPT to SSO, which accelerates learning slightly over the baseline.

function between tasks. In each task, the agent starts in a random cell of the bottom row of the world and the puddle locations are randomly chosen from the four possible locations. As with the remote switches domain, OPT performance is compared to a baseline learning without transfer, OPT when it is only provided with one of the possible mappings, PRQL (which is provided with all possible mappings as different policies) and SSO.

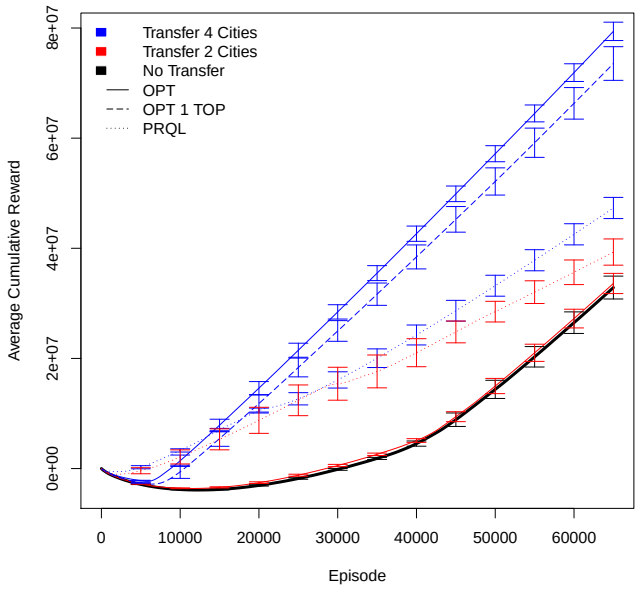
Q-learning was used for all learning and all tasks were learned for 1000 episodes. If an episode lasted for longer than 1000 steps, it was prematurely terminated. The Q-learning parameters were set as follows: $\alpha = 0.1$, $\gamma = 0.99$. The agent followed an ϵ-greedy policy with ϵ initialized to 0.1 for all states and is decayed to a minimum value of 0.01 based on the EUF of the state, according to $\epsilon(s) = \max(0.01, 0.1 \cdot 0.95^{EUF(s)})$. The Q-values were initialized to the semi-pessimistic value of -90. The OPT parameters were set as follows: $\upsilon = 20$, $\iota = 10$, and $\tau = 0.5$. A higher value of τ was selected for this domain because the optimal policy does not require many steps (between 10 and 16 depending on the initial state and puddle locations) and the higher value is used to prevent an overreliance on TOPs for the duration of an episode. Note that the shortness of the optimal policy was similar for the remote switches domain, which did not use a higher value of τ. However, not every target task state in the remote switches domain mapped to a valid source task state and a TOP would terminate when the TOP produced an invalid source task state. In contrast, all target task states of the puddles domain map to a valid source task state; therefore, a higher value of τ is necessary in the puddles domain to prevent an overreliance on the TOPs, whereas it is not necessary in the remote switches domain. PRQL parameters were $\upsilon = 0.95$, ψ was initialized to 1.0 at the start of each episode, τ was initialized to 0, and τ was incremented by 0.05 after each episode. The υ parameter was chosen empirically to be 0.95 due to its balance of learning acceleration and convergence properties.

The results showing the average cumulative reward from thirty separate trials are shown in Figure 4.3. All transfer variants of OPT learn faster than without transfer. As

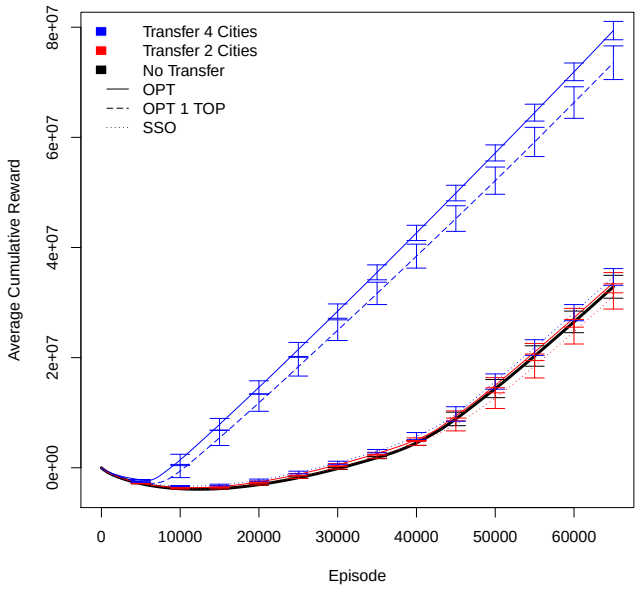
far as learning acceleration goes, OPT performs equally well from either the two-puddle task or the one-puddle task; however, the ultimate policy when transferring from the one-puddle task is sightly less optimal than the when transferring from the two-puddle task. This result indicates that the learning acceleration primarily comes from the the source task guiding the agent to the goal, but that transferring from a source task with a more similar number of puddles allows the agent to learn how to navigate the puddles better than when transferring from a task with a dissimilar number of puddles. OPT also learns faster than the baseline when it is only provided with one mapping to a source task, but worse than when it's provided with many mappings, indicating that it is beneficial for the agent to learn when to use each mapping. PRQL (shown in Figure 4.3 (a)) provides a substantial amount of learning acceleration, more so than OPT. However, it converges to a less optimal policy than any version of OPT. Although not shown, it was found that if the υ value was decreased, PRQL would provide less learning acceleration, without converging to a better policy, in which case there is no reason to decrease υ. It is worth noting that OPT's learning acceleration can also be increased to similar levels as PRQL in this domain by decreasing the value of τ; however, OPT would also converge to a similarly (to PRQL) less optimal policy due to an overreliance on the TOPs. SSO results are shown in Figure 4.3 (b), which shows that SSO accelerates learning slightly. Interestingly, however, the difference is only statistically significant when using transfer from the one-puddle task. It is not clear whether this difference is just statistical noise or whether there is some underlining cause.

4.2.3 City Trading

The city trading domain is a grid world with cities located in different cells. Each city may have a good it demands or a good it can supply from three different possible goods: wheat, sheep, and iron. If the city demands a good, there is a quantity associated with the city for how much it demands. Similarly, cities that can provide a good have a



(a) OPT compared to PRQL



(b) OPT compared to SSO

FIG. 4.4. Average cumulative reward over ten trials for the trading domain with transfer to a six-city task from sources with either four or two cities present. OPT provides substantial learning acceleration from the four-city task, and converges to a better policy compared to the one mapping variant and the baseline. Two-city transfer provides only slight learning acceleration. Figure (a) compares OPT to PRQL; PRQL accelerates learning from both tasks faster than OPT, but converges to a far less optimal policy. Figure (b) compares OPT to SSO, which accelerate learning slightly over the baseline when transferring from the four-city task, and not at all from the two-city task.

certain quantity of the good that they can supply. The agent's goal is to retrieve goods from different cities and supply them to the cities that demand the good such that no city demands any additional goods. Additionally, the agent wants to minimize the amount of distance traveled to accomplish the goal. The OO-MDP of the domain is defined by two object classes: an agent class and a city class. The agent class is defined by five attributes: the agent's x-y position and an attribute representing the quantity of each of the three goods that the agent is carrying. The city class is defined by six attributes: the city's x-y position, the good the city demands, the good the city can supply, and the quantity of goods demanded and supplied. The agent has two city-parameterized actions it can apply: load and provide. The load action causes the agent to go to the parameter-specified city (if the agent is not already there) and take one unit of the supplied good from the city. If the city does not have any more supplies, then the agent simply goes to the city. Conversely, the provide action causes the agent to go to the parameter-specified city and give it one unit of the good that the city demands. If the agent does not have any of the demanded good or if the city does not demand any goods, then it only causes the agent to go to the city. The reward function for this domain returns 2000 for the final supply action that causes all cities to be satisfied. If the agent travels to a different location, the reward function returns $-5 * ||a - a'||$, where $||a - a'||$ is the Euclidean distance between the agent's position in the previous state (a) and its position in the next state (a'). Otherwise, the reward function returns -1 if the agent does not travel to a different location.

Performance in this domain is tested on a six-city task, with the initial state configuration shown in Figure 4.1 (c). The agent starts in the center (cell 5,5), at an equal distance from each of the possible cities. Three of the cities only supply goods and do not demand any goods themselves; the other three cities do not supply any goods and only demand some good. The three supply cities are located on the left side of the world; each supplies a different good; and each provides eight units of their supplied good. The three cities that

demand goods are located on the right side of the world; each demands a different good; and each demands five units of their demanded good. The optimal path is to start with the top left supply city, and move around the circle so that all the supplies are gathered before the agent visits and supplies any of the cities demanding goods. Transfer performance is evaluated using two different source tasks: a two-city source task and a four-city source task. Because any city demanding to receive a good requires the corresponding city supplying the good to exist, initial states for the two-city task and four-city task are generated by randomly selecting a good and then adding the corresponding pair of supply and receive cities in the six-city task for that good. In the two-city task, one of the three possible goods is selected (resulting in two cities being selected); in the four-city task, two goods are randomly selected (resulting in four cities being selected). Because actions are parameterized to city objects, the OO-MDP state mapping between the tasks provides not only the state space mapping, but also the action space mapping. As with the previous OO-MDP domains, OPT performance is compared to a baseline without transfer learning, an OPT variant in which only one of the possible state mappings can be selected by the agent (in this case, the one mapping provided is the mapping that is optimal in the initial state of the task), PRQL, and SSO. Note that a number of possible object mappings from the six-city task to a lower city task can produce invalid states, because an object mapping in this domain determines which cities in the target task appear in the mapped source task state and it is possible for a mapping to retain a city that demands a certain good, but not the city that can provide it. For OPT, this problem is implicitly resolved by the initiation conditions of a TOP which require the mapped source task state to possess a certain number of episodic updates, for which there will be none if the mapping produces an invalid state. For PRQL, this is resolved explicitly by only supplying the PRQL algorithm with policies that use mappings that will produce valid states. For SSO, the problem is explicitly resolved by the initiation conditions of a TOP which require the mapped source task state to be valid

source task state.

Q-learning was used for all learning. The two-city source task was learned for 5000 episodes; the four-city source task was learned for 20000 episodes; and the six-city target task was learned for 65000 episodes. The Q-learning parameters were set as follows: $\alpha = 0.1$, $\gamma = 0.99$. The agent followed an ϵ-greedy policy with ϵ set to 0.1 (which was constant and never decayed). The Q-values were initialized to the semi-pessimistic value of 50. The OPT parameters were set as follows: $\upsilon = 100$, $\iota = 800$, and $\tau = 0.1$. PRQL parameters were $\upsilon = 0.95$, ψ was initialized to 1.0 at the start of each episode, τ is initialized to 0, and τ is incremented by 0.001 after each episode. The τ parameter was increased by a smaller than usual amount after each episode, since the target task in this domain takes much longer to learn than previous tasks that were tested.

The results showing the average cumulative reward from ten separate trials are shown in Figure 4.4. OPT using transfer from the four-city task learns much faster than the baseline learning without transfer. Moreover, OPT using the four-city task for transfer converges to a better policy than the baseline after the 65000 episodes of learning. The OPT variant that can only use a single mapping (the mapping that is most optimal from the initial state) learns nearly as fast as OPT when provided all mappings, but it converges to a slightly less optimal policy. OPT using the two-city task for transfer learns slightly faster than the baseline, but the gain is negligible compared to the gain from the four-city transfer and is a much smaller gain than what OPT has provided in other domains. A possible reason for this result is that a large negative reward is received for traveling to other cities and so a large negative reward is incurred by following the multi-step OPT, resulting in the agent learning to avoid using the TOP the next time. The four-city task, in contrast, may not suffer from the same problem because the four-city task results in the agent being much closer to the goal state than the two-city task, thereby allowing the goal state reward to mitigate the large negative reward from initially executing the TOP. The one-mapping

two-city transfer OPT variant is not shown in the results, because it is not substantially different from the normal two-city transfer results.

PRQL results are shown in Figure 4.4 (a). PRQL learns much faster than the baseline learning without transfer when provided policies from either the four-city task or the two-city task and it also initially performs better than OPT. However, PRQL ultimately converges to a much worse policy than OPT and learning without transfer for both transfer scenarios (with transfer from the four-city task converging to a slightly better policy than transfer from the two-city task). PRQL also has an abnormal behavior in its learning curve between episode 2000 and episode 3000 for both transfer scenarios, which is possibly due to changes in which policies it selects to use for each episode.

SSO results are shown in Figure 4.4 (b); SSO learns only slightly faster than the baseline when using transfer from the four-city task. There is no statistically significant difference in performance compared to the baseline when transferring from the two-city task.

4.3 Conclusions

This chapter described how the factored representation of OO-MDPs can be exploited to define sets of related tasks of the same domain and automatically generate a number of task mappings between target and source tasks, all of which OPT can use by creating a TOP for each. Specifically, tasks are related if they belong to the same OO-MDP domain and have the same reward function that can be compactly defined by using the propositional functions of the OO-MDP domain. State mappings between two tasks are automatically generated by finding all mappings of objects of the same class and action mappings are similarly provided if the actions in tasks are parameterized to objects. These approaches were empirically validated on three different OO-MDP domains: a remote switches do-

main, a puddles domain, and a city trading domain. Although the principal difference between target and source tasks in all of these domains was that there were a different number of objects present (thereby inducing a different state space between tasks), each of these domains tested a different form of transfer between the target and source tasks. The difference in the number of objects of the remote switches domain tasks resulted in different transition dynamics between the tasks. For the puddles domain, the different number of objects between tasks resulted in different reward functions (even though all tasks had the same abstract OO-MDP reward function). For the city trading domain, the different number of objects between tasks resulted in a different action space between tasks. In all of these domains, OPT was compared to a baseline learning without transfer on the target task and to two existing policy transfer algorithms: Policy Reuse in Q-learning (Fernández & Veloso 2005) and the options method by Soni and Singh (2006). OPT performed favorably compared to the baseline and existing transfer approaches in all domains.

One kind of domain that was not tested in this chapter was a continuous domain that requires value function approximation to learn. The reason such a domain was not tested is because existing value function approximation techniques typically rely on states being represented by a single feature vector, whereas OO-MDPs are represented by an unordered set of objects, each represented by their own feature vector. In the next chapter, methods for value function approximation in OO-MDP domains are introduced so that the transfer benefits enabled by OO-MDPs can be exploited in continuous domains.

Chapter 5

OBJECT-ORIENTED MDP VALUE FUNCTION APPROXIMATION

The previous chapter discussed how the representation of object-oriented MDPs (OO-MDPs) can be exploited to facilitate transfer learning with OPT. In particular, the OO-MDP representation provides a means to automatically create state (and possibly action) space mappings between tasks, which OPT can use in learning by creating a TOP for each mapping. Ideally, these approaches should extend to continuous domains, which require the use of value function approximation (VFA) to learn. However, existing VFA methods typically require a fixed feature vector that describes the state, whereas OO-MDP states are represented by an unordered set of objects, each represented by a separate feature vector. A naive attempt to resolve this problem might be to arbitrarily order the objects of each state the agent enters, concatenating objects' feature vectors into a single feature vector. However, such an approach may cause two identical OO-MDP states to have two different feature vectors if the objects are ordered differently for each state. For instance, consider an OO-MDP domain with one object class X that is defined by two numeric attributes. In this domain, the OO-MDP states $\{X(2,4), X(7,8)\}$ and $\{X(7,8), X(2,4)\}$ are the same because the ordering of the objects does not matter. However, if these states were merely concatenated into a single feature vector following the order of object appearance, two

different feature vectors would be produced: $\langle 2, 4, 7, 8 \rangle$ and $\langle 7, 8, 2, 4 \rangle$, respectively. Moreover, the similarity of two feature vectors would depend on which ordering was used. For instance, the state ordering producing feature vector $\langle 1, 4, 6, 9 \rangle$ has a smaller Euclidean distance from $\langle 2, 4, 7, 8 \rangle$ than that of an ordering producing $\langle 6, 9, 1, 4 \rangle$. Since VFA techniques are typically sensitive to these kinds of differences, concatenating the feature vectors of objects in an arbitrary order is not sufficient. Another alternative is for a designer to provide a canonical method for ordering objects based on some evaluation of their values. However, requiring an expert to design such an effective ordering method lowers the autonomy of the agent and may still suffer from the same vector similarity issues. This chapter introduces techniques for VFA in OO-MDP domains that does not suffer from this same problem and that can be applied in any VFA approach based on discretization, such as cerebellar model articulator controller (Albus 1971), or on instance-based regression, such as sparse distributed memories (Kanerva 1993). Empirical results using these approaches with OPT on a continuous domain are also presented.

5.1 Discretization-based VFA for OO-MDPs

The most trivial form of value function approximation is simply discretizing the state space so that any continuous state belongs to exactly one discrete state in a finite set of discrete states. Since there is a potentially infinite number of continuous states, discretizing the state space generalizes learning between continuous states that map to the same discretized state. Given a set of discrete states, linear gradient descent SARSA (see Algorithm 4) is performed by treating each discrete state as a binary feature that is "on" if the current continuous state maps to the respective discrete state, and "off" otherwise. As a result, only one feature for every state is ever on and linear gradient descent SARSA performs effectively the same as tabular SARSA would if the actual state space were the same as the

discrete state space. Given a feature vector of continuous state variables, discretization is often performed by using a rectangular tiling of the variables. That is, an interval *width* for each variable of the feature vector is specified and the variables are partitioned into *tiles* spanning rectangular intervals with the specified widths. For instance, if a continuous variable x was rectangularly tiled with a variable interval width of 2 and with interval delineation aligned with the value 0, then between the values of 0 and 10, there would be five tiles ($\{[0,2),[2,4),[4,6),[6,8),[8,10)\}$). If two variables ($x$,$y$) were rectangularly tiled with widths 2 and 0.5, respectively, and with interval delineations aligned with the origin, then between the points $(0,0)$ and $(10,10)$, there would be one hundred tiles (five intervals along the x variable and twenty intervals along the y variable).

Because OO-MDP states are described by a set of objects, each with their own feature vector, any discretization approach that is applicable to feature vectors can also be applied to each object of the OO-MDP state. For rectangular tilings, for instance, a width and interval delineation alignment for each attribute of each object class would need to be specified. Given a width and interval delineation alignment for each object class attribute, each object can be represented by a discretized object that has discrete values for each attribute. If all object values are discrete, then the entire state is discrete and any Q-value weights associated with it can be queried. Ideally, however, state querying should be both fast and memory efficient. One way to make querying fast and memory efficient is to compute a hash value for each state and store only visited states in a hash table data structure so that memory is conserved to those states needed, while still permitting fast access to each discrete state. For states represented by a single feature vector, a hash value for the state can be computed as a function of the discretized attribute values. If minimum and maximum values for each state variable are known, then a unique hash value for each tile can be computed in a similar way as the memory index of a cell in a matrix data structure would be computed. For instance, if a 2D matrix data structure with C columns and R

rows were defined, the memory index (m) of the cell located at column c and row r would be computed as $m = c + rC$. If the matrix had an arbitrary number of n dimensions, then the memory index calculation would be generalized to $m = \sum_{i=0}^{n} e_i \prod_{j=0}^{i-1} d_i$, where e_i is the element index along the ith dimension and d_i is the size of the ith dimension. For a continuous state's feature vector, this same equation can be used to compute the hash value for a state's tile by setting the dimension size along each variable to the number of intervals that span the variable from the minimum value to the maximum value and setting the element index for each variable to the interval in which the vector's value for that dimension belongs. Specifically, let M_i be the number of intervals along the ith variable, this interval is defined as:

$$M_i = \left\lceil \frac{R_i - L_i}{W_i} \right\rceil, \tag{5.1}$$

where R_i is the maximum "right" value of the ith variable, L_i is the minimum "left" value of the ith variable, and W_i is the width of intervals for the ith variable. Also let I_i be the interval to which the value for the ith variable in feature vector F belongs, which is defined as:

$$I_i = \left\lfloor \frac{F_i - L_i}{W_i} \right\rfloor, \tag{5.2}$$

where F_i is the value of the ith variable in feature vector F. Finally, the tile hash value for a continuous feature vector is defined as:

$$H(F) = \sum_{i=0}^{n} I_i \prod_{j=0}^{i-1} M_j. \tag{5.3}$$

Because OO-MDP states are represented as a set of objects, each with their own feature vector, a hash value using this approach could be computed for each object. However, for the hash function to produce the hash value for the tile of an entire state, it must operate on all object feature vectors. If there was at most one object instance of each class in a state,

then the state hash value could be computed (using Equation 5.3) from a single feature vector of all objects that was constructed by concatenating object feature vectors in some specified (but otherwise arbitrary) order of object classes. This method would ensure that all states belonging to the same tile produced the same the same hash value since the order of all object attributes for a state would always be the same. However, since there may be multiple object instances of the same class, there must be a defined way of ordering objects of the same class such that two sets of object instances belonging to the same tile are always ordered the same way. Although there are a number of possible ordering methods, the one proposed here is to compute the hash value of each object's feature vector and order objects of the same class in ascending order of the object hash values (descending order would also be acceptable). Given this ordering, the hash value for an OO-MDP state's tile is computed by using Equation 5.3 on a feature vector that is the concatenation of the feature vectors of all objects in the state, ordered primarily by the class of the objects and then by the hash value of object feature vectors. Since the hash value of each object's feature vector is computed before the hash value for the whole state, the hash equation can be re-expressed to reuse the hash values of each object. Specifically, if a lower limit, upper limit, and width for each attribute of each object class is provided, then the unique tile hash value for an OO-MDP state is computed as:

$$\hat{H}(s) = \sum_{i=0}^{N_s} H(f(o_{si})) \prod_{j=0}^{i-1} V(c(o_{si})), \tag{5.4}$$

where N_s is the number of object instances in state s, o_{si} is the ith object instance in state s (ordered by object class and then the hash value of each object's feature vector), $f(o)$, is the feature vector of object o, $c(o)$ is the object class to which object o belongs, and $V(c)$ is the *volume* of object class c. The volume of the object class is the number of tiles that span the space defined by the object class' minimum and maximum bounds for each attribute of

the class. Formally, the volume is defined as:

$$V(c) = \prod_{t \in \mathcal{T}_c} \left\lceil \frac{R(t) - L(t)}{W(t)} \right\rceil, \tag{5.5}$$

where $\mathcal{T}_c$ is the set of attributes defined for object class c, $L(t)$ is the minimum "left" bound for attribute t, $R(t)$ is the maximum "right" bound for attribute t, and $W(t)$, is the interval width for attribute t.

Since OO-MDPs also allow for actions parameterized to objects, the actions of a discretized state also need to be similarly discretized. Discretizing the actions, however, is trivial once the discretization of the state is performed. Specifically, the parameters can be replaced with the discrete version of each object parameter (or the hash value of each object parameter).

Although discretizing continuous states to a single tile provides generalization between between all continuous states belonging to the same tile, generalization is often too broad and also produces poor edge effects. For instance, if an interval for a variable is delineated on the value 1, then even though the values 0.99 and 1.01 are very similar, the values reside in different tiles and therefore will have no generalization between them. In contrast, if the interval width is 2, then there will be generalization between values 1.01 and 2.99, even though they are much more dissimilar values than 0.99 and 1.01. To gain back more discrimination between states, while maintaining the same amount of generalization, and diminish edge effects, other discretization-based approaches to VFA may be used. Cerebellar model articulator controllers (CMACs), for instance, are one of the most common forms of discretization-based VFA that improves these issues and is what was used for the lunar lander domain in Chapter 3. Instead of creating a single tiling of the state variables of a continuous state, CMACs create multiple tilings. Each tiling is offset by a different position so that there are different interval delineations. Figure 3.3 (a) shows two

offset tilings of a 2-variable continuous state. If a state resides in a tile of a given tiling, then it is said to "activate" that tile. Each state will activate one tile from each tiling and because the tilings are offset from each other, each of the activated tiles spans a different region of the state variables. As a result, some of the states that activate the same tile in one tiling will activate different tiles in other tilings. For instance, Figure 3.3 (b) shows the position of the 2D continuous state 'A' and the tiles of two tilings that it activates. Figure 3.3 (c) shows the position of a different state ('B'); while state 'A' and 'B' activate the same tile in the first tiling, they activate different tiles in the second tiling. The benefit of tiles from different tiles activating together for only certain common states is that it allows for greater discrimination between states while still generalizing learning across all states in the same tile. Moreover, this representation also reduces edge effects, since two states that are very similar but activate two different tiles may activate the same tile in another tiling.

Since CMACs use the same discretization techniques as a simple single tiling discretization, the same hash-based methods for OO-MDPs can be applied for CMACs, differing in that a different hash value is computed for each tiling. The difference in the hash values is produced by different tilings using slightly different minimum "left" bounds for each attribute of each object class, thereby producing a differently offset tiling.

5.2 Instance-based VFA for OO-MDPs

Another form of VFA that may be adapted to work with continuous OO-MDP states is instance-based regression. Instance-based regression involves storing a database of exemplar states. Some approaches have a predetermined fixed database of exemplar states, making the function approximation a radial basis network (Poggio & Girosi 1990; Kretchmar & Anderson 1997); others have a dynamic database of exemplars that is modified based on states visited by the agent (Smart & Kaelbling 2000; Ratitch & Precup 2004).

Each exemplar state has an associated feature. The value of the feature is typically some similarity measure between the query state (the state for which the Q-value is to be predicted) and the exemplar state to which the feature is associated. That is, if the the query state is very similar to the exemplar state, then the feature associated with the exemplar state will be very high; if the query state is very dissimilar from the exemplar state, then the feature associated with the exemplar state will be very low. Therefore, in order to adapt instance-based regression techniques to continuous OO-MDP states, a similarity measure between OO-MDP states must be provided.

If a continuous state is represented by a feature vector, then the similarity between states can be computed using a radial basis function. A radial basis function is a function of the distance between some query vector and a "center" vector that returns a real value; that is, a function $\phi : \mathbb{R}^n \rightarrow \mathbb{R}$ is a radial basis function if $\phi(\mathbf{x}) = \phi(||\mathbf{x}-\mathbf{c}||)$. For instance, a common choice for a radial basis function is the Gaussian radial basis function defined by:

$$\phi(r) = e^{-\frac{r}{b}}, \tag{5.6}$$

where $r = ||\mathbf{x} - \mathbf{c}||$, and $b > 0$ is a bandwidth factor that affects how strongly the function responds to query vectors that are distant from the center vector. That is, for all $b > 0$, ϕ returns a maximum value of 1 when the distance from a query vector to the center is zero and as b is increases, the value of ϕ increasingly approaches 1 for vectors that are a positive distance from the center. A complete review of radial basis functions and their uses in radial basis networks is provided by Poggio and Girosi (1990).

Because the objects of an OO-MDP are represented by feature vectors, a radial basis function can be used to compute the similarity of two objects of the same class. However, because the similarity must ultimately be computed as a function of states, a similarity metric between sets of feature vectors needs to be defined. To compare the distance between

sets of objects, I use the *minimum distance matching* between the sets of objects. A matching between two sets of objects (X and Y) of equal cardinality is a bijection between the two sets so that every object in X is matched with an object in Y. A bijection between the two sets ($f : X \rightarrow Y$) is a minimum distance matching if the sum of the distances between matched objects is the minimum possible value. Formally, the minimum distance matching between two sets of objects ($M(X, Y)$) is:

$$M(X, Y) = \underset{b}{\operatorname{argmin}} \sum_{x_i \in X} ||x_i - b(x_i)||, \tag{5.7}$$

where b is a bijection between object sets X and Y. This is effectively the same problem as the *assignment problem*, which can be solved in polynomial time with the hungarian algorithm (Kuhn 1955). Given the minimum distance matching between two sets of objects, a total similarity measure between two OO-MDP states can be computed using any of a number of possible approaches. Ideally, inappropriate similarity measure should be selected based on the kind of instance-based learning to be performed. In this work, the similarity between two states is defined as the minimum matched object similarity between the states, where the object similarity is determined by a radial basis function of the object feature vectors.

In instance-based linear VFA, there is a state feature for each exemplar in the database. The Q-value for a given query state s and action a is estimated as the linear combination of the state features values, which are based on the similarity from s to each state feature's corresponding exemplar state, and the weights for each state feature for action a. That is, $Q(s, a) = \sum_i \theta_a(i)\phi_s(i)$, where $\theta_a(i)$ is the action weight for action a for the ith exemplar state, and $\phi_s(i)$ is the similarity-based feature value between state s and the ith exemplar state. However, if actions are parameterized to the objects of the state in which the action can be applied, then each state has a different action set. Therefore, to estimate the

Q-value for a parameterized action in a query state, the action must be mapped to corresponding actions in the exemplar states. Because the similarity between states is based on the minimum-distance matching of objects, this mapping is already provided. Given a mapping between objects of the states, the parameterized actions in each state can be mapped by mapping their parameters. That is, action $\mathtt{a}(x, y)$ applied in state q would be mapped to action $\mathtt{a}(b(x), b(y))$ in state e where $b = M(q, e)$ is the minimum-distance matching between states q and e.

5.2.1 Sparse Distributed Memories

Given the above approach to measure OO-MDP state similarity, any instance-based VFA approach can be used. In particular, this work explores using *sparse distributed memories* (SDMs) (Kanerva 1993); specifically, Ratitch and Precup's (2004) work with SDMs for online learning of continuous states is followed. Sparse distributed memories are a form a local linear instance-based regression. Like other forms of instance-based regression techniques, SDMs store a database of exemplar instances with associated features that indicate the degree of similarity between a query state and the exemplar state. However, in SDMs, only a fraction of the exemplars will have a positive similarity with any given query state; the rest will return zero similarity with the query state. Exemplars that have a positive similarity value with a query state are said to be *activated* by the query state. The feature value associated with each exemplar is the normalized similarity. That is, the feature value (f_i) of the ith exemplar state for some query state q is:

$$f_i(q) = \frac{\mu_i(q)}{\sum_k \mu_k(q)}, \tag{5.8}$$

where $\mu_i(q)$ is the similarity between the ith exemplar state and query state q. Normalizing the similarity with respect to the sum of the query's similarity with all other states it acti-

vates improves the function approximation in regions in which the similarity of exemplar states to the query is small(Ratitch & Precup 2004; Kretchmar & Anderson 1997). For OO-MDP states, the similarity between states is measured as the minimum matched object similarity between the states. For the object similarity, a threshold-based radial basis function called the triangle kernel is used (Atkeson, Moore, & Schaal 1997). Using the triangle kernel, the similarity of two objects of the same class (c) is given by:

$$S_c(o_1, o_2) = \begin{cases} 1 - \frac{||f(o_1) - f(o_2)||}{\beta_c} & \text{if } ||f(o_1) - f(o_2)|| < \beta_c \\ 0 & \text{otherwise} \end{cases}, \tag{5.9}$$

where $f(o)$ is the feature vector of object o, $||f(o_1) - f(o_2)||$ is the Euclidean distance between the feature vectors of objects o_1 and o_2, and β_c is a user defined distance-threshold for object class c. By using a similarity measure that drops off to zero after a certain distance threshold, the set of activated exemplar states that need to be considered when computing the value for any given query state is restricted. Such a similarity measure also allows the value function to modeled for local regions of the state space. Note that Ratitch and Precup (2004) use a variant of the triangle kernel in which the similarity between two feature vectors is the minimum similarity of their attributes, and the attribute similarity is a triangle kernel of the attributes (for which each attribute has its own distance threshold). A similar approach could be used for computing the object similarity, since objects of the same class have the same attributes, but doing so would require specifying additional parameters (namely, a threshold for each attribute of each class, rather than a threshold for each class).

5.2.2 Efficient Exemplar Retrieval

In order for SDM-based VFA to be efficient, there needs to be an efficient means to query the activated exemplar states. If states were only represented by a single feature vector, then the exemplar states activated by a query state could be efficiently provided by indexing exemplars with a *kd-tree* (Bentley & Friedman 1979). A kd-tree is a data structure for indexing elements that have a k-dimensional key, where each key element represents a coordinate in Euclidean space. For states represented by feature vectors, the state key is the feature vector. Kd-trees are efficient at performing range searches, i.e., finding the set of instance in the tree whose keys are within a maximum distance from a query key. If the similarity between two states is defined by a triangle kernel or a similar kernel that returns zero after a threshold distance, then the exemplars activated by a query state can be efficiently retrieved from a kd-tree by using a range search with a maximum distance set to the distance at which the kernel returns zero. For OO-MDP states, however, a kd-tree cannot be used since states are not represented by a feature vector. If the similarity measure between two OO-MDP states satisfies the triangle inequality, then distance metric-based spatial indexing structures such as *vp-trees* (Bozkaya & Ozsoyoglu 1997; Fu *et al.* 2000) and *m-trees* (Ciaccia, Patella, & Zezula 1997) could in principle be used to perform a range search, however, it was found in experimentation that due to the uniform covering of exemplar states across the state space, such distance-based indexing structures are extremely inefficient and are made even slower because they require the full state-wise distance to be computed between the query state and nodes in the tree data structure. (In contrast, a kd-tree only compares a single dimension of the key at each node.) In this work, I propose a variant of an *inverted index* (Fox *et al.* 1992; Zobel & Moffat 2006) for object instances that is itself indexed with a kd-tree. An inverted index is a data structure typically used in information retrieval and text search. Specifi-

Algorithm 6 Find-Activated(q, K, I, β, m)

1: **Inputs** q: query state. K: set of kd-trees indexing all exemplar state object instances by object class. I: inverse index from object instance to exemplar state. β: set of object distance thresholds. m: minimum size of candidate set to search by inverse index.
2: candidates $\leftarrow \{\}$
3: For each object class c in q:
4: For each object instance o of class c in state q:
5: objectKeys $\leftarrow K_c$.range(o, β_c) // exemplar objects within distance β to o
6: refs $\leftarrow \{I(k) : k \in$ objectKeys $\}$ // set of exemplar states from inverse index
7: **if** candidates = $\{\}$
8: candidates $\leftarrow$ refs
9: **else**
10: candidates $\leftarrow$ candidates $\cap$ refs
11: **if** sizeOf(candidates) $<$ m
12: break out of both loops // go to line 13
13: activated $\leftarrow \{ i : i \in$ candidates, $\mu_i(q) > 0 \}$ // exemplars with similarity to $q > 0$
14: **return** activated

cally, given a collection of documents that are represented as bags of words, an inverse index stores a list of each possible word associated with a set of references to the documents in which the word appears. A word and reference set pair is retrieved quickly using a hash table (or some other efficient data structure for text keys). If a user wishes to search for documents containing key words $k_1, k_2, ..., k_n$, then the set of documents containing the key words is $\bigcap_i^n I(k_i)$, where $I(k_i)$ is the set of document references for keyword k_i that is returned from the inverse index. For OO-MDP SDMs, the database is a set of OO-MDP exemplar states (rather than text documents). Each state can be thought of as a bag of object instances, where an object instance is a feature vector and object class. Therefore, an element of an inverted index of OO-MDP exemplar states is an association from an object instance to the set of exemplar states that include the object instance.

A critical difference between the application of text document searching and finding the set of activated OO-MDP exemplar states is that in the former, query key words must match exactly with words in the database; in the latter, object instances in the query state

must only be within a certain distance of the objects in the exemplar states. Additionally, the set of exemplar states retrieved by the inverse object instance indexing may include exemplar states that are not activated by the query state. This disagreement may occur because the inverse index may match two objects in the query state to the same object in an exemplar state, which would not result in a bijection between the query and exemplar state. Therefore, after computing the minimum distance matching, one of the objects in the query state may be too far from its matched object in the exemplar state, thereby failing to activate the exemplar. To address these differences, a modified version of inverse indexing for finding activated OO-MDP exemplar states is shown in Algorithm 6. Given a set of exemplar states, an inverse index from the object instances of the exemplars to the exemplars is created. Additionally, a set of kd-trees (one for each object class) stores the set of exemplar object instances. Given such an indexing, the algorithm to find the exemplar states activated by a query state starts by iterating over each object instance in q (lines 3-4). For each object instance (o), a range search for exemplar object instances near o is performed using the kd-tree for o's class (line 5). The distance for the range search is set to β_c: the minimum distance at which the kernel function for objects of o's class (c) returns zero. As a result, all objects returned in the range search will have a positive similarity to o. From the set of exemplar object instances near o, the inverse index is queried to retrieve the set of exemplar states that contain them (line 6). Note that for state q to activate exemplar e, each object instance in q must have a positive similarity with an object instance in e. Therefore, only the intersection of exemplars returned from each object instance can be an activated exemplar because only those in the intersection will have a positive similarity match with each object. That being the case, the set of possible candidates is updated to be the intersection of all exemplar states returned from each previously searched object instance (lines 7-10). Because the set of activated candidates can never increase in size after it is intersected with the returned exemplar states from each subsequent object instance in q, there

may reach a point at which the size of the candidates is small enough that there is no more computational benefit from searching the kd-trees for each object instance. Therefore, if the size of the candidate set is below some threshold, the loops searching over each object instance are escaped (lines 11-12). The last steps of the algorithm compute the final similarity between q and each of the candidate exemplars, returning those that have a positive similarityi.e., the activated set of exemplars (lines 13-14).

5.2.3 Dynamic Resource Allocation

Before using an SDM, it must be determined what the database of exemplar states will be. One approach would be to uniformly pre-populate the database; however, this may lead to creating exemplars for regions of the state space that will be either never or rarely visited by the agent. Having the database contain exemplars that are unlikely to be visited is both memory inefficient and computationally inefficient (since the set of exemplar states must be searched to find the activated states). An alternative is to use dynamic resource allocation so that the database of exemplars changes with learning. Although there are a number of approaches for dynamic resource allocation in instance-based learning (Anderson & others 1993; Flynn, Flachs, & Flynn 1992; Fritzke 1997; Hely, Willshaw, & Hayes 1997; Platt 1991; Rao & Fuentes 1998; Sutton, Whitehead, & others 1993), this work follows the *N-based heuristic* (Ratitch & Precup 2004), which provides more stable results for online reinforcement learning than other approaches. The N-based heuristic starts with a completely empty database of exemplars and adds states based on the states observed by the agent. States are added to the database when the database is too sparse around a current state to make an adequate prediction. Additionally, states are added in a way that ensures that exemplars are evenly distributed across their local neighborhoods; specifically, each

pair of exemplar states ($\mathbf{h}^i$ and $\mathbf{h}^j$) must satisfy the criterion:

$$\mu(\mathbf{h}^i, \mathbf{h}^j) \leq \begin{cases} 1 - \frac{1}{N-1} & \text{if } N > 2 \\ 0.5 & \text{otherwise} \end{cases}, \tag{5.10}$$

where $\mu(\mathbf{h}^i, \mathbf{h}^j)$ is the similarity between exemplar states $\mathbf{h}^i$ and $\mathbf{h}^j$, and N is a user-defined parameter that indicates the minimum number of exemplars that should be activated by any instance. If the number of exemplars activated by a query state is less than N, then additional exemplars are added to the database according to two rules:

Rule 1 If fewer than N exemplars are activated by query state q, then add q as an exemplar if adding q does not violate condition 5.10.

Rule 2 If after applying Rule 1, there are $N' < N$ exemplars activated by query state q, then randomly generate and add $N - N'$ new exemplars within the activation neighborhood of q that satisfy condition 5.10.

In the original work, states were represented by feature vectors and activation required a query state to be within an rectangular region of the exemplar states; therefore, generating new random states around q was performed by randomly sampling the rectangular space around q. For OO-MDP states, a random state in the activation neighborhood of q is generated by randomly generating a new object instance for each object in q. Object o of class c randomly generates a new object o' of class c by randomly sampling within a sphere centered on o with a radius equal to the distance of the triangle kernel for c: β_c.

If there is a memory limit on the number of exemplar states in the database that is reached and a query state q activates fewer than N exemplars, then an existing state in the database is randomly selected and changed to a state within the activation neighborhood of q. If exemplar state h is to be repurposed, then before changing its values, the most similar

state to it (h') is first found and h' has its values changed so that h' is in a position between h and the previous position of h'. Resetting h''s values mitigates the effects of removing an existing exemplar. For an OO-MDP state, changing h' to be between h and h''s previous position first requires finding the minimum matching of objects between h and h' and then setting the object values of h' to be values that are between the matched objects of h and h'.

Given these mechanism for dynamic resource allocation, a useful database of exemplars for OO-MDP states can be formed.

5.3 Using Multiple Variable Subsets

One problem with discretization and instance-based VFA approaches is that they may not scale well as the dimensionality of the state increases. For CMACs, a common solution is to use multiple tilings over multiple subsets of variables. For instance, if states are defined by three state variables x, y, and z, instead of using n overlapping tilings of all three dimensions, n tilings *for each variable* might be created for $3n$ total tilings. Although not as commonly used for instance-based methods, a similar approach of storing multiple databases of exemplars, each using a different subset of state variables to define the states, could be taken. In either case, the result of using multiple state variable subsets is that different state features for the linear gradient descent SARSA algorithm may represent different aspects of the state.

For OO-MDP states, using variable subsets has a few more complications because it is unclear how to differentiate the variables of different objects of the same class. However, if variable subsets are determined on the object class level and the variables of all objects of the same class are included, then this problem is resolved. For instance, if an OO-MDP domain consisted of two object classes (X and Y) and class X was defined with two

attributes x_0, x_1, then a designer might define one variable subset (S_1) to be for the single attribute x_0 of class X and no attributes for class Y. If a state (s) consisted of two objects (o_1, o_2), each of class X, then variable subset S_1 would define s with two variables: $o_1.x_0$ and $o_2.x_0$. If s consisted of three objects (o_1, o_2, o_3) each of class X, then variable subset S_1 would define s with three variables: $o_1.x_0$, $o_2.x_0$, and $o_3.x_0$. One possible drawback of this approach is that if a designer defines a variable subset that does not include any attributes for class Y and a parameterized action operates on objects of class Y, there is no way for the VFA to distinguish between the Q-values for each parameterized version of the action (because it has no reference to what the objects are). Therefore, if a variable subset excludes classes that are parameters for an action, then the VFA action weight for a state feature based on that variable subset will be the same for all possible parameters of the action. For instance, if feature i is based on subset S_1 and there are two possible objects (c and d) of class Y to which action a can be parameterized, then $\theta_{a(c)}(i) = \theta_{a(d)}(i)$, where $\theta_x(i)$ is the action weight VFA uses for state feature i and action x. As a result, a designer would have to specify additional variable subsets to S_1 that included class Y in order for the agent to select between different parameterizations of action a.

5.4 Results

Two continuous state OO-MDP domains were tested; one was tested with OO-MDP CMAC VFA and the other with OO-MDP SDM VFA. The domain used for testing the CMAC VFA is a similar domain to the remote switches domain used in Chapter 4. As in the remote switches domain, the goal of the agent is to reach some gated exit point of the domain. However, unlike the remote switches domain, the gate in this domain is opened by turning wheels located in the world, which all must be turned to raise the gate. After the agent leaves a wheel, the wheel will slowly spin back to its original position. Therefore,

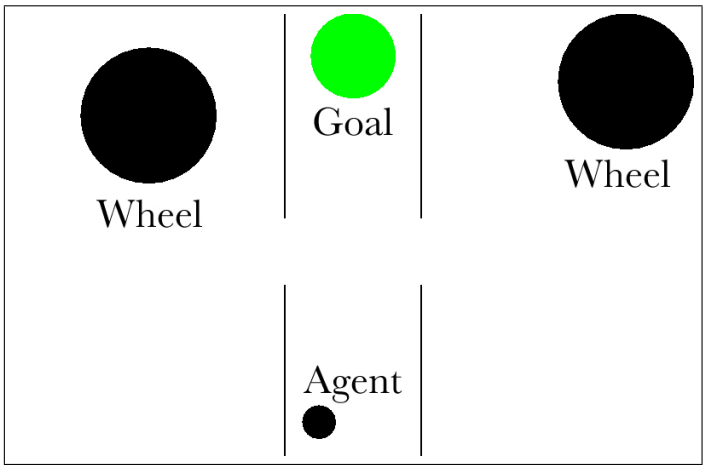


FIG. 5.1. A representation of the wheel domain. The agent needs to go to the green goal location, but before it can exit through it, the agent must first go each each wheel and turn them a number of times. With every step the wheels slowly turn back to the off position so the agent must travel to the goal region quickly enough to make it through. Wheel positions are randomly located in the top of each side room.

the agent must quickly visit each wheel and turn them enough to give the agent ample time to travel to the gate and pass through it. If the agent does not make it to the gate in time, it must go back and turn the wheels again (which means that mistakes are costly).

This domain is defined by an OO-MDP with three classes: agent, wheel, and goal. Each class has two attributes to describe the x and y position of the object. The wheel class has an additional "turned" attribute defining how much it has been turned, and also has two additional attributes that specify the wheel's x and position relative to the agent. In addition to these object classes, the world also has a number of walls that create rooms that the agent must navigate. The entire space spans a 20x13 region. The agent has four navigation actions (north, south, east, and west), which will stochastically move the agent in the indicated direction by a distance that is uniformly distributed between 0.8 and 1.0 units. The agent also has a fifth action for turning a wheel. When the agent is near a wheel

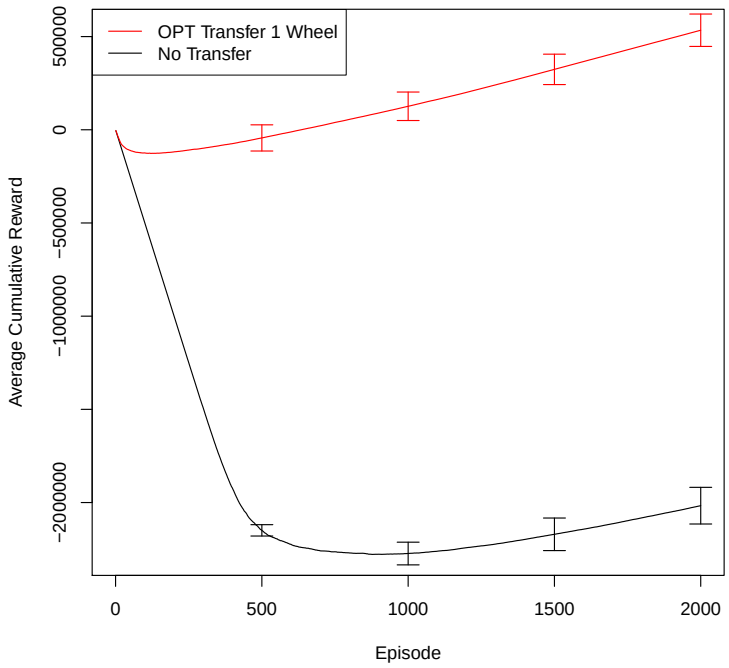
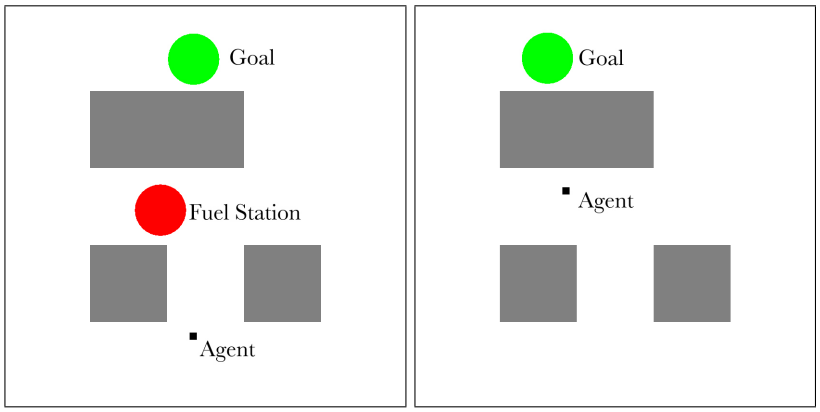


FIG. 5.2. Average cumulative reward over ten trials on the wheel domain with two wheels. The CMAC value function approximation effectively learns the task with transfer from a 1-wheel source task and without transfer; learning is significantly faster with transfer learning.

(within a distance of 2 units), the turn action will turn the wheel by five intervals, for a maximum of 20 intervals. If the agent is not near a wheel, then the turn action does nothing productive. After each timestep, every wheel that is further than 2 units from the agent turns backwards by 0.25 intervals.

Both traditional SARSA learning and transfer learning (using OPT) is tested on the wheel domain. The target task is a 2-wheel world with an example layout shown in Figure 5.1. The agent always starts at the position shown in Figure 5.1 and the goal is always in the same location. The location of the wheels, however, is randomly selected from the

top region of each side room. The source task is the same as the target task except that there is only one wheel present. The source task wheel can appear in the top region of either the left or right side room. When the agent completes the task, it receives a reward of 500; otherwise the agent receives a reward of -1. If an episode lasts longer than 5000 steps, it is prematurely terminated. To reduce the number of tiles stored, the CMAC was implemented with multiple variable subsets that were defined on a per-object class basis, as discussed in Section 5.3. Specifically, two variable subsets were created, each spanned by 10 overlapping tilings. The first variable subset was defined for two object classes: the agent class and the wheel class. For the agent class, both its x and y attributes were included in the representation and each attribute had a tile width of 3. For the wheel class, only the the turned attribute was included with a tile width of 5. The second variable subset only represented objects of the wheel class and was defined for the wheel's agent-relative x and y position wheel attributes and the turned wheel attribute. The tile width for all three of these attributes was set to 3. The purpose of the first variable subset is to represent features that would be useful in determining how to navigate the map and reach the goal. However, reaching the goal is contingent on the wheels being turned enough for the agent to reach it, which is why the first variable subset includes the "turned" attribute for wheel objects in addition to the agent's position. The second variable subset is meant to be useful in allowing the agent to navigate to the wheels when they are turned too little for the agent to be able to reach gate in time to exit. Q-value weights for each tile were defaulted to 0.0 (for a semi-optimistic initialization); the learning rate was set to 0.0025 (which is comparable to a 0.05 learning rate in a tabular paradigm); and the discount factor was set to 0.99. The agent also followed an ϵ-greedy policy with ϵ set to 0.05. The OPT parameters were set as follows: $\upsilon = 500$, $\iota = 3000$, and $\tau = 0.1$. Figure 5.2 shows the average cumulative reward using CMAC VFA on the two-wheel target task. Transfer learning using OPT from the 1-wheel source task significantly accelerates learning.



(a) Target Task (b) Source Task

FIG. 5.3. An example target task (a) and source task (b) episode in the fuel world. The grey blocks represent buildings; the green circle represents a goal location; the red circle represents a fuel station; and the small black square represents the agent. The agent must reach the goal location before it runs out of fuel and may pass through the fuel station to refuel if the goal is too far to reach with its current fuel.

An OO-MDP SDM was also tested on the wheel domain; however, results were significantly worse than using the CMAC. SDM performance on the domain also decreased if multiple variable subsets were used; as a result, a single high-dimensional state representation was required. This representation may explain SDM's relatively poor performance. In an alternate city navigation domain, however, the SDM was effective. In this domain, the agent must navigate a simplified city to reach a goal location. However, the agent has a limited amount of fuel and must reach the goal location before it runs out. If the agent is too far from the goal location to reach it without running out of fuel, it will have to pass through a fuel station, which automatically refills its fuel, and continue to the goal. The fuel domain is defined by an OO-MDP with five classes: agent, fuel tank, fuel station, goal, and building. All classes, except the fuel tank, are defined by two attributes specifying the object's x and y position in the world. The fuel tank class is defined by a single attribute

specifying how much fuel the agent has. The world is a 15x15 unit space and the agent has four navigation actions (north, south, east, and west), each moving the agent one unit in the specified direction. If an action would cause the agent to collide with a building object (each of which span a 3x3 rectangular region centered on the building's x and y position attributes), the action does not change the agent's position. The agent's fuel tank is initialized to 12 and moving causes the fuel level to drop by 1 unit. However, if the agent passes through a fuel station (which is a circular region centered on the station's x and y position attributes with a radius of 1), then the agent's fuel is refilled to the maximum. If the agent reaches the goal location, it receives a reward of 100. If the agent runs out of fuel before reaching the goal location, it receives a reward of -1000. All other actions cost a reward of -1.

Both traditional SARSA learning and transfer learning (using OPT) were tested on the fuel domain. Examples of a target task episode and source task episode are shown in Figure 5.3. The target task is a one-fuel-station task with four buildings. The four buildings were always in the same location for each episode; however, the location of the goal, fuel station, and were stochastically selected for each episode. To select locations for each of these objects, the goal location was first selected, which could be any real-valued x-y position that was above the two northern buildings. After the goal location was selected, the fuel station was selected to be any real-valued location below the the two northern buildings, above the two southern buildings, and within a manhattan distance less than 10 from the selected goal location. The agent location was then selected to be anywhere below the most northern edge of the southern buildings with a manhattan distance from the selected fuel station location greater than 4 and less than 10. Selecting locations in this way requires the agent to first navigate to the fuel station before navigating to the goal in nearly all scenarios. The source task is a zero-fuel-station task with four buildings. As with the target task, the building locations are always the same, but the agent and goal locations

are stochastically selected for each episode. The goal location is selected in the same way as in the target task and the agent location is selected in the same way as the fuel station in the target task (thereby ensuring that the agent is close enough to reach the goal before running out of fuel). The source task is trained for 40000 episodes and the target task is trained for 120000 episodes. The SDM object-wise distance thresholds for the agent class, fuel tank class, goal class, and fuel station class were set to 1, 2, 2, and 2, respectively. A minimum of eight activation states per query state was required. The default Q-values for newly visited states were set to the semi-pessimistic value of -90; the learning rate for the task was set to 0.1; and the discount factor was set to 0.99. The agent also followed an ϵ-greedy policy with ϵ set to 0.1. The OPT parameters were set as follows: $\upsilon = 200$, $\iota = 700$, and $\tau = 0.1$. Figure 5.4 shows the average cumulative reward using SDM VFA on the one-fuel station target task. It takes a significant amount of time to adequately learn the task with an SDM; however, learning is greatly accelerated when using OPT to transfer from the source task.

5.5 Conclusions

This chapter introduced methods to adapt any discretization or instance-based value function approximation (VFA) approach to work with object-oriented MDPs (OO-MDPs). OO-MDPs differ from typical factored representations because rather than using a single feature vector to describe the state, states are described by an unordered set of feature vectors. Since Chapter 4 introduced object-oriented OPT (OO-OPT), which facilitates transfer learning by leveraging OO-MDP state representations, providing VFA approaches for OO-MDPs allows OO-OPT to be used in continuous and very large state space domains that require VFA to be tractable. While these OO-MDP VFA approaches are general to any discretization or instance-based approach, two specific variants of these approaches were

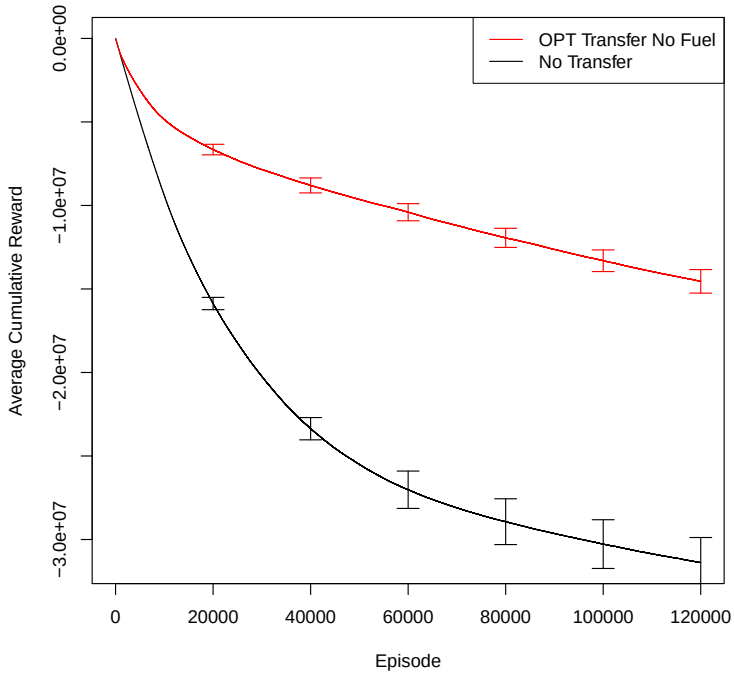


FIG. 5.4. Average cumulative reward over ten trials on the fuel domain with one fuel station using an SDM for learning. Learning the task takes the SDM a significant amount of time; however, it is greatly accelerated when using transfer learning from a source task without a fuel station.

examined in more detail: cerebellar model articulator controller (CMAC) VFA and sparse distribute memory (SMD) VFA. These approaches were then tested on continuous and large state space OO-MDP domains. CMACs were found to perform much better and benefited from using multiple variable subsets to reduce the dimensionality of each tiling and number of tiles stored, whereas SDMs only worked well when using a single higher-dimensional state representation. Results were also compared to using transfer with OPT, which significantly increases learning performance for both VFA approaches. In addition to the fact that SDMs did not perform as well as CMACs in these tests, SDMs also require significantly more computational time to find all activated memories (even when using a kd-tree to facilitate instance retrieval). Therefore, CMACs are recommended for continuous OO-MDP VFA. That said, there may be some domains that better benefit from the more continuous state representation SDMs provide compared to CMACs (due to features being continuous rather than binary). Since SDMs and CMACs are similar in concept, one promising approach for such domains would be to use a hybrid approach in which a CMAC is used to quickly retrieve tiles, but the feature associated with the tile is a real value indicating how centered the query state that lies in the tile is (where query states near the edge of the tile return a smaller value than query states centered in the tile). This would capture the more continuous representation of a SDM without the computational cost. It would, however, lose the SDMs dynamic memory capabilities; this might, however, be a reasonable trade off.

Chapter 6

LITERATURE REVIEW

Option-based Policy Transfer (OPT) is a form of transfer learning for reinforcement learning (RL). Although OPT is highly effective at transferring learned knowledge between tasks, there are many other approaches that utilize learning transfer in RL to accelerate learning in a novel task. Different approaches may transfer different kinds of knowledge, may have different restrictions on the permissible differences between tasks, or may require tasks to be learned in certain ways. In previous chapters, OPT was empirically compared to two existing policy transfer approaches: Policy Reuse in Q-learning (Fernández & Veloso 2005) and an option approach from Soni and Singh (2006). In this chapter, a review of many different forms of RL transfer learning is provided and compared conceptually to OPT. I categorize the previous approaches to transfer learning in seven groups: policy transfer (which is the form of transfer provided by OPT), Q-function transfer, subgoal transfer, feature transfer, model transfer, relational reinforcement learning, and learning task mappings. Policy transfer is when an agent reuses the policy from one or more source tasks in a target task. In Q-function transfer, the Q-functions from one or more source tasks inform the Q-value predictions in a target task. In subgoal transfer, the subgoal structure is extracted from one or more source tasks and used to inform the learning of a target task. Feature transfer is when abstract features from source tasks are extract to facilitate learn-

ing. Model transfer occurs when an agent learns and transfer the transition dynamics of a source task to a target task. Relational reinforcement learning is concerned with learning algorithms for relational domains that effectively learn reusable rules of behavior for tasks with different numbers of objects. Finally, learning task mappings is not strictly a form of transfer learning, but refers to methods for learning state and action space mappings between tasks that enable transfer learning algorithms to be used. At the start of each section is a brief overview of the transfer learning category and its general advantages and disadvantages; then a literature review for transfer algorithms in that category is provided.

6.1 Policy Transfer

Policy transfer, which is the form of knowledge transfer that OPT uses, allows the agent to apply the policy from one or more source tasks that it has previously solved. Policy transfer is most effective when the optimal policy of a target task requires few changes from the policy of a source task. The effectiveness of policy transfer decreases as the number of necessary changes from a source task policy to a target task policy increases; some approaches to policy transfer may perform especially poorly if the policies are not highly similar. OPT, however, was found to be effective as long there were some regions of the state space in which the policies of the source task and target task were similar. A requirement of policy transfer algorithms is to be able to map the state from the target task to the source task and the actions from the source task to the target task so that the policy of the source task can be queried from states in the target task and applied in the target task. OPT assumes that these mappings are given as explicit inputs to the algorithm; they may come from a designer or some other automated process, such as object-oriented OPT introduced in Chapter 4. Other approaches simply require the state and action spaces of tasks to be the same.

The most similar work to OPT is the option approach taken by Soni and Singh (2006) (SSO), against which OPT was empirically compared in previous chapters. Technically, SSO was designed strictly for learning between multiple possible mappings from the same source task rather than for policy transfer. However, because SSO can also be used for general-purpose policy transfer it is included in this section as a policy transfer algorithm. SSO takes a very similar approach to OPT: the policy for each source task (or possible mapping to a source task as it was originally designed) is represented with an option that the agent can use just as it would an actual action of the source task. The major differences between OPT and SSO lie in the specific learning algorithm used, the initiation conditions of options, and the termination conditions of the options. SSO can be initiated in any target state that maps to a valid source task state; it only terminates in terminal states of the source task (and in target task states that do not map to a valid source task state). If such initiation and termination conditions were used in OPT, performance would suffer greatly, due to an overreliance on the policies of the source tasks (as shown in the results of Appendix A). SSO does not exhibit this same problem, since the learning algorithm used by SSO only follows the policy of an option for at most one step. However, as shown in this dissertation, SSO's learning algorithm caused a severe underreliance on the source task policy and OPT greatly outperformed SSO in every experiment.

The next most similar approach to OPT is *policy reuse in Q-learning* (PRQL) (Fernández & Veloso 2005; 2006; Fernández, García, & Veloso 2010), which was also included as a source of empirical comparison for OPT in previous chapters. PRQL is provided with a library of previously solved source tasks. At the start of each episode for a given target task, PRQL probabilistically selects a source task for transfer (or none at all), where the probability of a source task being selected is based on the expected discounted reward when using the task for transfer. Computing the probability in this way biases the selection towards source tasks that are more useful for transfer. After a source task is se-

lected, policy transfer from the source task to the target task is performed using the *policy reuse* algorithm for the entire episode. At each step of the episode, policy reuse probabilistically selects to either follow the policy of the selected source task or to follow the currently learned target task policy. As the number of steps of the episode increases, the probability of following the selected source task's policy decreases. Originally, PRQL was only designed for transfer between tasks with identical state and action spaces; however, it was eventually shown to be effective for transfer between tasks with different state and action spaces if it was provided a mapping between state and action spaces (Fernández, García, & Veloso 2010). OPT has several significant advantages over PRQL. First, unlike PRQL, OPT can dynamically select between source task policies on a state-by-state basis rather than having to commit to a single source task for an entire episode. This allows OPT to combine multiple source task policies in useful ways. The other significant advantage for OPT is that because PRQL primarily follows a source task's policy in the beginning of an episode and then follows it less frequently in later steps of the episode, PRQL cannot significantly benefit from a source task's policy if it is only useful in later parts of the episode. This limitation could be especially problematic if a target task and source task are only similar near goal states, but not in initial states. OPT, however, does not suffer from this problem because OPT will follow a source task policy whenever it is deemed beneficial to do so. These theoretical advantages for OPT were found to be significant in practice: OPT outperformed PRQL in all of the experiments in previous chapters.

In addition to PRQL, Fernandez *et al.* (2005; 2006) also introduced a method called *policy library through policy reuse* (PLPR) for creating a library of source tasks. After a target task is solved, the similarity of the target task is compared to the similarity of each source task stored in the current library. If the most similar target task is dissimilar enough from the target task (by some threshold), then the target task is added to the library. The similarity of the target task and a source task is measured as the difference between the

expected discounted return when applying the target policy and the expected discounted return when using policy reuse with the given source task. In this work, it was shown how the OO-MDP state representation could be exploited to filter out source tasks that were less informative than other existing source tasks, but there was no formal method to construct the entire source task library. Since OPT learns Q-values for each TOP (where the Q-value is the expected discounted return of applying the given option in a given state and then following the agent's policy thereafter), it may be possible to adopt a similar approach as PLPR for generating the task library for OPT.

Policy transfer may also be especially useful for policy search algorithms (Stanley & Miikkulainen 2002; Taylor, Whiteson, & Stone 2006). Policy search algorithms differ from temporal difference methods like Q-learning and SARSA in that instead of learning the value function of the MDP, they directly learn a policy. Taylor *et al.* (2007) consider policy transfer between tasks that differ in the number of state variables and actions, and where the policies are represented by a neural network that is trained with a genetic algorithm (Stanley & Miikkulainen 2002). To provide transfer from the source task to the target task, the weights of the target task's neural network are initialized in accordance with the wights of the source task's neural network. Because the tasks differ in the number of state variables and actions, a mapping between state variables and actions is provided, which can then be used to determine how to map the weights from the source task's neural network to the target task's neural network. After initializing the target task's neural network weights with the learned source task's neural network weights, the genetic algorithm is used to further optimize the weights. Transfer of the weights was found to provide an improvement in the target tasks' initial performance and to accelerate the learning of the target task. One limitation of this approach to policy transfer is that transfer can only be provided from a single source task, rather than from multiple source tasks, as with OPT. Another possible disadvantage is that there is no clear way to transfer the policy between tasks

that use different policy controllers, such as transferring from a decision tree to a neural network. Since source and target tasks should be similar (otherwise transfer would not be considered), it is likely that the source and target tasks will always be learned with the same controller; but if not, OPT could theoretically use a source task that was learned with different controllers or VFA approaches.

In all of the approaches of policy transfer described above, the direct policy of one or more source tasks is used. However, another approach to policy transfer is to transfer the high-level behavior of a source task's policy rather than the specific behavior in each state of the source task. In *progressive RL* (Madden & Howley 2004), an agent starts in a simple task that is progressively made more difficult in terms of the size of the state space. The first source task is learned normally with Q-learning, after which its final policy is compressed into a set of rules based on high-level symbolic features that are present in all tasks. The next, more difficult, task is also learned with Q-learning, except when the agent is in a previously unvisited state, it chooses an action not based on the Q-value distribution, but based on the previously learned rules. Additionally, except for the action suggested by the rules, the Q-values of all other actions in the state are penalized so that when the state is visited again, the action that was preferred by the rule will again be favored. Using the transferred rules, initial performance on the target task is improved and learning on the target task is accelerated (compared to using no rules). If a constant set of symbolic features is available, progressive RL allows the agent to perform transfer learning across tasks with different state spaces; however, tasks must be discrete and this method cannot handle different action spaces or reward functions.

Torrey *et al.* (2005) present an approach in which after an agent solves a task, it extracts high-level rules, or advice, for favoring actions in certain circumstances. Advice is represented as rules to prefer one action over another when the Q-values for one action are much higher than those of the other actions. Unlike Progressive RL, these rules are

extracted from continuous state variables rather than requiring a set of high-level symbolic features to be available *a priori*. This advice is then transferred to a target task as constraints for a support vector regression algorithm (Maclin *et al.* 2005) used to approximate the value function of the target task. Source and target tasks can be defined with different state variables and actions if a mapping between the state variables and actions can be provided. In later work (Torrey *et al.* 2006), multiple sets of advice is extracted as first order logic rules that operate on objects in the world and are again provided as constraints for the support vector regression algorithm in the target task. Both of these approaches to policy transfer ultimately resulted in accelerated learning of the target task.

A similar approach of transferring high-level rules was used by Taylor *et al.* (2007). In this case, a set of rules was extracted from a learned source task policy using the RIPPER algorithm (Cohen 1995) and three different ways to utilize the rule set were tested: (1) increasing the Q-value initialization for the action suggested by the rule set; (2) adding a new action that followed the action suggested by the rules (the agent is also forced to take this added action for a fixed number episodes); and (3) adding an additional state feature, the value of which represents the index of the action suggested by the rule set for the current state (the agent is also forced to take the action suggested by the rule set for a fixed number of episodes). All cases resulted in better initial performance and faster learning; the approach that adds an additional action that executes the action preferred by the ruleset provided the most benefit. The added action approach has some similarities with OPT, except OPT uses the direct policy rather than high-level rules and allows for the source task policy to be followed for more than one step. OPT also does not require the agent to follow the source task for some number of episodes. No results for multi-source transfer were presented so it is unclear if this approach scales to multi-source transfer or what the best way to adapt to multi-source transfer would be.

Torrey *et al.* (2007) later used a more aggressive form of policy transfer using similar

concepts as in their earlier work. For a given source task, rules in the form of first order logic are again extracted, then sets of rules are combined together to form a finite state machine that they refer to as a *strategy*. A strategy effectively represents a set of rules that, when combined together, were successful in the source task. To transfer a strategy to a target task, the agent spends the first 100 episodes following the policy induced by the strategy and updating the Q-values of the MDP actions with the new experiences. After the initial 100 episodes, the agent ignores the strategies and learns as usual. By transferring strategies, the initial performance of the agent was improved and learning was accelerated.

One potential problem with this general approach of transferring high-level rules rather than the direct policy is that it is contingent on the ability for a source task policy to be largely representable with high-level rules; the more complex the policy, the less useful this approach will be.

6.2 Q-function Transfer

Q-function transfer is very similar in spirit to policy transfer. However, instead of transferring the direct policy from a source task to a target task, Q-function transfer transfers the learned Q-function from one or more source tasks. As a result, the Q-values in the target task reflect the Q-values of the source tasks. For a temporal difference learning agent that derives its policy from the Q-values, this initially has the same effect as policy transfer in that the agent will behave in the target task as it would in the source task. An advantage of using Q-function transfer over policy transfer is that there need not be any explicit mechanism designed for the agent to switch from the source task policies to the target task policy. Instead, this switch happens implicitly and gradually as the agent updates the Q-values from the initially transferred values. One downside to Q-function transfer is if an agent wants to transfer the Q-functions from multiple source tasks, there must be some safe

way of integrating each source task's Q-function that preserves the behavior of the different source tasks. Additionally, if the reward function of the source and and target tasks are very different, the transferred Q-values may not provide a good approximation of the optimal target Q-values, requiring the initialized values to be unlearned and preventing a significant learning acceleration. Policy transfer does not have this same limitation, because two very different reward functions can result in very similar policies or may at least have regions of the state space in which the policies are similar. As with policy transfer, Q-function transfer requires some way to map the states and actions between tasks so that the Q-values may be retrieved.

One of the earliest forms of transfer learning in reinforcement learning is provided by Selfridge *et al.* (1985). In this work, an actor-critic model with function approximation of the state *value* function (Barto, Sutton, & Anderson 1983) was used to learn the tasks (which technically makes this value function transfer, rather than Q-function transfer). Rather than learn the target task outright, initially the agent learned on a version of the target task with simpler transition dynamics. After learning the simplified version, the transition dynamics were modified to something more closely resembling the ultimate target task. By slowly adjusting the transition dynamics from a simplified model to the full model, learning was accelerated and initial performance was increased on the target task. The major disadvantage to this approach is that it requires the agent or a designer to have access to the transition dynamics of the target task. Additionally, the reward function, state space, and action space must be the identical in this approach.

Asada *et al.* (1994) provided another early version of transfer learning in reinforcement learning. As in Selfridge *et al.*'s approach, a complex task was solved by learning initially with an easy version of the task and progressively increasing the difficulty. However, in this case, the target task was made easier by starting the agent nearer to the goal state (rather than using an easier set of transition dynamics) so that the agent more quickly

received a goal reward signal. After learning a successful policy for initial states near the goal state, the initial state was moved further away. This process allowed the Q-values learned in the earlier task to be directly reused, since the states were identical. The primary problem with this approach is that it requires the designer to know what the goal state is and to be able to manually place the agent in states near the goal state. It also requires the state space, action space, transition dynamics, and reward function to be the same across the tasks.

Perkins and Precup (1999) considered transfer for a scenario in which different tasks were defined by different transition dynamics and the agent was not explicitly told which task was the current task. Two approaches to transfer were tested. In one approach, the agent maintains separate Q-values for each task, which are updated with respect to the agent's probabilistic belief for being in each task; these probabilities are updated for each task as the agent interacts with the world; by adjusting each value function with respect to the probabilistic belief of each task being the current task, the agent can generalize its knowledge across tasks. The alternate approach ignores task information entirely and instead learns a value function as if the task information did not exist, which effectively averages the Q-values across all tasks. Because all tasks were very similar, both approaches were effective at learning across all tasks, with the average value function performing the best. This approach, however, has the disadvantage that the state space, action space, and reward function must be identical across tasks.

A similar transfer problem, in which tasks differ only in the transition dynamics that are drawn from a defined distribution, was explored by Tanaka and Yamamura (2003). In this work, the agent is always aware of what the current task was (but not what the transition dynamics are). To provide transfer from previously solved tasks to a new target task, the average Q-value for each state-action pair across all previously solved tasks is used to initialize the Q-values of the new task. The standard deviation of the Q-values is

also maintained, which is used to bias the priority of simulated backups using prioritized sweeping (Moore & Atkeson 1993). This approach to transfer resulted in better initial performance; although there was a period in which performance dropped somewhat, the agent ultimately learned the optimal policy faster than without transfer. This approach to transfer learning suffers from requiring an identical state space, action space, and reward function across all tasks. Additionally, experiments were all performed in a discrete domain and it is unclear how to adapt this approach to continuous domains.

Konidaris and Barto's *autonomous shaping* (2006) is a method for Q-function transfer that is similar to Progressive RL (Madden & Howley 2004). In autonomous shaping, the state space between a set of tasks can differ, provided that a constant set of *agent features* are available across all tasks. Agent features differ from the symbolic features of Progressive RL in that they are continuous values. Additionally, rather than using agent features to extract policy rules, agent features are used to approximate an aggregate Q-function across all tasks (similar to how value function approximation is performed). When a new task begins, the Q-values of the task are initialized to the value approximated from the agent features, thereby improving initial performance and accelerating learning on the task. A limitation to this approach is that the tasks must be discrete (although the agent features may be continuous) and it is unclear how this approach would be adapted to VFA approaches. Additionally, the reward function and actions must be the same between tasks.

Taylor *et al.* (2005) provide perhaps the first example of using state variable and action space mappings between tasks; their method has subsequently been used in many other approaches to RL transfer learning (Torrey *et al.* 2005; 2006; Soni & Singh 2006; Taylor, Whiteson, & Stone 2007; Taylor & Stone 2007; Torrey *et al.* 2007; Fernández, García, & Veloso 2010), including OPT. Specifically, these mappings were meant to address the problem of providing transfer between tasks in continuous domains that were learned with value function approximation. Taylor *et al.* (2005) first showed how the

weights of CMAC VFA could be transferred between tasks—thereby transferring the continuous state Q-function—by leveraging the provided state variable and action mappings. Later, this approach was generalized to allow transfer between tasks learned with radial basis and neural network function approximation (Taylor, Stone, & Liu 2007). Transferring the Q-function in this way improved initial performance on the target task and accelerated learning. The most limiting factor of this approach is that transfer can only be provided by a single source task and the reward function should at least be very similar to ensure that the Q-values provide good initial estimates.

Banerjee and Stone (2007) consider Q-function transfer between different two-player, alternate move, complete information games. Since these games are expected to have very different state spaces and actions, the games are first mined for state features that represent the game tree structure at each state and are invariant to the specific game being played. After a specific game is learned (the source task), the Q-value as a function of these features is computed. When a new type of game is started (the target task), it initializes its Q-values in accordance with the Q-values estimated from the game features. If a target task state is represented by a game tree feature that was not present in the source task, then the Q-value is initialized to a default value. As a result, transferring is only provided between states of the source and target task that are similar. This approach is that is specific to two-player, alternate move, complete information games and cannot be readily applied to other problem spaces.

Lazaric (2008) provides transfer from multiple source tasks to a target task by transferring state-action-reward-state tuples from source tasks to the target task. The state space and transition dynamics between the tasks can be different and the transferred instances are weighted by the similarity of the tasks. Moreover, if certain regions of a source and target task are more similar than others, then such instances can be biased accordingly. A batch learning algorithm is then employed to learn from both target task instances and source

task instances. Limitations of this approach include being restricted to the batch learning algorithm and requiring the reward function to be the same.

6.3 Subgoal Transfer

Subgoal transfer involves transferring partial policies from source tasks to the target task. Specifically, after a source task is solved or explored in some way, it can be analyzed to extract subgoals that represent points of interest or necessary conditions that must be met to solve the source task. After the subgoals are extracted, learning in the target task proceeds by reasoning over the subgoals. If accomplishing some set of the subgoals is necessary for solving the target task, then this may accelerate the learning of the target task. An advantage to transferring subgoals is that the policies of the source and target task may be very different, but if they both share some common set of subgoals, then transfer can still be beneficial. A limitation of subgoal transfer with respect to other transfer algorithms is that if a source and target task are highly similar, then more of the learned knowledge should be shared than just subgoals. Additionally, it may be difficult to determine ahead of time what would constitute a relevant subgoal. For instance, there may be regions of the target task and source task in which the policies are similar, but if the extracted subgoals do not fully reflect those regions, then useful similarities between the tasks are lost. OPT, in contrast, does not exhibit this problem because it allows the agent learn which parts of the source task policies are useful and which are not.

Singh (1992) considered transfer between a set of *composite tasks*, each of which could be entirely solved by a common set of *elemental tasks*. Composite tasks all had the same state space, action space, and transition function and were differentiated by the reward function. Each elemental task had its own reward function and the reward function of the composite task was a linear combination of the elemental task reward functions and

a cost function for the composite task. Given that the composite task reward function was based on the reward function for each elemental task, knowledge about the value function for each elemental task could be used to seed the value function of new composite tasks (which makes this work an example of Q-function transfer as well as subgoal transfer). This transfer resulted in an acceleration in the learning of the the new target task. Later work by Drummond (2002) used similar ideas, except that the elemental tasks could be identified and combined together by leveraging visual features that were extracted from the domain. This variation resulted in an acceleration of learning of new tasks. Two limitations of these approaches are that they require the state space to be largely unchanged and they require composite tasks reward functions to be approximated from the reward function of elemental tasks. they also make the assumption that composite tasks can be solved entirely by stringing together elemental tasks.

A somewhat different form of transfer learning considered by Serhstov and Stone (2005) is transferring useful actions between tasks. This specifically tackles the issue of tasks with a very large number of actions that the agent would normally have to consider. However, if the optimal policy can be approximated with only a subset of the actions, then learning over the subset would decrease the problem's complexity. In such cases, transfer can be provided by transferring a relevant action subset from the source tasks to the target task; the target task can then be learned using only that action subset. To facilitate action transfer, it is assumed that the general outcomes of actions are known and that the class of states that is useful for predicting the outcomes of actions is known. Given the action outcome and state class information, subsets of actions for the outcome types can be extracted that are useful in the source tasks. Sherstov and Stone also introduce *random task perturbation*, which creates a set of synthetic source tasks from one actual source task. Extracting an action subset from the set of synthetic source tasks allows for a better action subset to be extracted than using only the actual source task. Transferring subsets of

actions in this way accelerates learning on the target task. This work was later extended by Leffler *et al.* (2007) to learn the transition dynamics and value function for a single task significantly faster by using the action outcome and state class information.

Possibly the most common form of subgoal transfer is *option identification*, a set of approaches that creates options from previously solved tasks or by analyzing the domain and then transfers the learned options to new tasks. In option identification, not only is the subgoal structure transferred, but the policy for achieving the subgoal is transferred as well. Creating an option requires finding initiation conditions (states in which the option can be executed) and termination conditions, which in this case represent subgoals of tasks. The approaches to option identification differ in how they identify the initiation conditions and subgoals. McGovern and Barto (2001a; 2001b) identity subgoals for options as states that are frequently visited in successful learning trajectories (where a trajectory is the executed sequence of states and actions in an episode) of a source task, but infrequently in unsuccessful trajectories. Initiation states are set to states that were visited in trajectories shortly before the subgoal was visited in the trajectory. Two limitations of this approach are that the state space must be discrete and the state space and action space between tasks must be identical.

The Q-cut algorithm (Menache, Mannor, & Shimkin 2002) identifies goal states as bottlenecks of the state space, which in turn are identified by creating a graph of the state space and using a max-flow/min-cut algorithm; the nodes that cut the state space graph serve as subgoals and the remaining states are initiation states. Using the cut sides of the graph, the algorithm is recursively applied to find even more subgoals. Transferring options that were learned in this way improved initial performance and accelerated learning on target tasks; however, this approach also has the limitations of requiring discrete state spaces and identical state and action spaces between tasks.

Simsek and Barto (2004) analyze visitation statistics of states in a source task to iden-

tify subgoals that are *relatively novel* states. Informally, a state s is considered relatively novel if for any given episode there is a set of states visited frequently before s is visited that are visited infrequently after s is visited. The initiation states for an identified relatively novel subgoal state are states that are commonly visited shortly before the subgoal. As with previous approaches, relative novelty requires a discrete state space and an identical state space and action space between tasks. Later, Simsek and Barto introduced *betweenness centrality* as a method for identifying options. Similar to Q-cut, betweenness centrality represents the state space and transition dynamics of a domain as a graph. For any state pair (s and v) a betweenness centrality score $c_s(v)$ represents the number of shortest paths from s to every other state t that must pass through v. If $c_s(v)$ has a large score for many choices of s, then this implies that v is an important state for traversing the domain; therefore, v is set as a subgoal for an option and the initiation states for subgoal v are the states s that provide v with a large score. Creating options in this way provides better initial performance on tasks and also accelerates learning. However, in addition to being limited to discrete state spaces and tasks with identical state and action spaces, this approach may require considerable computation time since the shortest path between states must be computed.

Unlike previously discussed option identification approaches, *skill chaining* (Konidaris & Barto 2009) identifies options in continuous domains with a goal state or goal state region (since the states are continuous). Skill chaining is a recursive process that starts by identifying state regions from which the agent has adequately learned how to reach the goal region of the source task (typically, this means states somewhat near the goal states). These identified states become the initiation conditions of an option whose subgoal is the goal region of the source task. The initiation state region of this option then becomes the next target goal region of another option. As before, an initiation state region is identified as a set of states from which the agent can adequately learn how to reach the target goal.

Recursively applying this process a number of times results in a sequence of options that chain from the initiation states of the source task to the goal region of the source task. If a new task specifies a new goal region, then these options can be reused when learning the solution, thereby accelerating learning and often resulting in better overall performance. A limitation of this approach is that it requires the state space and action space of the different tasks to be identical.

Rather than identifying options from source tasks that can then be reused in related tasks with the same state space, Konidaris and Barto (2007) tackle the problem of using *portable options*: options that can be used in tasks with different state spaces. To achieve this, the idea of agent features (Konidaris & Barto 2006)—special state features relative to the agent and present across all tasks—is applied. Specifically, rather than learning the policy of an option using the full state features, the policy is learned with respect to the agent features. Although the agent features may not be sufficient to fully describe the state space, it was found that they could still be effective enough to learn a reasonable policy and improve initial performance and to accelerate learning across tasks with different state spaces. In later work (Konidaris, Scheidwasser, & Barto 2012), an approach to finding agent features by identifying a subset of state variables that can always be mapped to state variables in every other task is suggested, although no empirical results were provided. A possible disadvantage of this approach is that some source tasks may share more state variables with a target task than others; limiting the agent features to the variable subset shared among all tasks may result in a less useful policy than transferring from the tasks that have many variables in common. For instance, in Chapter 4 of this dissertation, OPT's performance increased as the number of state variables shared between the source and target task increased.

Similar to the agent features work, Andre and Russell (2002) also explore transferring policies learned with a state abstraction to accelerate learning in new tasks with different

state spaces. Specifically, their work requires that partial policies for a set of tasks are defined in the form of a hierarchical decomposition of the task goals. They introduce a language called ALISP for specifying these partial policies. Given such a partial policy, safe state abstractions for each node of the hierarchy are determined. A state abstraction is a subset of the state variables. A state abstraction is *safe* if the optimal policy with the state abstraction will result in the same policy as when all state variables are included. (Determining whether the policies will be the same is facilitated by some user provided information in the ALISP language.) After safe abstractions are determined for each node, entire node policies may be explicitly transferred to new tasks with different state spaces that have a similar partial policy specified. Transferring the learned policy of nodes in a source task hierarchy to target tasks results in improved initial performance and accelerated learning. A disadvantage of this approach is that it requires a fair amount of user-provided information regarding the hierarchical partial policy.

Another way to transfer subgoals is to extract a hierarchical task goal decomposition from a source task and transfer the hierarchy to a target task *without* the policies for each node in the hierarchy. The target task can then be learned using a hierarchal learning algorithm such as MaxQ (Dietterich 1998), which learns the policies for each node simultaneously. The HI-MAT algorithm (Mehta *et al.* 2008) takes a successful trajectory of a source task (which could be learned with traditional techniques), along with a dynamic Bayesian network that specifies the transition dynamics of the task, and constructs a MaxQ hierarchy of subtasks that would result in a policy consistent with the trajectory. This hierarchy can then be directly transferred to a target task and used with the MaxQ algorithm to accelerate learning. The advantage of this approach is that the state spaces between the source and target tasks can be different, provided that they would still have the same task decomposition. A disadvantage to this approach is that if the target task and source task do not have the same task decomposition, but do have similar hierarchies, the agent cannot

learn to use the similar parts and ignore the rest. OPT, on the other hand. allows the agent to learn the regions of the state space in which a source task's policy is useful.

6.4 Feature Transfer

If abstract state features are provided to describe the state space, learning may be accelerated by adapting the Q-value predictions for many states (similar to value function approximation). Feature transfer is concerned with analyzing a source task for abstract state features that are useful in predicting the Q-functions for many states. If these abstract features can be used in a similar target task, then learning the target task can be accelerated. As with subgoals, one potential disadvantage with this approach alone is that if two tasks are highly similar, there may be even more information that can be shared between them. However, it may also be possible to use a combination of feature extraction and other transfer techniques to even more greatly accelerate learning.

Foster and Dyan (2002) consider using feature transfer between a set of tasks that all have the same state space structure. For instance, if the state space is spatially structured like a 2D gridworld, then this allows transfer to occur across tasks that are scaled-up or scaled-down versions of each other. Given a set of previously solved tasks, an expectation maximization algorithm is used to extract structurally connected regions of the state space called *fragmentations*. Given a new task, the state space is augmented with information of the fragmentation to which the state belongs and value functions are computed as linear combination of a value for the specific state and a value for the fragmentation. As a result, the values for each fragmentation generalize learning across states belonging to the same fragmentation, thereby accelerating learning on new tasks. Significant limitations of this approach are that tasks must be discrete, the action space between tasks must be the same, and the state space must be structurally the same between tasks.

Ferguson and Mahadevan (2006) extract features by first having the agent make random walks through the state space of a source task to form a graph repenting the states and transitions between states. Next, the resulting graph's Laplacian is computed and the k "smoothest" eigenvectors of the Laplacian are extracted and used to produce proto-value functions of the state space. From these features, the value function of source task is approximated and least-squares policy iteration (Lagoudakis & Parr 2003) is used to compute the policy. If a target task only differs from the source in terms of the reward function, then proto-value functions of the source tasks can be directly transferred over to the target task (without having to randomly walk the state space), since the proto-value functions only depend on the state space and transition dynamics. If a target task represents a scaled version of the source task state space, then the eigenvectors of the source task can be transformed to the appropriate eigenvectors of the target task using the Nyström method (Baker 1977), thereby allowing the source task's value function to be transferred. Transferring the transformed eigenvectors and value function resulted in a reasonable approximation of the target task policy that would reach the goal state with high probability. Disadvantages of this approach are that it requires a discrete state space and that the amount of knowledge transfer between tasks with different reward functions is limited.

The GATA algorithm (Walsh, Li, & Littman 2006) uses state abstractions from a set of source tasks that it can safely apply to a target task. Specifically, a method of extracting relevant features from a large set of features given a solved task is provided as an input to the algorithm. Given a set of previously solved source tasks, each is mined for its relevant features. A total set of relevant features is the union of the relevant features of all previously solved source tasks. When a new target task is presented, only the total set of relevant features is used to perform learning. Learning with the safe subset of features accelerates learning on the target task compared to learning with all features. Disadvantages to this approach are that a target task may not need to use all features that were relevant in at least

one of the previously solved source tasks and there there appears to be no way for it to identify more similar source tasks that could further restrict the set of features.

6.5 Model Transfer

Although reinforcement learning is often concerned with learning the policy for solving a task, another goal might be to learn the transition dynamics, or model, of the domain. Learning a model is often beneficial because if a model is known, then the agent can use planning to learn the task and accelerate the policy learning process beyond what is possible with just direct experience with the world. If a new task in a domain with the same model is provided, the benefits of having the model are even greater, since the agent does not have to learn the model from scratch. Several transfer learning algorithms have been developed to transfer model knowledge from a source task to a target task. Model learning is somewhat orthogonal to other transfer learning approaches and it could be possible to combine model transfer with other forms of transfer, such as policy transfer.

Atkeson and Santamaria (1997) demonstrate one of the simplest form of model transfer, in which the target and source task have identical transition models and only the reward function differs. In such a situation, all of the source task data related to learning the model is valid in the target task; therefore, the model can be immediately utilized for more efficient learning than learning without the model.

Sunmola and Wyatt (2006) consider model transfer between tasks that have the same state and action space, but different transition dynamics. Given a source task with a known model, the prior probabilities for the target task model parameters can be set to a useful value and the posterior probability of the transition dynamics of the target is updated with sampling using a Bayesian framework. Results indicated better initial performance using the transferred priors, but learning time was roughly equivalent. A similar approach is

taken by Wilson *et al.* (2007), who assumed tasks to be drawn from a fixed but unknown distribution of tasks that varies the reward function. The distribution is modeled with a hierarchical mixture model in which task reward functions are conditionally dependent on the class of the task and each class has some probability of being generated. The hierarchical structure is initially unknown and is estimated with data; that is, as new observations are found, new classes of tasks are created and added to the learned hierarchical structure. Leveraging the learned hierarchal structure, prior probabilities for the reward function of new tasks can be estimated and the specific reward function for a task is quickly adjusted using Bayesian techniques. Using informed priors improves initial performance and accelerates learning on new tasks compared to using completely uninformed priors. A limitation of this approach is that it assumes that the state space is the same for all tasks.

The TIMBREL algorithm (Taylor, Jong, & Stone 2008) considers model transfer between tasks with different continuous state variables and action spaces. To learn a target task's model, the instance-based model learning algorithm Fitted R-Max (Jong & Stone 2007) is used, in which instances are represented as state-action-reward-state tuples. When the amount of data near a target task state is too limited to estimate the model, instances from the source task are imported. Since the source task instances have different state variables and actions than the target task, they are first mapped to target task states and actions using a user-provided mapping between variables and states. Using transferred instances results in an increase in initial performance and accelerated learning. Fachantidis *et al.* (2012) later extended TIMBREL to COMBREL, which enabled the agent to select a different task mapping for different regions of the state space. Using multiple task mappings rather than one allows COMBREL to more rapidly accelerate learning on new tasks. This approach is limited to using a single source task, but it is possible that it could be extended to multiple tasks, since it allows for multiple mappings.

6.6 Relational Reinforcement Learning

Relational reinforcement learning (RRL) is a class of RL algorithms that are designed explicitly for domains whose states can be fully represented as relationships of objects. For instance, the blocks world is one of the most classic relational domains, since blocks are logically related to each other. Relational domains are similar to OO-MDPs (Diuk, Cohen, & Littman 2008), except that OO-MDPs also define objects by feature vectors. Rather than learning a policy for each possible relational state, RRL algorithms typically focus on learning a policy or value function as a function of object conditions. An advantage of this approach is that the policy is generalized for tasks with variable numbers of objects present and therefore has some similarity with transfer learning work. A natural disadvantage is that it cannot operate on non-relational domains, such as domains with continuous state variables.

Guestrin *et al.* (2003) propose a relational MDP (RMDP) learning algorithm that uses a representation similar to OO-MDPs, where objects belong to classes with associated attributes. An RMDP differs from OO-MDPs in that the features of objects are based on relations between objects, rather than on attributes of objects. Additionally, this approach assumes that value functions are linear combinations of individual reward functions for each object class. Using this assumption, a single aggregate value function can be learned and shared among environments with different objects and number of objects present. In particular, the domain assessed by Guestrin *et al.* was a real-time strategy game that consisted of many units, each with their own reward function.

The TG algorithm (Driessens, Ramon, & Blockeel 2001) learns a value function for relational domains using a regression tree in which nodes are split on the truth value of first-order logical predicates. When learning is performed across domains with different numbers of objects, the resulting tree can operate on tasks with variable numbers of ob-

jects, provided that the goal remains the same. The TGR algorithm (Ramon, Driessens, & Croonenborghs 2007) extends the TG algorithm to better enable transfer across tasks with different goals. Specifically, transfer between tasks is enabled by using a source task's learned regression tree as a starting point for the regression algorithm that is then modified as the agent interacts with the target task. Transfer results in better initial performance and an acceleration in learning compared to learning the task without transfer. A limitation of this approach is that transfer can only be provided from a single source task.

Crooenborghs *et al.* (2008) uses a combination of options and relational learning to accelerate learning in more complex tasks. Specifically, a set of parameterized subgoals for a set of tasks is provided and the agent begins by learning a policy for the subgoals using any RRL algorithm, such as TG. After the policy for the subtasks is learned, a set of rules is extracted for more compactly defining the policy and a parameterized option is created for the subgoal and resulting policy rule set. This parameterized option is then provided for use in more complex tasks. Because the option is learned with an RRL algorithm, it is robust to a varying number of objects in different target tasks and results in accelerated learning of the target tasks. A limitation of this approach is that it requires subgoals for the target task to be specified by a user.

6.7 Learning Task Mappings

In order for transfer to occur across domains with different state or action spaces, a designer must typically provide a mapping between the state and action spaces of source and target tasks, which naturally decreases the autonomy of the agent. Ideally, the agent should be able to determine which mappings to use on its own; a number of approaches to learning task mappings have been proposed. In this dissertation, objected-oriented OPT (OO-OPT) was introduced, which allows the agent to extract useful mappings from the structure of

OO-MDP states without any prior learning or significant computation required. However, states may not always be representable as OO-MDPs and OO-OPT cannot currently provided mappings between tasks that are represented by entirely different object classes. In such cases, other approaches to learning/generating task mappings may be useful to apply in conjunction with OPT.

Kuhlmann and Stone (2007) consider two-player fully observable deterministic games in which the transition dynamics and reward function are fully known. For each task, a *rule graph* is created that represents how the various state variables are affected by actions. To provide value function transfer from a source task to a target task, the variables from the source task are mapped to the target task variables by finding the mapping that would require the smallest number of changes from the source task graph to the target task graph.

Liu and Stone (2006) introduce a method for creating a mapping between a source task and target task given a *qualitative dynamic Bayesian network* (QDBN) for each task that describes the overall behavior of the transition dynamics. While a dynamic Bayesian network (DBN) describes how state variables probabilistically change after the application of actions, a QDBN specifies the *class* of variable changes. For instance, a "move north" action might cause the y-position state variable to be increased from the y-position variable in the current state; therefore, the the relationship would marked as belonging to the "increase" class. Given only these kinds of classes of relationships of the DBN, rather than actual probability distributions, Liu and Stone extract a state variable and action mapping that preserves the class of relationships as much as possible. Using such an extracted mapping with Q-function transfer (Taylor & Stone 2005) resulted in transfer performance comparable to a human-provided mapping.

The AtEase algorithm (Talvitie & Singh 2007) takes as input a set of candidate mappings from the target task to a source task. AtEase then tests the usefulness of each mapping in two-phases that are repeated multiple times. The first phase is an exploration phase, in

which the agent tests each mapping individually for a number of steps and measures the effectiveness of using that mapping. A mapping is tested by following the policy of the source task using the specified state mapping from the target task to the source task (similar to a TOP in OPT). After each mapping is tested, the agent goes through an exploitation phase in which the agent uses the best mapping for a number of steps. If the agent performs much worse than expected at any point before those number of steps are completed, then the mapping is removed from the set of possible mappings and the agent moves on to using the next best mapping. After the exploitation phase is complete, the agent returns to the exploration phase. These two phases alternate until either there is only one mapping left or a mapping has remained the best mapping for a sufficiently long period. Results indicated that the best mapping tended to be selected eventually. If a task is goal-oriented, this approach may not be effective, because while the mapped policy of a source task may bring the agent near the goal state, it may never actually reach the goal state. As a result, AtEase may determine all mappings in a goal-oriented task to be equally bad even if they are not. Additionally, because AtEase requires a set of candidate mappings to be taken as input, it may be better to simply use OPT, which can learn which mappings to use while learning the target task.

6.8 Conclusions

While many approaches to transfer learning in RL have been proposed in the past, OPT has a number of advantages. Compared to PRQL (Fernández & Veloso 2005) and the option approach to policy to transfer from Soni and Singh (2006), OPT was empirically found to be much more effective in this dissertation. Other policy transfer algorithms surveyed are restricted to single-source transfer or rely on the ability to extract high-level rules from the source tasks. Q-function transfer approaches typically require the reward func-

tion to be either identical or very similar across tasks. Option identification approaches extract partial policies from source tasks, which may not always transfer the useful parts of the source task policy or may not transfer enough of the source task policy if the source and target task are highly similar. Additionally, many option identification approaches are limited to use between tasks with the same state space. However, it may be possible to combine OPT with option-identification work to provide more transferable options that can be quickly adapted to new task state spaces. Other subgoal-based transfer approaches typically require the goals of the tasks to be very similar or to be fully solvable with the provided subgoals. Relational RL approaches are effective at transferring between tasks with variable numbers of objects, but are naturally restricted to relational domains. Feature transfer and model transfer represent fairly orthogonal types of transfer that in the future could be combined with OPT. Object-oriented OPT is more effective at generating mappings than other approaches to learning/generating task mappings, since it does not require any additional learning or processing time to generate the mappings. However, since object-oriented OPT requires tasks to be defined with OO-MDP states and to be in the same domain, other approaches to learning mappings may be useful for OPT.

Chapter 7

CONCLUSIONS AND FUTURE WORK

This dissertation focused on the problem of transfer learning in reinforcement learning. While reinforcement learning is effective at learning the policy for a specific task, learning must typically start over if the task is changed in any way. However, practical real-world tasks are often not identical every time they are solved; instead, there are often variations in the specific goal of the task or world configuration that would induce a new reward function or transition dynamics of the world compared to previously seen tasks. Transfer learning approaches, however, allow an agent to learn one specific task and reuse it in another variant of the task, which makes transfer learning critical if autonomous and capable agents are to be produced. This dissertation made three core contributions to transfer learning: option-based policy transfer (OPT); object-oriented OPT (OO-OPT), which identified sets of related tasks and automatically constructed numerous task mappings to use with OPT; and value function approximation approaches for the application of OO-OPT on object-oriented MDPs. These three contributions enable an agent to quickly and autonomously learn new tasks by reusing passing information. However, there are still many opportunities for extensions and improvements to these methods.

7.1 Future Work for OPT

OPT provides *policy transfer*, which allows an agent to follow the policy of a previously solved source task while solving a new task. In OPT, a source task's policy is encapsulated in an option (Sutton, Precup, & Singh 1999), which the agent can then select to use in the same way as a normal action in the domain. I call these source task options *transfer options* (TOPs). Because the agent treats the source task's policy as though it were an action, the agent learns in which states it is useful to follow the source task's policy, just as it learns in which states to take each of its possible actions. Treating a source task policy like an action has the additional benefit of naturally enabling transfer from multiple source tasks, because a separate TOP can be created for each source task. Multi-source transfer with OPT is especially effective because the state-by-state selection of TOPs allows the agent to learn how to combine source task policies to gain even better performance. OPT was empirically compared to two other multi-source policy transfer algorithms: a similar option-based approach introduced by Soni and Singh (2006) and *policy-reuse in Q-learning* (PRQL) (Fernández & Veloso 2005; 2006; Fernández, García, & Veloso 2010). In all experiments comparing OPT to these two approaches, OPT performed favorably and often performed significantly better. A similar transfer learning approach is Q-function transfer, for which a number of approaches were surveyed in Chapter 6, although these approaches often have stronger constraints than OPT, such as requiring the reward function structure to be very similar among tasks.

7.1.1 Automatic Parameter Estimation

One immediate area for future work with OPT is in automating the selection of OPT's parameters. Specifically, OPT requires a designer to set three parameters: a visitation threshold (v) that must be exceeded in the source task state to apply the TOP in the target

task, thereby ensuring that the source task policy is well learned in that state rather than being random; a visitation threshold (ι) in the target task state space that enables option interruption, which allows the agent to stop following a TOP if it is no longer useful; and a minimum termination probability (τ) that ensures that the agent does not become overreliant on a TOP (in which case the agent could follow the TOP not just when it is beneficial, but also detrimental). Optimal parameter selection is in general not critical to receive beneficial performance gains from OPT. For instance, τ was set to to 0.1 in all experiments except the puddles domain in Chapter 4, in which it was set to 0.5. The puddles domain was an exception because the task required fewer steps to reach the goal than in other domains; fewer steps to reach the goal means that a TOP should be terminated sooner to prevent an overreliance on its policy. Chapter 3 suggested some possible heuristics for a designer to choose parameter values. Specifically, since the *episodic update frequency* of states is measured for OPT, the υ and ι values could be selected based on the mean and standard deviation of these frequencies for the most recent learning episodes of the source task. The τ parameter could similarly be determined by examining the average length of episodes in the source task and setting τ to a value that resulted in only a user-specified fraction of the source task policy being followed. Specifically, if n is the average number of steps taken in the source task policy, and p is the fraction of the source task policy the designer would like the agent to follow for each application of the TOP, then τ could be set to $\tau = \frac{1}{pn}$. Although setting τ in this way requires the user to instead specify a different parameter (p), it is a more domain-independent parameter and there might thus be a more universal choice for p than there is for τ. Although none of these approaches were formally followed in the experiments in this work (rather, the average visitation rates and episode length of source task were examined and used informally), future work could directly implement them and compare them to the most optimal parameter choices. For instance, there might be a consistently good choice for p regardless of the domain and there

may be a relationship between the performance of OPT for values of p and the similarity of the source and target tasks. If such a relationship exists, knowledge of it would be beneficial for selecting parameter values.

7.1.2 Increasing Initial TOP Reliance

Another direction for future work is in increasing an agent's reliance on TOPs in the initial phases of learning. Specifically, learning was typically greatly accelerated when using OPT; however, it did not always increase initial performance. The reason for this is likely because an agent would be exploring TOPs and actions equally since they were initialized with the same initial Q-values. Since the option learning algorithm used would propagate back the discounted sum of reward accumulated when following it, the Q-value estimates for TOPs could also initially be lowered further than Q-values for other actions that only received one reward signal feedback. As a result, TOPs may not be relied on as much as they should until additional learning is performed and it is clear that the TOPs are effective at reaching goals. This phenomenon is perhaps best illustrated in the trading domain of Chapter 4. In that domain, transfer from the least similar task resulted in almost no improvement in learning performance. However, PRQL, which forces the agent to follow a source policy for the initial steps of the episode, did result in improved performance when transferring from the same source task. This result indicates that the source task policy was useful, but with OPT, the agent must not have learned that it was, perhaps because the benefits from the source task policy did not take the agent close enough to the goal for the reward to be propagated back to earlier states (which would be necessary for the agent to learn to use the TOP). A beneficial addition to OPT may be to promote TOP selection when the current state is unfamiliar. This approach would make intuitive sense because when a person does not know better, it is typically a good idea to rely on related experience. A possible way to promote TOP selection in such a circumstance would be to initialize the Q-values of

TOPs to be higher than those of other actions/options. If the agent were following a greedy policy or some stochastic approximation of a greedy policy, then in unfamiliar states, the TOPs would be more likely to be selected than the actions. Biasing Q-values in some way has also been used effectively in related transfer learning work (Madden & Howley 2004; Taylor & Stone 2007).

7.1.3 Multi-staged Task Transfer

OPT was primarily tested as a general means to reuse and transfer knowledge from multiple similar source tasks to a target task. Because OPT so greatly accelerates learning, it may be especially effective for learning a complex task through transfer of a series of progressively more complex tasks rather than learning the complex task from scratch. Some of the earliest transfer learning in reinforcement learning work was designed with this approach in mind (Selfridge, Sutton, & Barto 1985; Asada *et al.* 1994; Atkeson & Santamaria 1997). Although starting with a simple task and progressively making it more difficult does require a "teacher" of sorts to provide an agent a schedule of progressively more complex tasks, it is appealing compared to other more supervised learning approaches, because the teacher need not provide any information other than a sequence of complex tasks to solve. In earlier work, the differences between the tasks had to be small, because the agent behaved as if no change in tasks occurred. However, since OPT allows the agent to directly reason about previously solved source tasks and to accelerate learning even when the tasks are fairly dissimilar, it may be possible to provide an agent using OPT a series of tasks that are more dissimilar and still result in faster learning of the ultimate target task. If there is a direct path from a simple task to the complex task, then OPT could simply use the TOP for the most recently learned and complex task, rather than all previously solved versions of the task. A potential limitation may occur if the current target task (t_1) was learned with transfer from the source task s_1 and s_1 was learned with transfer

from source task s_2. In this case, if t_1 uses the policy of s_1 it may implicitly require the agent to also use the policy of s_2, since s_1 was learned with TOPs derived from s_2. However, it would be more computationally desirable if t_1 could use the policy of s_1 without s_1 having to use TOPs, which would require looking up the policy of s_2. However, because OPT learns the target policy (i.e., the policy derived solely from the task actions rather than actions and TOPs) simultaneously with the transfer policy (i.e., the policy derived from the task actions and TOPs), it may be possible to remove the TOPs from the policy of s_1 when learning on target task t_1 begins. However, it is possible that removing the TOPs from s_1 will result in worse performance if the target policy of s_1 has not been sufficiently learned (when compared to the transfer policy it learned). Therefore, to remove TOPs from the task without it negatively affecting the performance, a method to identify when the target policy is as good as the transfer policy may need to be developed. Alternatively, it might be possible to directly extract the policy of s_2 and encode it into s_1's policy wherever s_1 would select the TOP for s_2.

7.1.4 Planning with Policy Transfer

In this work, learning was always performed with model-free temporal difference algorithms such as Q-learning or SARSA. However, planning-based MDP algorithms might also benefit from policy transfer by speeding up the amount of time spent searching for a good policy. In particular, Monte Carlo tree rollout methods such as UCT (Kocsis & Szepesvári 2006) and PROST (Keller & Eyerich 2012), which is an adaptation of UCT to probabilistic planning environments, may benefit. These approaches take a generative model of the domain and simulate multiple episodes from the current state to estimate the state's Q-values, which can then be used to select an action in the real world. In each simulated rollout of an episode, an action must be selected; the performance of the algorithm is largely influenced by how actions are selected. Because options behave like actions, it

may be possible to adapt these algorithms to include the use of TOPs. Particular attention may be needed to address how to update the action selection for actions taken during the simulated execution of a TOP, since action selection in these algorithms is based on more than just the Q-values. However, it may be possible to simply update the various action statistics for actions taken by a TOP as if the planner chose them. As with the transfer experiments tested in this work, planning algorithms may benefit from transfer from tasks with different transition dynamics, reward functions, or state features.

7.1.5 Policy Transfer in Model Learning Paradigms

This dissertation used policy transfer to accelerate value function learning algorithms; that is, algorithms that learn the policy by estimating the Q-values directly from experience in the world. An alternative RL paradigm is *model learning*. Rather than learn Q-values directly from experience in the world, model-learning approaches learn the transition dynamics and reward function of the world from experience. The agent then learns a policy indirectly from the learned model rather than from the direct experience in the world. The advantage of this approach is that because the model is Markovian (transitions and rewards only depend on the previous state), less experience may be necessary to learn the world dynamics for any given state than the experience that is required to the learn the Q-values (since Q-values depend on the policy for all future states). If the model can be learned more quickly (in terms of world experience; more computation is typically required for model learning) than the Q-values, then the agent can use simulated experience from the model to derive the policy. Although model transfer approaches (see Section 6.5) are useful for model learning approaches, model learning approaches may also benefit from policy transfer approaches like OPT.

R-max (Brafman & Tennenholtz 2003) is one of the most common approaches for deriving learning policies for the model learning paradigm and numerous implementations of

R-max for different kinds of environments exist in literature. For example, Jong and Stone (2007) provide an implementation of R-max for environments that require a continuous state space model; Factored R-max (Guestrin, Patrascu, & Schuurmans 2002) implements model learning for environments that have a provided dynamic Bayesian network (DBN) structure that can be used to model the environment dynamics; SLF-Rmax (Strehl, Diuk, & Littman 2007) extends Factored R-max to also learn the structure of the DBN; and the OO-MDP representation (Diuk, Cohen, & Littman 2008) was originally designed to facilitate R-max model learning for complex deterministic worlds. In R-max, the agent starts with a fictitious model of the world, in which every state-action pair returns the maximum reward and transitions to a fictitious state that can only transition to itself and also always returns the maximum reward when doing so. Once the transition dynamics and reward function for a given state-action pair can be accurately estimated (which is usually determined by the number of times that state-action pair was taken in the real world), the fictitious transition is replaced with the estimated transition. The policy that the agent follows in the real world is always based on the optimal policy for the fictitious model (which is typically determined with planning algorithms like value iteration). The effect of deriving the optimal policy from the fictitious model is that the agent will be biased toward exploring state-action pairs in the real world that it has not yet sufficiently explored and which have the potential to be more optimal than the state-action pairs it has currently explored.

There are two possible ways in which OPT could facilitate model learning paradigms like R-max. One possible benefit is that OPT may allow for more computationally efficient planning when deriving the policy from the fictitious world model. In the original options work (Sutton, Precup, & Singh 1999), options were shown to significantly reduce the number of iterations required of value iteration to converge to the optimal policy; therefore, TOPs may also provide computational gains. If OPT can also be adapted to accelerate other planning algorithms (as discussed in the previous section), it could similarly pro-

vide computational gains for R-max algorithms that use different planners for the fictitious model. Another way in which OPT may be beneficial for model learning paradigms is to have the learning policy favor the selection of TOPs when neither the transitions for the primitive actions nor TOPs are well modeled in a state. This bias would have the effect of the agent reusing knowledge when it is unsure of the consequences. If TOPs were more difficult to model than the primitive actions, then an additional modification could be made in which the agent could only apply a TOP when the model for any of the primitive actions was inaccurate, which would only allow the agent to use transfer knowledge when it did not know any better.

7.1.6 Robotics Applications

Using reinforcement learning to learn behavior for a robot in the real world is often difficult because reinforcement learning can take many episodes to converge to a good policy. In virtual environments in which actions can be taken in fractions of a second, this typically is not a problem because many episodes can be run quickly. However, in the real world, the actions of a robot can take orders of magnitude longer to complete. As a result, reinforcement learning may take longer than is desirable. Learning an accurate model of the real world may also be difficult; therefore, the policy learned with a model may not be ideal for behavior in the real world. However, if a simplified model could be learned or provided, then the agent could quickly learn a policy for the simulated world and then use OPT to transfer the learned policy to the real world to greatly accelerate learning with respect to the actual physical dynamics.

7.2 Future Work for Object-Oriented OPT

A requirement for OPT is that a mapping between states and actions of tasks is provided. If the state and action spaces of a source and target task are identical, then no mapping is required, since each state and action maps to itself. However, if the state or action spaces are different, then the mappings are required in order for the policy of a source task to be followed in the target task. Object-oriented OPT enables multiple possible mappings between a source and target task to be generated automatically from domains defined with object-oriented MDPs (OO-MDPs) (Diuk, Cohen, & Littman 2008). OO-MDPs extend MDP definitions to include a set of object classes, a set of attributes for object classes, and a set of propositional functions defining relationships between objects. A state of an OO-MDP is an unordered set of object instances, which are instantiations of specific object classes. That is, an object instance provides a set of values for the attributes associated with its class. In effect, an OO-MDP state is a set of feature vectors. By leveraging this class information of objects in states of tasks, a set of possible mappings between objects of the same class is provided. Rather than picking a single mapping, OO-OPT uses all of these mappings by creating a TOP for each mapping. The results presented in Chapter 4 demonstrated that allowing the agent to use multiple mappings rather than just one mapping increased performance. In effect, the agent learned the circumstances in which each mapping was most useful. In addition to providing mappings between source and target tasks, the OO-MDP representation also provided an automatic way of defining sets of related tasks by using the propositional functions to define abstract goal requirements. Moreover, the *object signature* of tasks (i.e., the number of object instances of each class) allows the agent to filter out redundant source tasks that provide no more additional information than other source tasks, thereby decreasing the number of source tasks used for transfer. Although OO-MDPs facilitate transfer learning with OPT, the fact that states are

represented as an unordered set of feature vectors makes applying typical value function approximation (VFA) techniques to them problematic, because typical VFA approaches expect a single feature vector. To address this, I provided methods to adapt any discretization or instance-based VFA approach to OO-MDP states.

7.2.1 Improving the Task Mapping Set

Although providing OPT with all valid OO-MDP object-wise mappings between two tasks was found to be beneficial in the experiments tested in this work, as the number of objects in a task increases, providing all of them may become intractable and oversaturate the action space with TOPs. In principle, however, it may not be necessary to provide all mappings so long as a subset approximating all mappings is provided. For instance, rather than provide a mapping for every possible combination of objects, a set of mappings that ensures that all objects in a target task are represented in at least one mapping may be sufficient. Alternatively, there may be a way to analyze the values of each object and determine a small subset of mappings that spans the most important objects. For instance, if two objects are very similar, it may not be important to represent both them in mappings as much as objects that are less redundant to the problem space. It may also be possible for the agent to identify objects that are more important to the policy for the current state than others (perhaps by analyzing the policy of the source task) and limit the set of mappings to the most important objects.

Additionally, OO-OPT only provides mappings between a target task and source task when the source task object signature is *dominated* by the target task object signature (which means that the target task has at least as many object instances of each object class as the source task). Since transfer to a source task is typically provided from a source task that is less or equally complex as the target task, this constraint is not unreasonable. However, if a a novel target task is reasonably complex, but less so than a source task, it

would still be beneficial to transfer from the source task. In order to provide transfer in this situation, it would require that "synthetic" objects were created for every object that a target task lacked, relative to the source task. A possible way to provide this would be to have the agent create objects and values for the most frequently observed object values in episodes. Once these were provided, the policy of the more complex target task could be queried.

7.2.2 Using Higher-level State Representations

OO-OPT generates task mappings for transfer between tasks by identifying mapping between objects of the same class. However, while two object classes may be different, they may be similar enough to make mappings between them useful for transfer. Proving mappings between different object classes could be especially useful to provide transfer from a source task that is not dominated by the target task. For instance, if an OO-MDP domain defines three object classes (X, Y, and Z), the *object signature* (the number of instantiated objects of each class) of source task s is $\langle X(1), Y(2)\rangle$, and the object signature of target task t is $\langle X(1), Z(2)\rangle$, then the object signature of t does not dominate the object signature of s; therefore, OO-OPT would not provide transfer from s to t. However, if class Y is similar class Z, then it might be beneficial to map objects of class Z in task t to objects of class Y in task s, thereby enabling transfer from task s to task t. If the OO-MDP representation was extended to support *object inheritance*, then the agent could identify when object classes were similar by examining the class hierarchy.

Object inheritance for an OO-MDP would borrow concepts from object-oriented programming languages, in which a class can be a subclass of a superclass and a subclass inherits all of the properties of its superclass. For an OO-MDP, classes define a set of attributes; therefore, if object class B is a subclass of object class A, then object class B's attribute set would include all the attributes in A's attribute set, as well as any additional

attributes specific to class B. Formally, $\mathcal{T}_B \supseteq \mathcal{T}_A$, where $\mathcal{T}_B$ is the attribute set for class B and $\mathcal{T}_A$ is the attribute set of class A. If A is a superclass of B, then it is an ancestor of B and any ancestor of A is an ancestor of B. Inversely, if A is an ancestor of B, then B is a descendent of A. Since multiple classes may be subclasses of the same superclass, a class hierarchy may be formed for some set of classes of the OO-MDP domain. Given a class hierarchy, two classes can be determined to be similar if they share a common ancestor. If two classes are similar, then a mapping between each can provided. To map the values of object instance (o) of class c_1 into the attributes of a different object class (c_2), there are two cases to consider: (1) when c_2 *is* an ancestor of c_1, (rather than merely sharing a common ancestor with c_1) and (2) when c_2 is a descendent of c_1 or neither c_1 nor c_2 is the ancestor of the other and merely share a common ancestor different from either of them. In the first case, the attributes of c_2 is a subset of the attributes in c_1; therefore, mapping the object instance values of o into class c_2 only requires removing any values for attributes in c_1 that are not attributes for c_2. In the second case, c_2 may require values for attributes that are not defined for c_1. Therefore, while the values for attributes in common between c_1 and c_2 can be kept the same, a method to set values for the additional attributes in c_2 would also have to be defined. One possible way to set the values for attributes not defined for c_1 may be to set the values to the most common attribute values observed by the agent. Using that method or a similar method to set the values for additional attributes, OO-OPT would be able to provide transfer between a wider range of tasks than it is currently capable of providing.

7.2.3 Using Analogies to Identify Cross-domain Mappings

Work in analogical reasoning seeks to find relationships between different domains and assign similarity scores between domains that are based on those discovered relationships. Because analogical reasoning identifies relationships between domains, it is highly

related to transfer learning and is often used in conjunction with transfer learning. For instance, Klenk *et al.* (2007) and Ouyang *et al.* (2006) use the *Structure Mapping Engine* (Falkenhainer, Forbus, & Gentner 1989) to identify analogies between physics problems that allows previous solutions to assist in finding the solution to a new, but related, problem; Melis *et al.* (1999) use analogies to guide the search of a theorem prover; and Veloso *et al.* (1993) and Könik *et al.* (2009) use analogies between new and previously solved problems to assist in planning. In all of these approaches, the identification of analogies is facilitated by a high-level relational or functional representation of objects. Analogies are found by mapping objects, terms, and relations between domains that maintain the overall structure of the domains/problems. For transfer learning in RL, analogical reasoning could be especially useful for identifying task mappings. In fact, since OO-OPT identifies mappings between objects of the same class, OO-OPT can be viewed as performing a weak form of analogy. What OO-OPT lacks in its identification of mappings is consideration of the structural relationships between objects. However, because the OO-MDP representation provides support for propositional functions that describe the relationships between objects, the most relevant mappings could potentially be identified by assessing the logical structure of states like other work in analogies. If the OO-MDP representation was extended to include more expressive logical structures between the objects, it might also facilitate the identification of mappings between objects of different classes or between objects of different classes defined for different domains. Extended logical representations might also be paired with the object inheritance ideas discussed in the previous section for even better automatic mapping identification.

7.2.4 Learning with a Large Task Library

OO-OPT provides a method to identify certain kinds of source task redundancies that allows the agent to ignore redundant source tasks when learning a new target task. For

example, consider a target task (t_1) that dominates two source tasks, s_1 and s_2. A naive approach to implementing transfer learning for t_1 would be to learn t_1 with multi-source transfer from both s_1 and s_2. However, if source task s_1 dominates source task s_2 and both source tasks have the same *task objects* (objects whose values define different tasks), then s_2 probably does not provide any more useful information than s_1, in which case the agent should always prefer to use s_1. Therefore, in this situation, OO-OPT will automatically only use transfer from s_1. However, there are likely even more sophisticated ways to manage source tasks that would result in better source task utilization. For instance, if s_1 did not learn a policy for the entire state space, it is possible that s_2 has a learned policy for states that are missing from s_1's policy. Given such a possibility, it may be better to allow t_1 to transfer from s_2 when the initiation conditions of the TOP for s_1 are not satisfied (that is, s_1 has not adequately visited the current state to produce a valid policy).

Another scenario for transfer is when two source tasks have different task objects. In this case, they each could potentially provide different useful information to a target task and therefore it may be beneficial to use both for transfer. However, if the agent has learned many different source tasks, then using all source tasks for transfer could oversaturate learning on the target task, thus decreasing learning performance. Therefore, it may be beneficial to develop a method for limiting the number of source tasks. If tasks differ in terms of their task objects, one way to limit the number of source tasks is to measure a similarity between the task object values and filter out source tasks that have very similar task values. An alternative approach might be to implement something similar to *policy library through policy reuse* (PLPR) (Fernández & Veloso 2005; 2006). In PLPR, the similarity of a target task to one of its source tasks is estimated by comparing the expected discounted return of using a source task for transfer learning (with the policy reuse algorithm) for an entire episode with the expected discounted return of using the target task policy. If the expected discounted returns are similar, then the tasks

are considered to be similar. If a target task is dissimilar from each of the used source tasks, then it is added to the task library; otherwise, the target task is not added to the library and as a result is never used as a source task for transfer. One of the advantages of OPT over PRQL (which PLPR uses) is that OPT enables policy transfer on a state-by-state basis, rather than committing to a single source task for a whole episode; therefore, PLPR could not be directly implemented with OPT since OPT does not measure the expected discounted return of using a source task for a whole episode. However, because the Q-values of TOPs are learned, it might be possible to sample a set of states from the task and compare the Q-values values of the TOP with the Q-values of the maximum action in each of those sampled states. If the Q-values of a TOP and the maximum action are similar in many states, this may indicate that the tasks are very similar, in which case a similar approach to PLPR could be implemented.

7.2.5 Portable Subgoal Options

One of the ways in which subgoal options differ from TOPs is that a subgoal option requires the agent to identify the common parts of a source task policy a priori, whereas OPT allows the agent to learn which parts of the source task's policy are similar and useful. Even when good subgoals and local policies can be extracted a priori and reused in the future, subgoal options may not be be useable in many different tasks if the tasks if the tasks are too dissimilar from the task in which the subgoal options were created. For instance, many of the option identification techniques surveyed in Chapter 6 require the state space, action space, and transition dynamics to be identical across tasks. Ideally, however, subgoal options should be portable to tasks that differ in these ways. If a subgoal option used a transfer learning algorithm such as OPT, however, then it could be quickly adapted to the new task. Moreover, OO-OPT could be especially useful for portable subgoal options, because general option goal conditions could be defined with the OO-MDP propositional

functions.

7.2.6 Integration with Heuristic Search Planners

Subgoal options that use OO-OPT may also provide a useful bridge between domains with low-level non-relational state representations (e.g., spatial domains) and high-level relational domains that are best solved with relational heuristic search planning (RHSP) algorithms, such as FF (Hoffmann & Nebel 2001) or LAMA (Richter & Westphal 2010). In RHSP, tasks are typically represented with a STRIPS-like formalism (Fikes & Nilsson 1971): states are represented as logical propositional statements about objects of the world, actions make propositional statements true or false, and goals are defined as propositional statements that must be made true. Leveraging this relational representation, domain-independent heuristic functions, such as the FF heuristic (Hoffmann & Nebel 2001), may be computed, and used with search algorithms to efficiently find a plan that achieves the goal. With an OO-MDP, a domain could be fundamentally non-relational, but have high-level goals specified in terms of the OO-MDP propositional functions. Using OO-OPT specified subgoal options to serve as a relational action set, RHSP algorithms could be used to find a plan in terms of these subgoal options that are subsequently executed in the true non-relational state space; because the subgoal options use OO-OPT, they would be robust to variations in the task, such as the number of objects present.

7.2.7 Improving VFA and Integrating with Relational Features

In this work, methods for adapting discretization and instance-based VFA approaches to domains represented with OO-MDPs was introduced. CMACs (a form of discretization-based VFA) was found to perform better than SDMs (a form of instance-based VFA) and was computationally more efficient. However, there may be some domains that benefit from SDM's more continuous and smooth state feature representation (as opposed to CMACs

which use create binary features). A possible alternative is to adapt CMACs to use real-valued features that express how centered a query state is in a tile, thereby producing a more continuous representation while maintaining CMAC's computational benefits. In addition to this representational change, CMACs might be adaptable to exploit more information from the OO-MDP. Specifically, OO-MDPs not only provide state information in terms of the feature vectors of each object instance, but also in terms of the propositional functions. Adding features to a CMAC that represented the truth evaluation of possible propositional functions could augment the VFA with important high-level features that are useful for learning the policy.

If OO-MDP subgoal options were created, this would also provide an opportunity to perform high-level VFA learning with relational RL methods such as TGR (Ramon, Driessens, & Croonenborghs 2007). Such algorithms may also benefit from integration with RHSP algorithms. That is plans from RHSP algorithms could be used to provide example exception traces for the relations RL algorithm, thereby accelerating the learning.

7.3 Concluding Remarks

This work introduced options-based policy transfer (OPT), which is extremely effective at transferring knowledge learned in previously solved source tasks to new tasks and which compares very favorably to related reinforcement learning transfer learning algorithms. Object-oriented OPT (OO-OPT) was also introduced, which facilitates OPT by identifying sets of related of tasks, filtering out redundant source tasks, and automatically producing multiple task mappings, yielding even greater performance benefits then using a single mapping. Additionally, methods to adapt any discretization or instance-based value function approximation techniques to object-oriented MDPs were provided so that the advantages provided by OO-OPT could be exploited in continuous or large state space

domains. Although there are still opportunities for improvement, these approaches provide a solid foundation for producing autonomous agents that can quickly adapt to variations in the world.

Appendix A

ADDITIONAL RESULTS

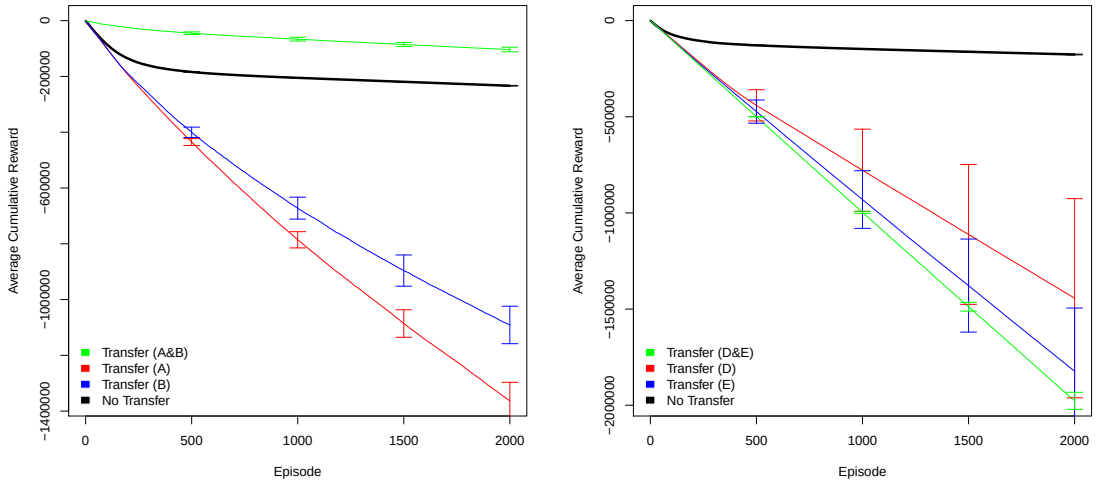
This appendix contains additional results for various experiments that were not presented in the relevant chapters.

A.1 Office Domain Results

This section presents additional results in the office domain that was explored in Section 3.4.1, including (1) OPT performance when less restrictive initiation and termination constraints for TOPs are used (that is, TOPs can be initiated in any state and only terminate when they reach a state that maps to a terminal state in the source task); (2) performance on target task F when a constant value of ϵ is used; and (3) performance of PRQL in different transfer and learning scenarios that are more similar to the original scenarios on which it was tested.

A.1.1 Less restrictive TOP initiation and termination conditions

To demonstrate that the initiation and termination conditions OPT imposes on TOPs (see Section 3.2.2) are necessary to prevent an overreliance on TOPs, this subsection presents results of OPT when TOPs can be initiated everywhere and only terminate in states that map to the termination states in the source task. Results for target tasks C and F



(a) Target C, source A & B

(b) Target F, source D & E

FIG. A.1. Average cumulative reward over ten trials for the office domain when TOPs can be initiated anywhere and only terminate in states that map to terminal states of the source task. Subfigure (a) shows results on target task C with source tasks A and B. Single-source transfer scenarios perform worse than the baseline without learning. Multi-source transfer is better than the baseline, but slower than when OPT uses its normal initiation and termination conditions. Subfigure (b) shows results on target F with source tasks D and E, where all transfer variants perform much worse than learning without transfer.

are shown in Figure A.1. As expected, the lack of constraints does result in an overreliance that produces much worse performance. For target C, single source transfer performs much worse than learning without transfer and the policy after 2000 episodes only modestly improves from the random initial policy. For multi-source transfer on C, the agent is able to learn a policy and it performs much better than learning without transfer. This result is likely seen because the agent can reach the target goal location by first going to one source task's goal and then the others. However, although not shown here, multi-source transfer is still slower than multi-source transfer when the normal OPT constraints on initial conditions and termination conditions are used, and the difference is statistically significant. For target F, single source and multi-source variants all perform much worse than learning without transfer and the policy has improved minimally after 2000 episodes. This experiment was also run on the malicious transfer scenario (target G, source A), but results are not shown, because results were so poor that the agent never reached the goal in any episode.

These results support the claim that the initiation and termination conditions OPT used are necessary to to keep a balanced reliance on TOPs, because if they are not used, it may result in worse performance than learning without transfer, let alone the transfer performance when they are used.

A.1.2 Target F with a constant ϵ value

This subsection provides the additional result of performing transfer learning on target F (see Figure 3.4 (a) for each task's goal location in the domain) with a non-deceasing value of ϵ. These results are shown in Figure A.2. Unlike in the previous results for this task (discussed in Section 3.4), multi-source transfer from sources D and E does not perform better (or worse) in a statistically significant way than single-source transfer from D (but does perform better than single-source transfer from E). The difference in the benefit of multi-source transfer with a non-decreasing ϵ versus a decreasing ϵ may be explained by

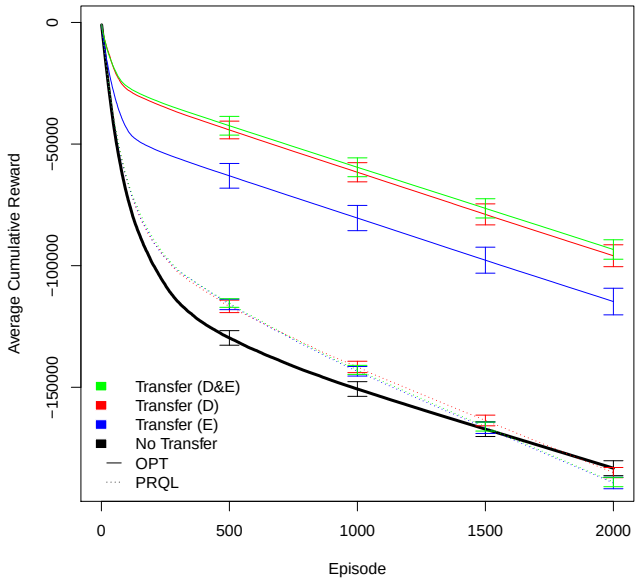


FIG. A.2. Average cumulative reward over ten trials for the office domain on target task F. This is the same experiment as the transfer experiment on F in Section 3.4, except that a constant non-decreasing ϵ value is used for the learning policy. In this case, multi-source transfer learning still performs the best, but performs no better or worse than single-source transfer from source task D.

the difficulty in learning how to exit from the rooms that contain the goals of source tasks D and E. With a constant ϵ, transfer performance in general may be better because when the agent terminates in the source task's room, it may more easily happen to then move out of the room. With a decreasing ϵ, this might not happen as easily, since the agent would behave more greedily, requiring the agent to more completely learn the policy for the new task before it can exit the room. If this is the case, it would explain why multi-source transfer is beneficial with a decreasing ϵ: with multi-source transfer, there is an additional source task that takes the agent out of the room without having to more completely learn the target policy. While this scenario shows less of an advantage for multi-source transfer, the fact that multi-source transfer performs no worse than the best single-source transfer (from D) indicates that OPT can learn which sources are best without a significant cost, thereby allowing a designer to provide multiple tasks to the agent without specifying how to use them.

A.1.3 Performance analysis of PRQL

The next set of additional results is meant to ascertain (1) the reasons why PRQL performs especially badly in the office navigation domain, (2) whether there are circumstances in which PRQL fares better than the baseline of learning without transfer, and (3) how OPT performs in these scenarios. One primary difference between the experiments in Section 3.4 and the original PRQL work (Fernández & Veloso 2005) is that in Section 3.4, there were no source tasks that were highly similar to the target task. In the first additional experiment, the situation with highly similar source tasks is explored by testing transfer from task B to task H; the locations of both of these task goal locations are shown in Figure 3.4 (a). These tasks have goal locations that only differ by a manhattan distance of six, both of which are located in the same room of the domain. In addition to setting the PRQL parameter to $\upsilon = 0.95$, as suggested in the original work, results are also shown

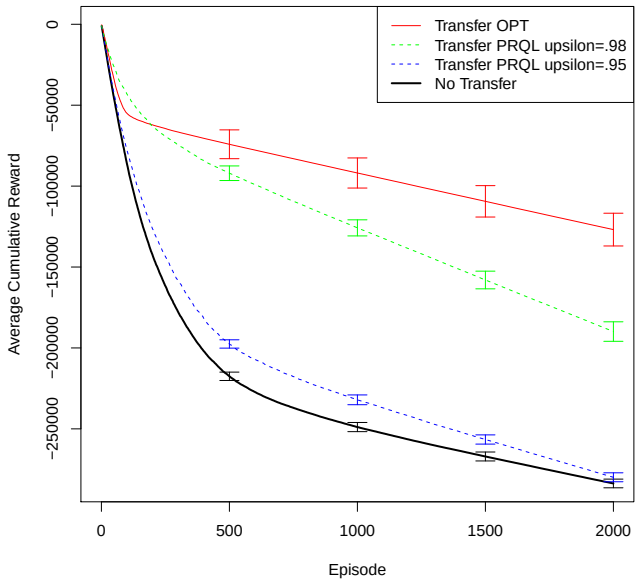
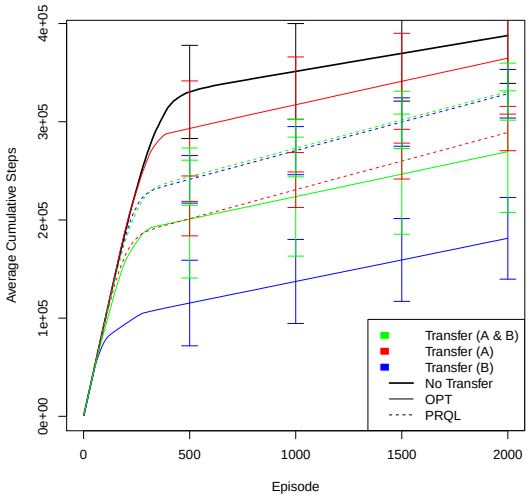


FIG. A.3. Average cumulative reward over ten trials for the Office domain on target task H. Transfer for both OPT and PRQL is provided from task B. OPT provides significant gains in learning performance. PRQL has increased learning speed for both tested values of υ (0.98 and 0.95). However, when $\upsilon = 0.95$, learning speed benefits are minimal and the ultimate policy followed is not as good. When $\upsilon = 0.98$, learning speed improvements are on par with OPT, but the final policy being followed is not as good.

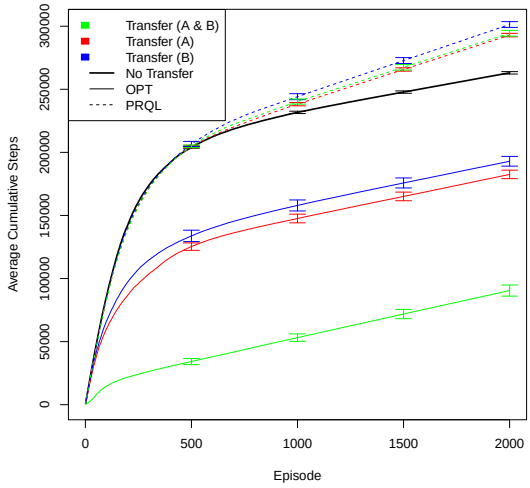
for $\upsilon = 0.98$. The υ parameter affects how long an agent favors following the source task policy within an episode before favoring the target task policy instead (where a value of $\upsilon = 1.0$ would correspond to the agent following the source task policy for the entire duration of an episode). Using a larger value was tested in addition to the recommended value of $\upsilon = 0.95$, because the source task is known to be highly similar and PRQL may therefore benefit from favoring the source task for more of the episode. Results of this experiment are shown in Figure A.3, which also shows the best performance with OPT.

PRQL does demonstrate better performance in this transfer scenario than in the ones tested in Section 3.4, but when $\upsilon = 0.95$, learning speed improvements are minor and ultimately do not result in an equally good policy as OPT. Results with $\upsilon = 0.98$ show that the learning speed of PRQL with a highly similar source and target tasks can match those of OPT, but ultimately do result in a less optimal policy. Further, using a value of $\upsilon = 0.98$ may not be practical in other scenarios because the less similar the tasks are, the less beneficial it would be to use a higher value of υ.

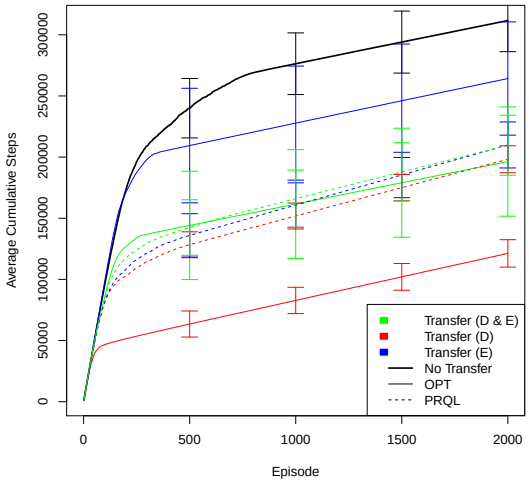
The next set of experiments is designed to test the role of the reward function and Q-value initializations in PRQL's performance compared to OPT. In the experiments in Section 3.4, a reward function that returns 1 for transitions to the goal and -1 everywhere else is used and the Q-values are semi-pessimistically initialized to -90. In the original PRQL work, the reward function returned 1 at the goal and 0 everywhere else and initialized Q-values pessimistically to 0. To determine the role of these factors, additional experiments are provided with a reward function that returns 1 and 0 and initializes Q-values either pessimistically to 0 or semi-pessimistically to 0.5. With a reward function that returns 0 instead of -1, the cumulative reward after each episode can no longer be used to measure the agent's performance, because it will always be zero or one after each episode. Therefore, performance is instead shown by the average cumulative number of steps taken after each episode. In this case, better performance is indicated by a lower cumulative number of



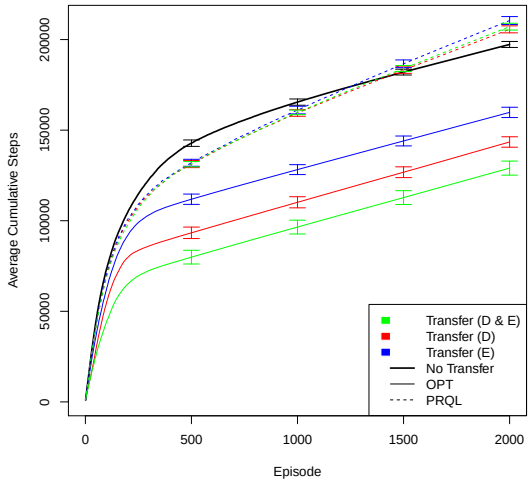
(a) Target C, source A & B; Q-init = 0.0



(b) Target C, source A & B; Q-init = 0.5



(c) Target F, source D & E; Q-init = 0.0



(d) Target F, source D & E; Q-init = 0.5

FIG. A.4. Average cumulative steps over ten trials for the office domain using a reward function that returns 1 at goal states and 0 otherwise. Subfigures (a) and (c) show results for two different target tasks and transfer scenarios with Q-values initialized to 0.0. Subfigures (b) and (d) show results on the same tasks with Q-values initialized to 0.5. Performance is both more random (with large confidence intervals) and worse overall when Q-values are initialized to 0.0. When they are initialized to 0.5, results mimic those found in Section 3.4.

steps and a flatter slope.

Figures A.4 (a) and (c) show the results for target task C and F, respectively, with a Q-value initialization of 0. Figures A.4 (b) and (d) show the results for target task C and F, respectively, with a Q-value initialization of 0.5. When the Q-values are initialized to 0.0, PRQL does appear to have more significant learning speed gains over the baseline learning without transfer than it did in the experiments in Section 3.4. Further, while OPT still has the transfer scenario with the best results, its advantages over PRQL are not as significant and in some transfer cases, PRQL does better. However, these differences occur because learning performance for both methods is worse and is highly variable as shown by the very large confidence intervals even on the baseline learning without any transfer. When the Q-value initialization is changed to a semi-pessimistic value (0.5), learning in general improves. In fact, for target C, baseline learning without transfer with a Q-value initialization of 0.5 performs better than the best PRQL transfer results (using a Q-value initialization of 0.0). Although not quite as significant, baseline learning with a Q-value initialization 0.5 is also more comparable to the best PRQL transfer results (with a Q-value initialization of 0.0) for target F. Further, when the semi-pessimistic Q-value initialization (0.5) is used, the results mimic those seen in Section 3.4, in which OPT has significant gains and performs much better than PRQL, indicating that the factor affecting performance is not the reward function, but how pessimistically the Q-values are initialized. The reason why the pessimistic Q-value initialization may result in much worse performance is because for most of the episode, no learning is actually being performed, because the reward will not alter the Q-values further. In this situation, the agent behaves randomly until a goal state reward is propagated back to a state-action pair. Such random behavior also explains why the confidence intervals for the pessimistic Q-value initialization results are so much larger.

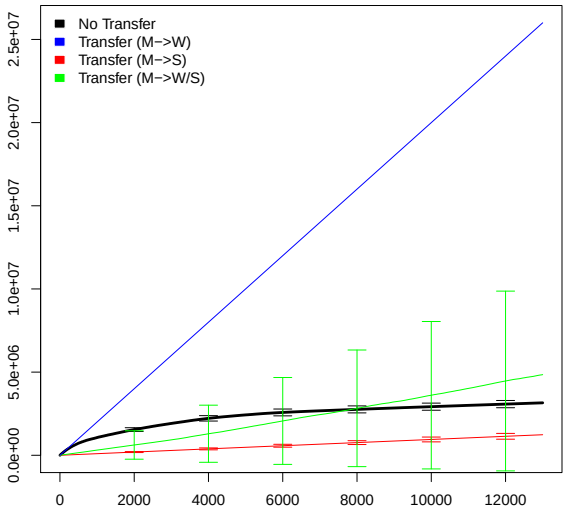
These results indicate that although PRQL may seem to do better when the Q-values are initialized very pessimistically, performance can also be similarly—or more—improved

by choosing a less pessimistic initialization of the Q-values, rather than by using transfer. Additionally, if less pessimistic value are chosen, OPT works better than PRQL.

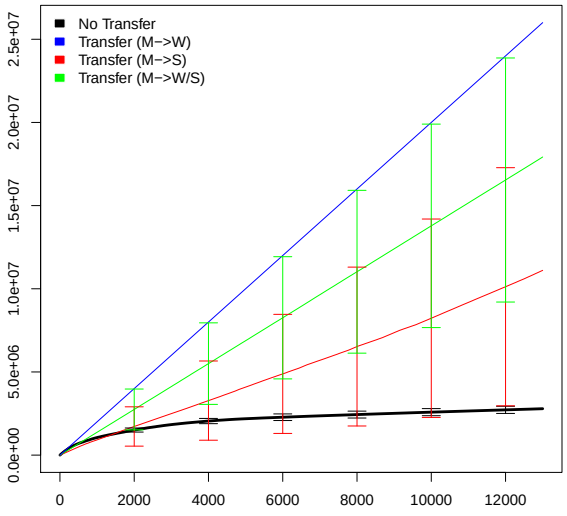
A.2 Lunar Lander Domain Results

In this section, additional results on the lunar lander domain, which was first explored in Section 3.4.2, are presented. These experiments are meant to demonstrate that the initiation and termination conditions OPT imposes on TOPs (see Section 3.2.2) are necessary to prevent an overreliance on TOPs, similar to the additional experiments presented in Appendix A.1.1 for the office navigation domain. Specifically, the performance of OPT is shown when TOPs can be initiated everywhere and only terminate in states that map to termination states in the source task. Performance under these circumstances for both transfer scenarios presented in Section 3.4.2 are shown in Figure A.5. Figure A.5 (a) shows results when the only difference between the source and target task is the thrust action set. In this case, single-source transfer from medium to weak thrust performs extremely badly an in none of the episodes does the agent reach the landing pad. In the multi-source transfer, the agent initially performs better, but ultimately converges to a worse policy than the baseline. Single-source transfer from medium to strong thrust is the only variant that performs well in this scenario, however, its average policy is not as good as the average policy when the normal OPT initiation and termination constraints are used (this is not shown in the figure). Figure A.5 (b) shows results when the source and target task differ both in the thrust action set and the position of the landing pad. In this case, all of the transfer variants perform much worse than the baseline learning and single-source transfer from medium to weak thrust never reaches the landing pad.

As with the office domain results, these results support the claim that OPT's initiation and termination conditions are necessary to prevent an overreliance on TOPs.



(a) Action set differences only



(b) Action set and landing pad location differences

FIG. A.5. Average cumulative steps over ten trials for the lunar lander domain when TOPs can be initiated anywhere and only terminate in states that map to terminal states of the source task. Subfigure (a) shows results when the only difference between the source and target task are the thrust action sets. Only single-source transfer from medium to strong thrust works well, but the average policy performance is worse than when OPT uses its normal initiation and termination constraints. Subfigure (b) shows results when the source and target task differ both in the thrust action set and when location of the landing pad; all transfer variants perform worse than the baseline.

REFERENCES

[1] Albus, J. 1971. A theory of cerebellar function. *Mathematical Biosciences* 10:25–61.

[2] Anderson, C., et al. 1993. Q-learning with hidden-unit restarting. *Advances in Neural Information Processing Systems* 81–81.

[3] Andre, D., and Russell, S. 2002. State abstraction for programmable reinforcement learning agents. In *Proceedings of the National Conference on Artificial Intelligence*, 119–125.

[4] Asada, M.; Noda, S.; Tawaratsumida, S.; and Hosoda, K. 1994. Vision-based behavior acquisition for a shooting robot by using a reinforcement learning. In *Proceedings of IAPR/IEEE Workshop on Visual Behaviors*, 112–118.

[5] Atkeson, C., and Santamaria, J. 1997. A comparison of direct and model-based reinforcement learning. In *Proceedings of Robotics and Automation*, volume 4, 3557–3564.

[6] Atkeson, C.; Moore, A.; and Schaal, S. 1997. Locally weighted learning. *Artificial Intelligence Review* 11(1):11–73.

[7] Baird, L. 1995. Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the International Conference on Machine Learning*, 30–37. Morgan Kaufmann Publishers, INC.

[8] Baker, C. 1977. *The Numerical Treatment of Integral Equations*. Academic Press.

[9] Banerjee, B., and Stone, P. 2007. General game learning using knowledge transfer. In *Proceedings of The International Conference on Artificial Intelligence*, 672–677.

[10] Barto, A.; Sutton, R.; and Anderson, C. 1983. Neuronlike adaptive elements that can solve difficult learning control problems. *Systems, Man and Cybernetics, IEEE Transactions on* 13(5):834–846.

[11] Bentley, J., and Friedman, J. 1979. Data structures for range searching. *ACM Computing Surveys* 11(4):397–409.

[12] Bozkaya, T., and Ozsoyoglu, M. 1997. Distance-based indexing for high-dimensional metric spaces. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 357–368.

[13] Bradtke, S. 1993. Reinforcement learning applied to linear quadratic regulation. *Advances in Neural Information Processing Systems* 295–295.

[14] Brafman, R., and Tennenholtz, M. 2003. R-max—a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research* 3:213–231.

[15] Ciaccia, P.; Patella, M.; and Zezula, P. 1997. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, 426–435.

[16] Cohen, W. 1995. Fast effective rule induction. In *Proceedings of the International Conference on Machine Learning*, 115–123.

[17] Croonenborghs, T.; Driessens, K.; and Bruynooghe, M. 2008. Learning relational options for inductive transfer in relational reinforcement learning. *Inductive Logic Programming* 88–97.

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

[19] Dietterich, T. G. 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Jouranl of Artificial Intelligence Research* 13:227–303.

[20] Diuk, C.; Cohen, A.; and Littman, M. 2008. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, 240–247. ACM.

[21] Driessens, K.; Ramon, J.; and Blockeel, H. 2001. Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner. *Proceedings of the European Conference on Machine Learning* 97–108.

[22] Drummond, C. 2002. Accelerating reinforcement learning by composing solutions of automatically identified subtasks. *Journal of Artificial Intelligence Research* 16:59–104.

[23] Fachantidis, A.; Partalas, I.; Taylor, M.; and Vlahavas, I. 2012. Transfer learning via multiple inter-task mappings. *Recent Advances in Reinforcement Learning* 225–236.

[24] Falkenhainer, B.; Forbus, K.; and Gentner, D. 1989. The Structure-Mapping Engine: Algorithm and examples. *Artificial Intelligence* 41(1):1–63.

[25] Ferguson, K., and Mahadevan, S. 2006. Proto-transfer learning in Markov decision processes using spectral methods. In *Proceedings of the International Conference on Machine Learning Workshop on Structural Knowledge Transfer for Machine Learning*, 151.

[26] Fernández, F., and Veloso, M. 2005. Building a library of polices through policy reuse. Technical Report CMU-CS-05-174, Carnegie-Mellon University School of Computer Science.

[27] Fernández, F., and Veloso, M. 2006. Reusing and building a policy library. In

Proceedings of the International Conference on Automated Planning and Scheduling, 378–381.

[28] Fernández, F.; García, J.; and Veloso, M. 2010. Probabilistic policy reuse for inter-task transfer learning. *Robotics and Autonomous Systems* 58(7):866–871.

[29] Fikes, R., and Nilsson, N. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3-4):189–208.

[30] Flynn, M.; Flachs, B.; and Flynn, M. 1992. Sparse adaptive memory. Technical Report 92-530, Stanford University.

[31] Foster, D., and Dayan, P. 2002. Structure in the space of value functions. *Machine Learning* 49(2):325–346.

[32] Fox, E.; Harman, D.; Baeza-Yates, R.; and Lee, W. 1992. *Inverted Files*. Prentice-Hall, Englewood Cliffs, NJ.

[33] Fritzke, B. 1997. A self-organizing network that can follow non-stationary distributions. *Proceedings of the International Conference on Artificial Neural Networks* 613–618.

[34] Fu, A.; Chan, P.; Cheung, Y.; and Moon, Y. 2000. Dynamic VP-tree indexing for n-nearest neighbor search given pair-wise distances. *The International Journal on Very Large Data Bases* 9(2):154–173.

[35] Guestrin, C.; Koller, D.; Gearhart, C.; and Kanodia, N. 2003. Generalizing plans to new environments in relational MDPs. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

[36] Guestrin, C.; Patrascu, R.; and Schuurmans, D. 2002. Algorithm-directed exploration for model-based reinforcement learning in factored MDPs. In *Proceedings of the International Conference on Machine Learning*, 235–242.

[37] Hely, T.; Willshaw, D.; and Hayes, G. 1997. A new approach to Kanerva's sparse distributed memory. *Neural Networks, IEEE Transactions on* 8(3):791–794.

[38] Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

[39] Jong, N., and Stone, P. 2007. Model-based exploration in continuous state spaces. In *The Seventh Symposium on Abstraction, Reformulation, and Approximation*, 258–272.

[40] Jonsson, A., and Barto, A. 2005. A causal approach to hierarchical decomposition of factored MDPs. In *Proceedings of the International Conference on Machine Learning*, 401–408. ACM.

[41] Kanerva, P. 1993. Sparse distributed memory and related models. In *Associative Neural Memories*, 50–76.

[42] Keller, T., and Eyerich, P. 2012. PROST: Probabilistic planning based on UCT. In *Proceedings of the International Conference on Automated Planning and Scheduling*.

[43] Klenk, M., and Forbus, K. 2007. Measuring the level of transfer learning by an AP physics problem-solver. In *Proceedings of the National Conference on Artificial Intelligence*, 446–451.

[44] Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *Proceedings of the European Conference on Machine Learning*, 282–293.

[45] Konidaris, G., and Barto, A. 2006. Autonomous shaping: Knowledge transfer in reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, 489–496.

[46] Konidaris, G., and Barto, A. 2007. Building portable options: Skill transfer in reinforcement learning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 895–900.

[47] Konidaris, G., and Barto, A. 2009. Skill discovery in continuous reinforcement learning domains using skill chaining. *Advances in Neural Information Processing Systems* 22:1015–1023.

[48] Konidaris, G.; Scheidwasser, I.; and Barto, A. 2012. Transfer in reinforcement learning via shared features. *Journal of Machine Learning Research* 13:1333–1371.

[49] Könik, T.; O'Rorke, P.; Shapiro, D.; Choi, D.; Nejati, N.; and Langley, P. 2009. Skill transfer through goal-driven representation mapping. *Cognitive Systems Research* 10(3):270–285.

[50] Kretchmar, R., and Anderson, C. 1997. Comparison of CMACs and radial basis functions for local function approximators in reinforcement learning. In *Proceedings of the International Conference on Neural Networks*, volume 2, 834–837. IEEE.

[51] Kuhlmann, G., and Stone, P. 2007. Graph-based domain mapping for transfer learning in general games. *Proceedings of the European Conference on Machine Learning* 188–200.

[52] Kuhn, H. 1955. The Hungarian Method for the assignment problem. *Naval Research Logistics Quarterly* 2(1):83–97.

[53] Lagoudakis, M., and Parr, R. 2003. Least-squares policy iteration. *Journal of Machine Learning Research* 4:1107–1149.

[54] Lazaric, A. 2008. *Knowledge transfer in reinforcement learning*. Ph.D. Dissertation, Politecnico di Milano.

[55] Leffler, B.; Littman, M.; and Edmunds, T. 2007. Efficient reinforcement learning with relocatable action models. In *Proceedings of the National Conference on Artificial Intelligence*, volume 22, 572.

[56] Liu, Y., and Stone, P. 2006. Value-function-based transfer for reinforcement learning using structure mapping. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, 415.

[57] Maclin, R.; Shavlik, J.; Torrey, L.; Walker, T.; and Wild, E. 2005. Giving advice about preferred actions to reinforcement learners via knowledge-based kernel regression. In *Proceedings of the National Conference on Artificial intelligence*, 819–824.

[58] Madden, M., and Howley, T. 2004. Transfer of experience between reinforcement learning environments with progressive difficulty. *Artificial Intelligence Review* 21:375–398.

[59] Marthi, B.; Kaelbling, L.; and Lozano-Perez, T. 2007. Learning hierarchical structure in policies. In *Proceedings of the Advances in Neural Information Processing Systems Workshop on Hierarchical Organization of Behavior*.

[60] McGovern, A., and Barto, A. 2001a. Accelerating reinforcement learning through the discovery of useful subgoals. In *Proceedings of the 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space*.

[61] McGovern, A., and Barto, A. 2001b. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the International Conference on Machine Learning*, 361–368.

[62] Mehta, N.; Ray, S.; Tadepalli, P.; and Dietterich, T. 2008. Automatic discovery and transfer of MAXQ hierarchies. In *Proceedings of the International Conference on Machine Learning*, 648–655.

[63] Melis, E., and Whittle, J. 1999. Analogy in inductive theorem proving. *Journal of Automated Reasoning* 22(2):117–147.

[64] Menache, I.; Mannor, S.; and Shimkin, N. 2002. Q-cut—dynamic discovery of subgoals in reinforcement learning. *Proceedings of the European Conference on Machine Learning* 187–195.

[65] Moore, A., and Atkeson, C. 1993. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning* 13(1):103–130.

[66] Ouyang, T. Y., and Forbus, K. D. 2006. Strategy variations in analogical problem solving. In *Proceedings of the National Conference on Artificial Intelligence*, 446–441.

[67] Peng, J. 1993. *Efficient dynamic programming-based learning for control*. Ph.D. Dissertation, Northeastern University.

[68] Perkins, T.; Precup, D.; et al. 1999. Using options for knowledge transfer in reinforcement learning. Technical Report 99-34, University of Massachusetts, Amherst.

[69] Platt, J. 1991. A resource-allocating network for function interpolation. *Neural Computation* 3(2):213–225.

[70] Poggio, T., and Girosi, F. 1990. Networks for approximation and learning. *Proceedings of the IEEE* 78(9):1481–1497.

[71] Ramon, J.; Driessens, K.; and Croonenborghs, T. 2007. Transfer learning in reinforcement learning problems through partial policy recycling. In *Proceedings of the European Conference on Machine Learning*, 699–707.

[72] Rao, R., and Fuentes, O. 1998. Hierarchical learning of navigational behaviors in an autonomous robot using a predictive sparse distributed memory. *Autonomous Robots* 5(3):297–316.

[73] Ratitch, B., and Precup, D. 2004. Sparse distributed memories for on-line value-based reinforcement learning. *Proceedings of the European Conference on Machine Learning* 347–358.

[74] Ravindran, B., and Barto, A. 2003. An algebraic approach to abstraction in reinforcement learning. In *Proceedings of the Twelfth Yale Workshop on Adaptive and Learning Systems*, 109–144.

[75] Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39(1):127–177.

[76] Rummery, G., and Niranjan, M. 1994. On-line Q-learning using connectionist systems. Technical Report 166, University of Cambridge, Department of Engineering.

[77] Selfridge, O.; Sutton, R.; and Barto, A. 1985. Training and tracking in robotics. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

[78] Sherstov, A., and Stone, P. 2005. Improving action selection in MDP's via knowledge transfer. In *Proceedings of the National Conference on Artificial Intelligence*, 1024–1029. AAAI Press.

[79] Simsek, O., and Barto, A. 2004. Using relative novelty to identify useful temporal

abstractions in reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, volume 21, 751.

[80] Simsek, O., and Barto, A. 2007. Betweenness centrality as a basis for forming skills. Technical Report 07-26, University of Massachusetts, Department of Computer Science.

[81] Singh, S., and Sutton, R. 1996. Reinforcement learning with replacing eligibility traces. *Machine Learning* 22(1):123–158.

[82] Singh, S. 1992. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning* 8(3):323–339.

[83] Smart, W., and Kaelbling, L. 2000. Practical reinforcement learning in continuous spaces. In *Proceedings of the International Conference on Machine Learning*, 903–910.

[84] Soni, V., and Singh, S. 2006. Using homomorphisms to transfer options across continuous reinforcement learning domains. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, 494.

[85] Stanley, K., and Miikkulainen, R. 2002. Evolving neural networks through augmenting topologies. *Evolutionary Computation* 10(2):99–127.

[86] Strehl, A.; Diuk, C.; and Littman, M. 2007. Efficient structure learning in factored-state MDPs. In *Proceedings of the National Conference on Artificial Intelligence*, 645.

[87] Sunmola, F., and Wyatt, J. 2006. Model transfer for Markov decision tasks via parameter matching. In *Proceedings of the 25th Workshop of the UK Planning and Scheduling Special Interest Group*.

[88] Sutton, R., and Barto, A. 1998. *Reinforcement Learning*. MIT Press.

[89] Sutton, R.; Precup, D.; and Singh, S. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence* 112(1):181–211.

[90] Sutton, R.; Whitehead, S.; et al. 1993. Online learning with random representations. In *Proceedings of the International Conference on Machine Learning*, 314–321.

[91] Sutton, R. 1988. Learning to predict by the methods of temporal differences. *Machine Learning* 3(1):9–44.

[92] Talvitie, E., and Singh, S. 2007. An experts algorithm for transfer learning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1065–1070.

[93] Tanaka, F., and Yamamura, M. 2003. Multitask reinforcement learning on the distribution of MDPs. In *Proceedings of the IEEE International Symposium on Computational Intelligence in Robotics and Automation*, volume 3, 1108–1113. IEEE.

[94] Taylor, M., and Stone, P. 2005. Behavior transfer for value-function-based reinforcement learning. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, 53–59.

[95] Taylor, M., and Stone, P. 2007. Cross-domain transfer for reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, volume 24, 879.

[96] Taylor, M.; Jong, N.; and Stone, P. 2008. Transferring instances for model-based reinforcement learning. *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases* 488–505.

[97] Taylor, M.; Stone, P.; and Liu, Y. 2007. Transfer learning via inter-task mappings for temporal difference learning. *Journal of Machine Learning Research* 8(1):2125–2167.

[98] Taylor, M.; Whiteson, S.; and Stone, P. 2006. Comparing evolutionary and temporal difference methods in a reinforcement learning domain. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, 1321–1328.

[99] Taylor, M.; Whiteson, S.; and Stone, P. 2007. Transfer via inter-task mappings in policy search reinforcement learning. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, 1–8. ACM.

[100] Torrey, L.; Walker, T.; Shavlik, J.; and Maclin, R. 2005. Using advice to transfer knowledge acquired in one reinforcement learning task to another. In *Proceedings of the European Conference on Machine Learning*, 412–424.

[101] Torrey, L.; Shavlik, J.; Walker, T.; and Maclin, R. 2006. Skill acquisition via transfer learning and advice taking. In *Proceedings of the European Conference on Machine Learning*, 425–436.

[102] Torrey, L.; Shavlik, J.; Walker, T.; and Maclin, R. 2007. Relational macros for transfer in reinforcement learning. In *Proceedings of the International Conference on Inductive Logic Programming*, 254–268.

[103] Tsitsiklis, J., and Van Roy, B. 1997. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control* 42(5):674–690.

[104] Veloso, M. M., and Carbonell, J. G. 1993. Derivational analogy in prodigy: Automating case acquisition, storage, and utilization. *Machine Learning* 10(3):249–278.

[105] Walsh, T.; Li, L.; and Littman, M. 2006. Transferring state abstractions between MDPs. In *Proceedings of the International Conference on Machine Learning Workshop on Structural Knowledge Transfer for Machine Learning*.

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

[107] Watkins, C. 1989. *Learning from delayed rewards*. Ph.D. Dissertation, Cambridge University.

[108] Wilson, A.; Fern, A.; Ray, S.; and Tadepalli, P. 2007. Multi-task reinforcement learning: a hierarchical Bayesian approach. In *Proceedings of the International Conference on Machine Learning*, 1015–1022.

[109] Zobel, J., and Moffat, A. 2006. Inverted files for text search engines. *ACM Computing Surveys* 38(2):6.