AN ABSTRACT OF THE DISSERTATION OF

Neville Mehta for the degree of Doctor of Philosophy in Computer Science presented on September 23, 2011.

Title: Hierarchical Structure Discovery and Transfer in Sequential Decision Problems

Abstract approved: ______________________________

Prasad Tadepalli

Acting intelligently to efficiently solve sequential decision problems requires the ability to extract hierarchical structure from the underlying domain dynamics, exploit it for optimal or near-optimal decision-making, and transfer it to related problems instead of solving every problem in isolation. This dissertation makes three contributions toward this goal.

The first contribution is the introduction of two frameworks for the transfer of hierarchical structure in sequential decision problems. The MASH framework facilitates transfer among multiple agents coordinating within a domain. The VRHRL framework allows an agent to transfer its knowledge across a family of domains that share the same transition dynamics but have differing reward dynamics. Both MASH and VRHRL are validated empirically in large domains and the results demonstrate significant speedup in the solutions due to transfer.

The second contribution is a new approach to the discovery of hierarchical structure in sequential decision problems. HI-MAT leverages action models to analyze the relevant dependencies in a hierarchically-generated trajectory and it discovers hierarchical structure that transfers to all problems whose actions share the same relevant dependencies as the single source problem. HierGen advances HI-MAT by learning simple action models, leveraging these models to analyze non-hierarchically-generated trajectories from multiple source problems in a robust causal fashion, and discovering hierarchical structure that transfers to all problems whose actions share the same causal dependencies as those in the source problems. Empirical evaluations in multiple domains demonstrate that the discovered hierarchical structures are comparable to manually-designed structures in quality and performance.

Action models are essential to hierarchical structure discovery and other aspects of intelligent behavior. The third contribution of this dissertation is the introduction of two general frameworks for learning action models in sequential decision problems. In the MBP framework, learning is user-driven; in the PLEX framework, the learner generates its own problems. The frameworks are formally analyzed and reduced to concept learning with one-sided error. A general action-modeling language is shown to be efficiently learnable in both frameworks.

Hierarchical Structure Discovery and Transfer in Sequential Decision Problems

by

Neville Mehta

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented September 23, 2011
Commencement June 2012

ACKNOWLEDGEMENTS

My tenure as a graduate student has been the best period of my life and I will attempt to verbalize my appreciation of the people who have made the experience so wonderful and enlightening.

I must begin by expressing my immeasurable gratitude for the unquestioning love and support of my parents and grandparents. Despite their trials, they have always encouraged me to pursue my dreams. Most importantly, they have taught me to never take anything for granted. They are, therefore I am.

Prasad Tadepalli has been integral to my graduate pursuits. Ever since he picked me as a graduate research assistant, he has allowed me to explore/exploit at will, while patiently supporting and guiding me throughout. He has helped shape my research ideas and has come through for me at crunch time on innumerable occasions. His diligent critique of every article that I have sent his way, including many versions of this dissertation, has always greatly improved the quality of the article.

In large part, my research follows the work done by Thomas Dietterich. Consequently, his mentoring and advice have been invaluable. Despite his jet-setting schedule, he has always found the time to respond in detail to my random musings. He combed through initial drafts of the dissertation and improved it immensely.

I have worked together with Alan Fern on a few projects and he has been an inspiring collaborator and advisor. His provoking questions and profound insights have continually motivated me to think bigger.

My committee members, Weng-Keen Wong and Margaret Niess, have been extremely helpful in all aspects of the process. They have also been very patient and understanding during my hectic dash to the finish.

I have had the pleasure of interacting with my excellent colleagues at Oregon State. Among them, I would especially like to thank Adam Ashenfelter, Ronald Bjarnason, Darren Brown, Diane Damon, Ethan Dereszynski, Robin Hess, Kimberly Mach, Charles Parker, Scott Proper, Aaron Wilson, and Michael Wynkoop for expanding my horizons.

Finally, in a nod to inanimate objects, I would like to acknowledge things with strings — guitars and tennis racquets in particular — for keeping me going.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF ALGORITHMS

To the strongest person I know, my mother.

Chapter 1: Introduction

Sequential decision problems are ubiquitous in the lives of people and range from the mundane (scratching an itch) to the sophisticated (curing disease through genetic engineering). These are problems in which decisions need to be made in response to sensing an environment to accomplish a goal. Every decision entails choosing an action from a predefined set that affects the state of the environment. If every action has a fixed outcome, then the environment is deterministic; if an action has a range of outcomes distributed according to a probability distribution, then the environment is stochastic. These action outcomes can either be known a priori or unknown. Because the outcomes could be stochastic and the environment might only be sensed partially by the agent, a fixed sequence of actions might not always reach the intended goal. Thus, the solution to a sequential decision problem is a behavioral policy that defines an action (or a distribution over actions) for every possible state of the environment. A plethora of problems from playing a game to acquiring a doctorate can be framed as sequential decision problems.

Reinforcement learning (RL) is a subfield of Machine Learning that studies how an agent can learn to optimize its behavioral policy in the process of interacting with an unknown, stochastic environment (Sutton and Barto, 1998). A Markov Decision Process (MDP) is a theoretical discrete-time formalism that is able to model sequential decision problems and is employed across a spectrum of paradigms from RL through decision-theoretic planning to classical planning. Despite this convenience, MDPs suffer from the curse of dimensionality, that is, their explicit state and action spaces grow exponentially with the number of state variables and action effectors respectively (Bellman, 1957). Thus, classical dynamic programming algorithms that exhibit polynomial-time complexity with respect to the size of the state space are actually intractable with respect to the number of state variables. Consequently, there has been a lot of work on developing more efficient algorithms to curb this state- and action-space combinatorial explosion. Some of these approaches strive for exact solutions while others attempt approximate ones, but both essentially leverage some of the structure (redundancies and symmetries) within domains to expedite learning. However, the degree to which the domain structure can be leveraged in RL is limited and the difficulty in sequential decision-making increases (at worst, exponentially) as a function of the distance to the goal and the number of relevant features.

In the spirit of the divide-and-conquer strategy to problem-solving, hierarchical reinforcement learning (HRL) is analogous to the way human beings tackle tasks — we normally look to break down the overall problem into its constituent parts in some intuitive way (that is normally based on achieving subgoals), solve the more manageable parts, and recombine them to produce the overall solution. HRL leverages hierarchical structure within the MDP to expedite the solutions by facilitating acting, learning, and planning at multiple levels of temporal abstraction. For instance, HRL could allow a robot to structure its low-level noisy sensors (camera, accelerometer, etc.) and parameterized effectors (joint actuators, grippers, etc.) to analyze and act at levels of temporal abstraction that correspond to the spectrum of tasks at hand, ranging from moving an arm to a certain angle to reconnoitering a planet. It would be just as frustrating for this robot to learn and plan for high-level functionality (for instance, helping a disabled person) based solely on its primitive state and action representations as it would be to formulate this entire dissertation in binary. The performance of HRL given well-designed hierarchical structure can be orders of magnitudes superior to that of non-hierarchical (flat) RL.

Models of the action dynamics are essential for many RL and HRL methods. The quality of a model depends on how accurately it can answer the key questions about the process being represented, especially answers that facilitate prediction of the process. For complex real-world domains, these models could either be impossible to describe exactly in the modeling language or the law of diminishing returns renders the computational investment required to plan with increased complexity not worthwhile. For example, to describe the precise workings of the universe would require a descriptive model the size of the universe. This unsettling proposition can be worked around via the mechanisms of abstraction and approximation. For example, the Newtonian laws of physics comprise a simpler model of motion that has since been superseded by the more accurate but more complicated models of the general theory of relativity. Despite the fact that the latter is more accurate in situations where the former fails, Newtonian laws still provide sound and efficient models at most levels of detail. Similarly, physical simulation engines use extremely deficient models for collisions with deformations, because not only are all the electrostatic forces acting within and between the objects extremely complex to model, but processing all this information would probably make the simulation computationally intractable. In many cases, stochasticity can provide a fuzzy blanket to cover up the excruciating details.

1.1 Motivation

People seldom learn in isolation — knowledge and skills are transferred across generations and peers. Traditionally, reinforcement learning has focused on learning in the context of a single problem or task at hand. Instead, transfer learning seeks to exploit the similarities between different tasks and transfer the knowledge gained from solving source problems to expedite learning in a target problem, with the cost of learning in the source problems being amortized over several target problems. Sequential decision problems provide plenty of opportunities for transfer because of the multitude of parameters that define these problems. For instance, RL might take a long time to discover long sequences of actions that achieve intermediate subgoals required for the successful solution to a problem, but structural knowledge transferred from smaller versions of a problem that share the same fundamental structure can significantly expedite learning in the original problem.

The biggest detraction from the impressive performance of HRL over RL is that, within most current state-of-the-art HRL systems, an expert must provide the hierarchical structure as prior knowledge to the agent. Ignoring the paradox that this expert needs to have an intimate understanding of the unknown MDP being solved, there is considerable cognitive burden to get the structure right, because providing the agent with inferior or faulty knowledge could lead to performance that is worse than flat RL! The dearth of methods for automating the discovery of good hierarchical structure is a serious lacuna in HRL.

Action models help reveal the underlying structure of a problem and the relatedness of two problems for the purposes of transfer. For example, the separate processes of electricity and magnetism were successfully combined into the unified field of electromagnetism after the initially distinct models were discovered to be deeply correlated and to share strong causal dependencies; this lead to a huge transfer of ideas. However, just as with hierarchical structure for most HRL systems, many methods that utilize action models assume that exact models are available a priori. It is important to analyze how these models can be learned autonomously and to study the trade-off between the model complexity and the sample and computational complexity required to learn them.

1.2 Contributions

My research contributions, described in this dissertation, are as follows:

- I analyze the transfer of predefined hierarchical structure in two settings: across multiple agents functioning within the same environment and across a family of environments that share the same transition dynamics but differing reward dynamics. For the multi-agent setting, I introduce the Multi-Agent Shared-Hierarchy (MASH) framework that improves and generalizes a previous HRL framework to allow knowledge sharing and decomposition among multiple agents. Empirical results show that not only does transfer across the cooperating agents mitigate the combinatorial explosion of the joint state and actions spaces, but MASH actually does much better than a single agent because of the parallel exploration of the larger space. In the setting of differing reward dynamics, I introduce an online algorithm, Variable-Reward HRL (VRHRL), that compactly stores the optimal value functions for several MDPs and uses them to optimally initialize the value function for a new MDP. Empirical results demonstrate the significant speedup in learning due to transfer.

- I introduce the HI-MAT approach for discovering hierarchical structure that transfers across a family of environments whose dynamics share the same qualitative causal dependencies. Under appropriate assumptions, HI-MAT induces hierarchies that are consistent with the single input domain, have compact space requirements without any approximations, and compare favorably to manually-designed hierarchical structure in learning performance in the target domains. I also introduce the HierGen approach that advances and generalizes HI-MAT in several ways including learning simpler action models and discovering structure that generalizes across multiple source domains. Both approaches are empirically validated in multiple domains.

- Because action models are an integral component of hierarchical structure discovery, I introduce and analyze two new frameworks for learning action models through interaction with an environment. I describe a new notion of approximation for action models in which the models are much simpler than the true models but are sufficiently exact for planning. In the mistake-bounded planning framework, the learner has access to a planner, an environment, and a goal generator and aims to learn with at most a polynomial number of faulty plans. The planned exploration framework is more demanding because the learner does not have access to a goal generator and must design its own goals, solve for them, and converge with at most a polynomial number of planning attempts.

1.3 Outline

This dissertation is hierarchically structured as follows:

- Chapter 2 introduces various fundamental formalisms and notation that will be employed throughout the manuscript.
- Chapter 3 introduces and describes the MASH framework that allows knowledge sharing among multiple agents. Moreover, the agents pool their experiences in parallel while coordinating at different levels of the hierarchical structure. The chapter describes a model-based average-reward reinforcement learning algorithm for the MASH framework and presents empirical results in a simplified real-time strategy game that show that the MASH framework is much more scalable than previous approaches to multi-agent learning.
- Chapter 4 considers transfer learning in the context of related RL problems that are derived from MDPs that share the same transition dynamics, but have different reward functions. Transfer is especially effective in the hierarchical setting because the overall policy is decomposed into sub-policies that are more likely to be amenable to transfer across different MDPs.
- Chapter 5 describes the HI-MAT framework that discovers hierarchical structure in a source domain by exploiting the relevant dependencies of actions and transfers that structure to speed up learning in a related target domain with the same

underlying relevant dependencies. The chapter demonstrates that relevantly motivated hierarchical structure transfers more robustly than other kinds of detailed knowledge that depend on the idiosyncrasies of the source domain and are hence less transferable.

- Chapter 6 describes the HierGen approach that significantly advances HI-MAT in several directions. A more robust annotation scheme, based on simple learned action models, facilitates the parsing of trajectories from multiple source domains simultaneously. Instead of relying on rich action models to provide the termination conditions, HierGen generalizes across the trajectories given a termination language and produces hierarchical structure with explicit parameterization that is comparable to manually-designed hierarchies.

- Chapter 7 introduces the mistake-bounded planning (MBP) and planned exploration (PLEX) frameworks for learning sufficiently exact action models for planning. In the MBP framework, the learner aims to learn the models with at most a polynomial number of faulty plans. In the PLEX framework, the learner must learn the models with at most a polynomial number of planning attempts. The chapter reduces learning in these frameworks to concept learning with one-sided error and also exhibits that a concrete family of hypothesis spaces for action modeling is efficiently learnable in both frameworks.

Chapter 2: Background

This chapter describes the Markov decision process (MDP) formalism, the hierarchical reinforcement learning (HRL) paradigm, and the relevant notation that will be employed throughout this dissertation.

2.1 Markov Decision Process

An MDP formally models any sequential stochastic process that progresses from one state to another depending on actions taken by an agent (at discrete time intervals) (Bellman, 1957). It is defined as $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where $\mathcal{S}$ is a finite set of states (the state space), $\mathcal{A}$ is a finite set of actions (the action space), $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0, 1] = \Pr(s' \in \mathcal{S} | s \in \mathcal{S}, a \in \mathcal{A})$ is the Markovian transition function specifying the probability of transitioning to state s' when action a is executed in state s, and $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$ is the reward function where $\mathcal{R}(s, a, s')$ specifies the immediate utility of executing action a in state s and transitioning to state s'. The expected reward function is denoted as $\mathcal{R} : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R} = \mathbb{E}_{s'}[\mathcal{R}(s, a, s')]$. The Markov assumption asserts that

$$\Pr(s_{t+1}|s_0, \dots, s_t, a_t) = \Pr(s_{t+1}|s_t, a_t),$$

that is, the state at time $t + 1$ is independent of all previous states and actions given the state and action at time t. This assumption is not restrictive, because any non-Markovian model of a system can be converted to an equivalent, albeit larger, MDP by including all the relevant information in the state. The other common assumption is that the transition dynamics are *stationary* or time-invariant, that is, the distribution over next states is independent of the time when the action is executed.

An MDP is *deterministic* if $\forall(s \in \mathcal{S})\, \forall(a \in \mathcal{A})\, \exists(s' \in \mathcal{S})\ \Pr(s'|s, a) = 1$; it is *stochastic* otherwise. Let $S \subset \mathcal{S}$. If $\forall(s \in S)\, \forall(a \in \mathcal{A})\, \forall(s' \notin S)\ \Pr(s'|s, a) = 0$ then S is a *closed* set of states. If no proper subset of S has this property, then S is a proper closed set or a *recurrent class* of states. A state s is *transient* if it does not belong in a recurrence class and s is *absorbing* if s is the only state in a recurrent class and

$\forall(a \in \mathcal{A})\,\forall(s' \in S)\ \mathcal{R}(s,a,s') = 0$.

A *generative* or *strong simulator* model of an MDP is a randomized algorithm that takes a state-action pair (s,a) as input and outputs the next state $s' \sim \Pr(\cdot|s,a)$ and the immediate reward $\mathcal{R}(s,a,s')$. An *online* or *weak simulation* model is a randomized algorithm with internal state s that takes an action a as input, transitions internally to $s' \sim \Pr(\cdot|s,a)$, and outputs s' and $\mathcal{R}(s,a,s')$; the agent has no recourse to resets (a jump to a state governed by an initial state distribution) of the simulation. The online MDP model is assumed throughout this dissertation.

2.1.1 Policy and Value Function

A *stationary deterministic* policy is a function $\pi : \mathcal{S} \mapsto \mathcal{A}$ defining the action $a \in \mathcal{A}$ taken in any state $s \in \mathcal{S}$. A *stationary stochastic* policy is a function $\pi : \mathcal{S} \times \mathcal{A} \mapsto [0,1] = \Pr(a \in \mathcal{A}|s \in \mathcal{S})$, denoted as $\pi(a|s)$, defining the probability distribution over actions conditional on the state. An MDP is *recurrent* or *ergodic* if every policy induces a Markov chain with a single recurrent class, and it is *unichain* if every policy induces a single recurrent class with some possible transient states. An MDP is *communicating* if there is a non-zero probability of getting from a state to any other state under some stationary policy.

A *trajectory* of length l is a sequence $\tau = (s_0, a_0, r_0, \ldots, s_{l-1}, a_{l-1}, r_{l-1}, s_l)$, where $s_t \in \mathcal{S}, a_t \in \mathcal{A}$, and $r_t \in \mathcal{R}(s_t, a_t, s_{t+1})$, denote the state, action, and reward, at time t. The probability of a trajectory τ being generated by a policy π is

$$\Pr(\tau|\pi) = \prod_{t=0}^{l-1} \pi(a_t|s_t)\Pr(s_{t+1}|s_t, a_t).$$

A subtrajectory τ_i going from state s_j to state s_k is the subsequence $(s_j, a_j, r_j, \ldots, s_k)$, and $\text{Actions}(\tau_i) = \{a_t : (s_t, a_t, r_t, s_{t+1}) \in \tau_i \wedge j \le t \le k\}$ represents the set of actions in the subtrajectory.

Given an MDP, the objective is to construct a policy that maximizes the expected long-term utility. The policy-based value function $V^\pi : \mathcal{S} \mapsto \mathbb{R}$ assigns to every state the true expected utility achievable from that state by following the policy π. A *Markov decision problem* (also ambiguously abbreviated MDP) is an MDP along with an objective criterion or performance metric that assigns a value function to any given policy. These criteria are as follows.

Expected total reward: The value of a state is defined as the sum of the rewards obtained by starting at that state and following policy π until an absorbing state is reached, that is,

$$V^{\pi}(s) = \mathbb{E}_{(s_0,a_0,r_0\ldots,s_h)\sim\Pr(\cdot|\pi,s_0=s)}\left[\sum_{t=0}^{\infty} r_t\right].$$

The total-reward criterion is the most direct, but can only be applied when every policy is guaranteed to eventually reach an absorbing state.

Expected cumulative discounted reward: The value of a state is defined as the discounted sum of the rewards obtained by starting at that state and following policy π from then on, that is,

$$V^{\pi}(s) = \lim_{h\to\infty} \mathbb{E}_{(s_0,a_0,r_0,\ldots,s_h)\sim\Pr(\cdot|\pi,s_0=s)}\left[\sum_{t=0}^{h} \gamma^t r_t\right].$$

Future rewards are discounted or diminished geometrically by a discount factor $\gamma \in [0, 1)$. Discounting is convenient because it is an analytically tractable method of keeping the value function bounded in an infinite-horizon setting. The parameter γ could be interpreted as the probability of surviving the next time-step. When $\gamma = 1$, this setting is equivalent to the total reward criterion.

Expected average reward: The value of a state is defined as the expected average sum of the rewards obtained by starting at that state and following policy π thereafter, that is,

$$V^{\pi}(s) = \lim_{h\to\infty} \frac{1}{h} \mathbb{E}_{(s_0,a_0,r_0,\ldots,s_h)\sim\Pr(\cdot|\pi,s_0=s)}\left[\sum_{t=0}^{h-1} r_t\right].$$

However, as the states in the recurrent class will be visited forever, the average reward will be identical across these states; also, the transient states can at most see a finite expected cumulative reward (before a recurrent state is visited), which vanishes under the limit. To address this issue, the value of a state is instead defined as the expected average-adjusted sum of the rewards obtained by starting at that state and following policy π thereafter, that is,

$$V^{\pi}(s) = \lim_{h\to\infty} \mathbb{E}_{(s_0,a_0,r_0,\ldots,s_h)\sim\Pr(\cdot|\pi,s_0=s)} \left[\sum_{t=0}^{h} \left(r_t - \rho^{\pi} \right) \right],$$

where

$$\rho^{\pi} = \lim_{h\to\infty} \frac{1}{h} \mathbb{E}_{(s_0,a_0,r_0,\ldots,s_h)\sim\Pr(\cdot|\pi,s_0=s)} \left[\sum_{t=0}^{h-1} r_t \right]$$

is the average expected reward or gain for a policy π. The value of a state represents the expected excess reward over what is expected in the duration of the action based on the current average reward and is also called the relativized bias of the state. The average-reward criterion is best suited for nonterminating processes that need to maintain a certain rate of efficiency.

The discounted value function of a policy π (or total reward when $\gamma = 1$) satisfies the Bellman equation

$$V^{\pi}(s) = \mathbb{E}_{s'\sim\Pr(\cdot|s,\pi(s))} \left[\mathcal{R}(s,\pi(s),s') + \gamma V^{\pi}(s') \right].$$

The unique value function for all optimal policies is $V^*(s) = \sup_{\pi} V^{\pi}(s)$, and it satisfies the Bellman equation

$$V^*(s) = \max_{a\in A} \Big(\mathbb{E}_{s'\sim\Pr(\cdot|s,a)} \left[\mathcal{R}(s,a,s') + \gamma V^*(s') \right] \Big).$$

The state-action value function $Q^{\pi} : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ defines the expected utility received by taking an action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$ and following policy π thereafter. In the discounted case, the state-action value function is related to the value function as

$$\begin{aligned} Q^{\pi}(s,a) &= \mathbb{E}_{s'\sim\Pr(\cdot|s,a)} \left[\mathcal{R}(s,a,s') + \gamma V^{\pi}(s') \right] \\ V^{\pi}(s) &= Q^{\pi}(s,\pi(s)). \end{aligned}$$

The unique state-action value function for all optimal policies is $Q^*(s,a) = \sup_{\pi} Q^{\pi}(s,a)$, and it satisfies the Bellman equation

$$Q^*(s,a) = \mathbb{E}_{s' \sim \Pr(\cdot|s,a)} \left[\mathcal{R}(s,a,s') + \gamma V^*(s')\right], \text{ where } V^*(s) = \max_{a \in A} Q^*(s,a).$$

Q-learning is a model-free, off-policy learning algorithm that is a direct implementation of this Bellman equation (Watkins, 1989). After taking action a in the current state s and observing reward r and next state s', the Q value is updated as

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\big(r + \gamma \max_{a'} Q(s',a')\big),$$

where α is a scalar that controls the rate of learning. Such a scaled update can also be denoted more compactly as

$$Q(s,a) \overset{\alpha}{\leftarrow} r + \gamma \max_{a'} Q(s',a').$$

If the learning rate α is initialized to 1 and updated as $\alpha \leftarrow \alpha/(1+\alpha)$, then the scaled update becomes an exponential moving average of the quantity on the right-hand side.

For any MDP that is either unichain or communicating, there exists a scalar ρ such that the average-reward value function for a policy π satisfies the Bellman equation

$$V^*(s) = \max_{a \in \mathcal{A}} \left(\mathbb{E}_{s' \sim \Pr(\cdot|s,a)} \left[\mathcal{R}(s,a,s') + V^*(s')\right] - \rho \right).$$

The policy that selects actions to maximize the right-hand side of this equation attains the optimal gain $\rho^* \geq \rho^\pi$ for all policies π. H-learning is a model-based, off-policy learning algorithm that implements this Bellman equation (Tadepalli and Ok, 1998). After taking action a in the current state s and observing reward r and next state s', V and ρ are updated as

$$V(s) \leftarrow \max_b \left(R(s,b) + \sum_{s'} \Pr(s'|s,b)V(s') \right) - \rho$$
$$\rho \overset{\alpha}{\leftarrow} R(s,a) + V(s') - V(s) \text{ when } a \text{ is chosen greedily.}$$

The reward model $R(s,a)$ and the transition model $\Pr(s'|s,a)$ have to be learned simultaneously along with the value function and average reward. For the average reward update, the immediate reward is adjusted by the difference in the values of s' and s to neutralize the effect of exploratory actions on the states visited by the agent and give an unbiased sample of average reward of a single action of the SMDP.

2.1.2 Semi-Markov Decision Process

MDPs are discrete time models and an action executed in the current time-step affects the state and the reward at the following time-step. The Semi-Markov decision process (SMDP), a generalization of the MDP, allows for temporally extended actions, that is, actions that might take variable periods of time instead of always taking a single time step. Action selections are made at distinct epochs in time at the controlled states, while the state of the system may continue changing during the action through multiple uncontrolled states. Any SMDP with uncontrolled and controlled states can be reduced to an SMDP with only controlled states such that the optimal policy is invariant to the reduction (Parr, 1998a).

For an SMDP, the probabilistic transition function is $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathbb{N} \mapsto [0, 1] = \Pr(s' \in \mathcal{S}, d \in \mathbb{N} | s \in \mathcal{S}, a \in \mathcal{A})$, where d is a random variable that denotes the action duration. Further, the reward function is $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathbb{N} \mapsto \mathbb{R}$. Consequently, a trajectory is generalized to the sequence $\tau = (s_0, a_0, r_0, d_0, \ldots, s_{l-1}, a_{l-1}, r_{l-1}, d_{l-1}, s_l)$, where $s_t \in \mathcal{S}, a_t \in \mathcal{A}, r_t \in \mathcal{R}(s_t, a_t, d_t, s_{t+1})$, and $d_t \in \mathbb{N}$, denote the state, action, reward, and action duration at time t.

For SMDPs, the discounted reward criterion is generalized to

$$V^\pi(s) = \lim_{h \to \infty} \mathbb{E}_{(s_0, a_0, r_0, d_0, \ldots, s_h) \sim \Pr(\cdot | \pi, s_0 = s)} \left[\sum_{t=0}^{h} \gamma^{d_t} r_t \right].$$

The average reward criterion is generalized to

$$V^\pi(s) = \lim_{h \to \infty} \mathbb{E}_{(s_0, a_0, r_0, d_0, \ldots, s_h, d_h) \sim \Pr(\cdot | \pi, s_0 = s)} \left[\sum_{t=0}^{h} \left(r_t - \rho^\pi d_t \right) \right],$$

where

$$\rho^{\pi} = \lim_{h\to\infty} \frac{\mathbb{E}_{(s_0,a_0,r_0,d_0,\ldots,s_h)\sim\Pr(\cdot|\pi,s_0=s)}\left[\sum_{t=0}^{h-1} r_t\right]}{\mathbb{E}_{(s_0,a_0,r_0,d_0,\ldots,s_h)\sim\Pr(\cdot|\pi,s_0=s)}\left[\sum_{t=0}^{h-1} d_t\right]}.$$

These equations simplify to those for MDPs when $\forall t, d_t = 1$. Consequently, the generalized Bellman equations for the optimal value functions in SMDPs are

$$V^*(s) = \max_{a\in A}\Big(\mathbb{E}_{s',d\sim\Pr(\cdot,\cdot|s,a)}\left[\mathcal{R}(s,a,s',d) + \gamma^d V^*(s')\right]\Big)$$

for discounted reward and

$$V^*(s) = \max_{a\in\mathcal{A}}\left(\mathbb{E}_{s',d\sim\Pr(\cdot,\cdot|s,a)}\left[\mathcal{R}(s,a,s',d) + V^*(s') - \rho\, d\right]\right)$$

for average reward.

Q-learning can be extended to SMDPs by generalizing the update after executing action a in state s and observing reward r, action duration d, and the next state s', to

$$Q(s,a) \overset{\alpha}{\leftarrow} r + \gamma^d \max_{a'} Q(s',a').$$

Here, the discount factor is exponentiated by the observed duration d of action a. Similarly, H-learning can be extended to SMDPs as

$$V(s) \leftarrow \max_b \left(R(s,b) - \rho\, D(s,b) + \sum_{s'} \Pr(s'|s,b)V(s')\right)$$
$$\rho \leftarrow \rho_r/\rho_t,$$

where

$$\rho_r \overset{\alpha}{\leftarrow} R(s,a) + V(s') - V(s)$$

is a measure of the average reward acquired during the execution of a temporally extended primitive action and

$$\rho_t \overset{\alpha}{\leftarrow} d$$

is a measure of the average duration of a primitive action. Both these quantities are updated only when a is chosen greedily. Here, a model of the action durations $D(s, a)$ also needs to be learned along with the models for rewards and transitions.

In the remainder of the manuscript, the notation for SMDPs is simplified by letting $\Pr(s'|s, a) = \sum_d \Pr(s', d|s, a)$ and $\mathcal{R}(s, a) = \mathbb{E}_{s',d}[\mathcal{R}(s, a, s', d)]$.

2.1.3 Factored Markov Decision Process

As defined originally, MDPs have explicit or extensional state spaces, that is, every state $s \in \mathcal{S}$ is an atomic entity. However, the state spaces of most domains exhibit internal structure, and it is helpful to describe states or sets of states in terms of their properties. An exponential degree of compactness can be achieved when a domain's state space can be represented by a set of state variables $\mathcal{X} = \{x_1, \ldots, x_n\}$ where each state variable x_i takes on values from its corresponding domain $\mathcal{D}(x_i)$. When the state space of the MDP is made implicit in such a way, it is called a *factored* MDP, and it can often be specified more concisely than a regular MDP because of regularities and localities in the reward and transition functions. This implicit state space over a set of variables $\mathcal{X}$ is $\mathcal{S} = \mathcal{D}(\mathcal{X}) = \times_{x \in \mathcal{X}} \mathcal{D}(x)$ and a particular state $s = \mathbf{x} \in \mathcal{D}(\mathcal{X})$ is an assignment of values to every variable in $\mathcal{X}$. For example, if the state space $\mathcal{S}$ is the set of n-bit binary numbers, then $\mathcal{X}$ could be the set of n bits (a variable could also represent more than a single bit), where $\mathcal{D}(x_i) = \{0, 1\}$.

For the sake of clarity, this dissertation sometimes uses the object-oriented notation $\langle \mathit{name} \rangle.\langle \mathit{attribute} \rangle$ for variable names. Factoring and temporally-extended actions are orthogonal extensions of MDPs, and together they result in a factored SMDP. We do not specifically qualify an MDP whenever the generic term can be used unambiguously.

2.1.4 State Abstraction

A *partition* of a set S is a set of sets $Z = \{Z_i\}_{i=1}^n$ such that $\forall i, j$, $Z_i \cap Z_j = \varnothing$ and $\bigcup_i Z_i = S$. For any $s \in S$, $[s]_Z$ denotes the block of states $Z_i \in Z$ to which s belongs. For any $s, t \in S$, $s \equiv_Z t \iff [s]_Z = [t]_Z$, where $\equiv_Z$ is an *equivalence* relation. A function $f : S \mapsto W$, where W is some arbitrary set, induces a partition of S where $\forall s_1, s_2 \in S$, $s_1 \equiv_f s_2 \iff f(s_1) = f(s_2)$. Given two partitions Y and Z, Y is a *refinement* of Z, denoted by $Y \preceq Z$, if $\forall x, y \in S$, $x \equiv_Y y \implies x \equiv_Z y$;

equivalently, Z is *coarser* than Y. The $\preceq$ relation is a partial order on the set of partitions of S. For example, if $S = \{x_1, x_2, \ldots, x_9\}$ and $f = \{(x_i, i \bmod 2) : x_i \in \mathcal{X}\}$, then f induces the partition $\{\{x_1, x_3, \ldots, x_9\}, \{x_2, x_4, \ldots, x_8\}\}$ and the partition $\{\{x_1\}, \{x_3, \ldots, x_9\}, \{x_2, x_4, \ldots, x_8\}\}$ is a refinement of that induced by f.

Given an MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, $\mathcal{S}$ is the ground or world state space. Let S be related to $\mathcal{S}$ via the surjection $\phi : \mathcal{S} \mapsto S$. Here, S is the *abstract* state space and every state s in the abstract state space defines an aggregate or an equivalence class over multiple ground states $\phi^{-1}(s)$.

We can extend the state abstraction theory developed for MDPs to a factored state representation. We first define the generic vector function $\Phi : \mathbb{R}^n \mapsto \mathbb{R}^m$ such that

$$\Phi(\mathbf{x} \in \mathbb{R}^n) = \big(\phi_1(\mathbf{x}), \ldots, \phi_m(\mathbf{x})\big), \text{ where } \phi_i : \mathbb{R}^n \mapsto \mathbb{R}.$$

In the context of an FMDP $\mathcal{M} = (\mathcal{X}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, the generic state-space projection is defined as $\Phi : \mathcal{D}(\mathcal{X}) \mapsto S_\Phi$ where S_Φ is some arbitrary abstract state space. Every projection Φ induces a partition of $\mathcal{D}(\mathcal{X})$ where $\forall \mathbf{x}_1, \mathbf{x}_2 \in \mathcal{D}(\mathcal{X})$, $\mathbf{x}_1 \equiv_\Phi \mathbf{x}_2 \iff \Phi(\mathbf{x}_1) = \Phi(\mathbf{x}_2)$. As a special case, if each ϕ_i simply picks out a unique component of $\mathbf{x}$, then $S_\Phi = \mathcal{D}(X_\Phi)$ where $X_\Phi = \{x_k : \exists i \, \phi_i(\mathbf{x}) = x_k\} \subset \mathcal{X}$. For example, if $\mathcal{X} = \{x_1, x_2, \ldots, x_9\}$ and $\Phi = (\phi_1(\mathbf{x}) = x_1, \phi_2(\mathbf{x}) = x_2)$, then $X_\Phi = \{x_1, x_2\}$. This subset projection is the most common form of state abstraction used in factored MDPs.

2.2 Hierarchical Reinforcement Learning

Reinforcement learning (RL) approaches to solving MDPs mitigate the state space explosion by leveraging the factored structure and reachability of the state space. However, they do not leverage structural constraints in the policy space. Hierarchical reinforcement learning (HRL) is a subfield of RL that combats the curse of dimensionality via hierarchical decomposition within the policy space of the problem. Not only does the decomposition allow the overall problem to be divided into smaller subproblems, it also facilitates faster learning through the reuse of solutions to shared subproblems and enables effective problem-specific state abstraction and aggregation. HRL applies temporal abstraction to the problem: decision-making should not be required at every step but instead temporally extended activities or macro-operators or behaviors or subtasks (which might have their own internal policies) can be selected to achieve subgoals. State abstraction is another powerful weapon in HRL's arsenal — the policy or value function of a certain subproblem only depend on the subset of state variables that actually affect the solution of that subproblem. Because they could depend on the agent's internal state as well, HRL policies could be non-Markovian with respect to the world state.

The three principal HRL frameworks are options (Sutton et al., 1999), HAMs/ALisp (Parr, 1998a; Andre and Russell, 2002), and MAXQ (Dietterich, 2000).

2.2.1 Options

Options lay down the minimal extension to RL allowing for a general treatment of temporally extended actions. An *option* is a closed-loop policy that operates over a period of time. It is defined by the tuple $\langle \mathcal{I}, \pi, \beta \rangle$, where π is its policy, $\mathcal{I} \subseteq \mathcal{S}$ is the initiation set of states, and $\beta(s)$ is the probability of termination in state s. The original MDP actions are also considered options that last exactly one time step. The original MDP along with a fixed set of options is an SMDP.

An option itself can select from any of the other available options as part of its internal policy. If the set of admissible options always contains the set of admissible actions, then the optimality of the policy over options in the core MDP remains unchanged; the absence of these actions could lead to suboptimal policies. In the original version of this framework, the internal policies of the options themselves were fixed a priori. However, this makes the agent's learning system depend heavily on expert knowledge and is brittle to changes. Intra-option learning methods were later introduced to analyze the inner workings of the options and learn their internal policies. Within the options framework, the agent's action set is augmented rather than simplified by the options, and could potentially lead to a blowup in action space. However, well-designed options prevent long-term random exploration within the domain.

2.2.2 Hierarchies of Abstract Machines and ALisp

The Hierarchies of Abstract Machines (HAMs) approach to reducing the number of decisions is by partially representing the policy in the form of finite state machines (FSMs) with nondeterministic choice points. HAMs also exploit the theory of SMDPs, but the emphasis is on restricting the policy space rather than augmenting the action space. A HAM policy is defined by a set of stochastic FSMs with internal states, stochastic transition functions, and a set of input states equal to that of the core MDP. A HAM has 4 types of states: ① action executes an action of the core MDP based on the state of both the core MDP and the HAM itself; ② call transfers control to an FSM within the HAM; ③ choice defines a set of FSM transitions from which the learned policy selects one; ④ stop terminates execution of the current FSM and returns control back to the FSM that called it. The composition of a HAM H and the core MDP $\mathcal{M}$ yields an SMDP $H \circ \mathcal{M}$. SMDP-Q-learning need only be applied at the choice states comprising the reduced SMDP. Just like options, these HAMs have to be expertly designed because they place strict restrictions on the final policy possible for the core MDP.

ALisp greatly generalizes the idea of HAMs to arbitrary Lisp programs with macro extensions to provide a platform for rich procedural prior knowledge with the ability to specify choice points that need to be determined through learning (Andre and Russell, 2002). This adds immense expressiveness in the form of interrupts, aborts, local state variables, parameter binding, and other generic programming language concepts for designing solutions to MDPs.

2.2.3 MAXQ

In the MAXQ framework, the temporally-extended actions or subtasks are organized in a task hierarchy $H = \{T_i : 0 \le i < m\}$, which is represented as a directed acyclic *task graph* that defines the task-subtask relationships (Dietterich, 2000). Unlike options and ALisp, the MAXQ framework does not reduce the entire original MDP into one SMDP. Instead, the original MDP $\mathcal{M}$ is replaced by induced sub-SMDPs, where each sub-SMDP $\mathcal{M}_i$ corresponds to a composite or primitive subtask T_i. Solving the root MDP $\mathcal{M}_0$ solves $\mathcal{M}$, and these sub-SMDPs can be solved simultaneously. The composite subtasks are primarily designed around subgoals (states or regions of the state space) such that achieving them facilitates the solution of the original MDP. Consequently, these sub-MDPs (with the possible exception of the root MDP) are episodic, that is, they have at least one absorbing or goal state. (The dynamics of an absorbing state s ensures that $\forall \pi$, $V^{\pi}(s) = 0$.)

A MAXQ task is defined as $T_i = (\Phi_i, G_i, C_i)$, where Φ_i is the state-space projection

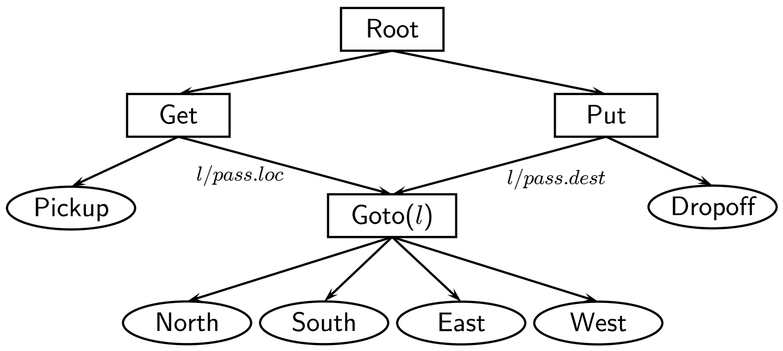


Figure 2.1: A manually-designed task hierarchy for the Taxi domain. The explicit binding for Goto's argument l to either the passenger's location (*pass.loc*) or destination (*pass.dest*) is part of the structural knowledge.

that induces a state space S_{Φ_i}, G_i is the goal or termination condition that determines when T_i has achieved its goal and is no longer applicable, and C_i is the set of child tasks. With slight abuse of notation, G_i is also used to denote the set of states $\{s \in S_{\Phi_i} : G_i(s) = \mathsf{true}\}$. For notational convenience, X_i denotes the subset of state variables when Φ_i is a subset projection and $S_i = S_{\Phi_i} - G_i$ denotes the set of nonterminal states. For a primitive task, $C_i = \{a \in \mathcal{A}\}$ and control always returns immediately back to the calling task after a is executed; as G_i is moot here, we define it to be empty. The order in which the children are represented in the task graph is arbitrary — it is the local policy $\pi_{T_i} : S_i \mapsto C_i$ within each subtask T_i that determines the selection of child tasks. (These local policies are normally represented and learned through Q functions.) A task hierarchy for the Taxi domain (Dietterich, 2000) is shown in Figure 2.1.

A hierarchical policy π for the overall hierarchy is an assignment of a local policy π_i to each sub-MDP $\mathcal{M}_i$. Any hierarchical policy with the highest possible utility is said to be *hierarchical optimal*. In order to maximize the modularity and reuse of context-free local policies, a *recursively optimal* policy is defined to be a hierarchical policy such that π_{T_i} is optimal for every T_i given the policies $\{\pi_c : c \in C_i\}$. The utility of such a policy could be suboptimal to that of a hierarchically optimal one, which could in turn be suboptimal to that of the optimal non-hierarchical policy.

Applying SMDP Q-learning to the MAXQ task hierarchy results in an algorithm called Hierarchical Semi-Markov Q (HSMQ) learning. Here, every task learns a local Q function simultaneously but independently, losing opportunities for sharing and compactness in the representation of the value function. The MAXQ value function decomposition rectifies this issue very elegantly. The Bellman equation for the decomposed action-value function at task T_i in the hierarchy is

$$Q_i^\pi(s,a) = V_a^\pi(s) + \sum_{s',d} \mathrm{Pr}_i^\pi(s',d|s,a)\gamma^d Q_i^\pi(s',\pi(s')),$$

where $V_a^\pi(s)$ is the expected discounted sum of rewards for executing the child task a starting in state s until it terminates. If the *completion* function is defined as

$$C_i^\pi(s,a) = \sum_{s',d} \mathrm{Pr}_i^\pi(s',d|s,a)\gamma^d Q_i^\pi(s',\pi(s')),$$

then the Q function can be expressed recursively as

$$Q_i^\pi(s,a) = V_a^\pi(s) + C_i^\pi(s,a),$$

where

$$V_i^\pi(s) = \begin{cases} Q_i^\pi(s,\pi_i(s)) & \text{if } T_i \text{ is composite} \\ \sum_{s'} \mathrm{Pr}(s'|s,i)\mathcal{R}(s,i,s') & \text{if } T_i \text{ is primitive.} \end{cases}$$

The corresponding learning algorithm is called MAXQ-0. Additionally, the MAXQ-Q algorithm can deal with pseudo rewards (not provided in the core MDP) designed by the expert to guide the learning of the individual subtasks, much like shaping rewards. The value function decomposition shown above extends beyond the MAXQ framework and has been modified by other researchers to ensure hierarchical optimality (Marthi et al., 2006) and safe abstraction in the face of discounting (Hengst, 2003).

Chapter 3: Multi-Agent Shared-Hierarchy Reinforcement Learning

This chapter introduces a multi-agent hierarchical reinforcement learning framework called Multi-Agent Shared-Hierarchy (MASH) framework. The central property of this framework is that it allows sharing of value functions between agents, facilitating faster learning. Sharing value functions across multiple agents was found to be effective in a non-hierarchical setting when the agents were homogeneous (Tan, 1993). However, a monolithic value function is not necessarily sharable across non-homogeneous agents as each agent may have a different reward function. Hierarchies divide the task into smaller subtasks, which makes them more amenable to sharing — the rewards within a subtask are more likely to be the same across different agents. Thus, if a robot has learned some useful navigational knowledge, a map for a region for example, it is much cheaper to communicate it to other robots, rather than having all robots learn it independently. Hierarchies make it possible to share the maps across different robots without also sharing the lower level controls or higher level task knowledge such as delivering mail. The packaging of scientific knowledge in short modular research papers has a similar underlying reason, namely high reusability of such knowledge and the low cost of its communication compared to its discovery.

This chapter has three principal parts. It introduces the multi-agent shared hierarchy (MASH) framework which facilitates selective sharing of subtask value functions across agents. It then describes how a previously published model-based hierarchical RL algorithm can be improved and generalized to the MASH framework. Finally, it presents empirical results in multiple versions of a simplified multi-agent real-time strategy (RTS) domain. The results show that, with suitable coordination information, the new MASH algorithm is more scalable with respect to multiple agents and outperforms previous approaches.

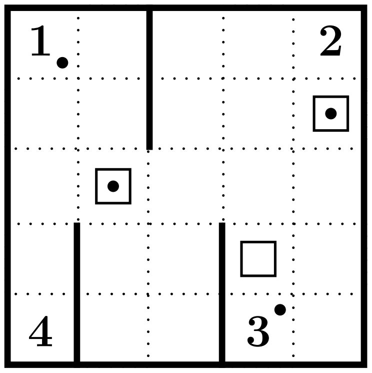


Figure 3.1: The multi-agent Taxi domain. The square boxes represent taxis, the dots represent passengers, and the numbers are the passenger pickup and drop-off sites.

3.1 Preliminaries

This section describes the multi-agent domains employed in this chapter and outlines the hierarchical average-reward learning (HARL) framework.

3.1.1 Multi-Agent Taxi Domain

The multi-agent version of the Taxi domain is a grid world, shown in Figure 3.1, where the agents are taxis (the squares located on the grid cells) that shuttle passengers (solid circles) from one of four marked cells (labeled 1, 2, 3, and 4) to their intended destinations (again, one of the four marked cells). At any time, there can be at most one passenger waiting at each of these special sites. The generation of passengers can be controlled in different ways, including those based on arrival probabilities. The passengers may also depart stochastically from the sites without being transported.

A full state description of this domain for n agents has 25^n values to indicate the taxi locations, 5^n values indicating the statuses of all the taxis (empty or an onboard passenger with a particular destination), and similarly 5^4 values indicating the statuses of the special sites. Hence, the size of the state space $|\mathcal{S}| = 25^n \times 5^n \times 5^4 = 5^{3n+4}$. The primitive actions available to each agent are moving one cell to the North, South, East, and West, Pickup a passenger, Dropoff the passenger, and Idle (no-op). The Idle action allows the agent to idle when there are no passengers currently waiting to be transported.

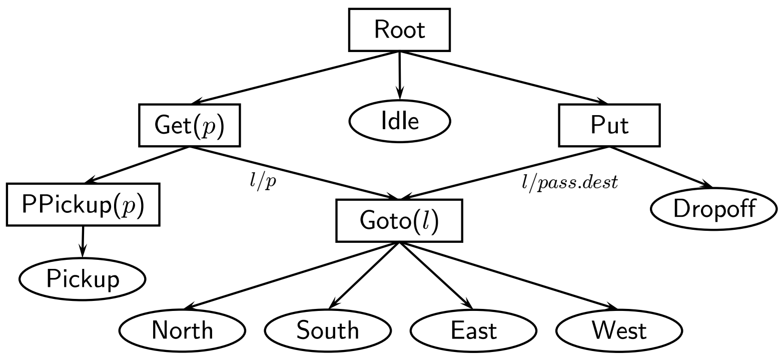


Figure 3.2: The task hierarchy for the multi-agent Taxi domain.

The task hierarchy for the multi-agent Taxi domain is shown in Figure 3.2. Get is parameterized by the passenger selected to be serviced, and the parameter binding needs to be learned by Root. Get(p) is terminated when either p is picked up by a taxi or p disappears from the grid. In turn, Get passes its intention (parameter binding) down to PPickup whose sole purpose is to ensure that no other passenger is picked up inadvertently, which can happen if the Pickup action could be executed at an occupied site before navigating to the selected passenger's site. It does this by being terminated in all states in which the taxi is not at the selected passenger's location.

3.1.2 Multi-Agent Real-Time Strategy Domain

We consider a simplified RTS domain shown in Figure 3.3. It is a grid world that contains peasants, the peasants' home base, resource locations (forests and goldmines) where the peasants can harvest wood or gold, and an enemy base that can be attacked. The primitive actions available to a peasant are moving one cell to the North, South, East, and West, Pick a resource, Put a resource, Strike the enemy base, and Idle (noop). The resources get regenerated stochastically. The enemy also appears stochastically and stays till it has been destroyed.

The task hierarchy for the RTS domain (Figure 3.4) is composed of the following tasks: Harvest(l) represents the subtask of going to a resource location l (goldmines/forests) and picking up the resource; Deposit represents the subtask of dropping off a harvested resource at the home base; Attack represents the subtask of suppressing

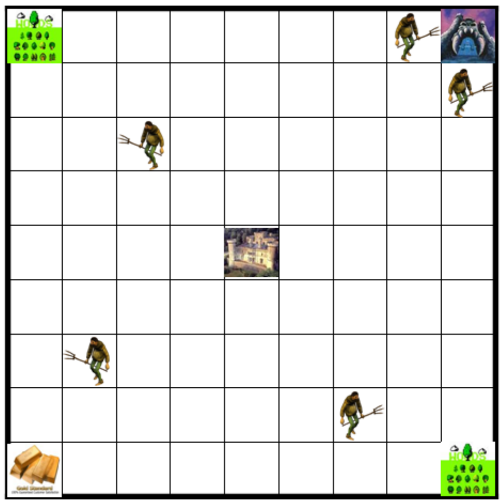


Figure 3.3: A simplified RTS domain with two resources, a home base, and an enemy.

the enemy whenever it appears; Goto(k) represents the subtask of going to location k on the grid.

3.1.3 HARL Framework

SMDP H-learning can be directly extended to task hierarchies in a couple of ways: one scheme is to employ SMDP H-learning only at the infinite-horizon root task and employ total-reward learning at all the other finite-horizon subtasks; the other scheme, in the spirit of HSMQ-learning, is to employ SMDP H-learning at every subtask (Ghavamzadeh and Mahadevan, 2001). While these schemes allow the subtask value functions to be context-independent and hence more reusable, they fail to find the hierarchically optimal policy even in very simple SMDPs.

The HARL framework is an adaptation of the MAXQ framework to the average-reward criterion (Seri and Tadepalli, 2002). Here, a hierarchical policy π is *recursively gain-optimal* if the local policy π_i at each subtask T_i maximizes its total expected average-adjusted reward with respect to the gain of the overall hierarchical policy, given fixed policies for T_i's descendants. The *result distribution invariance* (RDI) property holds for a task hierarchy if the probability distribution of termination states for any subtask is independent of the hierarchical policy used during that subtask. (A sufficient condition

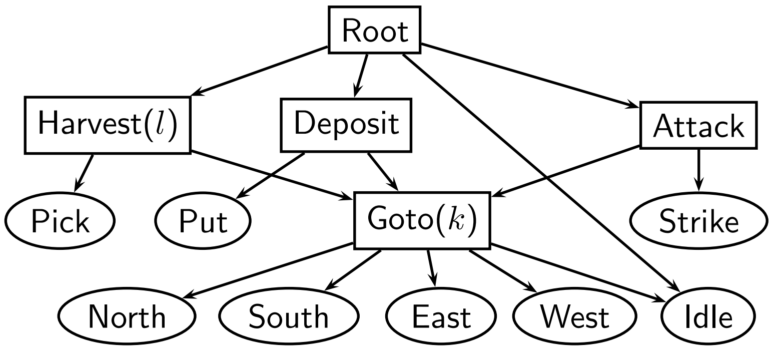


Figure 3.4: The task hierarchy for the RTS domain.

for RDI is when every subtask has a unique terminal state.) The recursively gain-optimal policy is also guaranteed to be optimal among all hierarchical policies when RDI holds.

The associated hierarchical H-learning (HH) algorithm leverages value-function decomposition by learning the value function of a task based on the value functions of the children. However, the value function at every task caches the values of the child tasks instead of computing them recursively as done in MAXQ's value decomposition. The global average reward ρ is computed at the leaves of the hierarchy, and the relativized bias values percolate up through the hierarchy.

3.2 Modified HARL Framework

The advantage of defining ρ globally with respect to the hierarchy in the HARL framework is that the recursively gain-optimal policy is also guaranteed to be optimal among all hierarchical policies when RDI holds. However, the tasks lose their context-independence and transferability, because every task's value function is affected by ρ and consequently depends on the global hierarchical policy. Moreover, as the update for ρ is not adjusted based on the overall value function as is done in SMDP H-learning, the value function at the root has been observed to diverge for non-episodic domains.

To facilitate reusability while guaranteeing hierarchical optimality when RDI holds, we follow a different strategy. Instead of storing the bias values for each task, we only store the expected total-reward function $V_i(s)$ and the expected total-duration function $D_i(s)$ for each task T_i. Given these two functions and an estimate of the global gain ρ, we can easily compute the bias for task T_i and state s as $V_i(s) - \rho\, D_i(s)$. As the expected total reward and time do not depend on ρ, these functions are context-independent and reusable as long as the local policy does not change; at the same time, the bias values are computable from them with respect to any ρ. We compute the global average reward at the root and utilize it for the bias and when choosing an action. Hence, the value function decomposition for a recursively gain-optimal policy satisfies the following set of Bellman equations for the non-root nodes

$$V_i(s) = \begin{cases} R(s,a) & \text{if } T_i \text{ is the primitive action } a \\ 0 & \text{if } s \text{ is a terminal state for } T_i \\ V_g(s) + \sum_{s' \in \mathcal{S}} \Pr(s'|s,g) V_i(s') & \text{otherwise} \end{cases}$$

$$D_i(s) = \begin{cases} D(s,a) & \text{if } T_i \text{ is the primitive action } a \\ 0 & \text{if } s \text{ is a terminal state for } T_i \\ D_g(s) + \sum_{s' \in \mathcal{S}} \Pr(s'|s,g) D_i(s') & \text{otherwise,} \end{cases}$$

where

$$g = \operatorname*{argmax}_a \Big(V_a(s) - \rho\, D_a(s) + \Pr(s'|s,a)\big(V_i(s') - \rho\, D_i(s')\big)\Big)$$

is the greedy subtask that maximizes the sum of the relativized bias of a child task and the bias of the task T_i from the resulting state. Because the total reward is unbounded at the root task and no parent tasks exist, we compute the biased value function as

$$V_{\text{root}}(s) = \max_a \left(V_a(s) - \rho\, D_a(s) + \sum_{s' \in \mathcal{S}} \Pr(s'|s,a) V_{\text{root}}(s') \right).$$

3.3 MASH Framework

Multi-agent RL is more challenging than single-agent RL because of two complementary reasons. Treating the multiple agents as a single agent increases the state and action spaces exponentially. On the other hand, treating the other agents as part of the environment makes it nonstationary and non-Markovian (Mataric, 1997).

We define the execution semantics of the multi-agent task hierarchy as follows. (The multi-agent semantics can be seen as a special case of the execution semantics of concurrent ALisp (Marthi et al., 2005).) The global Markov state of the system is defined by the world state w and the set of parameterized task stacks G of all the agents in the system. At every decision epoch, each agent at an internal task node is assigned to a child subtask. The agents at the leaf nodes wait until all agents are at their leaf nodes and then simultaneously execute their primitive actions, which affect the world state. The set of all global Markov states and the cross product of all the action assignments form a *task-management* MDP. Because the task-management MDP is well-defined, it has a joint gain-optimal policy. However, solving for this joint gain-optimal policy is computationally difficult, because it involves the global state and the joint action space. In previous work, it has been found that decomposing a joint MDP into weakly interacting sub-MDPs, solving them offline, and combining the solutions online yields effective solutions (Meuleau et al., 1998). We instead explore a completely on-line multi-agent approach described next.

3.3.1 Multi-agent HH

We factor the world state w into several agent-centric states s_z (e.g., an agent's location) for agent z, and a global state q (e.g., the availability of the resources), which is independent of all agents. We similarly decompose the reward into agent-centric rewards and learn the total expected reward of each agent z for each task a as a function of the combined state information s_z and q, appropriately abstracted for task a. The total reward for task T_i in the Markov state s when all agents are following a current joint policy π is given by $V_i(s) = \sum_z V_i^z(s_z, q)$, where V_i^z is the total expected reward of agent z for task T_i in the context of the joint policy π. Each agent z now learns a separate value function V^z and average reward, and has a separate task stack. V^z is z's value function in the context of the joint policy π, which is in turn determined by each agent choosing the actions greedily with respect to their own value functions. It is not guaranteed to be a globally optimal joint policy because the policies of the agents are not truly independent and the environment is no longer stationary due to all agents learning simultaneously. Although there is no explicit coordination here, implicit coordination emerges because the learned models within each agent's hierarchy reflect the effects of the other agents on the global state variables. For instance, keeping track of the probabilities of wood disappearing from the forest allows an agent to detect the effects of the other agents.

3.3.2 Coordinated HH

A more sophisticated extension to the method described above is to include some kind of explicit coordination information (Makar et al., 2001). One approach to coordinating is to make a large part of the world state w a part of the global state q. More generally, we allow each task node to define appropriate coordination variables that specify the state and task variables of the other agents which are relevant for making decisions for the current agent. We call this approach Coordinated HH. For example, in the RTS game domain, the top-level tasks of the agents make good coordination variables. If c represents the coordination variables at task T_i for agent z, we have the following Bellman backup at any task T_i:

$$V_i^z(s_z, q, c) \leftarrow \max_a \left(V_a^z(s_z, q, c) + \sum_{s'_z, q', c'} \Pr_i(s'_z, q', c' | s_z, q, c, a) V_i^z(s'_z, q', c') \right).$$

The V^z values here are with respect to a current joint policy of all agents, which is determined by each agent choosing its greedy action with respect to the equation above. This entails that each agent takes into account the coordination information of the other agents. To do this efficiently, we process the agents in a fixed order, where each agent uses the coordination information of the previous agents who already chose their actions. Given an order of the agents, we call a joint policy *serially optimal* if each agent's action is optimal given the choices of the previous agents. The goal of the Coordinated HH is to converge to a serially optimal policy.

Further optimization in Coordinated HH is obtained by anonymizing the other agents referred to in the coordination variables. For example, one would expect that the state in which agent 1 is collecting wood and agent 2 is collecting gold has the same value for agent 3 as the state in which agent 1 is collecting gold and agent 2 is collecting wood. It suffices to know that an agent is collecting gold and another is collecting wood. Thus, we reduce the state space of the coordination variables by anonymizing the agents. In particular, if we have n agents, the root task has m subtasks, and the root's subtasks selected by the agents are the coordination variables, then the root task's state space would grow by a factor of $(m+1)^{n-1}$. However, anonymizing the agents reduces this factor to $\binom{(n-1)+(m+1)-1}{n-1}$.

3.3.3 Shared Value Functions

The final optimization we make is to share the value functions across agents.[1] The compelling justification for sharing is that all agents are interacting with the same environment and every agent sees an isomorphic view of the world involving its own state variables s_z, the global variables q, and the coordination information c from the other agents. This perspective allows for subtask sharing across agents. MASH extends Coordinated HH by recombining the separate agent value functions into one.

Although the value functions are shared among the agents, every agent has its own independent task stack, and keeps track of its own average reward (global across all tasks) within the system. The agents pool their experiences together to more efficiently learn the value function of every subtask. For instance, the more agents we have moving around, the quicker the $\mathsf{Goto}(k)$ task is learned. The state abstraction and termination conditions within the task hierarchy are parameterized by the agents' identities and the subtask parameters. For example, $\mathsf{Harvest}(l)$ subtask's abstraction involves only looking at the resource available at location l and the agent's location.

Sharing the value functions precludes implicit coordination (as in Multi-agent HH), because each agent uses exactly the same value function. For example, all agents would rush to kill the enemy if the enemy appears. Hence, the MASH framework needs the explicit coordination scheme described in the previous section. In the experiments, the coordination information is in terms of the top-level subtasks of the other agents.

3.3.4 MASH Algorithm

Algorithm 3.1 is the pseudo-code for the MASH learning algorithm. The code is presented from one agent's standpoint. Every agent in the domain executes this code concurrently and only needs to synchronize with the other agents at two points within the algorithm (lines 3, 11). The actual execution is a little more complicated than the standard stack-based recursive execution shown here. The algorithm always scans the entire stack for terminated subtasks, starting at the root level. When the first terminated subtask is found, it is popped off along with all of its descendants.

In this algorithm, we use s to denote the state (s_z, q, c) which includes the agent-

[1]Although the MASH framework allows for the selective sharing of tasks across agents, this chapter only considers the case where the entire hierarchy is shared.

Algorithm 3.1 MASH

```
Input: Task $T_i$, agent-centric state $s_z$, agent-independent state $q$
Output: Resultant state $s'$
 1: $s \leftarrow (s_z, q, c)$  // $c \leftarrow$ coordination information
 2: if $T_i$ is primitive then
 3:    Wait for all agents to select their primitive actions
 4:    Execute primitive action
 5:    Observe reward $r_z$, elapsed time $d_z$, next state $s'_z$
 6:    Update transition, reward, and duration models
 7: else
 8:    while not TERMINATED$(T_i, s)$ do
 9:       $a \leftarrow$ exploratory or greedy subtask according to Equation 3.1
10:       $s' \leftarrow$ MASH$(a, s_z, q)$
11:       Synchronize and gather coordination info in $c'$
12:       $s' \leftarrow (s'_z, q', c')$
13:       Update transition probability model $\mathrm{Pr}_i(s'|s, a)$
14:       if $T_i \neq$ Root task then
15:          Update the time model $D_i(s)$ according to Equation 3.2
16:          Update $V_i(s)$ according to Equation 3.3
17:       else
18:          if $a$ was greedily selected then
19:             Update $\rho_z$ according to Equations 3.4, 3.5, 3.6
20:          Update $V_{\text{root}}(s)$ according to Equation 3.7
21:       $s \leftarrow s'$
22:    $V_i(s) = 0$
23: return $s'$
```

centric variables s_z, global variables q, and the coordination variables c (line 1). In lines 4–6, the primitive subtask updates its reward and time models after the primitive action has been executed. The transition model is updated by keeping counts of the occurrences of (s, a) and (s, a, s'); the reward and duration models are updated via moving averages. In line 9, the composite subtasks use an ε-greedy exploration strategy, that is an exploratory action is chosen with a fixed probability (8% in all our experiments), otherwise the average-adjusted greedy action is chosen according to

$$g \leftarrow \underset{a \in C_i(s)}{\operatorname{argmax}} \left(V_a(s) - \rho_z \, D_a(s) + \sum_{s' \in \mathcal{S}_i} \mathrm{Pr}_i(s'|s, a) \left(V_i(s') - \rho_z \, D_i(s') \right) \right). \tag{3.1}$$

This algorithm is called recursively for the selected subtask a, and the resulting state s' (when the control returns to the calling task) is observed (line 10). After the agents synchronize, the appropriately abstracted next state is computed (lines 11–12). The composite subtask then updates its transition probability model (line 13).

If the composite task being executed is not the root task, we update the time model with respect to the greedy action g in the current state (line 15).

$$D_i(s) \leftarrow D_g(s) + \sum_{s' \in \mathcal{S}_i} \mathrm{Pr}_i(s'|s,g)\, D_i(s') \tag{3.2}$$

We update the value of s to reflect the expected total reward obtained by selecting child task g in s, and following the task's policy thereafter until completion (line 16).

$$V_i(s) \leftarrow V_g(s) + \sum_{s' \in \mathcal{S}_i} \mathrm{Pr}_i(s'|s,g)\, V_i(s') \tag{3.3}$$

If the task being currently executed is the root task, and the child task a was selected greedily, we update ρ_z (lines 18–20).

$$\text{avg-reward}_z \overset{\alpha_z}{\leftarrow} V_a(s) + V_{\text{root}}(s') - V_{\text{root}}(s) \tag{3.4}$$

$$\text{avg-time}_z \overset{\alpha_z}{\leftarrow} D_a(s) \tag{3.5}$$

$$\rho_z \leftarrow \text{avg-reward}_z / \text{avg-time}_z \tag{3.6}$$

The expression $V_{\text{root}}(s') - V_{\text{root}}(s)$ in eqn. 3.4 nullifies the effects of the exploratory actions. Next, we update the root's average-adjusted value function (line 21).

$$V_{\text{root}}(s) \leftarrow \max_{a \in C_{\text{root}}} \left(V_a(s) - \rho_z\, D_a(s) + \sum_{s' \in \mathcal{S}} \mathrm{Pr}(s'|s,a)\, V_{\text{root}}(s') \right) \tag{3.7}$$

The current task continues until a termination state is reached. The value of all termination states = 0 (line 25).

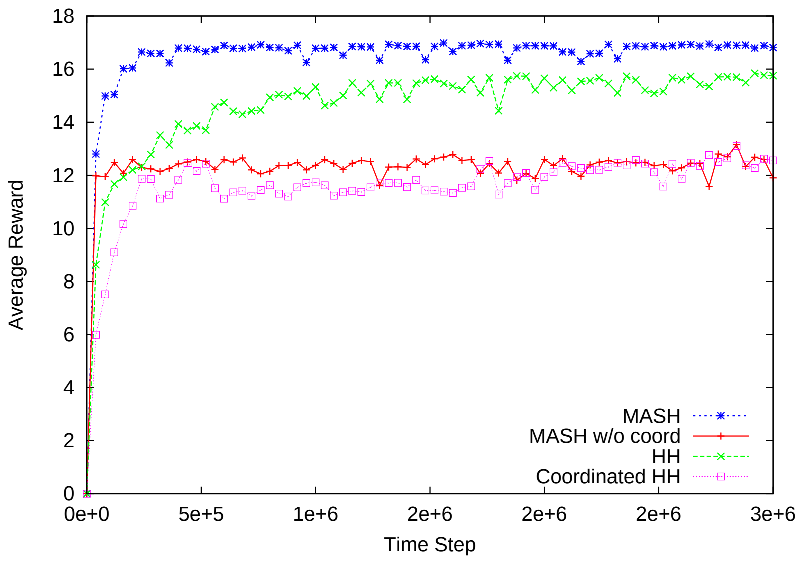


Figure 3.5: Performance in the 5×5 3-agent Taxi domain.

3.4 Empirical Evaluation

In the multi-agent Taxi domain described in Section 3.1.1, stochastic passenger generation does not require a lot of coordination between agents because even when two agents try to pick up a passenger from one site, the unsuccessful agent just has to wait a few time steps till the next passenger is generated. To necessitate coordination, the passenger generation is as follows: a destination site is picked and passengers headed to that destination are generated; no new passengers are generated until all passengers have been dropped off, after which a new destination site is selected, and so on. Every action has a 95% probability of success; it has no effect otherwise. There is a default reward of -0.1 for every action to allow the total-reward subtasks to optimize their policies, and a reward of $+100$ for a successful drop-off. All results shown here are averaged across 30 independent runs.

Figure 3.5 shows the performance of the algorithms in the 5×5 3-agent Taxi domain. The MASH algorithm learns the fastest and converges to the optimal policy, while the

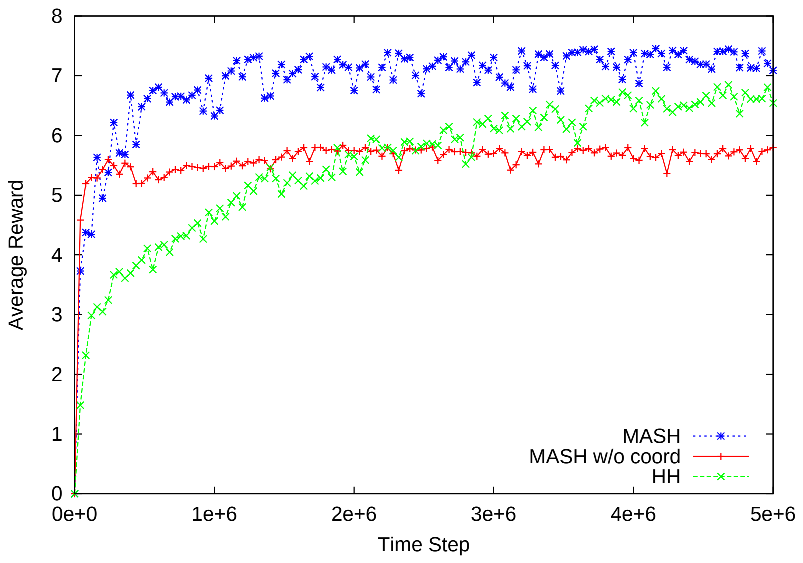


Figure 3.6: Performance in the 10×10 3-agent Taxi domain.

version without coordination converges to a sub-optimal policy. The Coordinated HH algorithm, slowed down by the sheer volume of values it must learn, improves very gradually, but does reach the performance of MASH without coordination. The Multi-agent HH algorithm does better than MASH without coordination, but converges to a slightly suboptimal policy. Figure 3.6 shows the performance of the algorithms in the 10×10 version of the 3-agent Taxi domain, demonstrating that the MASH framework scales well with the size of the domain. Coordinated HH cannot be run here because of its exorbitant demand for memory.

Figure 3.7 shows the performance of the algorithms in the 15×15 2-agent RTS domain described in Section 3.1.2. Again, the MASH algorithm learns the fastest and converges to the best policy. Both HH & Coordinated HH improve very gradually. Figure 3.8 shows the performance of the algorithms in the 25×25 version of the 4-agent RTS domain, demonstrating that the MASH framework scales very well with the size of the domain, but Coordinated HH requires too much space to even run. The larger number

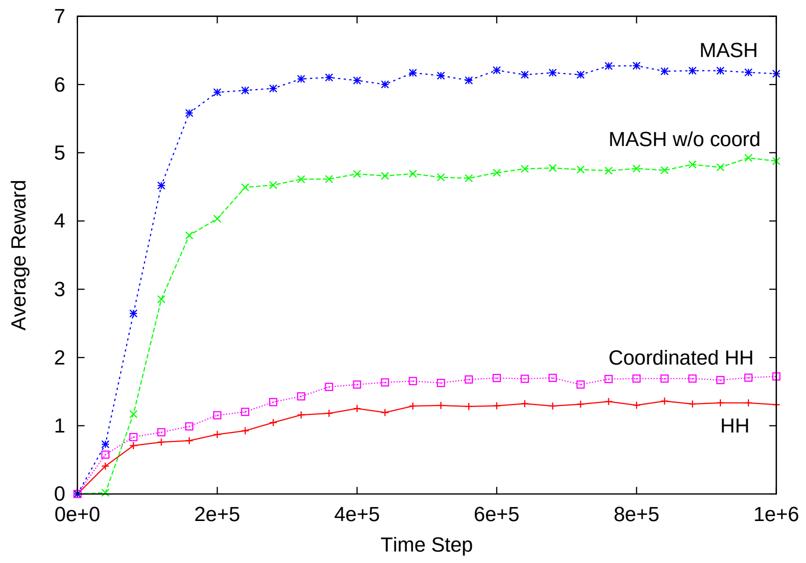


Figure 3.7: Performance in the 15 $\times$ 15 2-agent RTS domain.

of agents necessitates more coordination as evidenced by the fact that MASH without coordination does much worse than HH.

3.5 Conclusion

This chapter has presented a multi-agent HRL framework called the MASH framework. This framework allows agents coordinating in a domain to share their hierarchical value functions to be able to learn more effectively. With suitable coordination information, this framework can greatly boost learning in multi-agent domains as shown by the results in the multi-agent RTS domain. Moreover, the framework is extensible to sharing only portions of the task hierarchy (certain subtasks) across multiple agents with the constraint that if a task is shared, then all its descendants must be shared as well; this is required to ensure a consistent value function decomposition.

MASH does not consider the cost of communication between agents. Ghavamzadeh

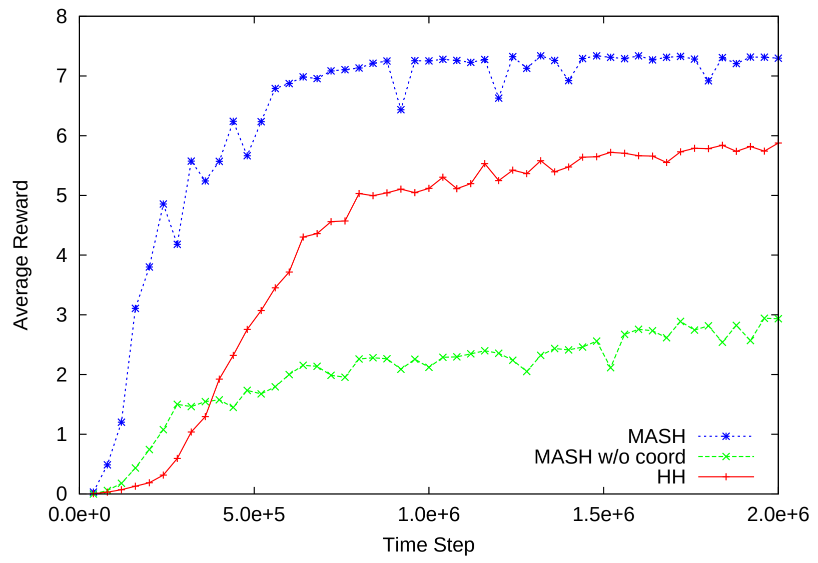


Figure 3.8: Performance in the 25 × 25 4-agent RTS domain.

and Mahadevan (2004) present an approach that treats communication of coordination information as explicit decisions to be made by the agents. However, unlike the coordination information, communicating the value function is not time-sensitive — all agents can record their experience tuples and update the shared value function at a later time. Intuitively, the cost of offline communication of value functions seems negligible compared to the reduction in training time achieved due to sharing.

Chapter 4: Variable-Reward Hierarchical Reinforcement Learning

Most work in transfer learning is in the supervised setting where the goal is to improve classification performance by exploiting inter-class regularities. This chapter considers transfer learning in the context of RL, that is, learning to improve performance over a family of semi-Markov decision processes (SMDPs) that share some structure. In particular, it introduces the *variable-reward transfer learning* problem where the objective is to speed up learning in a new SMDP by transferring experience from previous MDPs that share the same dynamics but have different rewards. More specifically, reward functions are weighted linear combinations of reward features, and only the reward weights vary across different SMDPs. This problem is formalized as converging to a set of policies which can solve any SMDP drawn from a fixed distribution in close to optimal fashion after experiencing only a finite sample of SMDPs.

SMDPs that share the same dynamics but have different reward structures arise in many contexts. For example, while driving, different agents might have different preferences although they are all constrained by the same physics (Abbeel and Ng, 2004). Even when the reward function can be defined objectively (e.g., winning as many games as possible in chess), usually the experimenter needs to provide other shaping rewards (such as the value of capturing a pawn) to facilitate efficient exploration. Any such shaping reward function can be viewed as defining a different SMDP in the same family. Reward functions can also be seen as goal specifications for agents such as robots and Internet search engines. Alternatively, different reward functions may arise externally based on difficult-to-predict changes in the world, e.g., rising gas prices or declining interest rates that warrant lifestyle changes. There is a large literature on multi-criteria decision problems where each criterion corresponds to an individual reward signal, and one of the standard ways of approaching this problem is to solve these decision problems with respect to a linear combination reward signal. An example of such a reward decomposition would be in a logistics domain where there are trade-offs between fuel consumption, delivery time, and number of drivers. Different companies might have different coefficients for these items and would require different solutions. Another example is that of trading where the prices of different commodities might change from day to day but the actions involved in trading them all have the same dynamics.

While this chapter focuses on the average-reward learning criterion, the general approach can be easily adapted to both the discounted and the total-reward settings. The key insight behind the work is that the value function of a fixed policy is a linear function of the reward weights. Variable-Reward Reinforcement Learning (VRRL) takes advantage of this fact by representing the value function as a vector function whose components represent the expectations of the corresponding reward features that occur during the execution of the policy. Given a new set of reward weights and the vectored value function of a policy stored in a cache (value function repository), it is easy to compute the value function and average reward of that policy for the new weights. VRRL initializes the value function for the new weights by comparing the average rewards of the stored policies and choosing the best among them; it then uses a vectorized version of an average-reward learning algorithm to further improve the policy for the new weights. If the average reward of the policy is improved by more than a satisfaction constant γ via learning, then the new value function is stored permanently in the cache. We derive an upper bound for the number of policies that will be stored in the worst case for a given γ and the maximum values of different reward weights.

This chapter extends VRRL to the MAXQ framework where a single task hierarchy is used for all SMDPs. Vectored value functions are stored in the tasks and represent the task policies. Given a new weight vector, our method initializes the value functions for each task by comparing the overall average rewards for the stored value functions and choosing the best. If the average reward improves during learning, then every task whose value function changes significantly is cached. In this hierarchical version, we expect the task policies to be optimal across a wide spectrum of weights, because the number of reward components affecting a particular task's value function is normally less than that affecting the non-hierarchical value function, and this results in significant transfer.

The approach is demonstrated empirically in a simplified real-time strategy (RTS) game domain. In this domain, a peasant must accumulate various resources (wood, gold, etc.) from various locations (forests, gold mines, etc.), quell any enemies that appear inside its territory, and avoid colliding with other peasants. The reward features are associated with bringing in the resources, damaging the enemy, and collisions. The actual value of these features to the agent is determined by the feature weights, which are different in each SMDP. The goal is to learn to optimize the average reward per time step. We show that there is significant transfer in both flat and hierarchical settings. The transfer is much more prominent in the hierarchical case, mainly because the overall task is decomposed into smaller subtasks which are optimal across a wide range of weights.

To cope with the huge state space (up to 3.7×10^{16} states), value-function approximation is employed within the hierarchical structure. Not only does the combination of hierarchical task decomposition and value function approximation allow our method to be applicable to a broad class of problems, but features for linear function approximation are easier to design for a decomposed value function than for a non-hierarchical one. Finally, this chapter also shows that standard perceptron-style learning can be used to induce the weight vector from a scalar reward function and the reward features.

4.1 Variable-Reward Transfer Learning

This section introduces the *variable-reward transfer learning* problem and presents an approach to solve it in a non-hierarchical setting.

4.1.1 Variable-Reward Transfer Learning Problem

A family of variable-reward SMDPs is defined as $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathbf{f})$, where $\mathcal{S}, \mathcal{A}, \mathcal{P}$ are the same as before, but $\mathbf{f}$ is an n-dimensional vector of binary reward feature components $\langle f_1, \ldots, f_n \rangle$. A weight vector $\mathbf{w} = \langle w_1, \ldots, w_n \rangle$ specifies a particular SMDP $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ in the family where $\mathcal{R} = \mathbf{w} \cdot \mathbf{f}$. Hence, n denotes the size of the minimal specification of any SMDP in the family, and is called the dimension of the SMDP family. The binary feature vector[1] indicates whether or not a particular component is active for (s, a); the weight vector's components are arbitrary real numbers that indicate the importance of the corresponding feature components. All the SMDPs in the variable reward family share the same states, actions, state transition function, and the expected execution times of actions, but may have different reward functions based on the particular weight vector. The basic transfer protocol is that the agent sees a stream of SMDPs from the family of variable-reward SMDPs; for every new SMDP, it is provided the associated

[1]The restriction to binary vectors is for explanatory purposes only; in practice, this could be an arbitrary real-valued vector.

weight vector and the agent is allowed to learn until convergence.

We now proceed with a formal definition of Variable Reward Transfer Learning, beginning with some preliminary definitions.

Definition 4.1. A *variable-reward transfer learning problem* is a pair $(\mathcal{F}, \mathcal{E})$, where $\mathcal{F}$ is a variable-reward SMDP family and $\mathcal{E}$ is an unknown distribution over weight vectors.

$\mathcal{E}$ defines a distribution over SMDPs in the family. The goal of the transfer learner is to quickly find an optimal policy for SMDPs drawn from the family (based on the weight vectors) given an opportunity to interact with it by taking actions and receiving rewards. A simple strategy would be to treat each SMDP independently and apply some reinforcement learning algorithm to solve it. While this approach cannot be improved upon if we need to solve only one SMDP in the family, it is inefficient when we are interested in solving a sequence of SMDPs drawn from the distribution over the SMDP family.

After solving a sequence of SMDPs drawn from the distribution, it would seem that we should be able to transfer the accumulated experience to a new SMDP drawn from the same distribution. Because all the SMDPs in the family share the same $\mathcal{P}$ and D, an obvious improvement over solving each SMDP separately is to learn models for the state-transition function $\mathcal{P}$ and the action-duration function D (collectively called the action models) and share them across different SMDPs. In the next section, we will see that even more efficient approaches are possible.

Definition 4.2. A *γ-optimal policy* is any policy π whose gain is at most γ less than that of the optimal policy, that is, $\rho^\pi \geq \rho^* - \gamma$.

Definition 4.3. An *ϵ, γ-approximate cover* for a variable-reward transfer learning problem $(\mathcal{F}, \mathcal{E})$ is a set of policies $\mathcal{C}$ such that, given an SMDP M chosen from $\mathcal{F}$ according to $\mathcal{E}$, $\mathcal{C}$ contains a γ-optimal policy for M with probability at least $1 - \epsilon$.

We refer to an ϵ, γ-approximate cover as an ϵ, γ-cover for short. This is an approximation in two senses. First, the policies in the cover might be up to γ worse than optimal. Second, there is some probability ϵ that the cover does not contain a γ-optimal policy for an SMDP in the family $\mathcal{F}$. Ideally, we would like a variable-reward transfer learner to produce an ϵ, γ-cover for any problem $(\mathcal{F}, \mathcal{E})$ after learning from a small number of training SMDPs drawn according to $\mathcal{E}$. However, as the training SMDPs are randomly chosen, we allow the learner to fail with a small probability δ. This is similar to the probably approximately correct (PAC) framework, where we only guarantee to find an approximately correct hypothesis with high probability.

Definition 4.4. A learner L is a *finite-sample transfer learner* if, for any variable-reward transfer learning problem $(\mathcal{F}, \mathcal{E})$ of dimension n and any set of parameters ϵ, γ and δ, there exists a finite bound $F(\epsilon, \delta, \gamma, n)$ such that, with probability at least $1 - \delta$, L finds an ϵ, γ-cover for $(\mathcal{F}, \mathcal{E})$ after learning from $F(\epsilon, \delta, \gamma, n)$ SMDPs drawn according to $\mathcal{E}$. In this case, we say that the learner has *sample complexity* $F(\epsilon, \delta, \gamma, n)$. Further, if $F(\epsilon, \delta, \gamma, n)$ is polynomial in n, $\frac{1}{\gamma}$, $\frac{1}{\epsilon}$, and $\frac{1}{\delta}$, then L is a *sample-efficient* transfer learner. If the run-time of L is polynomial in n, $\frac{1}{\gamma}$, $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, and the sample size $F(\epsilon, \delta, \gamma, n)$, then L is a *time-efficient* transfer learner.

The above definition can be generalized to a variety of SMDP families that share different types of properties. This chapter is restricted to the variable-reward family.

4.1.2 Variable-Reward Reinforcement Learning

Our approach to variable-reward reinforcement learning (VRRL) exploits the structure of the variable-reward SMDP family by caching value functions and reusing them. We begin with the following theorem which states that the value function for any fixed policy is linear in its reward weights.

Theorem 4.1. *Let $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ be an SMDP in the variable-reward SMDP family with a reward weight vector $\mathbf{w}$ and π be a policy. The gain ρ^π is $\mathbf{w} \cdot \boldsymbol{\rho}^\pi$, and the bias of any state s is $h^\pi(s) = \mathbf{w} \cdot \mathbf{h}^\pi(s)$, where the i^{th} components of $\boldsymbol{\rho}^\pi$ and $\mathbf{h}^\pi(s)$ are the gain and bias respectively with respect to the i^{th} reward feature for the policy π.*

Proof. The result follows directly from the fact that the immediate rewards for the SMDP are linear in the reward weights, and the bias and the average rewards are based on the sums of the immediate rewards. □

Policies can be represented indirectly as a set of parameters of these linear functions, that is, the gain and the bias functions are learned in their vector forms, where the components correspond to the expected value of reward features when all the weight

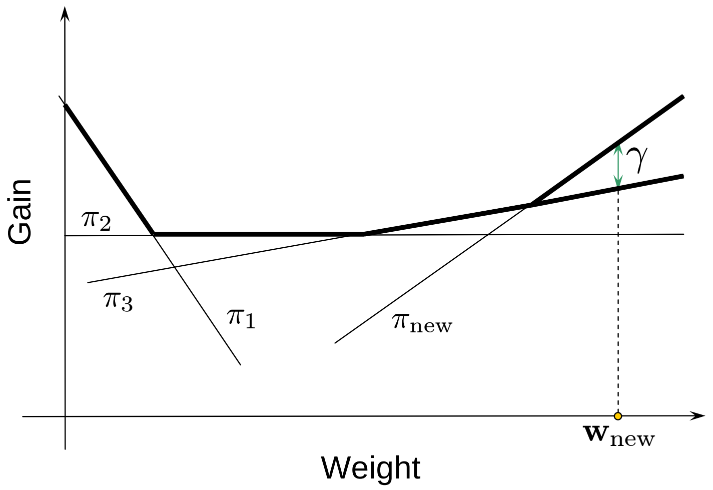


Figure 4.1: Policy plot for a 1-component weight vector. Every line in the above plot represents a single policy whose gain varies linearly with the reward weights. The dark lines represent the gain of the optimal policy as a function of weight.

components are 1. In the following section, we show that the set of optimal policies for different weights forms convex and piecewise linear gain and bias functions. If a single policy is optimal for different sets of weights, it suffices to store one set of parameters representing this policy.

4.1.2.1 Algorithm Description

To understand the intuition behind our approach, consider a gain vector $\boldsymbol{\rho}$ for a particular policy. Plotting $\mathbf{w} \cdot \boldsymbol{\rho}$ with respect to the weight components of $\mathbf{w}$ would result in a hyperplane. Every policy generates one such hyperplane in the weight space as demonstrated in Figure 4.1 for the case of a single-component weight vector.

The bold piecewise linear and convex function in the figure represents the best weighted gain for each possible weight. Extended to multiple dimensions, this becomes a convex piecewise planar surface[2]. Thus, when a new weight is considered, the learner might start off with the policy that registers the highest weighted average-reward, repre-

[2]The reasoning here is exactly the same as in the POMDPs, where the value function is a convex piecewise linear function over the belief states (Kaelbling et al., 1998).

sented by a point in the highlighted convex function in Figure 4.1. Initializing the value function with that of the dominant policy for the current weight vector will assure that the agent would learn a policy that is at least as good.

Let $\mathcal{C}$ represent the set of all optimal policies which are currently stored. Given a new weight vector $\mathbf{w}_{\text{new}}$, we might expect the policy $\pi_{\text{init}} = \text{argmax}_{\pi \in \mathcal{C}}(\mathbf{w}_{\text{new}} \cdot \boldsymbol{\rho}^{\pi})$ to provide a good starting point for learning. Our transfer learning algorithm works by initializing the bias and gain vectors to those of π_{init} and then further optimizing them via average-reward reinforcement learning.

After convergence, the newly learned bias and gain vectors are only stored in $\mathcal{C}$ if the gain of the new policy with respect to $\mathbf{w}_{\text{new}}$ improves by more than a satisfaction threshold γ. With this approach, if the optimal policies are the same or similar for many weight vectors, only a small number of policies are stored, and significant transfer can be achieved. The algorithm for Variable-reward Reinforcement Learning is presented in Algorithm 4.1. The counters i and c in the algorithm are primarily for the purposes of theoretical analysis. In practice, we keep looping through the algorithm and adding to the cache when necessary as long as new weight vectors are experienced.

We could use any vector-based average reward reinforcement learning algorithm in step 11 of the algorithm. In this work, we employ the vectorized version of H-learning algorithm (Tadepalli and Ok, 1998). In H-learning, action models are learned and employed to update the h-value of a state. In the vectorized version, the $\mathbf{h}$-values, the reward models (the binary feature values $\mathbf{f}(s, a)$), and the gain $\boldsymbol{\rho}$ are all vectors. The greedy action a is now defined as

$$a \leftarrow \underset{b}{\text{argmax}} \Big(\mathbf{w} \cdot \Big(\mathbf{f}(s,b) - \boldsymbol{\rho}\, D(s,b) + \sum_{s'} \Pr(s'|s,b)\mathbf{h}(s')\Big)\Big).$$

After the execution of a, the value of the state is updated as

$$\mathbf{h}(s) \leftarrow \max_{b} \Big(\mathbf{f}(s,b) - \boldsymbol{\rho}\, D(s,b) + \sum_{s'} \Pr(s'|s,b)\mathbf{h}(s')\Big).$$

For the gain $\boldsymbol{\rho}$, the only change from the non-vectorized version is in the update for the average reward $\boldsymbol{\rho}_r^{\pi}$, which is done as

$$\boldsymbol{\rho}_r^{\pi} \overset{\alpha}{\leftarrow} \mathbf{f}(s,a) + \mathbf{h}(s') - \mathbf{h}(s).$$

Algorithm 4.1 VRRL PROTOCOL

```
Output: Set of optimal policies $\mathcal{C}$.
1: $i \leftarrow 1$
2: $c \leftarrow 0$
3: $\mathcal{C} \leftarrow \varnothing$
4: $\pi_{\text{init}} \leftarrow \varnothing$
5: repeat
6:    Obtain the current weight vector $\mathbf{w}$
7:    if $\mathcal{C} \neq \varnothing$ then
8:       $\pi_{\text{init}} \leftarrow \text{argmax}_{\pi \in \mathcal{C}}(\mathbf{w} \cdot \boldsymbol{\rho}^{\pi})$
9:       Initialize the value function vectors
10:      Initialize the gain to $\boldsymbol{\rho}^{\pi_{\text{init}}}$
11:   Learn the new policy $\pi'$ through vector-based RL
12:   if $(\mathcal{C} = \varnothing)$ or $(\mathbf{w} \cdot \boldsymbol{\rho}^{\pi'} - \mathbf{w} \cdot \boldsymbol{\rho}^{\pi_{\text{init}}} > \gamma)$ then
13:      $\mathcal{C} \leftarrow \mathcal{C} \cup \pi'$
14:      $c \leftarrow 0$
15:      $i \leftarrow i + 1$
16:   else
17:      $c \leftarrow c + 1$
18: until $c \geq \frac{1}{\epsilon} \ln \frac{(i+1)^2}{\delta}$
19: return $\mathcal{C}$
```

We can also apply this vectorizing trick to R-learning, the model-free average-reward reinforcement learning algorithm (Schwartz, 1993). Some experimental results based on this are presented in Natarajan and Tadepalli (2005).

Though we expound on the application of the variable-reward concepts to the average-reward criterion, these ideas could easily be extended to the weighted total reward for the total-reward criterion or the weighted discounted reward for the discounted-reward criterion. For both these criteria, the value function of any fixed policy is a linear function of the reward weights. Hence, we could store the value function in a vectorized form just as in the average-reward case. In the average-reward setting, the gain vector provides a convenient way to find the best policy in the cache for a given set of reward weights. In the total-reward and the discounted-reward settings, we need to keep track of the expected returns for the start state distribution to serve a similar purpose; this will have a vectorized form whose dimension is the number of the reward components. Its inner product with the reward weights gives the expected returns of any policy, and it can be

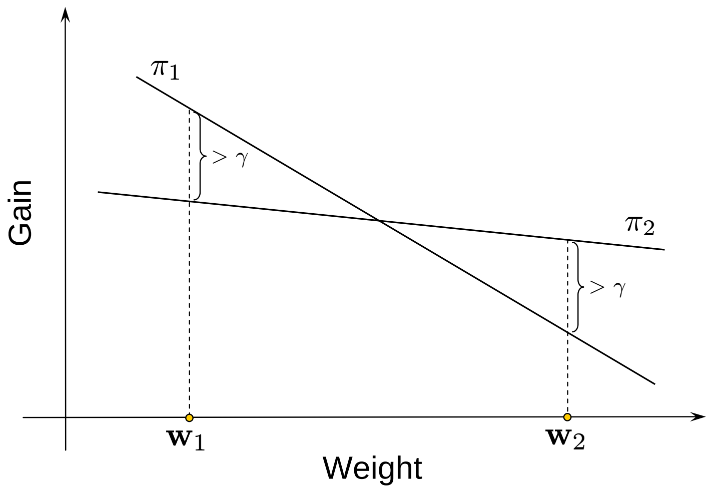


Figure 4.2: Two policies π_1 and π_2 learned for weights $\mathbf{w}_1$ and $\mathbf{w}_2$ respectively.

used to determine the initial policy as in step 8.

The algorithm presented here transfers seamlessly to any representation of policies, including value-function approximation, as long as the representation is the same during retrieval (step 8) and caching (step 13). This is because the algorithm caches the parameters of the representation as a proxy for the policy itself.

4.1.2.2 Theoretical Analysis

We now derive an upper bound on the number of policies stored by the VRRL algorithm and use it to derive the sample complexity.

Theorem 4.2. *Assuming that the vector-based RL algorithm finds optimal policies for each weight, the number of policies learned by VRRL is upper-bounded by* $O\left(\left(\frac{Wn}{\gamma}\right)^n\right)$*, where* n *is the number of components of the weight vector,* W *is the maximum range of the weight components, and* γ *is the satisfaction parameter.*

Proof. Let π_1 and π_2 be any two policies learned by our algorithm. Recall that the average rewards of the policies are linear functions of the reward weights as shown in Figure 4.2). Let the gain vectors of the two policies be $\boldsymbol{\rho}^{\boldsymbol{\pi_1}}$ and $\boldsymbol{\rho}^{\boldsymbol{\pi_2}}$ respectively. Let

$\mathbf{w_1}$ and $\mathbf{w_2}$ be the weights at which the policies π_1 and π_2 were learned. Because π_1 and π_2 are optimal for $\mathbf{w_1}$ and $\mathbf{w_2}$, we know that

$$\begin{aligned} \mathbf{w_1} \cdot \boldsymbol{\rho}^{\boldsymbol{\pi_1}} - \mathbf{w_1} \cdot \boldsymbol{\rho}^{\boldsymbol{\pi_2}} &\geq 0 \\ \mathbf{w_2} \cdot \boldsymbol{\rho}^{\boldsymbol{\pi_2}} - \mathbf{w_2} \cdot \boldsymbol{\rho}^{\boldsymbol{\pi_1}} &\geq 0. \end{aligned}$$

If π_1 was learned before π_2 then π_2 must have been judged better than π_1 at $\mathbf{w_2}$ by our algorithm or else it would not have been stored. Hence,

$$\mathbf{w_2} \cdot \boldsymbol{\rho}^{\boldsymbol{\pi_2}} - \mathbf{w_2} \cdot \boldsymbol{\rho}^{\boldsymbol{\pi_1}} > \gamma. \tag{4.1}$$

Similarly, if π_1 was learned after π_2, we will have

$$\mathbf{w_1} \cdot \boldsymbol{\rho}^{\boldsymbol{\pi_1}} - \mathbf{w_1} \cdot \boldsymbol{\rho}^{\boldsymbol{\pi_2}} > \gamma. \tag{4.2}$$

Because at least one of equations 4.1 and 4.2 is true, adding the two left-hand sides gives us

$$\begin{aligned} &(\mathbf{w_2} - \mathbf{w_1}) \cdot (\boldsymbol{\rho}^{\boldsymbol{\pi_2}} - \boldsymbol{\rho}^{\boldsymbol{\pi_1}}) > \gamma \\ &\qquad\implies |\mathbf{w_2} - \mathbf{w_1}|\,|\boldsymbol{\rho}^{\boldsymbol{\pi_2}} - \boldsymbol{\rho}^{\boldsymbol{\pi_1}}| > \gamma \\ &\qquad\implies |\mathbf{w_2} - \mathbf{w_1}| > \frac{\gamma}{|\boldsymbol{\rho}^{\boldsymbol{\pi_2}} - \boldsymbol{\rho}^{\boldsymbol{\pi_1}}|} \\ &\qquad\implies |\mathbf{w_2} - \mathbf{w_1}| > \frac{\gamma}{\max_{i,j}(|\boldsymbol{\rho}^{\boldsymbol{\pi_j}} - \boldsymbol{\rho}^{\boldsymbol{\pi_i}}|)} > \frac{\gamma}{\sqrt{n}}. \end{aligned}$$

The above equation implies that the weight vectors for which distinct policies are stored should be at least at a distance of $\frac{\gamma}{\sqrt{n}}$ from each other[3]. Let the maximum range of the weight space, that is, the difference between the highest and the lowest weight for any reward component be W. Hence, the maximum number of stored policies N is bounded by the number of points that can be packed in a hypercube of side W, where the distance between any two points is $\geq \frac{\gamma}{\sqrt{n}}$.

We estimate an upper bound on this quantity as follows: Suppose there are N such points in the hypercube. We fill the volume of the hypercube by surrounding each point

[3]Every component (dimension) of $\boldsymbol{\rho}$ is between 0 and 1 because it represents a feature expectation. The maximum distance between two such n-dimensional points is $\sqrt{n}$.

by a hypersphere of radius $\frac{\gamma}{2\sqrt{n}}$. No two such hyperspheres will overlap, because the centers of the hyperspheres are at least at a distance of $\frac{\gamma}{\sqrt{n}}$ from each other. Hence, the volume of the hypercube divided by the volume of the hypersphere will upper-bound the number of hyperspheres. As the volume of hypersphere of radius r is $\frac{r^n \pi^{\lfloor \frac{n}{2} \rfloor}}{\lfloor \frac{n}{2} \rfloor !}$ (Weeks, 1985),

$$N \leq \frac{W^n \lfloor \frac{n}{2} \rfloor !}{\left(\frac{\gamma}{2\sqrt{n}}\right)^n \pi^{\lfloor \frac{n}{2} \rfloor}} \leq O\left(\left(\frac{Wn}{\gamma}\right)^n\right).$$

□

Corollary 4.3. *The sample complexity of the VRRL algorithm is bounded by* $O\left(\frac{N}{\epsilon}\ln\frac{N}{\delta}\right)$, *where* N *is the upper-bound in the previous theorem.*

Proof. Algorithm 4.1 is a rough sketch of the VRRL algorithm. Notice that for the learner to terminate in stage i, it should have passed through $m_i = \frac{1}{\epsilon}(2\ln(i+1) + \ln\frac{1}{\delta})$ randomly chosen test weight vectors without learning a new policy. We first bound the probability of the learner terminating with a non-ϵ, γ-cover in stage i. A new policy is not learned only when the current policy cache produces a γ-optimal policy. A non-ϵ, γ-cover produces a γ-optimal policy on a random SMDP with probability at most $1-\epsilon$. The probability that all m_i weight vectors lead to γ-optimal policies is at most $(1-\epsilon)^{m_i} < e^{-\epsilon m_i} = e^{\ln\frac{\delta}{(i+1)^2}} = \frac{\delta}{(i+1)^2}$. The probability that the learner terminates with some non-ϵ, γ-cover in some stage is therefore bounded by

$$\sum_{i=1}^{\infty} \frac{\delta}{(i+1)^2} < \int_{i=0}^{\infty} \frac{\delta}{(i+1)^2} di < \delta.$$

From Theorem 4.2, we know that the learner learns at most N policies. The value of i is upper-bounded by N. Each time i is incremented, $m_i = O\left(\frac{1}{\epsilon}\ln\frac{(N+1)^2}{\delta}\right)$ SMDPs are tested. Thus, we have a sample complexity of $O\left(\frac{N}{\epsilon}\ln\frac{N}{\delta}\right)$. □

VRRL is not sample-efficient, because N is exponential in the natural parameter of the SMDP family n. However, it is easy to see that VRRL runs in time linear in the sample size and polynomial in n (not counting the cost of solving the SMDPs, and assuming that the policy cache is indexed efficiently for a constant-time retrieval). Thus, according to Definition 4.4, VRRL is a time-efficient transfer learner. This worst-case sample complexity bound suggests that the algorithm scales poorly in the number of reward components. While we may be able to improve our bounds with a tighter analysis, we argue that in many practical domains the number of reward components is small. VRRL is highly effective in such domains in transferring knowledge across different SMDPs. We provide empirical evidence for these claims in the later sections.

In light of this worst-case result, it is interesting to consider whether the algorithm can achieve better sample complexity for problems $(\mathcal{F}, \mathcal{E})$ when there exists a small set of policies that are sufficient to cover the problem. It turns out that even if a problem $(\mathcal{F}, \mathcal{E})$ has a small ϵ, γ-cover, the algorithm may require an arbitrarily large number of samples to find such a cover. To see this, consider a distribution $\mathcal{E}$ that places a probability mass of $1-\epsilon-\epsilon'$ on weight vector $\mathbf{w}_1$, a probability mass of ϵ' on $\mathbf{w}_2$, and spreads the remaining probability mass ϵ uniformly over the K weight vectors $\mathbf{w}'_1, \ldots, \mathbf{w}'_K$. It is possible to construct a variable-reward SMDP family $\mathcal{F}$ such that, for a pair of weight vectors generated according to $\mathcal{E}$, the optimal policy for one weight vector is not γ-optimal for the other. Hence, the ϵ, γ-cover of such a problem $(\mathcal{F}, \mathcal{E})$ consists of the optimal policies for $\mathbf{w}_1$ and $\mathbf{w}_2$ (all the remaining weight vectors together have a probability mass of ϵ) and is of size 2. However, making ϵ' arbitrarily small and K arbitrarily large will cause the algorithm to require arbitrarily many samples to generate an ϵ, γ-cover for $(\mathcal{F}, \mathcal{E})$.

The above example shows that a small cover does not necessarily guarantee a small sample complexity. This result does not seem to be specific to our algorithm but is a consequence of the fact that certain problem distributions are not exploitable in a transfer context. In the above example, there are many inherently different MDPs, each generated with a very small probability — this is a pathological scenario for any transfer learner.

If we place additional constraints on a policy cover, it is possible to show that our algorithm is efficient when there exists a small cover.

Definition 4.5. An ϵ, γ-cover for $(\mathcal{F}, \mathcal{E})$ is p-constrained if for each policy in the cover, with at least probability p, it is γ-optimal for a randomly drawn weight vector from $\mathcal{E}$.

With this definition in hand, we can show the following result.

Theorem 4.4. *Let $(\mathcal{F}, \mathcal{E})$ be a variable-reward transfer learning problem. If there exists a p-constrained $\epsilon, \gamma/2$-cover for $(\mathcal{F}, \mathcal{E})$ with M policies, then with probability at least $1-\delta$, the above algorithm will store an ϵ, γ-cover after at most $\frac{1}{p} \ln \frac{M}{\delta}$ samples.*

Proof. Let $\mathcal{C}$ be the $\epsilon, \gamma/2$-cover assumed in the theorem. Consider a policy π in the cover and a weight vector w for which π is $\gamma/2$-optimal. If π' is the optimal policy for w, it follows that π' is γ-optimal for any weight vector for which π is $\gamma/2$-optimal. Thus, if we can guarantee that for each π in the cover, we sample a weight vector for which it is $\gamma/2$-optimal then the set of policies stored for those weights will effectively cover all of the weight vectors that $\mathcal{C}$ covers. Hence the set of learned policies will be an ϵ, γ cover for $(\mathcal{F}, \mathcal{E})$.

It remains to bound the number of samples required to guarantee that, with high probability, we get a set of weights such that each policy in $\mathcal{C}$ is $\gamma/2$-optimal for at least one weight. Given m samples, the probability that a single policy is not $\gamma/2$-optimal for any of the m samples is no more than $(1-p)^m$. Using the union bound, the probability that at least one policy is not $\gamma/2$-optimal for any sample is bounded by $M(1-p)^m$. We want this probability to be less than δ, that is, $M(1-p)^m \leq \delta$. Solving for m gives the bound of the theorem. $\square$

Although this bound may look like it is only logarithmic in M, it will in fact at least scale as $M \ln M$. To see this, note that we must have $pM < 1$ which means that $1/p$ is at least as large as M. This result shows that if there is a small cover such that the probability of each policy being optimal for a problem is not too small, then we can obtain efficient sample complexity. This agrees with the intuition that transfer learning is most useful in situations with a small number of inherently distinct SMDP types, each of which is not too unlikely to be experienced.

4.1.3 Empirical Evaluation

We present results of the performance of the VRRL system within the simplified RTS domain shown in Figure 4.3. It is a map with a fixed number of peasants, the peasants' home base, multiple resource sites where the peasants can harvest resources, and an enemy base which can be attacked when it appears. Only one peasant can occupy a grid cell. Although there are multiple peasants in the world, there is only a single *learning* peasant; all other peasants execute fixed policies. The state variables in this domain are the locations of the peasants, what they are currently carrying (gold, wood, ..., nothing), the availability of resources at each of the resource locations, and the enemy's status

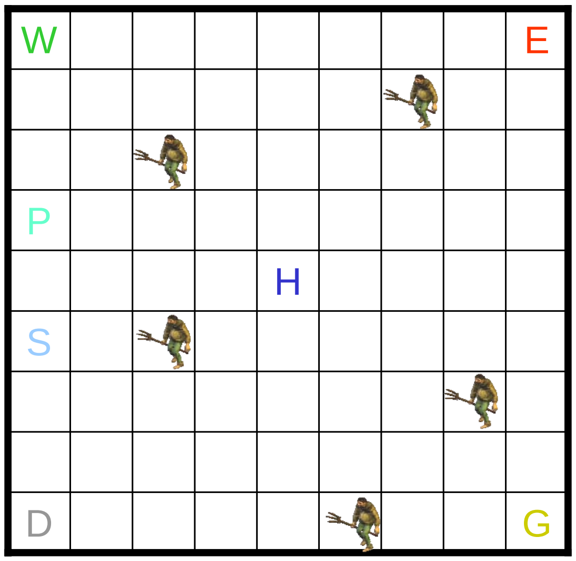


Figure 4.3: The simplified RTS domain with 5 peasants, multiple resources (W, G, ...), a home base H, and an enemy base E.

(present or absent). The peasants can move one cell to the North, South, East, and West, Pick a resource from a resource site, Put a resource at the home base, Strike the enemy base, and Idle (no-op); the probability of failure for an action is 0.05. Every resource is stochastically generated at its designated site with a probability of 0.5. The probability of the appearance of an enemy base is 10^{-4}, and it persists until it is attacked. Due to the lack of scalability of non-hierarchical learning, these performance curves are based on a 25×25 RTS map with 1 peasant, 5 fixed resource sites (2 resources), a home base, and an enemy base. The rate of $\epsilon-$greedy exploration is 0.1.

The reward feature vector has components associated with dropping off each of the resources, enemy elimination, and a time-step penalty that provides shaping to the flat learner. Theoretically, each of the weight components $w_i \in (-\infty, \infty)$. Empirically, we defined a set of seed values that we shuffle to generate the training and testing weights. These seed values are chosen to make one component dominate, and this is consequently reflected in the policies learned. For a concrete illustration of how the reward feature and weight vectors generate the scalar reward, consider that the peasant is at its home base carrying gold (state s_1), it is executing put, and the immediate reward feature

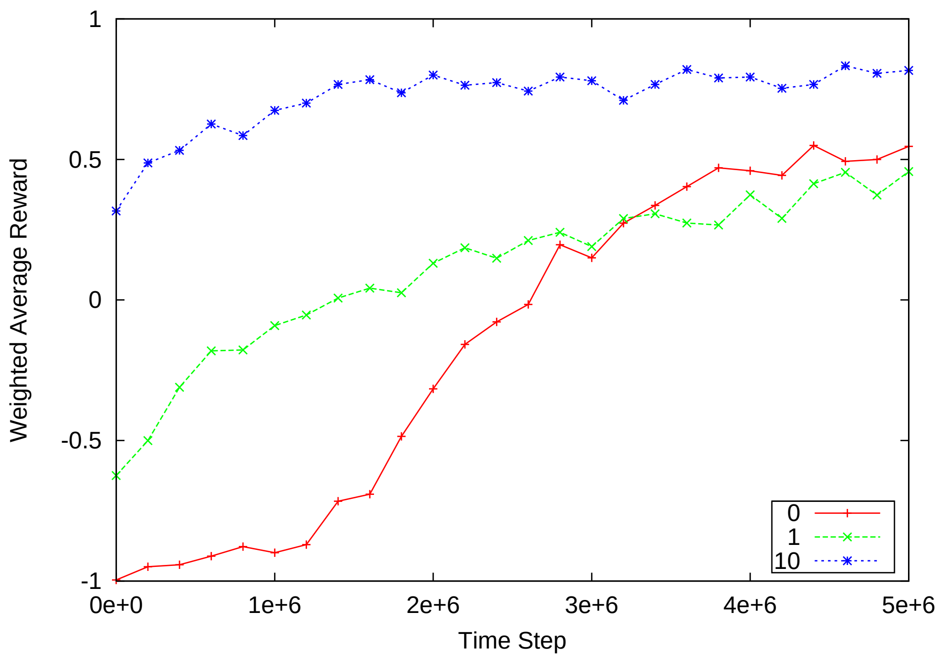


Figure 4.4: Performance of VRRL on a test weight after seeing 0, 1, and 10 training weight vectors (averaged across 10 trials).

vector $\mathbf{f}(s_1, \mathsf{put}) = (0, 1, 0, 1)$. If the current weight vector $\mathbf{w} = (10, 50, 30, -10)$, then the immediate scalar reward $R(s_1, \mathsf{put}) = \mathbf{w} \cdot \mathbf{f}(s_1, \mathsf{put}) = 40$.

Figure 4.4 shows the learning curves for the VRRL learner. All curves are averaged across 10 trials. The experiments are designed to show the performance of the learner on a particular test weight after having seen 0 through 10 training weights. Curve i represents the performance on the test weight having already seen i training weights; for the sake of clarity, the plots only show the learning curves for $i = 0, 1, 10$. As the variable-reward framework is designed principally to deal with dynamically changing weights, we evaluate the performance on the test weight vector by incrementally introducing training weight vectors. Curve 0 measures the performance of the algorithm on the test weight, given no prior experience. Next, one training weight is introduced, and the optimal policy for this weight is learned and cached. Curve 1 now measures the performance on the test weight vector given the single training weight. When the second training weight is introduced, a cached policy is retrieved for initialization, and a new policy is learned; this new policy is cached if it is substantially better than the initializing policy. Importantly, none of the learning done during performance evaluation (on the test weight vector) spills over into the training phases — the policy learned for the test weight is *never* cached.

From the results, we can observe that the learning curve for the test weight given no prior training weights (i.e., an empty policy cache) is the slowest to converge in both figures. However, after accumulating the 10 training weight vectors, the VRRL agent demonstrates a high jump-start[4] and quicker convergence for the test weight. With only one training weight vector, VRRL exhibits some negative transfer, that is, the initialization to the policy learned for the first training weight hurts the convergence for the test weight. This is unsurprising given that currently we always attempt to transfer past experience even when the experience is limited.

Because the MDPs have the same dynamics, the flat learner does not need to relearn the transition model from scratch when the reward weights change. Figure 4.5 shows the results of repeating the VRRL experimental setup with one crucial difference — no policy is ever cached. Instead, the only transfer here is due to the transfer of transition models. In this case, we do observe a slight speed-up in convergence but no jump-start.

4.2 Variable-Reward Hierarchical Reinforcement Learning

VRRL exploits the decomposition of reward into reward components, whereas HRL is based on the idea of decomposing the tasks into subtasks. In this section, we explicate the variable-reward hierarchical reinforcement learning (VRHRL) algorithm that synergistically combines the two ideas. We seek to incorporate the variable-reward transfer mechanism into a hierarchical framework to benefit from value-function decomposition.

The task hierarchy for the RTS domain is the same as that shown in Figure 3.4, the principal difference being that the Harvest task has more bindings (resources) here. As the Root task solves the entire SMDP, it sees the entire world state. The Harvest task see an abstract state space based on the location of the peasant, what it is carrying, and

[4]Jump-start is defined as the immediate benefit via transfer without any learning in the new setting.

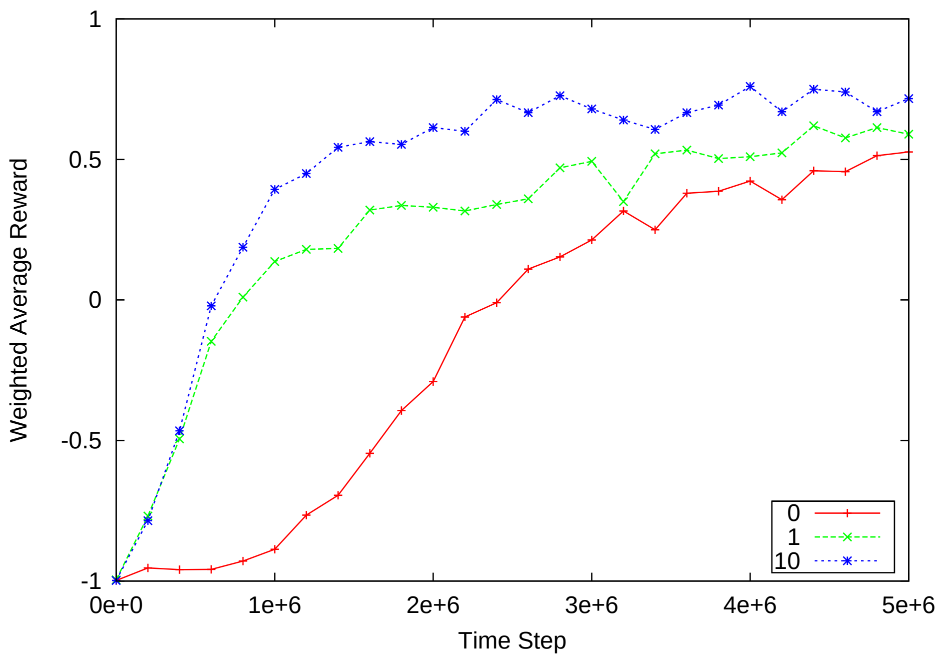


Figure 4.5: Performance of VRRL without the policy-caching mechanism on a test weight after seeing 0, 1, and 10 training weight vectors (averaged across 10 trials).

the resource present at the resource locations. The Deposit task only sees the location of the peasant and what it is carrying. The Attack task keeps track of the location of the peasant and the status of the enemy. The Goto task only see the location of the peasant. The primitive movement actions, Pick and Idle, only store one value each. Put needs to keep track of the location and the resource being toted. Finally, Strike looks at the location of the peasant. We have designed the task hierarchy to include a parameterized Harvest task. Because the set of terminal states of this task depends solely on the binding value of the parameter, RDI holds, and this task can be called optimally based on the external context. On the other hand, the terminal states of an unparameterized Harvest task depend on the local policy, and RDI no longer holds.

In non-hierarchical learning, the optimal policy is represented by a monolithic value function. This means that changes in the reward function will often result in non-local changes to this value function. Storing a new value function for every new weight would consequently result in a large policy cache. Moreover, besides taking a longer time to converge, a monolithic value function is also more prone to lead to negative transfer especially when only a small number of policies are stored. Instead, a hierarchical value function is less prone to negative transfer especially when the hierarchical decomposition is closely aligned with the reward decomposition. Every task has a local value function, and local changes in rewards can be better managed. For instance, if none of the reward variations affect navigation, the Goto task only needs to learn its local value function once; perfect transfer is achievable for this task across the family of MDPs (and consequently every task below it in the task hierarchy). In the case of the RTS domain with multiple colliding peasants, the Goto task is only affected by the collision component of the reward, and it transfers over perfectly to all MDPs for which the corresponding reward weight is identical. Thus, negative transfer can be significantly reduced with better design of hierarchical structure.

4.2.1 Bellman Equations for VRHRL

The vectorized value function $\mathbf{V}_i(s)$ for a non-root task T_i represents the vector of total expected reward components during task T_i starting from state s following a recursively optimal policy π. Hence, the value function decomposition for a non-root task satisfies the following equations:

$$\mathbf{V}_i(s) = \begin{cases} \mathbf{f}(s,i) & \text{if } T_i \text{ is a primitive task} \quad (4.3a) \\ 0 & \text{if } s \text{ is a terminal state for } T_i \quad (4.3b) \\ \mathbf{V}_g(s) + \sum_{s' \in \mathcal{S}} \Pr(s'|s,g) \cdot \mathbf{V}_i(s') & \text{otherwise,} \quad (4.3c) \end{cases}$$

where the recursively optimal child task

$$g = \operatorname*{argmax}_a \left(\mathbf{w} \cdot \left(\mathbf{h}_a(s) + \sum_{s' \in \mathcal{S}} \Pr(s'|s,a)\, \mathbf{h}_i(s') \right) \right) \tag{4.4}$$

maximizes the weighted bias (the objective of action selection is to maximize the weighted gain, i.e., the dot product of the weight vector with the gain vector), and the bias vector is computed as

$$\mathbf{h}_i(s) = \mathbf{V}_i(s) - \boldsymbol{\rho}\, D_i(s).$$

Storing the bias vector indirectly in a form that is independent of the gain gives a limited form of reusability in that the value function for a subtree of the task hierarchy may be transferred across MDPs with different global gain vectors as long as the optimal policy for the subtree remains the same. Storing the value functions as vectors facilitates transfer across different MDPs in the variable-reward family just as in the non-hierarchical variable-reward RL. The Bellman equations for the task durations D_i remain the same as before. Because the root task is recurrent and has no parent tasks, we keep track of the bias values of the root task directly as

$$\mathbf{h}_{\text{root}}(s) = \max_a \left(\mathbf{h}_a(s) + \sum_{s' \in \mathcal{S}} \Pr(s'|s,a) \cdot \mathbf{h}_{\text{root}}(s') \right).$$

4.2.2 Transfer Mechanism in VRHRL

The VRHRL agent has three components: the task hierarchy with the current task value functions and the associated global gain, the task stack, and a cache of previously learned optimal policies $\mathcal{C}$ that comprise the convex piecewise function. The policies in the cache are indirectly represented by the task value and duration functions and the global gain. The policy cache $\mathcal{C}$ is specific to the hierarchical transfer mechanism, while the other components are part of a basic hierarchical agent.

Initially, the agent starts out with an empty policy cache. The agent proceeds to learn a recursively optimal policy π_1 for the first weight $\mathbf{w}_1$. When the agent receives a new weight $\mathbf{w}_2$, it first caches π_1 (the task functions and the global gain achieved for $\mathbf{w}_1$) in the policy cache $\mathcal{C}$. Next, it determines $\pi_{\text{init}} = \text{argmax}_{\pi \in \mathcal{C}}(\mathbf{w}_2 \cdot \boldsymbol{\rho}^{\pi})$, which in this case is π_1, and initializes its task functions and global gain based on π_{init}. It then improves the value functions using the vectorized version of model-based hierarchical RL, which works as follows.

At any level of the task hierarchy, the algorithm either chooses an exploratory child task, based on ϵ-greedy exploration, or a greedy one according to Equation 4.4. It estimates the transition model $\Pr(s'|s,a)$ for each task by counting the number of resulting states for each state-subtask pair to estimate the transition probabilities. In doing so, the states are abstracted using the abstraction function defined at that task, so that the transition probabilities can be represented compactly. Every time a task terminates in state s', equations 4.3a–4.3c are applied to update the total expected reward vector $\mathbf{V}_i(s)$ of task T_i, and scalar equations are used to update the scalar duration function $D_i(s)$. The global gain is computed as

$$\boldsymbol{\rho}_r^\pi \overset{\alpha}{\leftarrow} \mathbf{V}_a(s) + \mathbf{h}_{\text{root}}(s') - \mathbf{h}_{\text{root}}(s),$$

where α is the learning rate, and s and s' are the states before and after executing the highest level subtask a of the root task; the update for ρ_t^π can be used unchanged. The updates for $\boldsymbol{\rho}_r^\pi$ and ρ_t^π are performed only when a is selected greedily and $\boldsymbol{\rho}^\pi \leftarrow \boldsymbol{\rho}_r^\pi / \rho_t^\pi$.

On sensing a new weight $\mathbf{w}_3$, the agent only caches the learned hierarchical policy π_2 for $\mathbf{w}_2$ if $\mathbf{w}_2 \cdot (\boldsymbol{\rho}^{\pi_2} - \boldsymbol{\rho}^{\pi_{\text{init}}}) > \gamma$. If this condition is not satisfied, then the newly learned policy is not sufficiently better than the cached version. When adding π_2 to the policy cache, we could just store the value function of every task in the task hierarchy. However, although the hierarchical policy has changed, many of the local task policies could still be the same. To leverage this fact, for every task being stored, we check the policy cache to see if any of the previously-stored versions of the task are similar to the current one; if so, then we need only store a reference to that previously-stored version. Two versions of a task are similar if none of the values for the vector components of the value and duration functions for any state differ by more than a similarity constant σ. Once caching is complete, the policy that maximizes the weighted gain with respect to $\mathbf{w}_3$ is chosen from the policy cache for initialization. This process is repeated for every new weight encountered by the system. Thus, every weight change is accompanied by the internal process of caching and value function initialization for the agent.

VRHRL is also applicable when employing value-function approximation. In the experiments described in the next section, we employ a linear value function over a set of predefined features for every task in the hierarchy. Instead of updating the value function for each possible value of the abstracted feature vector, the weights of the linear value function are updated in proportion to the gradient of the temporal difference error. Representationally, linear function approximation benefits greatly from the nonlinearity of value function decomposition, because it facilitates the contextual activation of the features within tasks. For example, only the feature that encodes the distance to a particular site bound to the parameter of Goto is made active within that task. This contextual activation in the non-hierarchical setting requires knowing the policy that is yet to be learned.

4.2.3 Empirical Evaluation

The performance results for the hierarchical agent are based on a 25×25 RTS map (discussed in Section 4.1.3) with 5 peasants, 5 fixed resource sites (5 resource types), a home base, and an enemy base. As this is a huge state space, in addition to the hierarchical decomposition, we also employ linear value-function approximation. For a vectored value function, this entails maintaining a set of parameters for each vector component; for a hierarchical value function, we maintain such a set of parameters for each task. The state features used are distances to sites, indicators for what the peasant is carrying, indicators for what resource is present at the sites, and an indicator for the appearance of the enemy base. The learning rate for updating the weights of the linear function approximation is 10^{-4}. The rate of $\epsilon-$greedy exploration is set to 0.1. The reward feature vector has 7 components that are associated with 5 resources, peasant collisions, and enemy elimination.

The experiments are designed exactly as in Section 4.1.3. Two important parameters that govern the learning behavior of the VRHRL algorithm are the satisfaction constant $\gamma = 0.01$ and the similarity constant $\sigma = 0.01$; these parameters are fixed for all experiments. Just as the learning rate and the rate of exploration are key parameters in regular RL, the satisfaction and similarity constants trade-off speedup in learning against the size of the policy cache. For instance, the smaller the satisfaction, the larger the number of policies stored in the cache. When a new weight is detected, the algorithm is more likely to find a policy that is very close to optimal from this heavily-filled cache. The smaller the similarity, the more new task data is stored by the algorithm instead of maintaining references to previously stored task information.

Figure 4.6 shows the learning curves for VRHRL. The learning curve for the test weight given no prior training weights (i.e., an empty policy cache) is the slowest to converge. However, the jump-start and speed of convergence for the test weight improve

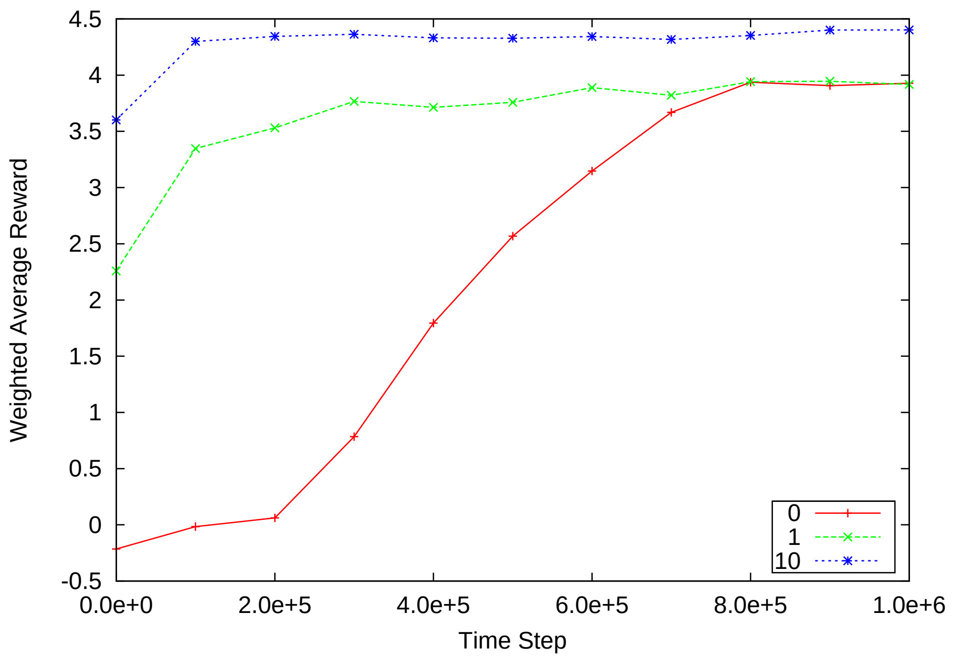


Figure 4.6: Performance of VRHRL on a test weight after seeing 0, 1, and 10 training weight vectors (averaged across 10 trials).

as more training weight vectors are experienced.

We have previously noted that the flat learner does not need to relearn the transition model from scratch when the reward weights change. In the hierarchical learner, the state-transition models at the composite and primitive tasks do not need to be relearned when RDI holds, because then the distribution of the termination states at every task is invariant to the changes in the policies of the child tasks. Figure 4.7 shows the results of repeating the VRHRL experiment without the policy-caching mechanism. Here, reusing the learned models is only slightly beneficial to the hierarchical learner, leading to a small initial speed-up in learning but without any jump-start or faster convergence. A complete trial of the VRHRL experiment (10 training weights, 1 test weight) takes 7266.1 secs on average, while that of the experiment without caching takes 7204.3 secs on average —

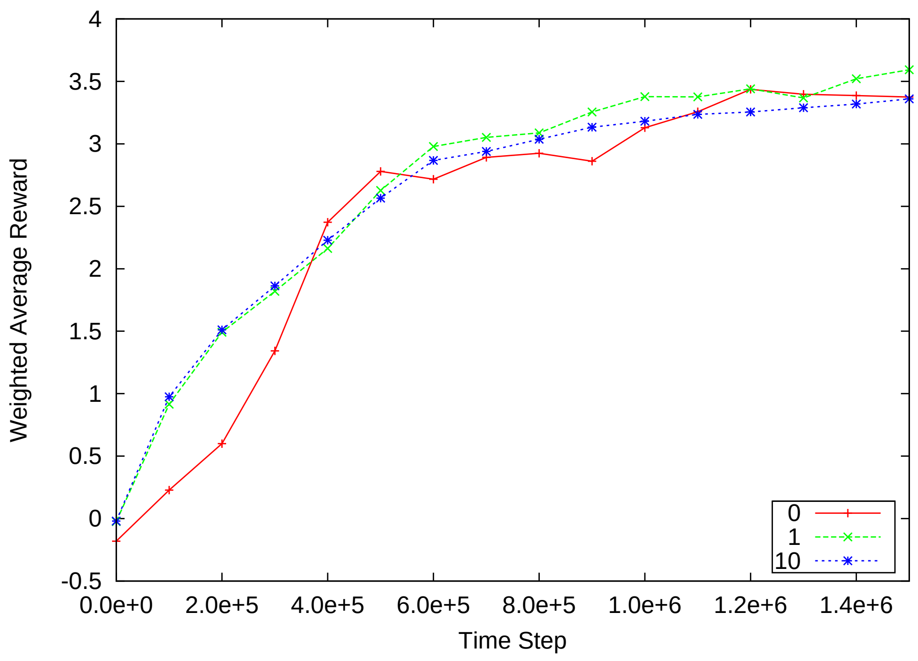


Figure 4.7: Performance of VRHRL without the policy-caching mechanism (averaged across 10 trials).

the caching mechanism adds an overhead of only 0.85% to the overall running time of the experiment.

We have mentioned that VRHRL benefits from having certain tasks like Goto transfer successfully because they are affected by only a small number of reward components. Figure 4.8 demonstrates that with caching only the Goto task (and reinitializing all other tasks to zero), the learner sees a jump-start that is smaller than that for complete policy caching, but the performance is still better than when not caching at all.

We have also conducted experiments in which the weight vector $\mathbf{w}$ is induced from the scalar reward signal R using perceptron-style incremental learning:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(R - \mathbf{w} \cdot \mathbf{f})\mathbf{f},$$

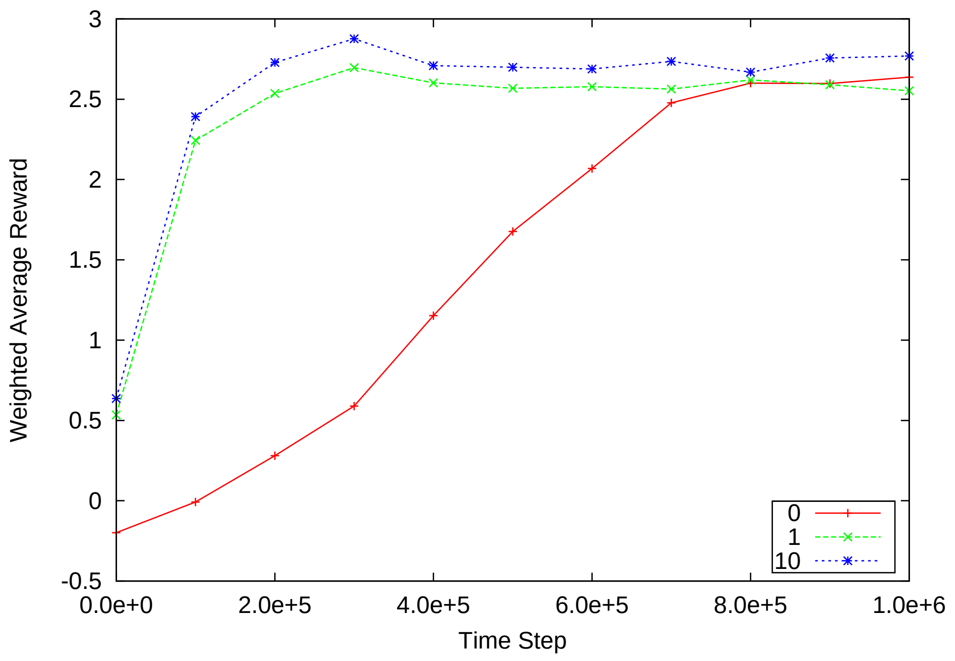


Figure 4.8: Performance of VRHRL when caching only the Goto task (averaged across 10 trials).

where **f** is the reward feature vector. The agent is allowed to take random actions for a certain number of steps in order to sense the scalar reward and the corresponding reward features. The number of steps can be tailored to allow the agent to get a good estimate of the true weight vector — the less likely it is for the agent to experience a certain reward component, the more time it needs in order to estimate the weight for that component accurately. In our experiments, we allow the agent to learn an extremely good estimate of the weight vector. The resulting learning curves look exactly like the plots shown for VRRL and VRHRL except that they are shifted to the right by the number of reward-learning steps.

4.3 Related Work

Taylor et al. (2005) propose a value-function mapping approach for transfer between source and target reinforcement learning tasks. In their approach, hand-coded similarity functions guide the transfer of the Q-value functions. These functions are applied to specify similar state spaces, action spaces, and representation mappings between the source and target tasks. Liu and Stone (2006) have extended this work to automate the determination of the similarity mappings by employing expertly-designed qualitative dynamic Bayesian nets (QDBNs) to represent the structural information about the source and target tasks. Based on the structure mapping, an algorithm discovers the similarities between the QDBNs of the source and target tasks to facilitate the transfer of value functions. As our work is restricted to assuming that the dynamics are exactly the same in the source and target tasks, it would be interesting to combine the two methods to handle both varying dynamics and reward functions.

Torrey et al. (2007) apply inductive logic programming (ILP) to learn macros that transfer Q-functions across different problems. Their method (applied to the m-on-n Breakaway domain within RoboCup soccer) proceeds as follows: the learner collects training examples while playing a few source games; relational macros that describe a successful strategy (policy) in the source task are learned through ILP; these macros are then employed as default policies in the target task for a fixed number of iterations; finally, a better policy is learned via Q-learning. In effect, the relational macros serve as exploration devices in the target domain. While their approach generalizes over several source games to learn the ILP rules, we consider each source task independently for caching. This greedy approach could result in a larger policy cache, and it could increase negative transfer in the pathological case where successive weights are radically different from one another.

Guestrin et al. (2003) employ linear programming techniques to solve relational MDPs and compute a set of value functions that can be applied across the grounded MDP instances. Mausam and Weld (2003) take the approach of converting the relational MDPs into several propositional MDPs and solving the propositional MDPs; here, the value functions are represented by first-order regression trees. In both these methods, the relational structure is exploited for learning the value functions for similar classes of MDPs. Price and Boutilier (2003) have looked at the problem of transferring knowledge from one learner to another as imitation. Our goal is chiefly to facilitate transfer in the presence of time-varying rewards.

Our approach to variable-reward transfer learning borrows ideas from multi-criteria reinforcement learning (Gabor et al., 1998). These ideas are related to earlier work on solving multi-criteria MDPs, where weights indicate the importance of different reward components. For example, White (1982) employs vector-based generalizations of successive approximation techniques to solve MDPs and Feinberg and Schwartz (1995) formulate the problem as optimizing a weighted sum of the total discounted rewards for the different components of the reward function. Russell and Zimdars (2003) consider the additive decomposition of rewards for solving MDPs, while Guestrin et al. (2001) apply reward decomposition to make multi-agent coordination tractable. Parr (1998b) decomposes the problem of solving a big MDP into one of solving smaller weakly-coupled sub-MDPs. The fact that the value function of a fixed policy over the sub-MDP is linear in the values of its exit states is effectively exploited to speed up the solution of the overall MDP.

4.4 Conclusion

This chapter has shown that vector-based value function learning and caching of policies can lead to effective variable-reward transfer learning. It has also demonstrates that hierarchical reinforcement learning can accelerate transfer across variable-reward MDPs more effectively than is possible with non-hierarchical methods. Our results are in the model-based setting and have the added advantage that the models need not be relearned from scratch when the rewards change; it is easy to learn them from non-decomposed scalar rewards, because the scalar reward is linear in the reward weights.

Possible future directions include sharing tasks across multiple agents as in the MASH framework and transferring across MDP families that share only part of the reward dynamics. In RTS games, we could consider MDPs that contain different objects such as peasants, footmen, and archers in different proportions and locations. Although the dynamics of local interaction for each peasant may be the same, the changes in the peasants' locations and their numbers mean that, technically, the different MDPs have different dynamics. Nevertheless, people seem to be able to effectively transfer their strategies from one such scenario to another. Duplicating this ability in machines would be a big advance for the field of machine learning.

So far in this dissertation, we have discussed the transfer of hierarchical structure while finessing the issue of its design. The next chapter addresses this issue by describes a framework that discovers hierarchical structure and analyzes its scope of transfer.

Chapter 5: Hierarchical Structure Discovery

Scaling up reinforcement learning (RL) to large domains requires leveraging the structure in these domains. Hierarchical reinforcement learning (HRL) and its close cousin, hierarchical planning (Nau et al., 2003), provide mechanisms through which domain structure can be exploited to constrain the value function and policy space of the learner, and lead to faster learning and efficient planning. In the MAXQ framework, a task hierarchy is defined (along with relevant state variables) for representing the value function of the overall task. This allows for decomposed subtask-specific value functions that are easier to learn than the global value function.

Large sequential decision tasks such as trip planning and real-time strategy games also offer special challenges and opportunities for studying transfer learning. These domains are complex, and good performance requires selecting long chains of actions to achieve subgoals needed for ultimate success. Reinforcement learning in these domains, because it involves a process of exploratory trial-and-error, can take a very long time to discover these long action chains. Fortunately, it is often possible to study smaller versions of these domains that share the same fundamental structure, but that involve fewer objects and smaller state spaces. Reinforcement learning on these smaller domains is much faster. If it can discover the shared structure and transfer it to the large scale domains, then this provides a much more efficient way of achieving good performance.

The automated discovery of task hierarchies is compelling for at least two reasons. First, it avoids the significant human effort and expertise in programming the task-subtask structural decomposition, along with the associated state abstractions and subtask goals. Second, if the same hierarchy is useful in multiple domains, it leads to significant transfer of learned structural knowledge from one domain to the other. The cost of learning can be amortized over several domains. Several researchers have focused on the problem of automatically inducing temporally extended actions and task hierarchies (Thrun and Schwartz, 1995; Reddy and Tadepalli, 1997; McGovern and Barto, 2001; Menache et al., 2001; Pickett and Barto, 2002; Hengst, 2002; Şimşek and Barto, 2004; Jonsson and Barto, 2006; Langley and Choi, 2006; Hogg et al., 2009). Because tasks are like subroutines, automatically learning tasks addresses the question of what constitutes a good subroutine, a long-standing fundamental question in computer science.

This work builds on previous research and automatically induces task hierarchies based on two key claims: first, it is possible to uncover the hierarchical task structure in a domain by analyzing the relevant relationships between actions in successful trajectories; second, the hierarchical task structure is more robustly transferable across domains than other kinds of knowledge such as the utilities of being in different states (the value function) or the utilities of executing different actions (the action-value function).

The Hierarchy Induction via Models and Trajectories (HI-MAT) approach is based on the claim that the key to transfer learning is to discover and represent deep forms of knowledge that are invariant across multiple domains. Consider the problem of driving to work. There are surface aspects, such as the amount of time it takes to get from home to the office or the selection of the best route, that may be highly regular, but they are unlikely to transfer when you move to a new city. On the other hand, the task structure involved in driving, such as starting the car, driving, obeying traffic laws, parking, depends only on the relevant structure of the actions involved, and hence transfers more successfully from one city to another. We are interested in transferring task knowledge between source and target domains that share the same relevant structure, that is, the actions in both domains depend upon and influence the same state variables. This is weaker than assuming that the behavior of actions is exactly identical in two domains. For instance, although two different cars may have very different engines that accelerate at different rates, they both speed up when you press the accelerator and slow down when you hit the brakes.

The high-level schema of the HI-MAT approach is shown in Figure 5.1. We focus on the asymmetric knowledge transfer setting where we are given access to solved source RL problems. The action models and a successful trajectory are extracted from a source task and analyzed to identify the relevant relationships between the actions in the trajectory. Our hierarchy discovery algorithm, HI-MAT (Hierarchy Induction via Models and Trajectories), leverages this relevantly annotated trajectory to discover a coherent task hierarchy that minimizes the number of inter-task relevant links. This task hierarchy is transferable for faster learning to all target tasks that share the relevant structure of the source task.

This chapter shows how a task hierarchy can be efficiently discovered from a sin-

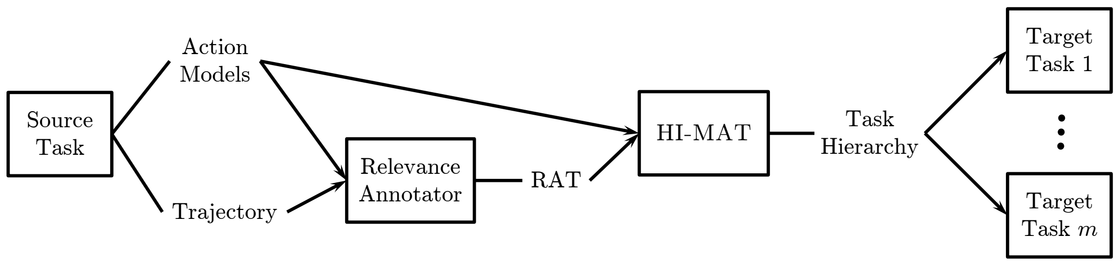
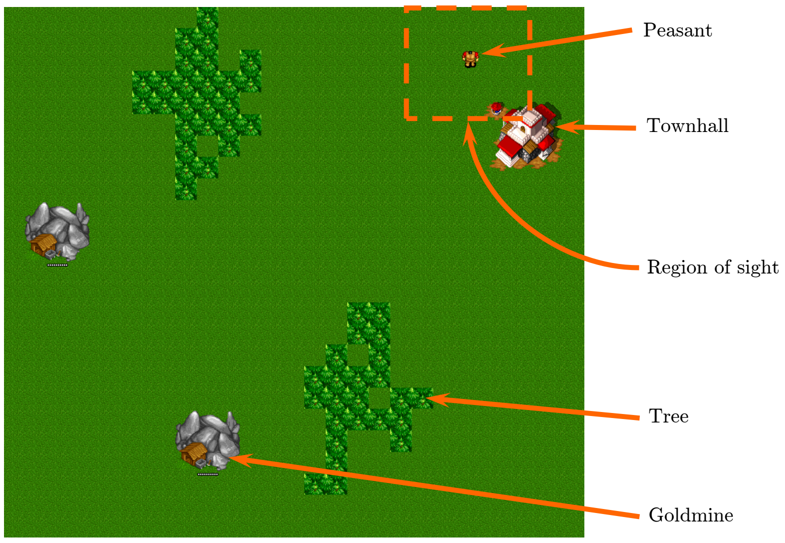


Figure 5.1: The architecture of the HI-MAT system. Action models and a single trajectory from the source task are analyzed to produce a relevantly annotated trajectory (RAT). The RAT and the action models are then provided to the HI-MAT algorithm, which discovers a task hierarchy. The structure of this task hierarchy facilitates faster learning in a range of target tasks that share the relevant structure of the source task.

gle source trajectory by exploiting the knowledge of the domain dynamics instead of a search-based approach (Appendix A sketches out the huge size of the search space). This work resembles explanation-based learning (EBL), which generates explanations of successful trajectories to deduce sound knowledge (Minton, 1988; Tadepalli and Dietterich, 1997; Nejati et al., 2006). However, unlike the EBL paradigm, which relies on complete knowledge of the actions to learn sound control rules, HI-MAT only learns based on the qualitative relevant structure of actions, which is more readily available for domains and yields more widely-transferable knowledge. The transferred knowledge is the hierarchical task structure, which can be further specialized to learn detailed action policies in the target task through further experience.

We analyze HI-MAT both theoretically and empirically. Our theoretical results show that, under appropriate conditions, the task hierarchies induced by HI-MAT are consistent with the observed trajectory and possess compact value-function tables that are safe with respect to state abstraction. Empirically, we show that ① using a successful trajectory can result in more compact task decompositions than when using only action models in the form of dynamic probabilistic networks (DPNs), ② the induced hierarchies are comparable to manually-engineered hierarchies on target RL tasks, and MAXQ-learning converges significantly faster than flat Q-learning on those tasks, and ③ transferring hierarchical structure from a source task can speed up learning in target RL tasks when transferring value functions cannot.



(a) Wargus map.

Variable	Description
$p.l$ or $p.x, p.y$	Peasant's location
$p.r$	Peasant's resource (gold/wood)
$r.g, r.w, r.t$	Indicators for goldmine, forest, or townhall regions
$q.g, q.w$	Quota indicators for gold and wood.

(b) State variables.

Figure 5.2: The Wargus Resource-Gathering domain.

5.1 Preliminaries

This section introduces the Wargus resource-gathering domain and describes the action model representation.

5.1.1 Wargus Resource-Gathering Domain

Wargus is a real-time strategy game. Two players inhabit a world that contains entities such as peasants, goldmines, and townhalls, and resources such as gold and wood. To win, a player must collect resources, build various kinds of units (e.g., townhalls, lumber mills, footmen, dragons), and then deploy those units to defeat the enemy in battle. We focus on the resource-gathering aspect of Wargus, where our agent needs to collect a specified quota of gold and wood as quickly as possible given an arbitrary map as shown in Figure 5.2a. To achieve the resource goal, the agent must command peasants to collect gold from goldmines and harvest wood from forests and deposit those resources at townhalls. The game state is described through the following variables: $p.l$ represents the coordinate location of peasant p; $p.r$ indicates if the peasant is empty-handed (empty) or carrying a resource (gold or wood); the binary variables $r.g, r.w$, and $r.t$ indicate if there is a goldmine, forest, or townhall respectively in the peasant's immediate vicinity; the binary variables $q.g$ and $q.w$ indicate whether the required quotas of gold and wood have been met.

The actions available to the agent are MineGold, ChopWood, Deposit, Goto(l), for mining gold when in the vicinity of a gold mine, chopping wood when in the vicinity of a forest, and navigating to a location l on the map (this uses the internal path-finding routine of the game). If MineGold is executed when an empty peasant p is in the vicinity (within the effective range) of a goldmine, then this will change $p.r$ to gold; otherwise, this action will have no effect (the game state does not change). Similarly, a successful ChopWood will change $p.r$ from empty to wood. A successful Deposit will deposit the peasant's resource at the townhall, set $p.r$ to empty, and provide a positive reward to the agent. The overall goal of achieving the requisite quotas $q.g = 1 \wedge q.w = 1$ is attained by having peasants repeatedly collect gold and wood from goldmines and forests, and deposit these resources at townhalls. Even though the actions have deterministic outcomes, they appear to behave nondeterministically due to the partial observability of the true state through the defined variables. Employing Boolean variables to represent the map's topography and the resource quotas actually results in non-Markovian system dynamics and is approximated with probability distributions within the MDP. For example, transitioning from an unmet quota to meeting it actually depends upon the number of deposits, but this is approximated by the distribution seen at the leaf for $q.g'$ in Figure 5.4. The primitive Goto actions are themselves temporally extended and the parameter l can be bound to a location on the map. To limit the number of Goto actions, we divide the map into regions whose radii are the effective range of the peasants and restrict l to be bound only to the centroids of these regions.

The Wargus resource-gathering domain is too complex for a non-hierarchical solution, because the larger the map, the larger the state space and the larger the number of navigation actions available at each state. However, the task hierarchy shown in Fig-

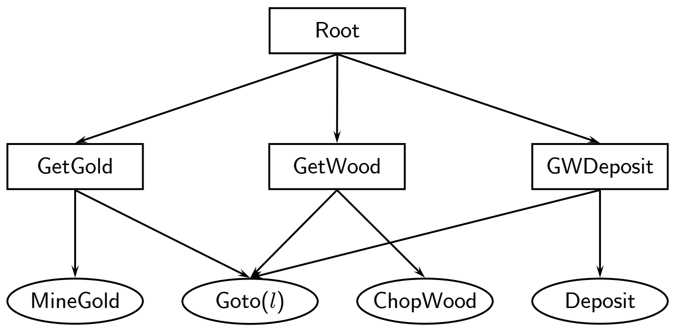


Figure 5.3: A task hierarchy for the Wargus resource-gathering domain.

ure 5.3 decomposes the overall problem into simpler subproblems. The Root task tries to learn a policy for meeting the overall resource quota in the original MDP by solving and combining the solutions of three subtasks: a GetGold task that moves a peasant to the vicinity of a goldmine and then mines for gold, a GetWood task that does the same for wood, and a GWDeposit task that brings a peasant carrying either gold or wood back to the townhall to deposit the resource. GetGold and GetWood are terminated when the peasant is carrying something ($p.r \neq$ empty) and GWDeposit is terminated when the peasant is not carrying anything ($p.r =$ empty). Because GetGold is not involved in getting wood or depositing anything, it does not need to know anything about the townhall ($r.t$) and wood ($r.w$) regions. Similarly, GWDeposit does not need to know about mining gold or chopping wood and is active only when the peasant is carrying something. Thus, each of these subtasks describes a sub-MDP with fewer state variables and actions than the original problem. This particular hierarchy was expertly designed (by me!) to emphasize subtask sharing, where the GWDeposit task deposits a resource regardless of its type.

5.1.2 Action Models

An action model represents the effect that executing an action has on the state variables. For instance, depositing gold at a townhall changes the peasant's resource variable $p.r$ from gold to empty. The factored state space allows for system dynamics to be specified using a more natural and intuitive representation instead of an $|\mathcal{S}| \times |\mathcal{S}| \times |\mathcal{A}|$ matrix for transitions and a $|\mathcal{S}| \times |\mathcal{A}|$ matrix for rewards. A 2-stage dynamic probabilistic

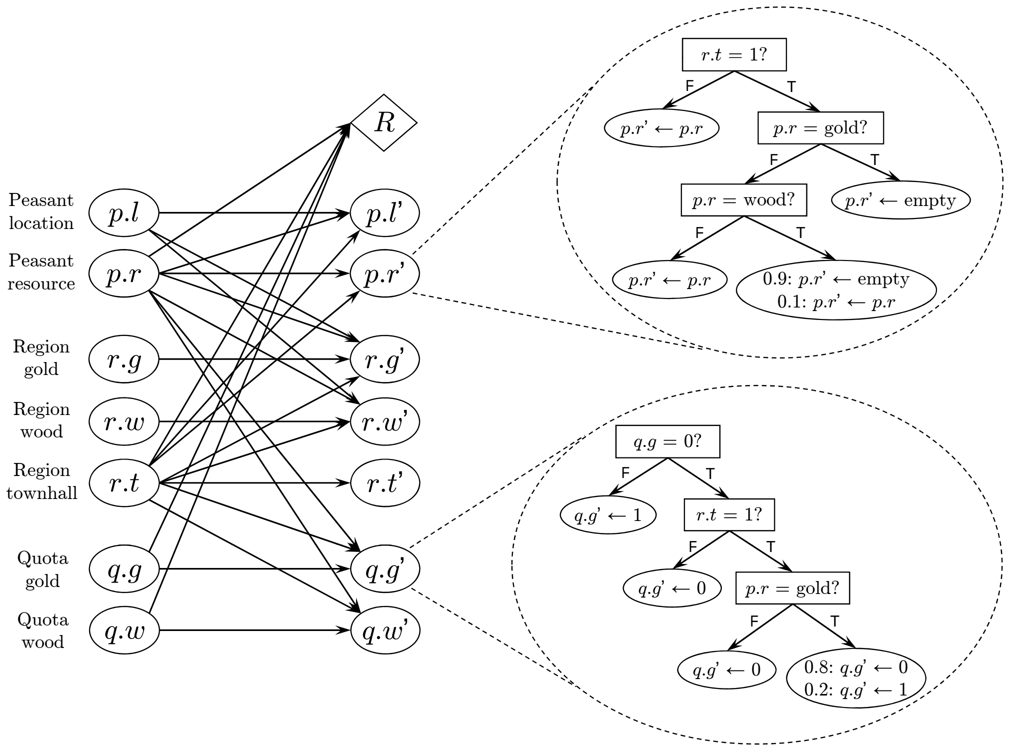


Figure 5.4: The dynamic probabilistic network model for the Deposit action in the Wargus domain. The left and right columns of elliptical nodes represent the state variables before and after the action is executed. The diamond node labeled R represents the immediate reward received after the action is executed. The peasant resource $p.r'$ and the gold quota $q.g'$ nodes have been expanded to show the decision tree representation of their conditional probability distributions. The internal nodes of the tree check conditions involving the state variables, such as the peasant being near a townhall ($r.t = 1$?), and the leaves contain the distributions over resultant values, such as executing Deposit near a townhall when carrying gold results in the peasant's resource being empty ($p.r' \leftarrow$ empty) with probability 1, or that the gold quota is met with probability 0.2 if all the conditions are satisfied.

network (DPN) model for action a is a directed acyclic bipartite graph $\mu_a = (\mathcal{X} \cup \mathcal{X}', E_a)$ where $\mathcal{X}$ is the set of vertices in the first stage and $\mathcal{X}' = \{x'_1, \ldots, x'_n, R\}$ is the set of vertices in the second stage. The two stages represent the values of the (random) state variables at times t and $t+1$ when a is executed at time t. A DPN for the Deposit action in Wargus is shown in Figure 5.4. The reward function is also represented within the action-based DPNs by incorporating a special (diamond-shaped) reward vertex R in the second stage.

Every edge $(u, v) \in E_a$ represents a direct probabilistic dependency of v on u when a is executed (Dean and Kanazawa, 1990). Let $\text{Parents}_{\mu_a}(v) = \{u : (u, v) \in E_a\}$. For a set of variables X, $\text{Parents}_{\mu_a}(X) = \bigcup_{x \in X} \text{Parents}_{\mu_a}(x)$. Diachronic edges are directed from variables at time t to variables at time $t+1$; synchronic edges are directed between vertices in the same time-step. A simple DPN lacks synchronic edges, that is, $\forall (u, v) \in E_a$, $u \in \mathcal{X} \wedge v \in \mathcal{X}'$; the associated graph is bipartite, and $\forall x$, $\text{Parents}_{\mu_a}(x) \subseteq \mathcal{X}$.[1] Events that cause state transitions but are considered beyond the control of the agent are called *exogenous* events. Such events are not the effect of any particular action, but implicit-event models merge exogenous events together with the transition dynamics of the actions (Boutilier et al., 1999). The system dynamics of a set of actions A is defined as $\mu_A = \bigcup_{a \in A} \mu_a = \left(\mathcal{X} \cup \mathcal{X}', \bigcup_{a \in A} E_a\right)$. Thus, the complete transition dynamics can be captured by a single DPN with an additional decision node representing the action choice. While this has certain advantages such as exploiting regularities across actions and succinctly representing exogenous effects, it normally introduces many unwanted dependencies between the variables because the individual DPN edges are unioned together.

5.1.2.1 Structured Conditional Probability Distribution

A conditional probability distribution (CPD), $\Pr(x' | \text{Parents}_{\mu_a}(x'), a)$, is associated with each node x' in the second stage. As the Markov property renders prior history irrelevant, the two stages suffice to capture the complete transition probabilities succinctly, and the variables do not influence each other within the same time stage (no synchronic effects).

[1]A generic DPN can be made simple by agglomerating the synchronically related variables into compound variables.

The transition function is now calculated as

$$\mathcal{P}(\mathbf{x}, a, \mathbf{x}') = \Pr(\mathbf{x}'|\mathbf{x}, a) = \prod_{x' \in \mathbf{x}'} \Pr(x'|\text{Parents}_{\mu_a}(x'), a).$$

The HI-MAT approach to discovering task hierarchies relies on compact, inspectable action models. We assume that this conditional probability distribution is represented as a decision tree that captures context-specific independence. The internal choice nodes of the decision tree at DPN node x' test the values of the Parents(x') and the leaf nodes specify a probability distribution over the values of x' in the resulting state. The tree structure for $p.r'$ represents the fact that it remains unchanged if the peasant is not near a townhall or when it is not carrying gold or wood; otherwise, the variable changes (with high probability) to reflect the fact that the Deposit succeeds and $p.r$ becomes empty. This captures the fact that the probability distribution over $p.r'$ depends on the context — it is more compact than representing the probability distribution as a table that enumerates all possible combinations of values of the parents (Boutilier et al., 1996). Similarly, a conditional reward tree is associated with the reward node R and Parents$_{\mu_a}(R)$ is the set of the state variables that are tested at R. Technically, DPNs with reward (and decision) nodes are called *influence diagrams* or *decision networks* (Russell and Norvig, 2003), but we refer to these models of the system (transition and reward) dynamics as DPNs when the usage is unambiguous.

5.1.2.2 Variable Closure

State abstraction is an integral part of hierarchical decomposition. When discovering and tackling subproblems, it is the ability to suitably abstract the state information that actually expedites the solution to the overall problem. In fact, defining subproblems without state abstraction normally results in a decomposition that is harder to solve than the original undecomposed problem. The structural information of a DPN model can be analyzed to determine the state abstraction for a task; this analysis hinges on the concept of closure.

Definition 5.1. The i^{th}-stage *closure* of variable x with respect to the DPN model μ_a for action a is defined as $\chi^i_{\mu_a}(x) = \text{Parents}_{\mu_a}(\chi^{i-1}_{\mu_a}(x)') \cup \{x\}$, where $\chi^0_{\mu_a}(x) = \{x\}$.

According to this definition, the set $\chi^i_{\mu_a}(x)$ keeps growing strictly monotonically with respect to i until it converges for some $i \leq |\mathcal{X}|$, because $\forall i, \chi^i_{\mu_a}(x) \subset \mathcal{X}$. This maximal set, denoted as $\chi^*_{\mu_a}(x)$, is simply called the closure of x with respect to μ_a.

The closure of a variable x with respect to a set of actions A is denoted as $\chi^*_{\mu_A}(x)$, where $\mu_A = \bigcup_{a \in A} \mu_a$. If $A \supseteq B$, then $\chi^*_{\mu_A}(x) \supseteq \chi^*_{\mu_B}(x)$, because an edge exists in μ_A if it exists in μ_B. The following example illustrates that $\chi^*_{\mu_A}(x) \neq \bigcup_{a \in A} \chi^*_{\mu_a}(x)$. Let $\mathcal{X} = \{x_1, x_2, x_3\}$, $A = \{a_1, a_2\}$, $\mu_{a_1} = (\mathcal{X} \cup \mathcal{X}', \{(x_1, x_2')\})$, and $\mu_{a_2} = (\mathcal{X} \cup \mathcal{X}', \{(x_2, x_3')\})$. Thus, $\chi^*_{\mu_{a_1}}(x_3) = \{x_3\}$ and $\chi^*_{\mu_{a_2}}(x_3) = \{x_2, x_3\}$. Now, $\mu_A = (V, \{(x_1, x_2'), (x_2, x_3')\})$, and $\chi^*_{\mu_A}(x_3) = \{x_1, x_2, x_3\}$. Therefore, $\chi^*_{\mu_A}(x_3) \neq \bigcup_{a \in A} \chi^*_{\mu_a}(x_3)$.

For a set of variables X, the i-th stage closure with respect to the DPN model μ_a for action a is $\chi^i_{\mu_a}(X) = \text{Parents}_{\mu_a}(\chi^{i-1}_{\mu_a}(X)') \cup X$, where $\chi^0_{\mu_a}(x) = X$; $\chi^*_{\mu_a}(X)$ is the corresponding maximal set.

Lemma 5.1. *Given a set of variables X and a set of actions A, $\chi^*_{\mu_A}(X) = \bigcup_{x \in X} \chi^*_{\mu_A}(x)$.*

Proof. We first show inductively that $\forall i$, $\bigcup_{x \in X} \chi^i_{\mu_A}(x) = \chi^i_{\mu_A}(X)$.

- Base step:
$$\bigcup_{x \in X} \chi^0_{\mu_A}(x) = \bigcup_{x \in X} \{x\} = X = \chi^0_{\mu_A}(X).$$

- Inductive step: Assume that $\chi^n_{\mu_A}(X) = \bigcup_{x \in X} \chi^n_{\mu_A}(x)$.

$$\begin{aligned}
\therefore \bigcup_{x \in X} \chi^{n+1}_{\mu_A}(x) &= \bigcup_{x \in X} \Big(\text{Parents}_{\mu_A}\big(\chi^n_{\mu_A}(x)'\big) \cup \{x\}\Big) \\
&= \bigcup_{x \in X} \text{Parents}_{\mu_A}\big(\chi^n_{\mu_A}(x)'\big) \cup \bigcup_{x \in X} \{x\} \\
&= \text{Parents}_{\mu_A}\Big(\textstyle\bigcup_{x \in X} \chi^n_{\mu_A}(x)'\Big) \cup X \\
&= \text{Parents}_{\mu_A}\big(\chi^n_{\mu_A}(X)'\big) \cup X = \chi^{n+1}_{\mu_A}(X).
\end{aligned}$$

As every variable's i^{th}-stage closure converges by at most $|\mathcal{X}|$ stages,

$$\chi^*_{\mu_A}(X) = \chi^{|\mathcal{X}|}_{\mu_A}(X) = \bigcup_{x \in X} \chi^{|\mathcal{X}|}_{\mu_A}(x) = \bigcup_{x \in X} \chi^*_{\mu_A}(x).$$

□

As the immediate reward node R is also included in the DPNs, the above definition of closure extends to R, barring the minor change that the closure excludes R itself. For a goal condition G, $\chi^*_{\mu_A}(G)$ denotes the closure of the set of variables in G.

Lemma 5.2. *For two action models $\mu_A = (V, E_A)$ and $\mu_B = (V, E_B)$, $\chi^*_{\mu_A}(\cdot) = \chi^*_{\mu_B}(\cdot)$ if $E_A = E_B$.*

Proof. This follows directly from the definition of closure. □

5.1.2.3 Variable Properties

Based on the action model μ_a for action a, a state variable $x \in \mathcal{X}$ can have the following properties.

Definition 5.2. x *persists* through a iff $\forall \mathsf{x} \in \mathcal{D}(x)$, $\Pr(x' = \mathsf{x} | x = \mathsf{x}, a) = 1$, that is, x stays unchanged every time a is executed. Otherwise, x is *changed* by a.

Definition 5.3. x is *checked* by a iff $x \in \text{Parents}_{\mu_a}(R)$ or there exists $y \in \mathcal{X}$ such that y does not persist through a and $x \in \text{Parents}_{\mu_a}(y')$.

Although an edge exists from x to x' in μ_a, x is not checked by a if it is the parent of only x' and it persists through a. In the action model for the Deposit action in Figure 5.4, $r.t$ persists through Deposit and all the state variables are checked by Deposit.

If the CPDs in μ_a are tree-structured, then state s induces a path from the root to a leaf in every CPD. The contextual DPN is denoted as $\mu_a(s)$ and $\text{Parents}_{\mu_a(s)}(\cdot)$ is the contextual equivalent of $\text{Parents}_{\mu_a}(\cdot)$.

Definition 5.4. x *context-persists* through a in s iff $\Pr(x' = \mathsf{x} | x = \mathsf{x}, s, a) = 1$, where x is the value of x in s. Otherwise, x is *context-changed* by a in s.

Definition 5.5. x is *context-checked* by a in s iff $x \in \text{Parents}_{\mu_a(s)}(R)$ or there exists $y \in \mathcal{X}$ that does not context-persist through a in s and $x \in \text{Parents}_{\mu_a(s)}(y')$.

When the Deposit action is executed in any state s in which the peasant is carrying wood, $q.g$ context-persists through Deposit in s and all state variables except $q.g$ are context-checked by Deposit in s.

$$\begin{tikzcd}
\mathsf{Start} \arrow[r, "p.l", "r.*"'] \arrow[rr, bend left, "p.r"] \arrow[rrrrrrrr, bend left=40, "q.w"] \arrow[rrrr, bend right, "q.g"'] & \mathsf{Goto} \arrow[r, "p.l", "r.*"'] & \mathsf{MG} \arrow[r, "p.l", "r.*"'] \arrow[rr, bend left, "p.r"] & \mathsf{Goto} \arrow[r, "p.l", "r.*"'] & \mathsf{Dep} \arrow[r, "p.l", "r.*"'] \arrow[rr, bend left, "p.r"] \arrow[rrrrr, bend right, "q.g"'] & \mathsf{Goto} \arrow[r, "p.l", "r.*"'] & \mathsf{CW} \arrow[r, "p.l", "r.*"'] \arrow[rr, bend left, "p.r"] & \mathsf{Goto} \arrow[r, "p.l", "r.*"'] & \mathsf{Dep} \arrow[r, "{p.*,\ r.*,\ q.w}"] & \mathsf{End}
\end{tikzcd}$$

Figure 5.5: A relevantly annotated trajectory (RAT) for the Wargus domain. Variables: $p.l$ = peasant location; $p.r$ = peasant resource; $r.g$, $r.w$, $r.t$ = regions; $q.g$, $q.w$ = quotas. Actions: Goto, MG = MineGold, CW = ChopWood, Dep = Deposit; the trajectory is prepended with Start which sources all the variables, and appended with End which sinks all the variables. An edge goes from one action to another (later in the trajectory) when a variable is relevant to both, but to no intervening action; edges are labeled with the associated variables. For succinctness, $p.* = \{p.l, p.r\}$, and $r.* = \{r.g, r.w, r.t\}$.

5.2 The HI-MAT Approach

We consider MDPs where the agent is solving a known conjunctive goal. This is a subset of the class of stochastic shortest-path MDPs. In such MDPs, there is a goal state (or a set of goal states), and the optimal policy for the agent is to reach such a state as quickly as possible. We assume that we are given factored DPN models for the source MDP, where the conditional probability distributions are represented as trees (CPTs). Further, we are given a successful trajectory that reaches the goal in the source MDP. With this in hand, the objective is to automatically induce a MAXQ hierarchy that can suitably constrain the policy space when solving a related target problem, and therefore transfer to achieve faster convergence in the target problem. This is achieved via recursive partitioning of the given trajectory into subtasks using a top-down parse guided by backward chaining from the goal. We use the DPNs along with the trajectory to define the termination predicate, the set of subtasks, and the relevant abstraction for each MAXQ subtask.

5.2.1 Relevance Annotation

The input trajectory is a sequence of actions that achieves the overall goal in the source problem. A trajectory in the Wargus domain is a sequence of Goto, MineGold, ChopWood, and Deposit actions that together achieves the requisite quota of gold and wood. The trajectory is first annotated with relevant information using the DPN models before it is fed to the HI-MAT algorithm. The intent of this annotation is to identify how executing actions affects the state variables that are relevant to future actions and, ultimately, the goal of the overall task. For example, the annotation must indicate that the MineGold and ChopWood actions enable the subsequent Deposit action by setting the peasant's resource $p.r$ to something other than empty, which is required for Deposit to succeed.

The trajectory annotation is based on the relevance of variables to actions which is gleaned from the DPNs.

Definition 5.6. A variable v is *relevant* to an action a in state s if v is either context-checked or context-changed by a in s; it is *irrelevant* otherwise.

In Figure 5.4, the peasant being near a townhall ($r.t$) is always relevant to Deposit, because it is in the root node of the tree, but the peasant's resource ($p.r$) is only relevant when the peasant is near a townhall.

Definition 5.7. A *relevant* edge $a \xrightarrow{v} b$ connects a to another action b (b following a in the trajectory) iff v is relevant to both a and b and irrelevant to all actions in between. A *relevantly annotated trajectory* (RAT) is the original trajectory annotated with all the relevant edges, sandwiched between dummy Start and End actions for which all variables are defined to be relevant.

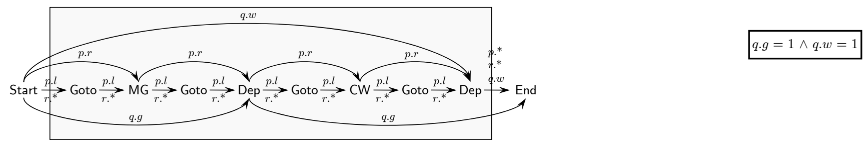
Given $a \xrightarrow{v} b$, the literal on a relevant edge refers to a formula of the form $v = \mathsf{v}$, where v is the value taken by v in the state before b is executed. The RAT is preprocessed to remove any cycles present in the original trajectory (this includes failed actions, such as an unsuccessful Deposit when the peasant is not near a townhall). A sample RAT for the Wargus domain is shown in Figure 5.5.

5.2.2 The HI-MAT Algorithm

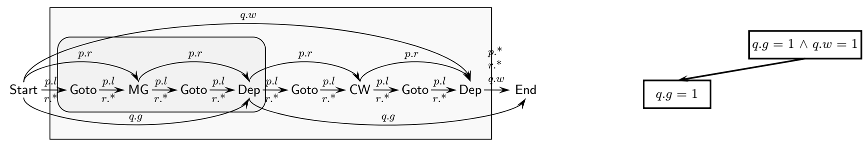
Given a RAT, the DPN models, and the MDP's goal (or recursively, the current subtask's goal) as input, the HI-MAT algorithm discovers hierarchical structure by recursively partitioning the RAT. Each partition corresponds to a candidate subtask. This partitioning process works backward from the goals of the task. It regresses each goal through the action models to discover preconditions and employs a heuristic procedure

Algorithm 5.1 HI-MAT

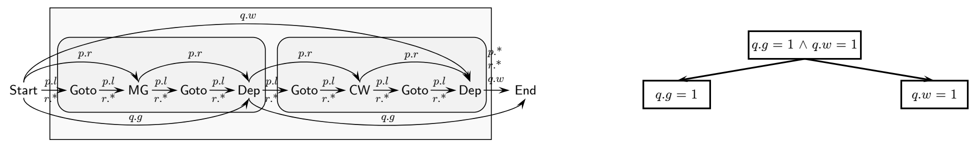
```
Input: Action models $\{\mu_a\}$, RAT $\Omega$, Goal predicate $G$.
Output: Task $(X, G, C)$.
1: $n \leftarrow$ number of actions in $\Omega$ excluding Start and End
2: if SameActionFamily$(\{\mu_a\}, \Omega)$ then
3:     $A \leftarrow$ set of actions in $\Omega$
4:     return (RelevantVariables$(\mu_A, G), G, A)$
5: $\Psi \leftarrow \varnothing$ // Set of trajectory segments
6: $U \leftarrow$ Literals$(G)$
7: while $U \neq \varnothing$ do
8:     Pick $u \in U$
9:     $j \leftarrow$ index of the action in $\Omega$ that achieves $u$
10:    $i \leftarrow$ RAT-Scan$(\Omega, j)$
11:    if $i = 1 \wedge j = n$ then
12:        $\Psi \leftarrow \Psi \cup \{(1, n-1, v) : v \in$ Precondition$(a_n)\}$
13:        $\Psi \leftarrow \Psi \cup \{(n, n, \varnothing)\}$
14:    else if $j > 0$ then // Last segment action $\neq$ Start
15:        $\Psi \leftarrow \Psi \cup \{(i, j, u)\}$
16:        $U \leftarrow U \cup \{v : \exists k < i \; \exists l \; a_k \xrightarrow{v} a_l \in \Omega, i \leq l \leq j\}$
17:    $U \leftarrow U - \{u\}$
18: while $\exists (i, j, u_1), (i, j, u_2) \in \Psi$ do
19:    $\Psi \leftarrow (\Psi - \{(i, j, u_1), (i, j, u_2)\}) \cup \{(i, j, u_1 \wedge u_2)\}$
20: $X \leftarrow \varnothing$
21: $C \leftarrow \varnothing$
22: for $t \in \Psi$ do
23:    $(X_t, G_t, C_t) \leftarrow$ HI-MAT(Extract$(\Omega, t_i, t_j), t_u)$
24:    $X \leftarrow X \cup X_t$
25:    $C \leftarrow C \cup \{(X_t, G_t, C_t)\}$
26: $A \leftarrow$ set of the primitive descendants of $T$
27: return $(X \cup$ RelevantVariables$(\mu_A, G), G, C)$
```



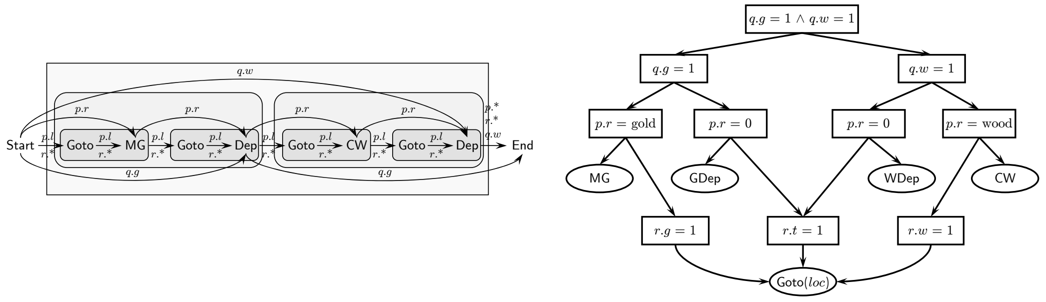
(a) The entire RAT is associated with the root task of collecting gold and wood. The root task terminates when the requisite amount of gold and wood have been collected.



(b) The segment corresponding to collecting the requisite amount of gold ($q.g = 1$) is extracted yielding the associated subtask.



(c) The segment corresponding to collecting the requisite amount of wood ($q.w = 1$) is extracted. Other literals entering this segment result in segments that are merged together with the $q.g = 1$ segment (merged goal condition not shown for clarity).



(d) The resultant task hierarchy after an entire parse of the RAT. The multiple invocations of the task that gets the peasant to the townhall ($r.t = 1$) within the RAT are merged into one unique task. However, the tasks that empty the peasant's resource ($p.r = 0$) are not merged because, while GDep and WDep are proxies for the Deposit action, the former's abstraction includes $q.g$ while that of the latter includes $q.w$ instead. The primitive Goto(loc) action is wrapped by tasks that get the agent to particular types of regions.

Figure 5.6: Illustrating the HI-MAT algorithm for the Wargus domain. The left-hand side shows the RAT scanning process, while the right-hand side shows the task hierarchy as it is being built.

to determine small RAT segments responsible for achieving the goal. For example, to achieve the overall goal in the Wargus domain, a Deposit action must be executed. By analyzing the DPN for Deposit and considering the conditions that allow $q.g$ to be set to 1, the subgoal of $p.r$ = gold is identified, which then leads to the discovery of the subsequence of Goto and MineGold actions that achieve this new subgoal, and so on. The HI-MAT algorithm is outlined in Algorithm 5.1 and illustrated in Figure 5.6.

HI-MAT first checks if the base criteria is satisfied (line 2), that is, the trajectory only contains actions from the same family (e.g., the Goto actions). This limits the decomposition in order to reduce the number of parameters that need to be learned and to facilitate the transfer of the hierarchical structure. Otherwise, it first initializes the set of unprocessed goals to the set of literals in the goal conjunction (line 6). It then selects any unprocessed goal u, and extracts the corresponding segment (line 10). In Figure 5.6c, the segment associated with collecting the wood quota ($q.w = 1$) has two incoming edges: one for the peasant's resource $p.r$, and the other for the peasant's location $p.l$. Consequently, HI-MAT extracts two segments associated with these literals. However, both these segments overlap with the segment that collects the gold quota ($q.g = 1$). (Extracted segments can only overlap fully when their shared ultimate action achieves the literals of all these segments.) HI-MAT merges the overlapping segments by replacing them with one that is assigned a conjunction of the subgoal literals (line 19). Thus, the merged goal condition is $q.g = 1 \wedge p.r =$ empty $\wedge\, p.l =$ townhall, but the second and third literals are always true when the first is true; we leave them out to reduce clutter.

RAT-Scan returns the index i of the first action in the RAT segment that achieves the literal under consideration. If this RAT segment is nontrivial (neither just the initial state nor the whole trajectory), it is stored (line 15), and the literals on relevant edges that enter it (from earlier in the trajectory) are added to the unprocessed goals (line 16). This ensures that the algorithm parses the entire trajectory barring redundant actions. If the trajectory segment is equal to the entire trajectory, this implies that the trajectory achieves only the literal u after the ultimate action. In this case, the trajectory is split into two segments: one segment contains the prefix of the ultimate action a_n with the preconditions of a_n forming the goal literals for this segment (line 12); the other segment contains only the ultimate action a_n (line 13). For example, the RAT associated with collecting gold cannot be split further based on the annotation so it is forcibly split

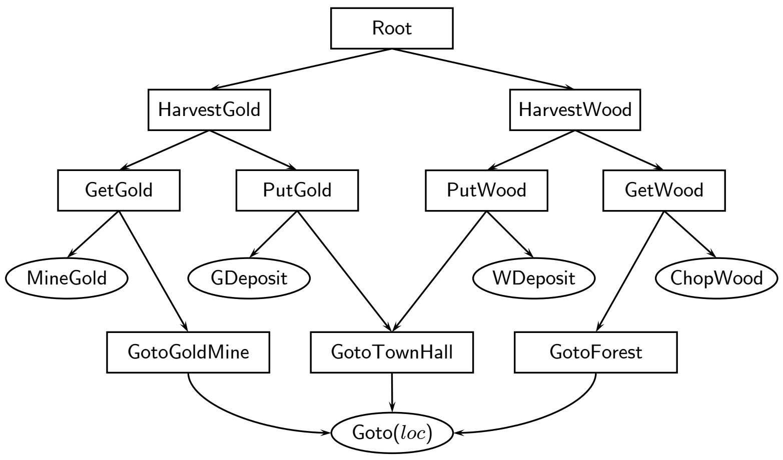


Figure 5.7: The HI-MAT hierarchy for the Wargus domain. The task names have been assigned intuitively based on the termination conditions.

into two segments: one containing only the ultimate Dep action and another containing all actions prior to Dep within the RAT whose goal is to achieve the goal of getting the peasant back to the townhall with some gold. RAT scanning is repeated until all subgoal literals are accounted for.

The HI-MAT algorithm partitions the RAT into unique segments, each achieving a single literal or a conjunction of literals due to merging. It is called recursively on each element of the partition (line 23). Figure 5.7 shows the final task hierarchy induced by HI-MAT in the Wargus domain with intuitive names for the subtasks based on their termination conditions. The TGoto(townhall) subtasks have been merged by HI-MAT, because discovered subtasks with identical termination conditions and identical child tasks (recursively) are recognized as multiple calls to a unique subtask.

5.2.2.1 Subtask Discovery

Given a literal, a subtask is determined by finding the set of temporally contiguous actions that are closed with respect to the relevant edges in the RAT such that the final action achieves the literal. The idea is to group all actions that contribute to achieving the specific literal being considered. Given an action index, Algorithm 5.2 extracts the longest segment in the RAT such that no other actions within the segment have any

Algorithm 5.2 RAT-Scan

```
Input: RAT $\Omega$, end index $j$.
Output: Start index $i$.
1: $i \leftarrow j - 1$
2: while $i > 0$ and $\forall v\ \exists k\ a_i \xrightarrow{v} a_k \implies k \leq j$ do
3:     $i \leftarrow i - 1$
4: return $i + 1$
```

causal influences outside it. Because of the way the RAT is constructed, a segment will never include an action whose set of relevant variables is not a subset of the set of variables relevant to the final action in the segment. The temporal contiguity of the segment that we assign to a task is required by the subroutine semantics of a hierarchical policy — a hierarchical MAXQ policy cannot interrupt an unterminated subtask, start executing a sibling subtask, and then return to executing the interrupted subtask. In Figure 5.6b, HI-MAT extracts the segment responsible for collecting the gold quota (achieving $q.g = 1$), and discovers the subtask associated with the segment as a child of the root task. Figure 5.6c shows how HI-MAT extracts the segment for collecting the wood quota (achieving $q.w = 1$) by stopping the head of the segment at the action after the Dep action that achieves $q.g = 1$. The causal annotation allows HI-MAT to correctly incorporate any extra Goto actions that might be present in either of these segments.

5.2.2.2 Child Tasks and Termination

To define a task, we must specify the set of child tasks that it can invoke and the termination predicate. The child tasks are those tasks that are associated with any of the trajectory segments at the next recursive level of the HI-MAT algorithm. The termination predicate is computed by taking the literal that was achieved by the segment and generalizing it subject to the conditions that appear in the DPNs; we consider the relational test(s) t_u in the action and reward DPNs involving the variable u on the relevant edge leaving the subtask (line 23 of Algorithm 5.1). For example, although the specific condition for getting the peasant to a townhall involves both the peasant's location $p.l$ and the townhall indicator $r.t. = 1$, the DPN only checks the latter and consequently only $r.t = 1$ is the termination condition. Moreover, if the DPNs also provide the primary effects of the actions, then further decomposition is possible. For instance, the Put Gold and Put Wood tasks are discovered because the DPN for Deposit specifies that the primary effect of the action besides satisfying the requisite quotas is to empty the peasant's resource. The root task is associated with the entire RAT and its termination condition is equal to the MDP's goal predicate.

5.2.2.3 State Abstraction

Every task's local policy in the hierarchy is dependent only on a subset of the state variables. For instance, the agent can be oblivious to the location of forests when executing the task for collecting gold. HI-MAT tries to assign the smallest set of relevant state variables to every task in its induced hierarchy to help expedite the learning of the local task policies. The state abstraction for a primitive task a is $\chi^1_{\mu_a}(R)$ if the reward is not stochastic and it is $\chi^2_{\mu_a}(R)$ otherwise (line 4). For a composite task, HI-MAT first constructs the merged DPN μ_A for the set of actions A in the segment associated with the task. The state abstraction for a task $T = (X_T, G, C)$ is $X_T = \chi^*_{\mu_A}(R) \cup \chi^*_{\mu_A}(G)$ if the primitive actions all have the same relevance (line 4) and $X_T = \chi^*_{\mu_A}(R) \cup \chi^*_{\mu_A}(G) \cup \bigcup_{c \in C} X_c$ otherwise (lines 24 and 27). The next section on the theoretical analysis of HI-MAT justifies these choices for state abstraction.

Adding the variables that do not change within a task to its state abstraction has the effect of creating a task with these added variables as the formal parameters or arguments. In HI-MAT, this parameterization is implicit in the termination conditions rather than being explicit as in Dietterich (2000).

Computing the relevant variables is similar to explanation-based reinforcement learning (Tadepalli and Dietterich, 1997) except that here we care only about the set of relevant variables and not their values. Moreover, the relevant variables are computed over a set rather than a sequence of actions.

5.2.2.4 Generalizing the Task Hierarchy

Because the RAT is based on a single successful trajectory in the source problem, not all primitive actions might be observed. Incorporating only the observed actions might limit the scope of transfer. To make the task hierarchy more generally applicable, HI-MAT checks if there are unobserved primitive actions that can be incorporated into a task

T with primitive child tasks without adding more state variables to T's current state abstraction (not shown in Algorithm 5.1). It does this by incorporating an unobserved action if its DPN is a subgraph of μ_A where A is the set of primitive children of T. The rationale here is to add unobserved actions that have the same causal effects as those of the child actions. For example, the given trajectory might only involve a few of the Goto actions, but all unobserved Goto actions are added to the induced task hierarchy, because they all affect the same set of variables. However, if one of the Goto actions affects a different variable, then it will not be incorporated even if it might turn out to be useful in a target task.

5.3 Theoretical Analysis

This section establishes certain theoretical properties of the hierarchies induced by the HI-MAT algorithm. A task hierarchy H is a directed acyclic graph in which the nodes are tasks and the edges represents the task-subtask relationships. We begin by analyzing the consistency of the HI-MAT hierarchy with respect to the trajectory.

Definition 5.8. A trajectory $(s_1, a_1, \ldots, s_n, a_n, s_{n+1})$ is *non-redundant* if no subsequence of the action sequence in the trajectory can be removed such that the remaining sequence still achieves the goal starting from s_1.

Definition 5.9. H is consistent with a trajectory-task pair $\langle \tau, T_i \rangle$, where $\tau = (s_1, a_1, \ldots, s_n, a_n, s_{n+1})$ and $T_i = (X_i, G_i, C_i)$, if $T_i \in H$ and

1. T_i is the primitive task a_1 and $n = 1$
2. T_i is not primitive, $\{s_1, \ldots, s_n\} \cap G_i = \varnothing$, $s_{n+1} \in G_i$, and there exists a set of trajectory-task pairs $\{\langle \tau_j, T_j \rangle : 1 \leq j \leq p\}$ consistent with H such that τ is a concatenation of $\tau_1, \ldots, \tau_p$ and $T_1, \ldots, T_p \in C_i$.

H is consistent with a trajectory τ if H is consistent with $\langle \tau, T_0 \rangle$ and T_0 is the root task in H.

Theorem 5.1. *If a trajectory τ is non-redundant, then HI-MAT produces a hierarchy H that is consistent with τ.*

Proof. Let $\tau = (s_1, a_1, \ldots, s_n, a_n, s_{n+1})$ be the trajectory. The algorithm extracts the conjunction of literals that are true in s_{n+1} (and not before), and assigns it to the goal, G_i. Such literals must exist — otherwise, some suffix of the trajectory can be removed while the rest still achieves the goal, violating the property of non-redundancy. Whenever the trajectory is partitioned into a sequence of subtrajectories, each subtrajectory is associated with a conjunction of goal literals achieved by that subtrajectory and the above argument applies recursively to each such subtrajectory. □

We now analyze the state abstraction of the HI-MAT hierarchy. For a set of variables X, let $\overline{X} = \mathcal{X} - X$ and let $\mathbf{x}$ denote the values taken by the variables in X. Let $V_T(s)$ represent the total expected reward received during a task T starting in state s.

Definition 5.10. If X and Y partition the set of state variables $\mathcal{X}$ and π is a hierarchical policy, then Y is *irrelevant* to the value function V_T^π of task T if, for any pair of states $s_1 = (\mathbf{x}, \mathbf{y}_1)$ and $s_2 = (\mathbf{x}, \mathbf{y}_2)$, $V_T^\pi(s_1) = V_T^\pi(s_2)$.

The leaf-irrelevance lemma (Lemma 5) in Dietterich (2000) states that if, for all pairs of states $s = (\mathbf{x}, \mathbf{y})$ and $s' = (\mathbf{x}', \mathbf{y}')$,

$$\mathcal{R}(s, a, s') = \mathcal{R}(\mathbf{x}, a, \mathbf{x}') \text{ and } \Pr(\mathbf{x}', \mathbf{y}' | \mathbf{x}, \mathbf{y}, a) = \Pr(\mathbf{x}' | \mathbf{x}, a) \Pr(\mathbf{y}' | \mathbf{x}, \mathbf{y}, a) \tag{5.1}$$

at a primitive task a, then Y is irrelevant to V_a. The notation $\mathcal{R}(s, a, s') = \mathcal{R}(\mathbf{x}, a, \mathbf{x}')$ is shorthand for representing the fact that $\mathcal{R}$ depends only on the values of the variables in X in states s and s'. The following lemma relates leaf irrelevance to the closure of the reward node $\chi^*_{\mu_a}(R)$ in the action model μ_a of a.

Lemma 5.3. *$\overline{X}$ is irrelevant to V_a if $X = \chi^*_{\mu_a}(R)$.*

Proof. If $X = \chi^*_{\mu_a}(R)$ and $Y = \overline{X}$, then for all pairs of states $s = (\mathbf{x}, \mathbf{y})$ and $s' = (\mathbf{x}', \mathbf{y}')$, $\mathcal{R}(s, a, s') = \mathcal{R}(\mathbf{x}, a, \mathbf{x}')$, because the reward depends on $\text{Parents}_{\mu_a}(R)$, which is a subset of $\chi^*_{\mu_a}(R)$ by definition. Moreover, $\Pr(\mathbf{x}', \mathbf{y}' | \mathbf{x}, \mathbf{y}, a) = \Pr(\mathbf{x}' | \mathbf{x}, a) \Pr(\mathbf{y}' | \mathbf{x}, \mathbf{y}, a)$, because an arc cannot exist from Y (outside the closure) to X (within the closure) by definition. Consequently, the leaf-irrelevance lemma applies. □

We improve on the leaf-irrelevance result by showing that a potentially larger set of variables is irrelevant to V_a.

Lemma 5.4. *Let $Z \subseteq \mathcal{X}$ be a set of variables. If $\mathcal{R}(s,a,s') = \mathcal{R}(\mathbf{z},a,\mathbf{z}')$ and $X = Z \cup \text{Parents}_{\mu_a}(Z')$, then $\overline{X}$ is irrelevant to V_a.*

Proof. The value of state s for the primitive task a is

$$\begin{aligned} V_a(s) &= \sum_{s'} \Pr(s'|s,a)\mathcal{R}(s,a,s') \\ &= \sum_{\mathbf{z}',\bar{\mathbf{z}}'} \Pr(\mathbf{z}',\bar{\mathbf{z}}'|\mathbf{z},\bar{\mathbf{z}},a)\mathcal{R}(\mathbf{z},a,\mathbf{z}') \\ &= \sum_{\mathbf{z}',\bar{\mathbf{z}}'} \Pr(\mathbf{z}'|\mathbf{z},\bar{\mathbf{z}},a)\Pr(\bar{\mathbf{z}}'|\mathbf{z},\bar{\mathbf{z}},a)\mathcal{R}(\mathbf{z},a,\mathbf{z}'). \end{aligned}$$

The distribution factors because there is no synchronic arc from $\bar{Z}$ to Z. If an arc does exist, then $\mathcal{R}(s,a,s') \neq \mathcal{R}(\mathbf{z},a,\mathbf{z}')$ as the reward would depend on $\bar{Z}$ as well.

$$\begin{aligned} \therefore V_a(s) &= \sum_{\bar{\mathbf{z}}'} \Pr(\bar{\mathbf{z}}'|\mathbf{z},\bar{\mathbf{z}},a)\sum_{\mathbf{z}'} \Pr(\mathbf{z}'|\text{Parents}_{\mu_a}(\mathbf{z}'),a)\mathcal{R}(\mathbf{z},a,\mathbf{z}') \\ &= \sum_{\mathbf{z}'} \Pr(\mathbf{z}'|\text{Parents}_{\mu_a}(\mathbf{z}'),a)\mathcal{R}(\mathbf{z},a,\mathbf{z}'). \end{aligned}$$

Hence, V_a depends only on the variables in $X = Z \cup \text{Parents}_{\mu_a}(Z') = \chi^2_{\mu_a}(R)$ and not on those in $\overline{X}$. □

This lemma proves that $\mathcal{X} - \chi^2_{\mu_a}(R)$ is irrelevant to V_a, which is an improvement over showing that $\mathcal{X} - \chi^*_{\mu_a}(R)$ is irrelevant (Lemma 5.3), because $\chi^2_{\mu_a}(R) \subseteq \chi^*_{\mu_a}(R)$.

Lemma 6 in Dietterich (2000) states that if $\mathcal{R}(s,a,s')$ is constant, then $\mathcal{X}$ is irrelevant to V_a. We generalize this result to show that $\overline{X}$ is irrelevant to V_a if the reward depends only on the values that the variables in X take in the current state.

Lemma 5.5. *$\overline{X}$ is irrelevant to V_a if $\mathcal{R}(s,a,s') = \mathcal{R}(\mathbf{x},a)$, that is, the reward only depends on the values that the variables in X take in state s.*

Proof. The value of state s for the primitive task a is

$$V_a(s) = \sum_{s'} \Pr(s'|s,a)\mathcal{R}(s,a,s') = \sum_{\mathbf{x}',\bar{\mathbf{x}}'} \Pr(\mathbf{x}',\bar{\mathbf{x}}'|\mathbf{x},\bar{\mathbf{x}},a)\mathcal{R}(\mathbf{x},a) = \mathcal{R}(\mathbf{x},a).$$

Hence, V_a depends only on the variables in $X = \text{Parents}_{\mu_a}(R) = \chi^1_{\mu_a}(R)$ and not on those in $\overline{X}$. $\square$

Lemma 5.6. *For an MDP $\mathcal{M} = (\mathcal{X}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, $\overline{X}$ is irrelevant to the value function V of $\mathcal{M}$ if $X \supseteq \chi^*_{\mu_\mathcal{A}}(R)$.*

Proof. If $X \supseteq \chi^*_{\mu_\mathcal{A}}(R)$, then no path can exist from a variable $v \in \overline{X}$ to a reward node in any probabilistic network that results from unrolling $\mu_\mathcal{A}$. If such a path exists, then $v \in \chi^*_{\mu_\mathcal{A}}(R)$ by definition. This implies that $v \in X$, which is a contradiction. Consequently, no variable in $\overline{X}$ has any influence on the reward received any time in the future and $\overline{X}$ is irrelevant to V. $\square$

To make a stronger claim, we identify a property of the DPNs.

Definition 5.11. A DPN model μ_a is *maximally sparse* if the following two conditions are satisfied:

1. If $x \in \mathcal{X}, y \in \text{Parents}_{\mu_a}(x')$, and $Z = \text{Parents}_{\mu_a}(x') - \{y\}$, then

$$\exists \mathsf{y}_1, \mathsf{y}_2 \in \mathcal{D}(y),\ \Pr(x'|\mathsf{y}_1, \mathbf{z}) \neq \Pr(x'|\mathsf{y}_2, \mathbf{z}).$$

2. If $y \in \text{Parents}_{\mu_a}(R)$ and $Z = \text{Parents}_{\mu_a}(R) - \{y\}$, then

$$\exists \mathsf{y}_1, \mathsf{y}_2 \in \mathcal{D}(y),\ R(\mathsf{y}_1, \mathbf{z}) \neq R(\mathsf{y}_2, \mathbf{z}).$$

Maximal sparseness implies that the parents of a variable or reward node have non-trivial influences on it — no parent can be removed without affecting the next-state distribution or the reward. For the maximal sparse DPN to be unique, all state variables must be non-redundant, that is, no two state variables should be identical in all states.

Lemma 5.7. *Assuming that $\mu_\mathcal{A}$ is maximally sparse for an MDP $\mathcal{M} = (\mathcal{X}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, $\overline{X}$ is irrelevant to the value function V of $\mathcal{M}$ if and only if $X \supseteq \chi^*_{\mu_\mathcal{A}}(R)$.*

Proof. (If) Lemma 5.6 applies.
(Only If) If $X \not\supseteq \chi^*_{\mu_\mathcal{A}}(R)$, then $\forall v \in \chi^*_{\mu_\mathcal{A}}(R) - X$, there exists an unrolled probabilistic network such that v is on a path to a reward node. Because $\mu_\mathcal{A}$ is maximally sparse, V must depend on v. Thus, $\overline{X}$ is not irrelevant to V. $\square$

By the application of Lemma 5.2, $\mathcal{X} - \chi^*_{\mu_\mathcal{A}}(R)$ is irrelevant to the value function V' of any MDP $\mathcal{M}' = (\mathcal{X}, \mathcal{A}', \mathcal{P}', \mathcal{R}')$ whose $\mu_{\mathcal{A}'}$ has the same structure as $\mu_\mathcal{A}$. Further, $\chi^*_{\mu_\mathcal{A}}(R)$ is the smallest set of variables that is relevant to V' if $\mu_{\mathcal{A}'}$ is also maximally sparse. From the transfer perspective, specifying only the edges of a DPN defines an abstract DPN that represents a family of domains. A particular domain is manifested by a specific instantiation of the parameters that satisfy the structural constraints, that is, no node x' in the instantiated DPN is affected by any node that is not a parent of x' in the abstract DPN. Consequently, the set of irrelevant variables computed from the abstract DPN is irrelevant to the value function of every member of this family. Further, if the maximally sparse DPN of a particular domain is structurally identical to that of the abstract DPN, then the set of relevant variables computed from the abstract DPN is also the smallest set of relevant variables for the particular domain.

We now determine the relevant variables for the value function of a composite task with primitive children.

Lemma 5.8. *If a task $T = (X, G, C)$ has only primitive children, then $\overline{X}$ is irrelevant to the value function of T if $X = \chi^*_{\mu_C}(R) \cup \chi^*_{\mu_C}(G)$.*

Proof. Let $\mathcal{M}' = (\mathcal{X}, C', \mathcal{P}', \mathcal{R}')$ be the MDP induced by overlaying the terminating task T on the original MDP $\mathcal{M} = (\mathcal{X}, C, \mathcal{P}, \mathcal{R})$. If we can show that $\chi^*_{\mu_{C'}}(R) = \chi^*_{\mu_C}(R) \cup \chi^*_{\mu_C}(G)$, then Lemma 5.6 applies and the proof is complete.

$\mathcal{M}'$ is an episodic MDP, that is, any state s that satisfies G is an absorbing state. As $\mu_{C'}$ must specify that $\mathcal{R}'(s, a, s') = 0$ and $\Pr(s' = s|s, a) = 1$ in an absorbing state s of $\mathcal{M}'$, there must exist arcs from the variables of G to every node in the second stage of $\mu_{C'}$. Besides these arcs, $\mu_{C'}$ has the same structure as μ_C. Therefore, in order to simulate the closure with respect to $\mu_{C'}$, we must explicitly add the variables in G at the first stage of the closure with respect to μ_C; equivalently, $\chi^*_{\mu_{C'}}(R) = \chi^*_{\mu_C}(R) \cup \chi^*_{\mu_C}(G)$. □

Finally, we determine the relevant variables for the value function of any task in the hierarchy. For this, we rely on Lemma 3 in Dietterich (2000): assuming X and Y partition $\mathcal{X}$, Y is irrelevant to the value function of task T if ① Equation 5.1 holds for every primitive descendant of T and ② for each composite task b that is equal to T or a descendant of T, $G_b(\mathbf{x}, \mathbf{y})$ is true iff $G_b(\mathbf{x})$ is true, that is, the terminal condition of b only depends on X.

Lemma 5.9. *If A_T is the set of primitive descendants of a task $T = (X_T, G_T, C_T)$ and B_T is the set of composite descendants of T, then $\overline{X}_T$ is irrelevant to V_T if*

$$X_T = \chi^*_{\mu_{A_T}}(R) \cup \chi^*_{\mu_{A_T}}(G_T) \cup \bigcup_{b \in B_T} \chi^*_{\mu_{A_b}}(G_b).$$

Proof. If $X_T = \chi^*_{\mu_{A_T}}(R) \cup \chi^*_{\mu_{A_T}}(G_T) \cup \bigcup_{b \in B_T} \chi^*_{\mu_{A_b}}(G_b)$, then it satisfies both conditions of Lemma 3 in Dietterich (2000). Consequently, $\overline{X}_T$ is irrelevant to V_T. □

A simple example shows that $\bigcup_{a \in A} \chi^*_{\mu_a}(R)$ cannot be substituted in for $\chi^*_{\mu_A}(R)$. Let $A = \{a_1, a_2\}$ be the set of actions. Let $\chi^*_{\mu_{a_1}}(R) = \varnothing$ and $\forall x' \in \mathcal{X}', \text{Parents}_{\mu_{a_1}}(x') = \mathcal{X}$. Let $\chi^*_{\mu_{a_2}}(R) = \{x_1\}$ and $\forall x' \in \mathcal{X}', \text{Parents}_{\mu_{a_2}}(x') = \varnothing$. Now, $X = \bigcup_{a \in A} \chi^*_{\mu_a}(R) = \{x_1\}$ and $\Pr(\mathbf{x}', \mathbf{y}' | \mathbf{x}, \mathbf{y}, a_1) \neq \Pr(\mathbf{x}' | \mathbf{x}, a_1) \Pr(\mathbf{y}' | \mathbf{x}, \mathbf{y}, a_1)$.

Definition 5.12. A hierarchy H is safe with respect to $\mu_{\mathcal{A}}$ if, for any task $T = (X_T, G_T, C_T)$ in H, $\overline{X}_T$ is irrelevant to V_T.

The above definition says that the state variables in each task are sufficient to capture the value of any trajectory consistent with the sub-hierarchy rooted at that task node.

Theorem 5.2. *If the procedure HI-MAT produces a task hierarchy H from τ and $\mu_{\mathcal{A}}$, then H is safe with respect to $\mu_{\mathcal{A}}$.*

Proof. HI-MAT computes the state abstraction for a task $T = (X_T, G_T, C_T)$ as

$$X_T = \begin{cases} \chi^*_{\mu_{A_T}}(R) \cup \chi^*_{\mu_{A_T}}(G_T) \cup \bigcup_{c \in C_T} X_c & \text{if } T \text{ is composite} \\ \chi^1_{\mu_{A_T}}(R) & \text{if } T \text{ is primitive and } \mathcal{R}(s, a, s') = \mathcal{R}(\mathbf{x}, a) \\ \chi^2_{\mu_{A_T}}(R) & \text{otherwise,} \end{cases}$$

where A_T is the set of primitive descendants of T. For a composite task T,

$$\begin{aligned} X_T &= \chi^*_{\mu_{A_T}}(R) \cup \chi^*_{\mu_{A_T}}(G_T) \cup \bigcup_{c \in C_T} X_c \\ &= \chi^*_{\mu_{A_T}}(R) \cup \chi^*_{\mu_{A_T}}(G_T) \cup \bigcup_{c \in C_T} \left(\chi^*_{\mu_{A_c}}(R) \cup \chi^*_{\mu_{A_c}}(G_c) \cup \bigcup_{b \in C_c} X_b \right), \end{aligned}$$

where B_T is the set of composite descendants of T. From the structural definition of a hierarchy, $A_T \supseteq \bigcup_{c \in C_T} A_c$, that is, the set of primitive descendants at a task must be a superset of the union of the primitive descendants of all its children. Consequently, $\chi^*_{\mu_{A_T}}(R) \supseteq \bigcup_{c \in C_T} \chi^*_{\mu_{A_c}}(R)$.

$$\begin{aligned}
\therefore X_T &= \chi^*_{\mu_{A_T}}(R) \cup \chi^*_{\mu_{A_T}}(G_T) \cup \bigcup_{c \in C_T} \left(\chi^*_{\mu_{A_c}}(G_c) \cup \bigcup_{b \in C_c} X_b \right) \\
&= \chi^*_{\mu_{A_T}}(R) \cup \chi^*_{\mu_{A_T}}(G_T) \cup \bigcup_{c \in C_T} \left(\chi^*_{\mu_{A_c}}(G_c) \cup \bigcup_{b \in C_c} \chi^*_{\mu_{A_b}}(G_b) \cup \dots \right) \\
&= \chi^*_{\mu_{A_T}}(R) \cup \chi^*_{\mu_{A_T}}(G_T) \cup \bigcup_{b \in B_T} \chi^*_{\mu_{A_b}}(G_b).
\end{aligned}$$

Thus, $\overline{X}_T$ is irrelevant to V_T according to Lemmas 5.4, 5.5, and 5.9. □

All of the analysis regarding the state abstraction have been based on the variables that are irrelevant to the value function and not the MAXQ completion function. In the MAXQ value decomposition, the value at a task is determined by combining the value computed at the child (recursively) with the completion value for the child. Consequently, a set of variables Y is irrelevant to all completion functions at T if Y is irrelevant to the value function at T. Because a task need only store the completion functions for its children, there are more opportunities for state abstraction. However, the following lemma shows that the savings are limited, because the completion function is simply the delayed value function — it represents the full value of the resulting state after the particular subtask is executed.

Lemma 5.10. *Let $T = (X, G, A)$ be a task such that every child can be executed after some subtask $a \in A$. A set of variables X is relevant to the completion function of T for a if X is relevant to V_T.*

Proof. For any hierarchical policy π, the completion function of T for a in state s is defined as

$$C^\pi_T(s, a) = \sum_{s', d} \Pr^\pi_T(s', d | s, a) \gamma^d V^\pi_T(s').$$

Thus, $C^\pi_T(s, a)$ depends on the variables that $V^\pi_T(s')$ depends on. If every subtask in A can be executed in state s', then $V^\pi_T(s')$ must depend on the variables that are relevant to V_T. □

For example, the completion function of a task T with primitive children will have fewer opportunities for further abstraction, because T is likely to execute every one of its children multiple times.

The analysis presented in this section does not address state abstraction arising from the funneling property of tasks, where many starting states are funneled to a small number of terminal states. Funnel abstractions permit a task to ignore variables that, while relevant inside a subtask, do not affect the termination distribution of the subtask. Nevertheless, the analysis captures some of the key properties of HI-MAT and sheds some light on its effectiveness.

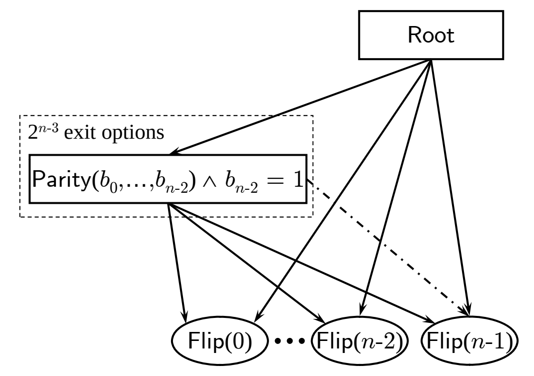
5.4 Empirical Evaluation

We test three hypotheses: ① employing a successful trajectory along with the action models will allow the HI-MAT algorithm to induce task hierarchies that are much more compact than (or at least as compact as) just using the action models; ② the hierarchies induced by HI-MAT will speed up convergence to the optimal policy in related target problems; ③ the HI-MAT hierarchies will be applicable to and speed up learning in target RL problems even when value functions from the source problems either are impossible to transfer or result in poor transfer.

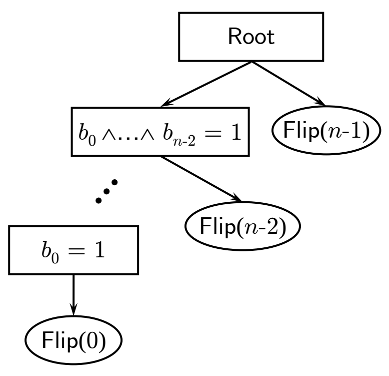
In all experiments, the DPNs are implemented procedurally through an application programmer interface (API). Given an action in a domain, the API provides the relevant variables (for annotation), the closure variables (for state abstraction), the preconditions and primary effects (for task terminations), and the entire set of actions related to a given action (for hierarchy generalization).

5.4.1 Contribution of the Trajectory

For this experiment, we compare HI-MAT's induced hierarchies to the exit-option hierarchies generated autonomously by the VISA system (Jonsson and Barto, 2006). Exit-option hierarchies are similar to MAXQ hierarchies, but every subtask has an associated exit condition (instead of a termination condition) and an exit option that is always called when the exit condition is true. VISA analyzes the influence of state variables on one another by constructing a causal graph, where the nodes are the variables; an edge,


Root
$2^{n\text{-}3}$ exit options
Parity(b_0,...,$b_{n\text{-}2}$) ∧ $b_{n\text{-}2} = 1$
Flip(0)
Flip(n-2)
Flip(n-1)

(a) VISA hierarchy. Task labels are the exit conditions; dash-dot arrows indicate exit options.



(b) HI-MAT hierarchy. Task labels are the termination conditions.

Figure 5.8: Task hierarchies for the ModBitFlip domain.

labeled by action a, exists from variable v_1 to variable v_2 in the causal graph if the DPN for a has an arc from v_1 to v_2'. Strongly-connected components in the causal graph are grouped into variable factors and there is an acyclic influence relationship between these factors. Factors correspond to exit options and the influence relationships correspond to task-supertask relationships in the exit-option hierarchy. A key difference between VISA and HI-MAT is that the latter analyzes a successful trajectory in addition to analyzing the DPNs.

To highlight our first hypothesis, a modified version of the Bitflip domain (Diuk et al., 2006) that we call ModBitflip is designed as follows. The state was represented by n bits, $b_0, b_1, \ldots, b_{n-1}$. There are n actions denoted by $\mathsf{Flip}(i)$. For $0 \le i < n-1$, $\mathsf{Flip}(i)$ toggles b_i if $b_0, \ldots, b_{i-1}$ are set; if not, it resets the bits $b_0, \ldots, b_i$. $\mathsf{Flip}(n-1)$ toggles b_{n-1} if both b_{n-2} is set and the parity across bits $b_0, \ldots, b_{n-2}$ is even (odd) when n is odd (even); if not, it resets all the bits. All bits are reset at the initial state and the goal is to set all bits.

The resulting hierarchies for this domain with $n = 7$ are shown in Figure 5.8. We observe that VISA constructs an exponentially sized hierarchy even with subtask merging activated within VISA. There are two reasons for this. First, VISA relies on the full action set to construct its causal graph, and does not take advantage of any context-specific independence among its variables that may arise when the agent acts according to certain policies. Specifically, for this domain, the causal graph constructed from

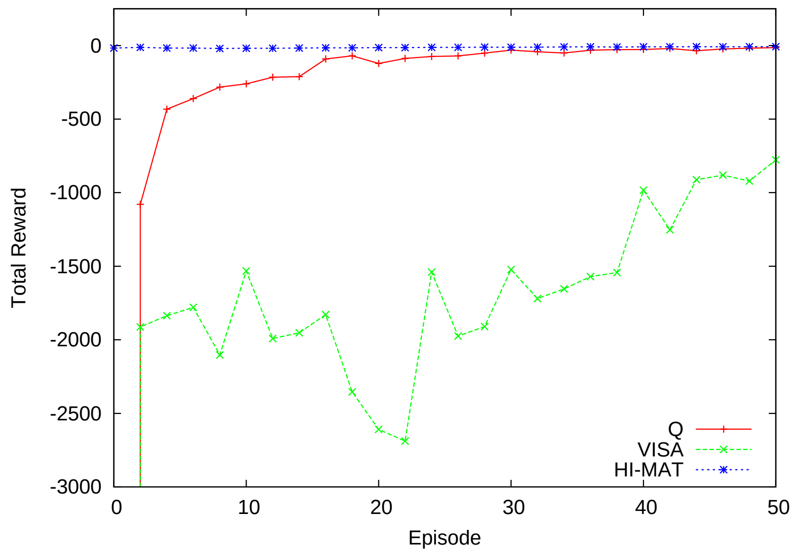


Figure 5.9: Performance of Q, VISA, and HI-MAT in the 7-bit ModBitFlip domain (averaged across 20 runs).

DPN analysis has only two strongly connected components (SCCs): one component contains $\{b_{n-1}\}$, and the other contains $\{b_0, \ldots, b_{n-2}\}$. The second SCC cannot be further decomposed using only information from the DPNs. Second, VISA creates exit options for all strongly connected components that transitively influence the reward function, whereas only a few of these may actually be necessary to solve the problem. Specifically, for this problem, VISA creates an exit condition for any instantiation that satisfies $\mathsf{Parity}(b_0, \ldots, b_{n-2}) \wedge b_{n-2} = 1$, resulting in exponential number of subtasks as shown in Figure 5.8a. The successful trajectory provided to HI-MAT achieves the goal by setting the bits going from left to right and results in the hierarchy in Figure 5.8b. Access to a successful trajectory focuses structure discovery on fruitful parts of the state and action spaces resulting in significantly more compact hierarchies than when analyzing only the global action models.

The performance results are shown in Figure 5.9. VISA's hierarchy converges even slower than the basic Q learner, because the root has $O(2^n)$ children as opposed to $O(n)$. The ModBitFlip domain has been engineered to highlight the case where access to a successful trajectory allows for significantly more compact hierarchies than without. Nonetheless, we expect that access to a solved instance will usually improve the compactness of the resulting hierarchy.

5.4.2 Transfer of the Hierarchy

We test the transfer performance of task hierarchies within the Wargus domain. We consider target problems whose specifications—number of peasants, goldmines, forests, and the size of the map—are scaled up from those of the source problems. This scaling along with the random placement of the entities in the source and target problems ensures that, although the probability parameters of the transition dynamics differ between the two, they are structurally the same. For instance, although the townhall is in a different location in the target map, depositing wood in its vicinity employs the same relevant relationships as in the source map. As coordination does not affect the policy significantly in this domain, we learn a hierarchical policy for the peasants using the MASH framework without coordination.

The following learners are contrasted: ① non-hierarchical Q-learning (Q), ② the VISA algorithm (VISA), ③ MAXQ-0 applied to a manually-designed hierarchy (Manual), and ④ MAXQ-0 applied to the HI-MAT induced hierarchy (HI-MAT). The HI-MAT algorithm first solves the source problem using flat Q-learning and generates a successful trajectory from it. Figure 5.10 shows the total duration of an episode as a function of the number of episodes experienced, averaged across 10 runs.

HI-MAT's hierarchy is faster to converge than the manually-designed one because, by analyzing the solved source problem, it is able to find a more refined task hierarchy with stricter termination conditions for each subtask than our hand-designed hierarchy. Consequently, the reduced policy space in the target problem yields a greater speedup in learning than reducing the number of value parameters via subtask sharing as in the manually-designed hierarchy. The improved rate of convergence is in spite of the fact that HI-MAT does not merge different invocations of the same explicitly parameterized subtask, so there is room for further improvement. VISA's performance suffers initially due to a large branching factor at the root option (which directly includes all

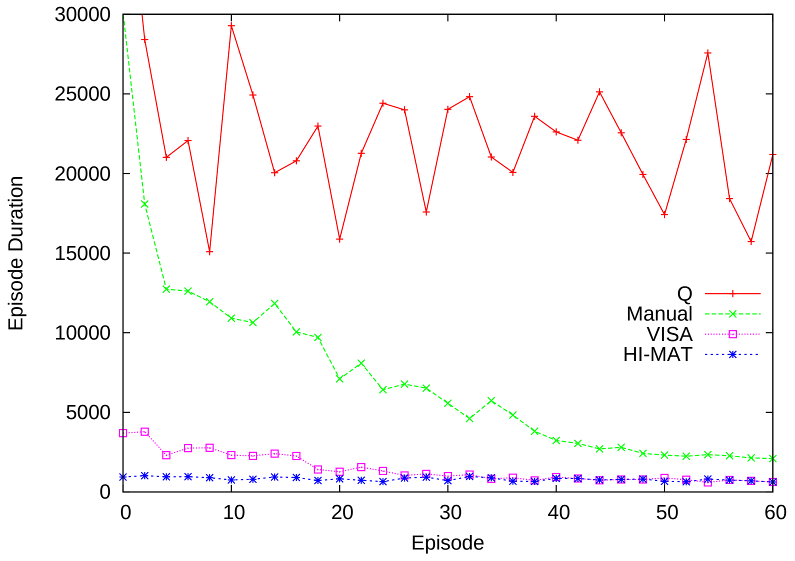
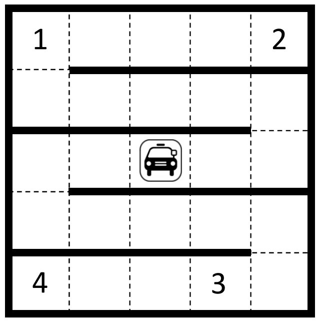


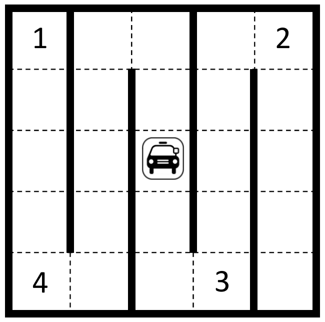
Figure 5.10: Performance in a target Wargus problem (averaged across 10 runs). Source problem: 25 × 25 grid, 1 peasant, 2 goldmines, 2 forests, 1 townhall, 100 units of gold, 100 units of wood; target problem: 50 × 50 grid, 3 peasants, 3 goldmines, 3 forests, 1 townhall, 300 units of gold, 300 units of wood. Learning algorithms: Q-learning (Q), the VISA algorithm (VISA), MAXQ-0 applied to the manually-designed hierarchy (Manual), and MAXQ-0 applied to the HI-MAT induced hierarchy (HI-MAT). The HI-MAT learning curve looks flat, because of the scale of the vertical axis required to show the learning curves of the much slower learners.

the navigation actions).

This experiment reveals the spectrum of performance based on the structure of the task hierarchy transferred across problems. The performance of Q is equivalent to that of transferring the shallowest task hierarchy (where all the primitive actions are children of the root task). Although such a hierarchy is applicable to a broader range of target problems, no knowledge is transferred from the source problem and the net effect is identical to learning in the target from scratch. The manually-designed task hierarchy is slightly more constrained than the shallowest hierarchy, and consequently Manual performs better in the target problem, but it has a more limited scope of transfer than



(a) Source problem.



(b) Target problem.

Figure 5.11: Source and target problems in the Taxi domain. In this domain, a taxi needs to get a passenger from one numbered location to another on the grid. Source and target problems differ only in the configuration of the grid walls.

Q. Although the HI-MAT hierarchy is even more constrained and consequently HI-MAT performs the best, it still has the same transferability as the manually-designed hierarchy, because the additional policy constraints are only made at the level of the relevant relationships in the transition dynamics and not at the parameter level.

5.4.3 Transfer of Structure over Value

To test whether hierarchical structure transfers better than value functions, we design source and target problems in the Taxi domain (Dietterich, 2000), where a taxi transports a passenger from a source location to a destination within a grid world. The source and target problems differ only in their wall configurations, while the passenger sources and destinations stay the same as shown in Figure 5.11. This setup is specifically engineered so that a value function from the source domain is a syntactically legal value function in the target domain (i.e., the state and action spaces are identical). However, the differing wall configurations affect the transition dynamics in the two problems. The difference between the source and target problems in the Wargus domain renders this kind of direct value-function transfer impossible.

Figure 2.1 shows a manually-designed task hierarchy for the Taxi domain. The decomposition uses the knowledge that the destination of the passenger is irrelevant when the taxi first goes to pick up the passenger, that the source is irrelevant once the

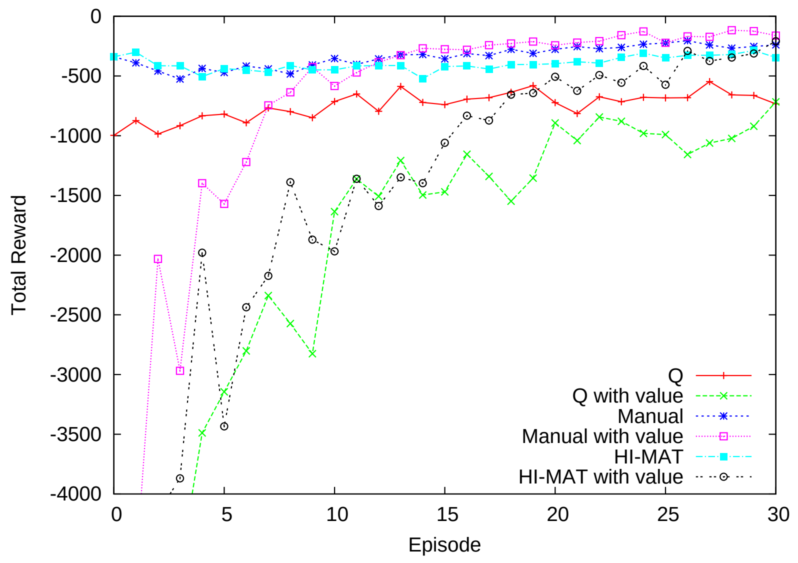


Figure 5.12: Performance in the target problem of the Taxi domain (averaged across 20 runs). Source and target problems differ only in the configuration of the grid walls. Learning algorithms: the non-hierarchical RL algorithm Q-learning (Q), the hierarchical RL algorithm MAXQ-0 applied to the manually-designed hierarchy (Manual), MAXQ-0 applied to the HI-MAT induced hierarchy (HI-MAT), and their variants in which the optimal value function in the source is also transferred along with the structure, denoted by the phrase "with value".

taxi has picked up the passenger, and that the location of the passenger is irrelevant when the taxi is navigating to a preselected location. The HI-MAT induced hierarchy is exactly the same except that it has two navigation tasks for picking up and dropping off the passenger instead of the single parameterized Goto(l) task. Both task hierarchies encode strong policy constraints, such as hard-coding the goal for navigation based on the passenger's information, and facilitate quicker convergence to the optimal policy.

Figure 5.12 shows the performance of the three learners, Q, Manual, HI-MAT, in the Taxi domain along with their variants (suffixed with the phrase "with value") in which the value functions are initialized with optimal value functions learned in the source problem. The performance of the HI-MAT induced hierarchy converges to the optimal policy at a rate comparable to that of the manually-designed hierarchy. Although the source-target problem pairs in the Taxi domain allow value-function transfer to occur, the target problems are still different enough that the agent has to "unlearn" the old policy through epsilon-greedy exploration. Transferring either the flat or the hierarchical value functions lead to worse rates of convergence to the optimal policy than transferring just the hierarchy structure with uninitialized policies (zeroed value functions) or even flat learning from scratch. For instance, transferring the navigation policies from the source problem initially causes the agent to keep running into walls in the target problem. Thus, transferring structural knowledge can be superior to transferring value functions, especially when the target problem differs significantly in terms of its optimal policy.

An explanation for the result is that an optimal policy is tuned to a particular instance of a domain. It will remain optimal in a very small set of related instances, because it depends intimately on the transition probabilities and the immediate reward values of the instances. Consequently, the corresponding optimal value function has very limited transferability. On the other hand, a task hierarchy embodies domain knowledge at a more abstract level than the value function and is more broadly applicable across instances of the domain. Given a transferred task hierarchy, the hierarchical policy can then be optimized individually for the target problems.

5.5 Related Work

This section discusses related work in the fields of autonomous hierarchical decomposition and transfer learning.

5.5.1 Autonomous Hierarchical Decomposition

Several researchers have tackled the problem of automatically inducing temporally extended actions and task hierarchies employing the various mechanisms. The SKILLS algorithm counterbalances compaction in subtask policy representation versus loss in optimality using minimum description length principles (Thrun and Schwartz, 1995). The PolicyBlocks algorithm creates subtasks by identifying commonalities in policies for related problems (Pickett and Barto, 2002). Bottleneck states or access states are states that connect two or more strongly-connected regions, and methods to discovering such subgoal states include applying the diverse density algorithm to trajectories deemed successful in a multiple-instance learning setting (McGovern and Barto, 2001), applying the relative-novelty metric that is based on visitation frequencies (Şimşek and Barto, 2004), or by taking a graph-theoretic approach of applying the max-flow/min-cut algorithm to the transition dynamics graph (Menache et al., 2001).

Similar to bottleneck states, exit states are states that satisfy the preconditions of actions. The HEXQ algorithm (Hengst, 2002) employs a heuristic based on the frequency of change in the state variables to determine exit states and partition the state space into exit-option subtasks — the most frequently-changing variable is associated with the lowest-level subtask, and the least frequently-changing variable is associated with the root. Plans that have been formulated for a domain can be partially generalized via appropriate annotation to be applicable in a related domain (Kambhampati and Hendler, 1989). Heuristics like perfect causality (Yamada and Tsuji, 1989) and static/dynamic filtering (Iba, 1989) have been applied to selectively learn new macros, because unchecked macro creation can result in performance that is worse than using no macros at all. Some hybrid methods use planning to automatically construct task hierarchies with high-level operators and then apply RL to flesh out these behaviors or to learn the choices for ambiguous plans (Ryan, 2002). VISA can also be considered a hybrid method, because it analyzes the action models to find policies that achieve preconditions for the actions (exit options) but binds them to the root option based on the (known) reward function.

5.5.2 Transfer Learning

Various elements in reinforcement learning are amenable to knowledge transfer including value functions, policies, models of system dynamics, and task hierarchies (as we have considered in this dissertation). These elements lie along the spectrum going from transferring detailed knowledge with a narrow scope to more abstract knowledge with a broader scope of transfer. Transferring detailed knowledge, when possible, leads to huge speedup, but transferring more abstract knowledge has a broader scope of transfer.

Value functions or policies transfer to a new task only when the target task is almost identical to the source task in terms of the action dynamics and rewards. For instance, ① the value function can be transferred as an auxiliary reward signal for reward shaping when part of the state space is identical (Konidaris and Barto, 2006), ② compact policy constraints (such as when the agent should pass versus shoot in soccer-like games) can be employed for advice in the target task (Torrey et al., 2007; Taylor and Stone, 2009), or ③ previously learned policies can be leveraged as macro actions when exploring in a new task (Fernández and Veloso, 2006).

Task hierarchies are transferable to target tasks that are different from the source in terms of action dynamics or rewards, as long as the qualitative relevant structure and variable dependencies of actions are preserved. Extracting relevant structure from the trajectories to discover the dependencies is common to the EBL approach adopted by other systems (Tadepalli and Dietterich, 1997; Nejati et al., 2006). However, by relying on only the qualitative structure of the domain dynamics and abstracting away the detailed quantitative model, our approach learns more widely transferable knowledge than these other approaches. Due to the generality of the transferred knowledge, the learner does require further experience in the target task to perform optimally. Besides being transferred as structural knowledge, task hierarchies also facilitate modular value-function transfer between two dissimilar RL problems that share common subtasks even though the overall flat value function may not transfer as seen in Chapter 4.

All of the transfer methods discussed above require some correspondence between the state representations of the source and target tasks. As in the MASH framework, Konidaris and Barto (2006) also transfer knowledge based on an agent-centric representation that is common across all the tasks. In our work, an agent-centric representation allows for the scaling of the domains of the state variables from the source to the target task, because the state abstractions of the subtasks only specify which variables are involved and the hierarchical value function can be relearned for different variable domains. Alternatively, Taylor and Stone (2009) provide/learn an explicit mapping between state variables of the source and target problems. In contrast, relational reinforcement learning addresses transfer via a higher-order generalization of the state space. Here, the world is represented as a first-order MDP, where the states are represented via predicates in first-order logic. The relational value function or policy generalizes to all grounded instances of the first-order MDP (Wang et al., 2008).

5.6 Conclusion

This chapter presented an approach to automatically induce task hierarchies from source problems and transfer these hierarchies to related target problems that share the causal dependencies of the system dynamics. Given action models, HI-MAT analyzes the causal and temporal relationships among the actions in a successful trajectory and partitions the trajectory recursively into a task hierarchy. We have shown that the learned hierarchies are consistent, safe, and have compact value-function tables. The empirical results indicate that leveraging the observed trajectory allows HI-MAT to learn structurally compact hierarchies. In a transfer setting, these hierarchies perform comparably to manually-designed hierarchies and provide more effective transfer than the direct transfer of the value function.

In spite of its success, HI-MAT's abilities are severely limited for the following reasons. ① It only analyzes a single successful trajectory, which it assumes is the execution trace of a hierarchical policy. In some domains, there are many good policies, only some of which will exhibit hierarchical structure (e.g., Kaelbling (1993)), so a trajectory extracted from a learned policy is unlikely to possess this structure. ② It relies heavily on expert-designed action models to construct generalized termination conditions for the subtasks. ③ It fuses the concepts of relevance and causality into a compact annotation scheme, which could lead to poor subtask discovery as shown in detail in the next chapter. Also, parsing based on the sequential ordering of the original trajectory could lead to the discovery of many fragmented subtasks. ④ It does not explicitly parameterize the subtasks, whereas hand-designed hierarchies exploit parameterization in the task hierarchy to encode non-Markovian intentions which can greatly improve the compactness of the policies. ⑤ HI-MAT has a rudimentary method of identifying similar subtasks and, along with the lack of explicit parameterization, this causes it to miss many more opportunities for subtask unification. The next chapter describes the advanced HierGen framework that addresses all of these issues.

Chapter 6: Advances in Hierarchical Structure Discovery

This chapter presents the HierGen approach to hierarchical structure discovery. It significantly advances HI-MAT that was presented in the previous chapter. The high-level schema of HierGen is outlined in Figure 6.1.

6.1 Overview

HierGen's principal advances over HI-MAT are ① learning and employing simple action models, ② a robust trajectory annotation and parsing mechanism, ③ the ability to generalize over multiple trajectories, and ④ increased subtask sharing through explicit parameterization. This section motivates and describes the first three advances.

6.1.1 Action Models

In most real-world domains, it is almost impossible to procure perfect models and we would like HierGen to utilize simple models so that they can be learned directly from the trajectories. We settle on utilizing simple decision trees (in which every internal node tests the value of a variable and every leaf contains a single value) within the DPN action models. We employ the J48 classification algorithm in Weka (v3.6.5) to learn such trees (Hall et al., 2009). To learn the tree for a variable v within the model for action a, a training instance is the factored state that a is executed in and the value of the variable v in the next state (class label); for the reward tree, a training instance is the factored state that a is executed in and the immediate reward (class label). The decision trees modeling the passenger's x-coordinate and the reward for the Dropoff action in the Taxi domain are shown in Figure 6.2.

Although these models are only capable of representing deterministic functions (the trees do not contain probability distributions at the leaves), they are sufficient for causal annotation of trajectories from stochastic domains. This is because the variable-checking information comes from the internal variables in the trees, while the variable-changing

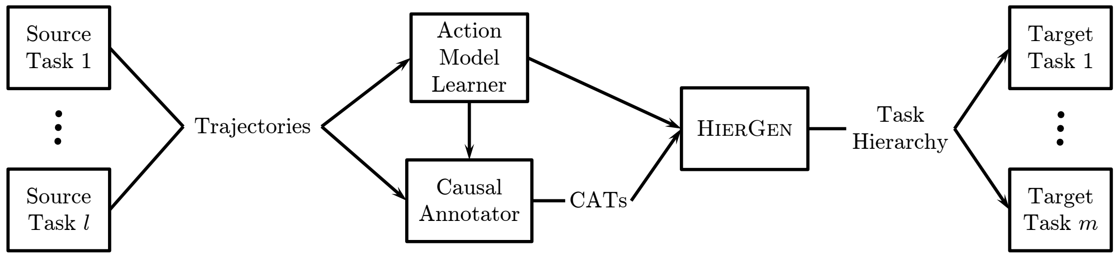
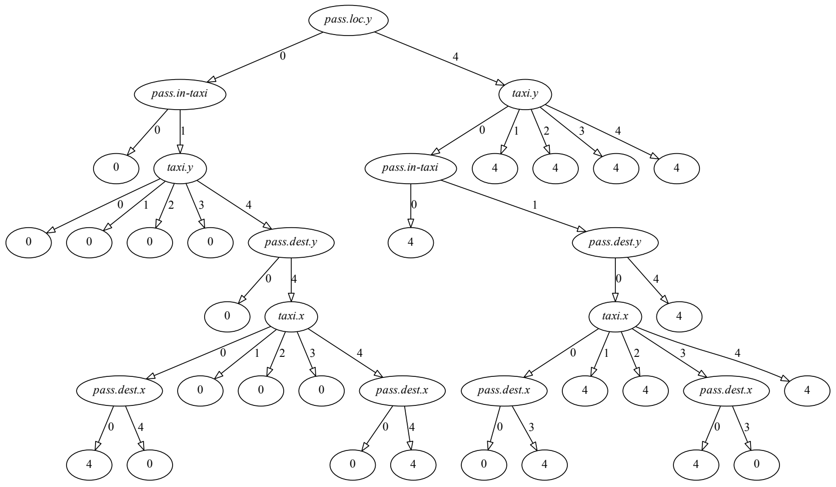


Figure 6.1: The HierGen architecture. Trajectories from multiple source tasks and the learned action models are analyzed to produce causally annotated trajectories (CATs). The CATs, along with the action models, are provided to the HierGen algorithm, which discovers hierarchical structure that can transfer across all target tasks whose actions share the same causal dependencies as those of the source tasks.

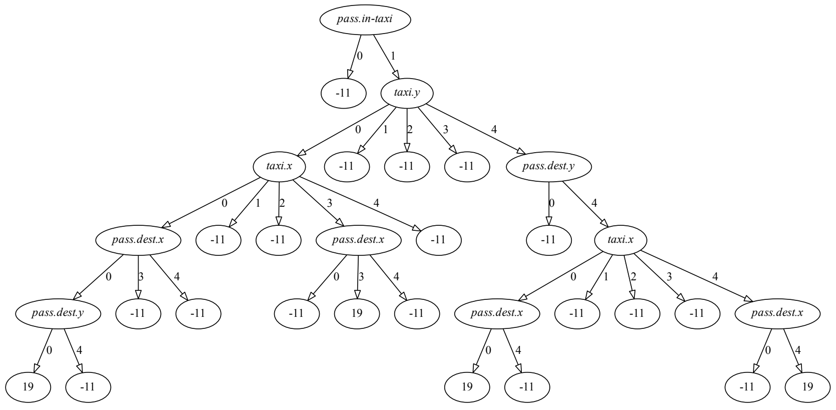
information can be gleaned directly from the trajectories. The ability to exploit simple deterministic models allows us to employ Weka even though it is unable to learn distributions at the leaves (the leaf values are simply the majority class labels). Moreover, we assume that the reward function is deterministic, that is, $\mathcal{R}(s, a, s') = \mathcal{R}(s, a)$. We considered employing the discrete mixture trees developed by Wynkoop and Dietterich (2008), but preliminary empirical tests showed that they were less stable than Weka's J48 trees, especially when learning from fewer input trajectories.

6.1.2 Trajectory Analysis

HI-MAT assumes that the trajectory is generated by a hierarchical policy, but this is a stringent requirement and structure discovery should be able to deal with non-hierarchically generated trajectories. Trajectory parsing in HI-MAT involves creating the RAT to assist in the scanning process. However, as HI-MAT builds subtasks based on contiguous action subsequences, subtask discovery is governed by the sequential ordering of actions in the original trajectory. While the successful trajectory is preprocessed (cycles are eliminated) to avoid discovering redundant structure, even optimal trajectories based on non-hierarchical policies could cause HI-MAT to discover extraneous subtasks. To illustrate, we concoct the Increment domain in which action a_1 increments x_1 if $x_3 = 1$, a_2 increments x_2 if $x_3 = 1$, and the goal is $(x_1 = 3 \land x_2 = 4)$. As only $\{x_1, x_3\}$ is relevant to a_1, and only $\{x_2, x_3\}$ is relevant to a_2, no dependencies exist

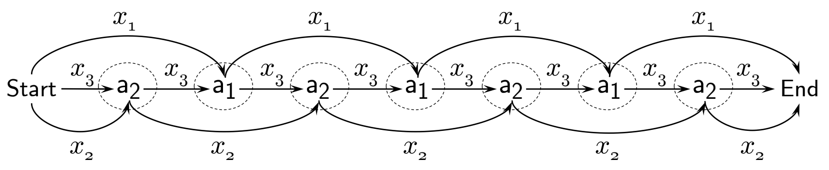


(a) Modeling *pass.loc.y*.

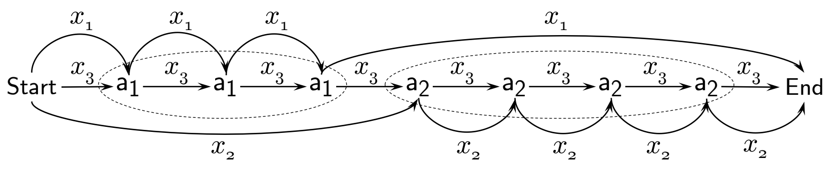


(b) Modeling the reward.

Figure 6.2: Simple decision trees for the *pass.loc.y* variable and the reward within the action model for Dropoff learned from 25 random trajectories in the Taxi domain.



(a) Fragmented juxtaposition.



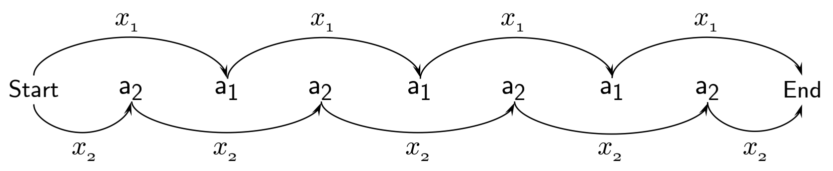
(b) Hierarchical juxtaposition.

Figure 6.3: Relevant annotation for two trajectories in the Increment domain. The dashed ellipses indicate the subtask segments discovered by HI-MAT.

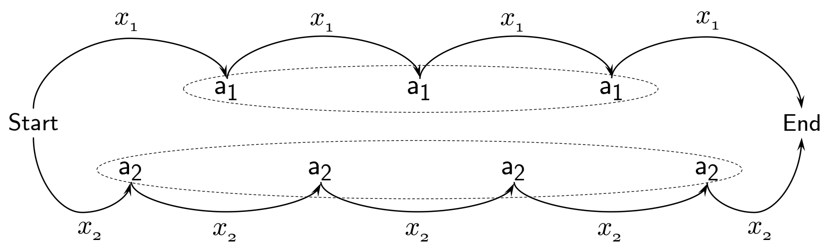
between a_1 and a_2. Consequently, these actions might be randomly juxtaposed in an optimal trajectory; two such juxtapositions are shown in Figure 6.3.

HI-MAT's parsing mechanism will discover one subtask per action in the fragmented trajectory as opposed to the two subtasks it discovers in the trajectory with hierarchical juxtaposition. Although the trajectories are certainly different, they are two possible totally-ordered *linearizations* of the underlying *causal ordering* of actions. In this view, the causal annotation process performs inverse partial-order planning, that is, it discovers the partial ordering from the linearizations.[1] This line of reasoning begs the question: can the trajectory be annotated and parsed directly based on the partial order it specifies instead of the sequential order of a particular trajectory? Relevance annotation combines the concepts of relevance and causality for succinctness. As long as all actions check and change all the variables, these two concepts overlap. However, introducing actions that only check or only change variables results in relevant edges that are not causal and vice versa, precluding the possibility of causality-based parsing. In the Increment domain, following the arc for x_3 back from the goal leads to the last a_2 even though none of the domain's actions actually affect x_3.

[1]Partial-order planning satisfies open preconditions individually by working in the space of partial-order plans. Here, a *causal link* $a \xrightarrow{p} b$ goes from action a to action b if a achieves the precondition p for b, and an *temporal-ordering link* $a \dashrightarrow b$ exists if either the effects of a clobber an effect of b that is required for another action or the effects of b clobber a precondition for a (Russell and Norvig, 2003).



(a) Fragmented linearization.



(b) Causal dependencies.

Figure 6.4: Causal annotation of the fragmented trajectory from the Increment domain. To reduce clutter, causal arcs for x_3 that go from Start to every action in the CATs have been suppressed.

6.1.2.1 Causal Annotation

The causal annotation for a trajectory is defined as follows.

Definition 6.1. A *causal* arrow $a \xrightarrow{x} b$ connects action a to an action b (that comes after a in the trajectory) iff a changes x, b checks x, and x is not changed by any action in between. A *causally annotated trajectory* (CAT) is the original trajectory annotated with all the causal arcs, sandwiched between dummy Start and End actions for which all variables are defined to be changed and checked respectively.

The trajectory is preprocessed to remove any cycles; this includes unsuccessful actions that do not alter the state. The CAT is postprocessed to remove actions that do not have outgoing arcs, because these actions have no effect within the CAT. This is only done for convenience — we will see that the parsing mechanism ignores such actions even if they are left in the CAT.

The CAT for the fragmented trajectory from the Increment domain is shown in Figure 6.4a. In contrast to the RATs in Figure 6.3, there are no arcs going between a_1 and a_2, because the causal arcs for x_3 (suppressed for clarity) all emanate from Start. Tracing x_3 back from End leads appropriately to Start and not to an action that does not affect x_3 as in the RAT. Figure 6.4b shows that the underlying dependencies in the CAT define the same two subtasks that are extracted from the RAT shown in Figure 6.3b. In fact, any CAT for this domain has this property.

6.1.2.2 Parsing Mechanism

The overall objective of the HierGen algorithm is to transform a set of CATs into a task graph. This is done by way of the *precedence graph* (PG), in which every node corresponds to a set of valid sub-CATs (at most one from every CAT).

Definition 6.2. A sub-CAT is *valid* if ① there is at most one outgoing arc for every variable, ② there could be multiple incoming arcs for a variable, but they should all emanate from the same action in the CAT, and ③ every member action, besides the ones from which the outgoing arcs emanate, must have at least one outgoing arc and every outgoing arc should be to another member action.

The PG has a unique leaf node and it corresponds to the sub-CATs (one from every CAT) with only the End action. An edge from node p to q in the PG indicates that p satisfies a precondition of q and corresponds to one or more causal arcs in the CATs. The restriction on the incoming arcs of a valid sub-CAT ensures that every variable's precondition is achieved by a unique node in the PG. Because a CAT is a directed acyclic graph, so is the PG.

The PG can be transformed into a (directed acyclic) task graph by converting every node to a task and flipping the causal arcs, that is, if node p is a parent of q in the PG, then task p is the child of task q in the task graph. Thus, the unique leaf node of the PG transforms to the root of the task graph. Besides this root task, any task corresponding to a node p in the PG also has children corresponding to the hierarchical decomposition of the sub-CATs at p. This scheme hardcodes the precedence of the subtrajectories into the hierarchical structure as opposed to HI-MAT's alternative — making the two subtasks siblings — which requires that the parent task learn the correct precedence of the subtasks. Having at most one outgoing arc per variable in a valid sub-CAT ensures that a valid termination condition can be constructed for every task.

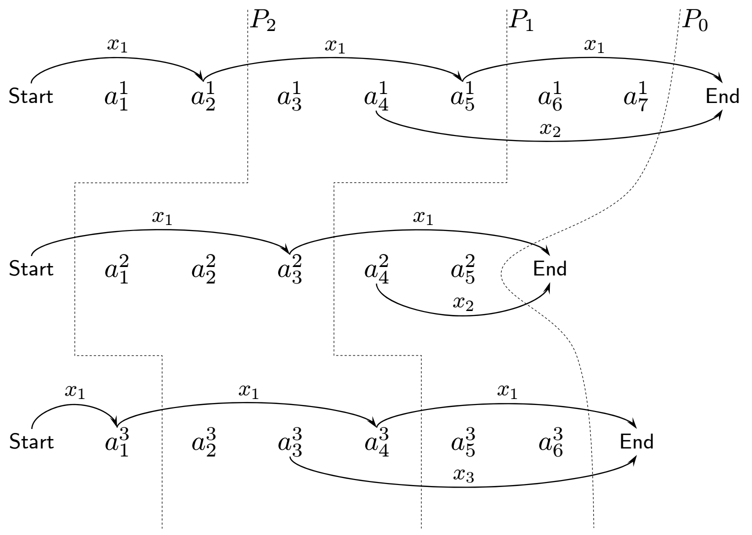
HI-MAT's parsing mechanism keeps absorbing temporally continuous actions in the RAT as long as the set of checked variables does not increase; this is equivalent to absorbing contiguous actions as long no relevant arc affects something beyond the ultimate action. Transferring this mechanism over to CATs would preclude the benefits of causal annotation and only parse the particular linearization of the CAT. To extract the underlying partial order, the new parsing mechanism works in the causal space by following causal arcs to assimilate the sourcing actions into subtrajectories. In Figure 6.4a, this entails independently following the x_i arcs to collect the same sequence of a_i as shown in Figure 6.3b.

Even in the causal space, the extraction of a sub-CAT p stops when an action that has an outgoing arc to another action outside p is encountered. Obviously, this action achieves a precondition that is checked by at least two sub-CATs. If this action is incorporated into p, then the termination condition of the corresponding task must include the condition on the outgoing arc to ensure that it is achieved by the task. However, this results in a larger or more specific termination condition. Instead, biasing the termination condition to have fewer variables (making it more general) facilitates better subtask reuse and leads to more compact state abstraction and more shielding for the subtasks.

6.1.3 Multi-CAT Analysis

Multiple trajectories provide a more complete picture of the system dynamics and the useful state and policy spaces than just one successful trajectory. Given multiple trajectories, we can first learn the simple action models from the (s, a, r, s') tuples in the trajectories. As these simple models do not provide the general conditions for termination, we instead parse the trajectories in parallel and generalize the conditions using the goal language. In this work, the termination language is the conjunction of literals, where every literal is either a variable equal to a constant or a variable equal to another variable. The trajectories also give us only a sample of the primitive actions required to achieve the goal condition from arbitrary initial states. Unlike HI-MAT's hierarchical structure which has to be explicitly generalized by tacking on the actions unobserved in the single provided trajectory to the final hierarchy, the structure here need only consist of the observed actions.

For illustrative purposes, we concoct a domain where $\mathcal{X} = \{x_1, x_2, x_3\}$, and the goal condition for the task is $x_1 = \mathsf{x}_1 \wedge (x_2 = \mathsf{x}_2 \vee x_3 = \mathsf{x}_3)$. Figure 6.5 shows three hypothetical



P_2
P_1
P_0
x_1
x_2
x_3
Start
End
a_1^1 a_2^1 a_3^1 a_4^1 a_5^1 a_6^1 a_7^1
a_1^2 a_2^2 a_3^2 a_4^2 a_5^2
a_1^3 a_2^3 a_3^3 a_4^3 a_5^3 a_6^3

Figure 6.5: Simultaneous parsing of multiple CATs. a_i^j represents the ith action (more generally, subtrajectory) in the jth trajectory. For clarity, the trajectories are only partially annotated.

(partially) annotated trajectories from this domain, where a_i^j represents the ith action (more generally, the ith subtrajectory) in the jth trajectory. The goal condition can be inferred by generalizing the state information along the cut P_0 according to the termination language. Next, if the variable x_1 is picked, then tracing the arcs labeled x_1 in reverse from End leads to the actions a_5^1, a_3^2, a_4^3; in turn, following the arcs labeled x_1 in reverse from a_5^1, a_3^2, a_4^3 leads to $a_2^1, \mathsf{Start}, a_1^3$. Just as is done for the goal condition, it is possible to discover generalized conditions for x_1 along both P_1 and P_2. The states collected from the trajectories for generalization are those that occur immediately after the actions at the tails of the cut causal arcs.

The cuts can also be utilized to determine admissibility conditions for the tasks by generalizing across the states that immediately precede the actions at the head of the cut causal arcs. An admissibility condition C can be incorporated into a task $T = (X, G, C)$ by making the new termination condition $G' = \overline{C} \vee G$. However, this could possibly increase the state abstraction of the task and also precludes opportunities for sharing among subtasks that have the same termination conditions but differing admissibility conditions. Now, a primitive task does not have an associated termination condition in the hierarchy and can always be executed by its parent in any state. To shield it from being executed unsuccessfully, it can be wrapped in a composite task whose termination condition is the negation of the admissibility condition for the primitive action. Preliminary experiments demonstrate that these shielding tasks can enhance the performance of the hierarchy. Obviously, the difference in performance of the hierarchies with and without shielding tasks increases with higher costs for executing actions illegally.

Besides providing the generalized conditions, the technique of simultaneously parsing also guides the generalization of the hierarchy being constructed. If at any stage, the preconditions cannot be unified across the trajectories, then this behooves the creation of variable factors that consist of sets of variables. In the limit, all the variables are in one factor and no decomposition is possible. This concept will be made more concrete in the next section that discusses the details of the HierGen approach.

6.2 The HierGen Approach

We assume that we are only given a set of random trajectories that all achieve the goal. With this, the objective is to automatically induce a task hierarchy that can suitably constrain the policy space when solving a target problem whose actions share the same causal dependencies as those in the source problems. This is achieved via the simultaneous parsing of the CATs into tasks using a top-down recursive procedure, guided by backward chaining from the goal in the causal space at every level. HierGen parses the CATs to determine the termination conditions and the set of child tasks, and employs the action models to determine the state abstraction for the hierarchy. We use the Taxi domain to explicate the details of the HierGen approach. A sample CAT for the Taxi domain is shown in Figure 6.6.

The hierarchy discovery algorithm is outlined in the mutually-recursive procedures Algorithms 6.1 and 6.2. HierGen first determines a goal condition G, based on the termination language (a conjunction of literals that are either "$v_i = v_j$" or "$v_i = \mathsf{v}$", where $v_i, v_j \in \mathcal{X}$ and $\mathsf{v} \in \mathcal{D}(v_i)$), that generalizes across the final state of every CAT in Ω. Next, if there is at least one CAT in Ω with multiple actions, then HierGen passes Ω and G to HierBuilder, which returns a set of tasks. If the set is empty, HierGen splits up Ω into two sets of sub-CATs: one with the sub-CATs (based on ψ) that contain the ultimate actions in the CATs and the other with the sub-CATs (based

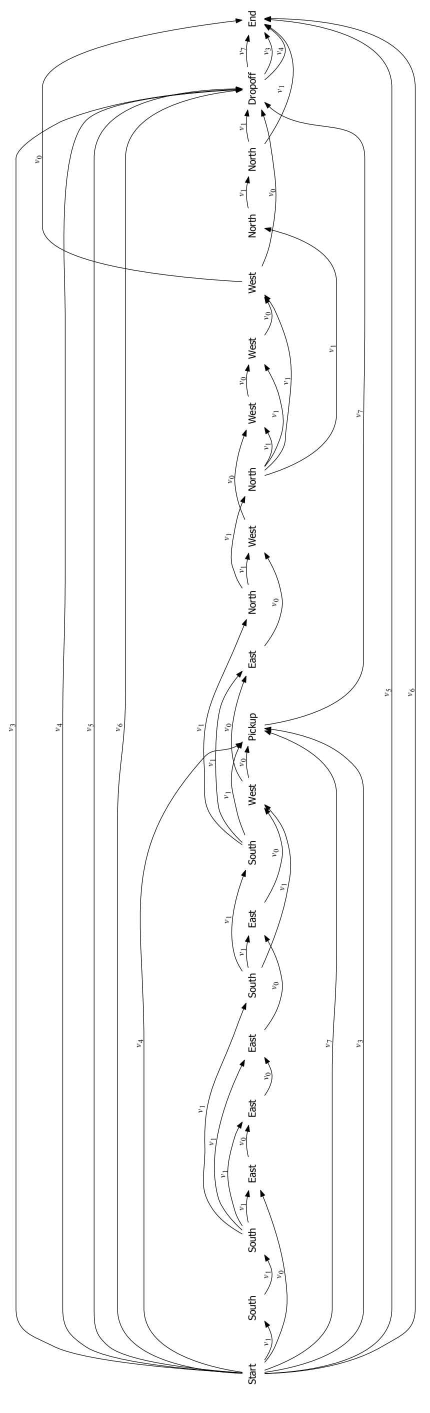


Figure 6.6: A CAT for the Taxi domain. Variables: $v_0 = taxi.x$, $v_1 = taxi.y$, $v_3 = pass.loc.x$, $v_4 = pass.loc.y$, $v_5 = pass.dest.x$, $v_6 = pass.dest.y$, $v_7 = pass.in\text{-}taxi$. ($v_2 = taxi.fuel$ is not used.)

Algorithm 6.1 HierGen

```
Input: Set of action models $\{\mu_a\}$, set of CATs $\Omega$.
Output: Task $(X, G, C)$.
1: $G \leftarrow$ GoalCondition($\Omega$)
2: if MultipleActions($\Omega$) then
3:     $C \leftarrow$ HierBuilder($\{\mu_a\}, \Omega$)
4:     if $C \neq \varnothing$ then
5:         $A \leftarrow$ set of primitive descendants in $C$
6:         $X \leftarrow \bigcup_{c \in C} X_c \cup$ RelevantVariables($\mu_A, G$)  // $X_c$ = abstraction of task $c$
7:         return $(X, G, C)$
8:     $\psi \leftarrow$ ExtractUltimateActions($\Omega$)  // $\psi$ = set of action indices
9:     $Q \leftarrow$ HierBuilder($\{\mu_a\}$, Extract($\Omega, \overline{\psi}$))  // $\overline{\psi}$ = everything but $\psi$
10:    if $Q \neq \varnothing$ then
11:        $(X, G, C) \leftarrow$ HierGen($\{\mu_a\}$, Extract($\Omega, \psi$))
12:        $C \leftarrow C \cup Q$  // Add the tasks in $Q$ to the set of children $C$
13:        $A \leftarrow$ set of primitive descendants in $C$
14:        $X \leftarrow X \cup$ RelevantVariables($\mu_A, G$) $\cup \bigcup_{q \in Q} X_q$
15:        return $(X, G, C)$
16: $C \leftarrow$ set of actions in $\Omega$
17: $X \leftarrow$ RelevantVariables($\mu_C, G$)
18: return $(X, G, C)$
```

Algorithm 6.2 HierBuilder

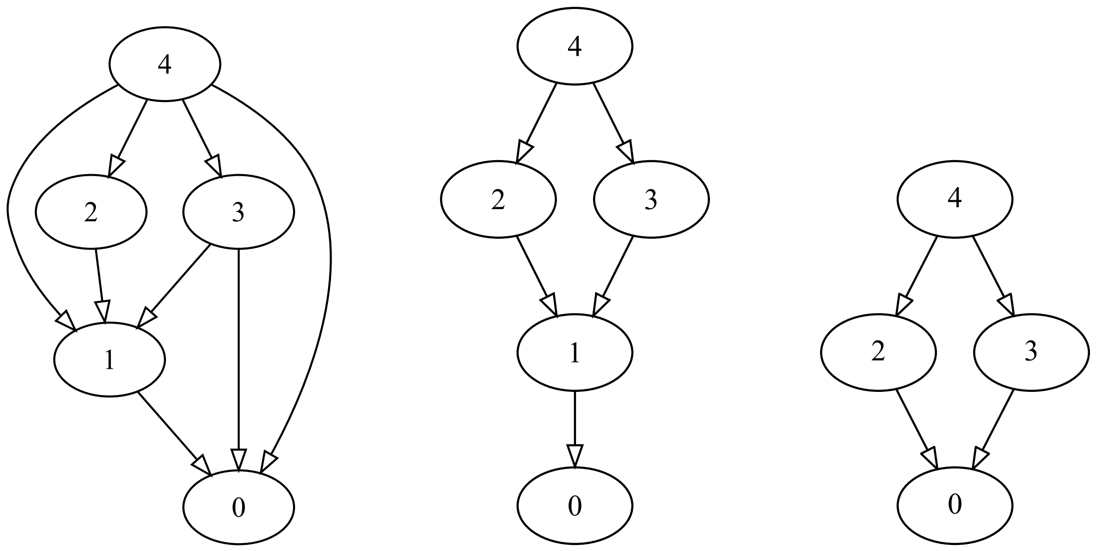
```
Input: Set of action models $\{\mu_a\}$, set of CATs $\Omega$.
Output: Set of tasks $\{(X, G, C)\}$.
 1: $G \leftarrow$ GoalCondition($\Omega$)
 2: if $G =$ false then  // No goal condition generalizes the final states in $\Omega$
 3:   return $\varnothing$
 4: $\Psi \leftarrow \varnothing$  // Set of sets of sub-CATs
 5: for $v \in$ Variables($G$) do
 6:   $\psi \leftarrow$ CAT-Scan($\Omega, \{v\}$)
 7:   $\Psi \leftarrow \Psi \cup \{\psi\}$
 8: $H \leftarrow \varnothing$
 9: $\widehat{\Psi} \leftarrow$ Unify($\Psi$)  // Unify the partition of goal variables across trajectories
10: if $\widehat{\Psi} \neq \varnothing$ then
11:   $\Psi \leftarrow \widehat{\Psi}$
12:   for $\psi \in \widehat{\Psi}$ do
13:     $Q \leftarrow$ HierBuilder($\{\mu_a\}$, Extract($\Omega, \overline{\psi}$))  // $\overline{\psi}$ = everything preceding $\psi$
14:     if $Q \neq \varnothing$ then
15:       $(X, G, C) \leftarrow$ HierGen($\{\mu_a\}$, Extract($\Omega, \psi$))
16:       $C \leftarrow C \cup Q$
17:       $A \leftarrow$ set of primitive actions in $C$
18:       $X \leftarrow X \cup$ RelevantVariables($\mu_A, G$) $\cup \bigcup_{q \in Q} X_q$
19:       $H \leftarrow H \cup \{(X, G, C)\}$
20:       $\Psi \leftarrow \Psi - \{\psi\}$
21: if $\Psi \neq \varnothing$ then
22:   $\psi \leftarrow$ Merge($\Psi$)  // Merge unsuccessful sets of sub-CATs into one set
23:   if $\psi = \varnothing$ then
24:     return $\varnothing$
25:   else
26:     $Q \leftarrow$ HierBuilder($\{\mu_a\}$, Extract($\Omega, \overline{\psi}$))
27:     if $Q = \varnothing$ then
28:       return $\varnothing$
29:     else
30:       $(X, G, C) \leftarrow$ HierGen($\{\mu_a\}$, Extract($\Omega, \psi$))
31:       $C \leftarrow C \cup Q$
32:       $A \leftarrow$ set of primitive actions in $C$
33:       $X \leftarrow X \cup$ RelevantVariables($\mu_A, G$) $\cup \bigcup_{q \in Q} X_q$
34:       $H \leftarrow H \cup \{(X, G, C)\}$
35: return $H$
```

on $\overline{\psi}$) that contain the remaining actions. HierGen then calls HierBuilder with the second set of sub-CATs. If a set of tasks Q is returned, then HierGen returns a task corresponding to the sub-CATs based on ψ that also includes the tasks in Q as children. If no decomposition is possible, then HierGen simply returns a task with the set of primitive actions in Ω as children.

HierBuilder first checks if any goal condition G can generalize across the final states of Ω. If not, it returns immediately; otherwise, CAT-Scan extracts sub-CATs from Ω that affect the goal variables. As each sub-CAT can affect more than a single variable, Unify merges the sub-CATs so that the affected variables are partitioned within Ψ. For example, let the variables in G be $\{v_1, v_2, v_3\}$. One CAT in Ω might one sub-CAT affect $\{v_1, v_2\}$ and the other affect $\{v_3\}$; another CAT might have one sub-CAT affect $\{v_1\}$ and the other affect $\{v_2, v_3\}$. In order to have a global partition of the state variables that satisfies all the CATs, the two sub-CATs in each CAT have to be merged to produce a single sub-CAT that affects $\{v_1, v_2, v_3\}$. If such a partitioning is possible, then HierBuilder calls itself recursively on the CATs extracted according to $\overline{\psi}$. If it returns with a set of tasks Q, then Q is added to the children of the task corresponding to ψ. If Q is empty, then ψ is not removed from Ψ. Merge tries to coalesce the elements remaining in Ψ ($|\Psi| > 1$) into one by extracting sub-CATs that affect the variables in Ψ (this is equivalent to building factors of variables); Merge calls CAT-Scan internally to accomplish this. If the merging is successful, then HierBuilder is called on the sub-CATs that precede the merged sub-CATs. If either Merge or HierBuilder return empty, then this emptiness is propagated to the calling procedure. Otherwise, the task that corresponds to the merged sub-CATs is augmented with children that correspond to the preceding tasks.

The pseudo-code for Algorithms 6.1 and 6.2 has been kept simple for explanatory purposes. Instead of recursing, HierBuilder actually constructs a matrix of sub-CATs — each row corresponds to a CAT and each column corresponds to a node in the PG. Processing the matrix results in more efficient space and computational complexity than recursion, because multiple causal arcs can emanate from an action and the matrix prevents duplication of the effort of extracting a sub-CAT for an action seed that has already been processed (memoization).

While the generation of the PG is implicit in the pseudo-code, a detail completely omitted from the pseudo-code is the manipulation of the PG before the hierarchical



(a) Initial PG. Node 4 is connected to nodes 0 and 1, because *pass.loc.y* might be equal to *pass.dest.y* in some trajectories.

(b) PG after transitive reduction.

(c) Final PG after combining node 0 with its single parent.

Figure 6.7: Processing the precedence graph produced at HierGen's first level of recursion based on 25 random trajectories from the Taxi domain. Every node in this graph corresponds to a composite task in the final hierarchy: 0 = Root, 1 = dropping off the passenger, 2 = picking up the passenger, 3 = navigating to the passenger's destination, 4 = navigating to the passenger's source.

structure is finalized. Recall that the PG is the result of the simultaneous graph transformation of all the CATs by HierGen. The initial PG for the Taxi domain is shown in Figure 6.7a.

Two refinement operations can be performed on the PG to produce a more compact task graph after transformation:

- The PG is transitively reduced to get rid of extraneous linkages between tasks. If node *a* precedes both *b* and *c*, and *b* precedes *c*, then *c* will be a parent of *b* and *b* will be a parent of *a* in the task graph. The trajectories demonstrate that the effects of *b* do not clobber the effects of *a* that are checked by *c*; otherwise, the link would not exist from *a* and *c*. Thus, a direct link from *a* to *c* is redundant in

Algorithm 6.3 CAT-Scan

```
Input: Set of CATs $\Omega$, set of variables $V$.
Output: A set $\psi$ of sets of action indices in $\Omega$
 1: for $\omega \in \Omega$ do
 2:   for $v \in V$ do
 3:     $\psi_\omega \leftarrow \{i : (a_i \xrightarrow{v} \mathsf{End}) \in \omega\}$  // Seed action indices
 4:   repeat
 5:     $I \leftarrow \big\{i : \big(\exists j \in \psi_\omega,\ \exists u,\ (a_i \xrightarrow{u} a_j) \in \omega\big) \wedge \big(\forall k \notin \psi_\omega,\ \forall u,\ (a_i \xrightarrow{u} a_k) \notin \omega\big)\big\}$
 6:     $\psi_\omega \leftarrow \psi_\omega \cup I$
 7:   until no more action indices can be added to $\psi_\omega$
 8:   $\psi_\omega \leftarrow$ UniquePrecondition($\psi_\omega$)
 9:   if $V \not\subseteq \psi_\omega$ then
10:     return $\varnothing$
11: return $\psi$
```

the PG. As the original PG is a directed acyclic graph, the transitive reduction is known to be unique. The transitively reduced PG is shown in Figure 6.7b.

- If the unique leaf node r of the PG (corresponding to the root task) has a single parent p, then p and r are merged. This operation prevents the root task from having only a single child in the task graph. The final PG after this operation and transitive reduction is shown in Figure 6.7c. Besides r, a node p with a single parent in the PG will not convert to a task with a single child, because the task will have at least one other child corresponding to the hierarchical decomposition of the sub-CATs at p.

6.2.1 Sub-CAT Extraction

The procedure to extract sub-CATs is shown in Algorithm 6.3. For a particular CAT, a sub-CAT is determined by searching for causally linked actions, starting with the actions that affect the given set of variables. An action is incorporated into the sub-CAT only if all outgoing arcs land within the sub-CAT. Once no more actions can be incorporated, the remaining incoming arcs correspond to preconditions of the sub-CAT. As the preconditions of the sub-CAT must be unique, UniquePrecondition enforces that all the incoming arcs labeled with a particular variable v come from the same causal action in the CAT; if this condition is violated, then the sub-CAT is truncated at the action with the highest temporal index that has an incoming arc for v. It is possible that no valid sub-CAT exists for a given set of variables and a set of CATs.

We explain this procedure by the example of tracing the variable v_7 back from End in Figure 6.6. This leads to the seed action Dropoff, from which v_0 leads to West. However, as West has an outgoing arc for v_0 to End, it is not incorporated into the current sub-CAT. From Dropoff, v_1 is traced back to North, which also has an outgoing arc to End and cannot be incorporated. v_3, v_4, v_5, and v_6 all lead to Start which can be ignored. v_7 traces back to Pickup, which is added to the current sub-CAT. No further actions can be incorporated. Now, both Pickup and Dropoff have an incoming arc for v_0 that does not emanate from the same action in the CAT. Thus, Pickup with the lower temporal index is pruned and the sub-CAT returned only has Dropoff. When the set $\{v_0, v_1\}$ is traced back from Dropoff by Merge, then both the final North and West are seed actions and the extracted sub-CAT is the entire navigation segment between Pickup and Dropoff.

6.2.2 State Abstraction

Just as in HI-MAT, HierGen computes the state abstraction for a primitive task a as $\chi^1_{\mu_a}(R)$ if the reward is not stochastic and it is computed as $\chi^2_{\mu_a}(R)$ otherwise. For a composite task, HierGen first constructs the merged DPN μ_A for the set of actions A in the segment associated with the task. The state abstraction for a task $T = (X_T, G, C)$ is $X_T = \chi^*_{\mu_A}(R) \cup \chi^*_{\mu_A}(G)$ if the task has only primitive actions and $X_T = \chi^*_{\mu_A}(R) \cup \chi^*_{\mu_A}(G) \cup \bigcup_{c \in C} X_c$ otherwise.

6.2.3 Task Parameterization

All the variables in the termination condition are relevant to the value function. HI-MAT includes these termination variables in the task's state abstraction. The variables that are not modified within the task become implicit parameters to the task. HierGen converts these variables to formal parameters and passes the parameter binding to the parent. This explicit task parameterization facilitates more subtask sharing across the hierarchy. For example, consider tasks T_1 with termination $v_1 = v_2$ and T_2 with termination $v_1 = v_3$; both T_1 and T_2 only affect v_1. With explicit parameterization, T_1 and T_2 can be coalesced, because their termination conditions can be unified by the parameterized form $v_1 = x_1$, and x_1/v_2 for the parents of T_1 and x_1/v_3 for those of T_2.

6.2.4 Task Coalescence

When the task hierarchy is being constructed, HierGen merges tasks that have the same termination condition (exact for the state variables, but parameters might be permuted) by coalescing their children. With the exception of parent-child tasks, tasks along a path from the root to a leaf are not merged to avoid loops (recursive relationships) in the hierarchy.

6.3 Theoretical Analysis

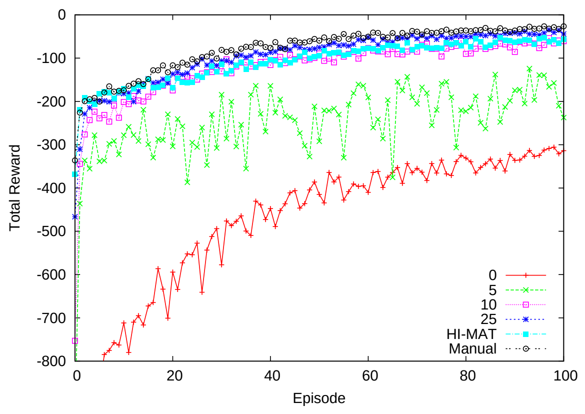
We prove that HierGen produces hierarchies that are safe with respect to the employed action models.

Theorem 6.1. *If the procedure HierGen produces a task hierarchy H from a set of trajectories and $\mu_{\mathcal{A}}$, then H is safe with respect to $\mu_{\mathcal{A}}$.*

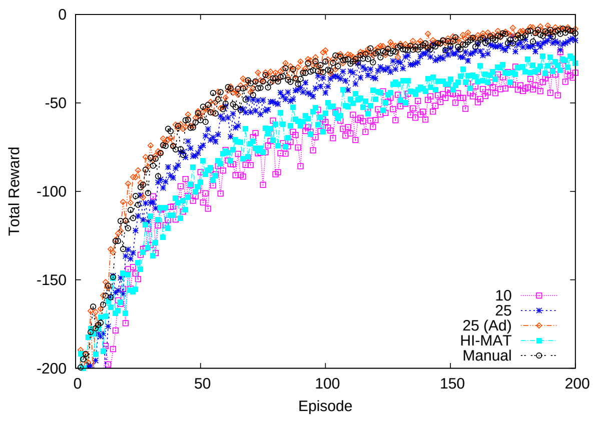
Proof. As HierGen uses the same technique as HI-MAT to compute the state abstraction for every task in H, the proof of Theorem 5.2 applies here as well. □

6.4 Empirical Evaluation

The experimental setup is to provide HierGen with randomly generated trajectories from a number of source tasks and then test the performance of the discovered task hierarchy in a target task. Every trial is a succession of episodes; every episode is a target task. Hierarchies with degenerate or overly-specific termination conditions could fail to complete an episode, because either a task with an unreachable condition spins endlessly or all subtasks are inadmissible (terminated). As soon as a threshold number of time steps is reached for an episode, the entire trial is deemed a failure. We evaluate HierGen on the same domain families as we did HI-MAT: Taxi, ModBitflip, and Wargus.



(a) Learning curves. The numbers in the key correspond to the number of random input trajectories. The curve for `0` corresponds to the performance of the shallowest hierarchy, which is equivalent to Q-learning.



(b) Learning curves: zoomed in. The "`(Ad)`" suffix indicates that the hierarchy has admissibility-based shielding tasks.

Number of trajectories	1	5	10	25	25 (Ad)
Successful trials	0%	52%	89%	100%	100%

(c) Success rate.

Figure 6.8: Performance of HierGen in the Taxi domain (across 100 trials).

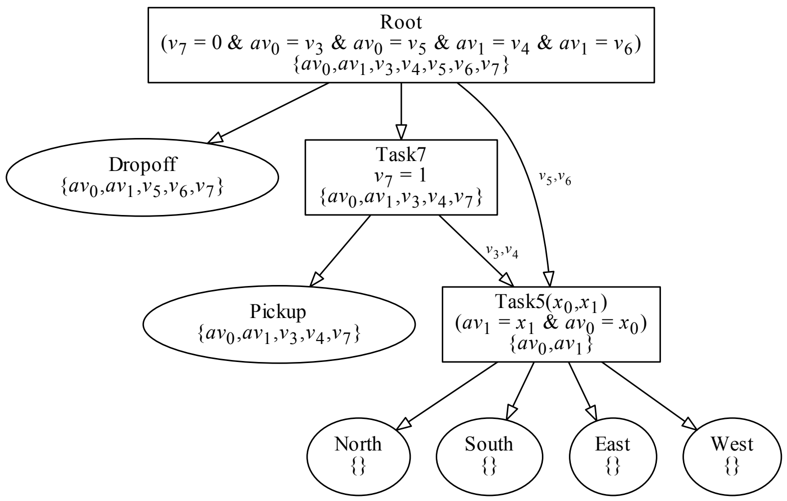

Root
(v_7 = 0 & av_0 = v_3 & av_0 = v_5 & av_1 = v_4 & av_1 = v_6)
{av_0,av_1,v_3,v_4,v_5,v_6,v_7}
Dropoff
{av_0,av_1,v_5,v_6,v_7}
Task7
v_7 = 1
{av_0,av_1,v_3,v_4,v_7}
v_5,v_6
v_3,v_4
Pickup
{av_0,av_1,v_3,v_4,v_7}
Task5(x_0,x_1)
(av_1 = x_1 & av_0 = x_0)
{av_0,av_1}
North
{}
South
{}
East
{}
West
{}

Figure 6.9: HierGen's hierarchy based on 25 trajectories from the Taxi domain. Every composite task is rectangular and is labeled with the task name (uniquely numbered), the termination condition, and the state abstraction; every primitive task is elliptical and is labeled with the name of the corresponding primitive action and the state abstraction. av_i denotes an agent-centric variable for MASHing.

6.4.1 Taxi Domains

In the Taxi domain, the source tasks are random configurations of the taxi and the passenger's location/destination, and every episode in a trial is also a random configuration of the domain. The performance of HierGen's hierarchies contrasted against that of the HI-MAT and the manually-designed hierarchies is shown in Figure 6.8. Given no input trajectories, HierGen outputs the shallowest hierarchy — the root task with only the primitive tasks as children — and the performance of this hierarchy is equivalent to that of Q-learning. Obviously, this hierarchy has no policy constraints and can solve all of the target configurations. When providing HierGen with 25 trajectories, the discovered hierarchy (Figure 6.9) also achieves a 100% success rate, but with a performance that is only slightly worse than that of the manually-designed hierarchy. The principal reason that this hierarchy performs better than that of HI-MAT is explicit task parameterization —

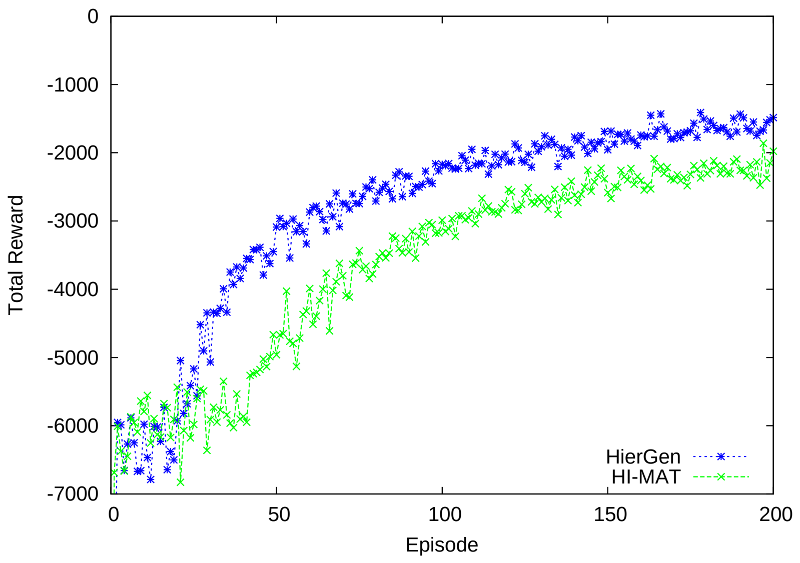


Figure 6.10: Performance of HI-MAT and HierGen (based on 25 random trajectories) in the scaled-up 25 $\times$ 25 Taxi domain (averaged across 100 trials).

HI-MAT's hierarchy cannot coalesce the navigation tasks and has to consequently learn many more parameters. Figure 6.10 shows that this issue is exaggerated when the target task is scaled up to the 25 $\times$ 25 version of the domain. As expected, the success rate of the discovered structure drops off as fewer trajectories are provided to HierGen. The hierarchy discovered from a single random trajectory (Figure 6.11) is consistent with just that trajectory and fails in some episode in every trial.

Figure 6.12 shows HierGen's hierarchy with the admissibility-based shielding tasks that prevent their primitive children from being executed unsuccessfully. For instance, Task1 prevents Dropoff from being executed if either the taxi is not at the destination or the passenger has not been picked up. Figure 6.8b shows that the performance of this hierarchy in the Taxi domain is slightly better than that of HierGen's regular hierarchy (Figure 6.9) and comparable to that of the manually-designed hierarchy.

We initially represented the fact of the passenger being in the taxi by setting the

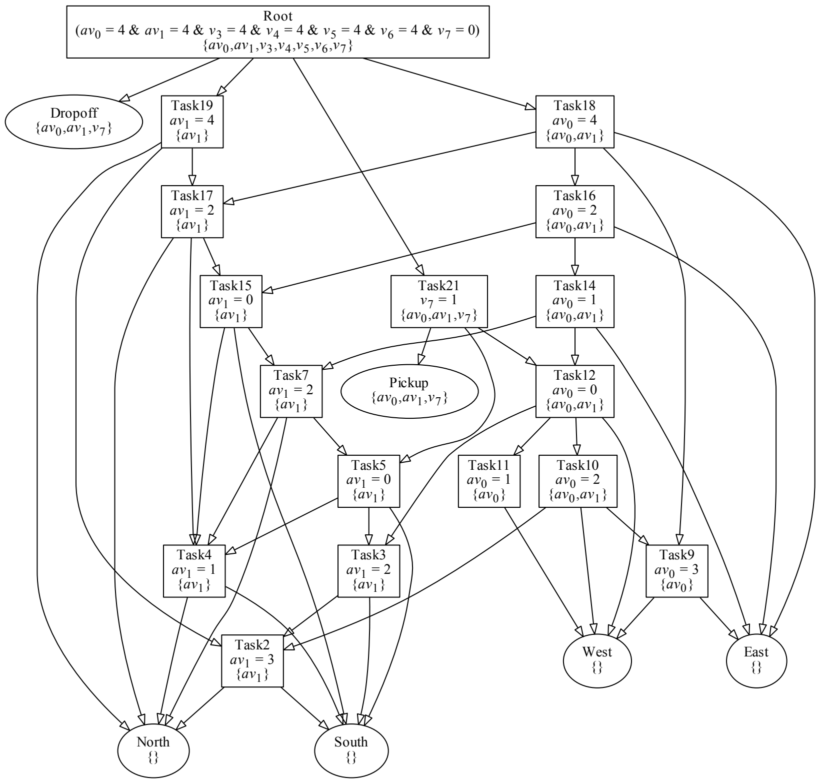

Root
(av_0 = 4 & av_1 = 4 & v_3 = 4 & v_4 = 4 & v_5 = 4 & v_6 = 4 & v_7 = 0)
{av_0,av_1,v_3,v_4,v_5,v_6,v_7}
Dropoff {av_0,av_1,v_7}
Task19 av_1 = 4 {av_1}
Task18 av_0 = 4 {av_0,av_1}
Task17 av_1 = 2 {av_1}
Task16 av_0 = 2 {av_0,av_1}
Task15 av_1 = 0 {av_1}
Task21 v_7 = 1 {av_0,av_1,v_7}
Task14 av_0 = 1 {av_0,av_1}
Task7 av_1 = 2 {av_1}
Pickup {av_0,av_1,v_7}
Task12 av_0 = 0 {av_0,av_1}
Task5 av_1 = 0 {av_1}
Task11 av_0 = 1 {av_0}
Task10 av_0 = 2 {av_0,av_1}
Task4 av_1 = 1 {av_1}
Task3 av_1 = 2 {av_1}
Task9 av_0 = 3 {av_0}
Task2 av_1 = 3 {av_1}
West {}
East {}
North {}
South {}

Figure 6.11: HierGen's hierarchy based on a single trajectory from the Taxi domain.

passenger's location to a coordinate beyond the reachable grid area (e.g., $(5, 5)$ in the $5{\times}5$ Taxi domain), but this affected the causal dependencies and the hierarchy was unable to transfer to any target configuration in which the chosen coordinate was different (e.g., $(26, 26)$ in the 25×25 version of the domain). Further complications arose due to the termination conditions being unreachable if the out-of-grid coordinate was bound to the navigation task in the discovered structure. Another motivation to change to the current state representation was developing a variant of the Taxi domain that we discuss next.

In the moving-passenger Taxi domain, the passenger moves with the taxi after being picked up, instead of the location variable being stuck at the original spot. The discovered hierarchical structure is shown in Figure 6.13. Here, the navigation tasks cannot be coalesced, because the termination conditions are different despite the explicit parame-

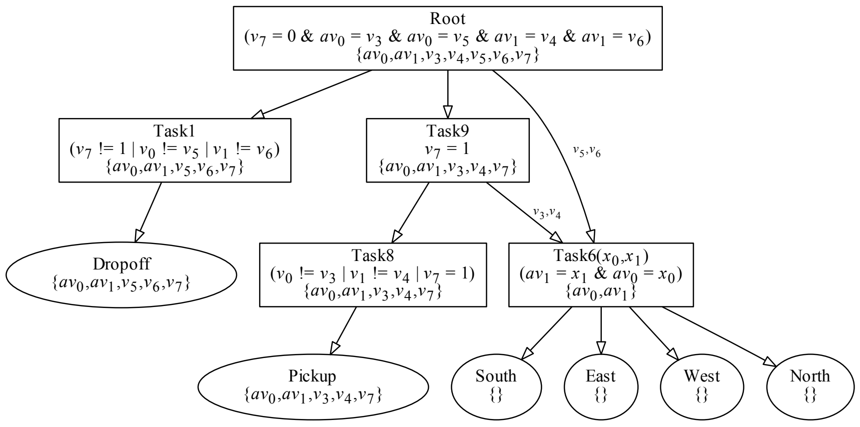

Root
(v_7 = 0 & av_0 = v_3 & av_0 = v_5 & av_1 = v_4 & av_1 = v_6)
{av_0,av_1,v_3,v_4,v_5,v_6,v_7}
Task1
(v_7 != 1 | v_0 != v_5 | v_1 != v_6)
{av_0,av_1,v_5,v_6,v_7}
Task9
v_7 = 1
{av_0,av_1,v_3,v_4,v_7}
v_5,v_6
v_3,v_4
Dropoff
{av_0,av_1,v_5,v_6,v_7}
Task8
(v_0 != v_3 | v_1 != v_4 | v_7 = 1)
{av_0,av_1,v_3,v_4,v_7}
Task6(x_0,x_1)
(av_1 = x_1 & av_0 = x_0)
{av_0,av_1}
Pickup
{av_0,av_1,v_3,v_4,v_7}
South
{}
East
{}
West
{}
North
{}

Figure 6.12: HierGen's hierarchy with admissibility-based shielding tasks based on 25 trajectories from the Taxi domain. The only purpose of a shielding task is to prevent its primitive child from being executed unsuccessfully.

terization. In another variant of the Taxi domain in which the passenger can be dropped off at any location (not just its intended destination), HierGen only discovers the shallowest hierarchy, because the random trajectories have redundant pickups/drop-offs and it is unable to generalize across them.

We also test the efficacy of HierGen in the stochastic Taxi domain, where all actions fail (no-op) with probability 0.4. Here, the unpruned trees of the learned action models introduce many redundant dependencies resulting in dense CATs. Consequently, this domain allows us to focus on how the pruning of the learned action models affects the hierarchical structure discovery. Weka accepts a confidence factor for pruning — smaller values imply heavier pruning. Figure 6.14 shows the performance of HierGen's hierarchies given various values of this confidence factor.

With no pruning, the resulting hierarchy (Figure 6.15) only succeeds in 14% of the trials, because if the taxi navigates to the destination without picking up the passenger, then Task8 is terminated and the taxi can no longer pick up the passenger. HierGen discovers this structure because the annotation incorrectly states that the navigation actions check the passenger's status variable v_7 = *pass.in-taxi*, resulting in a spurious dependence of Task8 on Task7. This implies that Task8 can only terminate when the

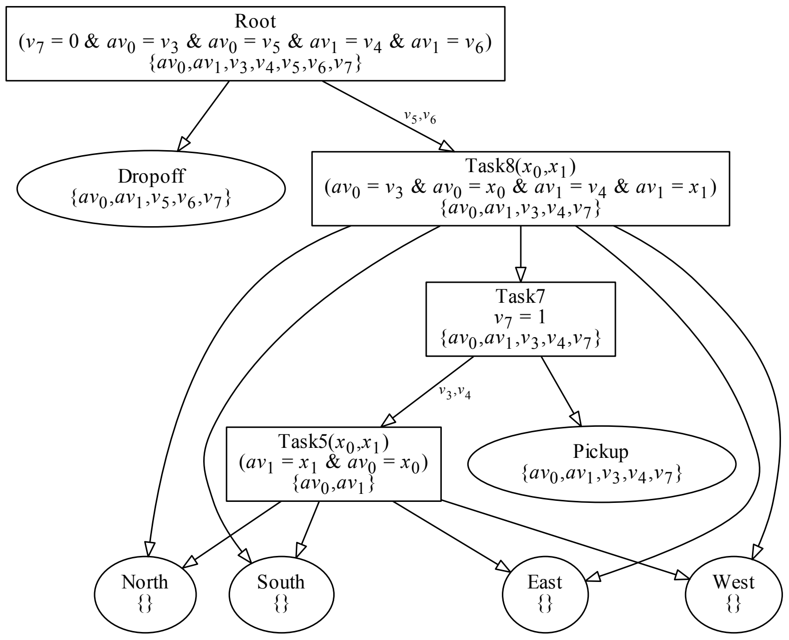
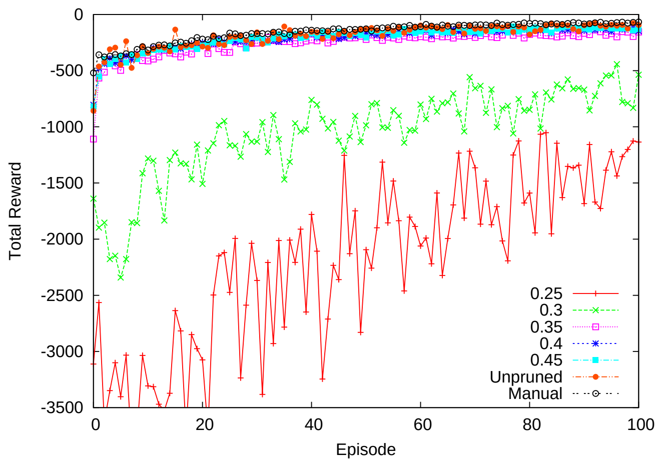

Root
$(v_7 = 0$ & $av_0 = v_3$ & $av_0 = v_5$ & $av_1 = v_4$ & $av_1 = v_6)$
$\{av_0, av_1, v_3, v_4, v_5, v_6, v_7\}$
v_5, v_6
Dropoff
$\{av_0, av_1, v_5, v_6, v_7\}$
Task8(x_0, x_1)
$(av_0 = v_3$ & $av_0 = x_0$ & $av_1 = v_4$ & $av_1 = x_1)$
$\{av_0, av_1, v_3, v_4, v_7\}$
Task7
$v_7 = 1$
$\{av_0, av_1, v_3, v_4, v_7\}$
v_3, v_4
Task5(x_0, x_1)
$(av_1 = x_1$ & $av_0 = x_0)$
$\{av_0, av_1\}$
Pickup
$\{av_0, av_1, v_3, v_4, v_7\}$
North
{}
South
{}
East
{}
West
{}

Figure 6.13: HierGen's hierarchy based on 25 trajectories from the moving-passenger Taxi domain.

passenger is in the taxi, which is not true. If the precedence graph is not transitively reduced, then this situation is avoided because the dependence of Root on Task7 is preserved as a subtask link and the taxi is able to pick up the passenger even after first navigating to the passenger's destination.

When the learned action models are maximally pruned (confidence = 1e-6), Pickup does not even appear in the CATs. This is because no action model checks *pass.in-taxi*, resulting in Pickup having no outgoing arcs and being removed. All the navigation actions are absorbed into the single task that culminates with the taxi at the destination and the resulting hierarchy is shown in Figure 6.16. Without Pickup, this hierarchy fails in the first episode of every trial.

The success rate of HierGen's hierarchies peaks at a confidence factor of 0.35 for model learning, because most of these hierarchies are equivalent to the structure of the



(a) Learning curves. The numbers in the key correspond to the confidence factor that regulates the pruning of the learning action models (smaller number = more pruning).

Confidence	Max. pruning	0.25	0.3	0.35	0.4	0.45	Unpruned
Successful trials	0%	22%	39%	95%	91%	63%	14%

(b) Success rate.

Figure 6.14: Performance of HierGen (given 50 trajectories) in the stochastic Taxi domain (across 100 trials).

hierarchy shown in Figure 6.9, except that Pickup and Dropoff have all the variables in their abstraction. A slightly lower confidence factor (= more pruning) results in Pickup having no variables in its abstraction. When provided with hand-designed action models, the HierGen hierarchy for this domain is identical to the one shown in Figure 6.9.

In a variant of the stochastic Taxi domain, the motion is perpendicular to the intended direction (uniformly randomly on either side) when the navigation actions fail. For example, when North is executed (it fails with probability 0.4), the taxi moves East with probability 0.2 and West with probability 0.2. HierGen's performance in this domain is similar to that in the stochastic Taxi domain.

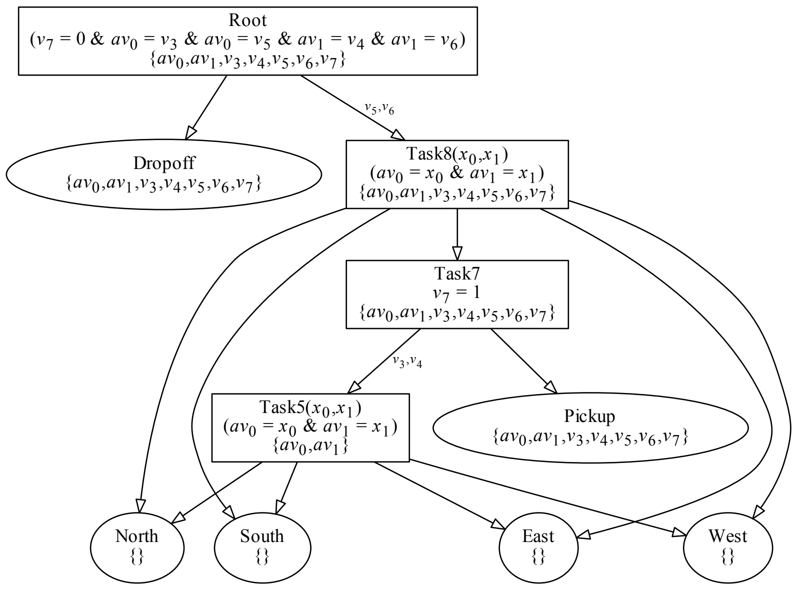

Root
($v_7 = 0$ & $av_0 = v_3$ & $av_0 = v_5$ & $av_1 = v_4$ & $av_1 = v_6$)
{$av_0,av_1,v_3,v_4,v_5,v_6,v_7$}
v_5,v_6
Dropoff
{$av_0,av_1,v_3,v_4,v_5,v_6,v_7$}
Task8(x_0,x_1)
($av_0 = x_0$ & $av_1 = x_1$)
{$av_0,av_1,v_3,v_4,v_5,v_6,v_7$}
Task7
$v_7 = 1$
{$av_0,av_1,v_3,v_4,v_5,v_6,v_7$}
v_3,v_4
Task5(x_0,x_1)
($av_0 = x_0$ & $av_1 = x_1$)
{av_0,av_1}
Pickup
{$av_0,av_1,v_3,v_4,v_5,v_6,v_7$}
North
{}
South
{}
East
{}
West
{}

Figure 6.15: HierGen's hierarchy based on 50 trajectories and unpruned action models from the stochastic Taxi domain.

6.4.2 ModBitflip Domain

In this domain, a single CAT (Figure 6.17a) is sufficient to build the complete hierarchy (Figure 6.17b), because the termination conditions do not generalize further across multiple trajectories. The learning curves are identical to those in Figure 5.9 as HierGen's hierarchy is identical to that of HI-MAT (despite the CAT being different from the RAT). Interestingly, HI-MAT's RAT-Scan fails to decompose the RAT, because the relevance of an action is always a superset of the action preceding it and the entire trajectory is absorbed into a single task; it is the recursive extraction of the ultimate action that leads to the discovered structure. Instead, HierGen directly parses the CATs into the resulting hierarchy without resorting to this extraction process.

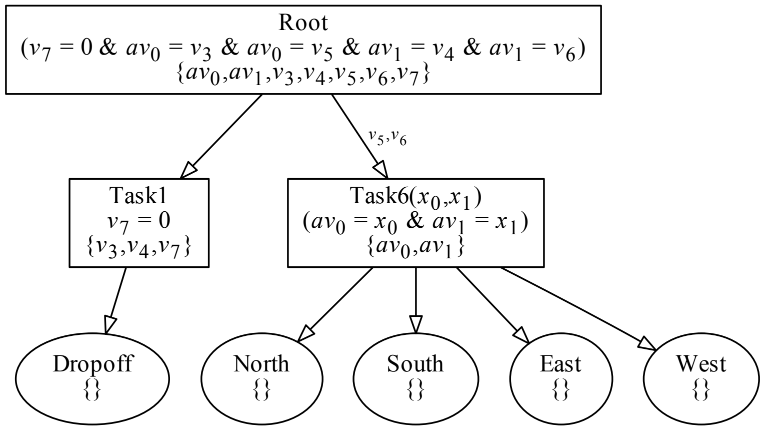
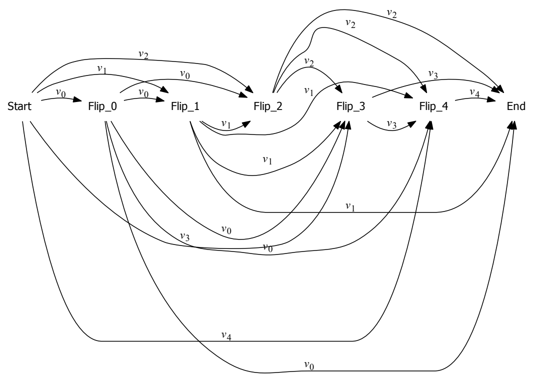

Root
($v_7 = 0$ & $av_0 = v_3$ & $av_0 = v_5$ & $av_1 = v_4$ & $av_1 = v_6$)
$\{av_0, av_1, v_3, v_4, v_5, v_6, v_7\}$
v_5, v_6
Task1
$v_7 = 0$
$\{v_3, v_4, v_7\}$
Task6(x_0,x_1)
($av_0 = x_0$ & $av_1 = x_1$)
$\{av_0, av_1\}$
Dropoff
{}
North
{}
South
{}
East
{}
West
{}

Figure 6.16: HierGen's hierarchy based on 50 trajectories and maximally-pruned action models from the stochastic Taxi domain.

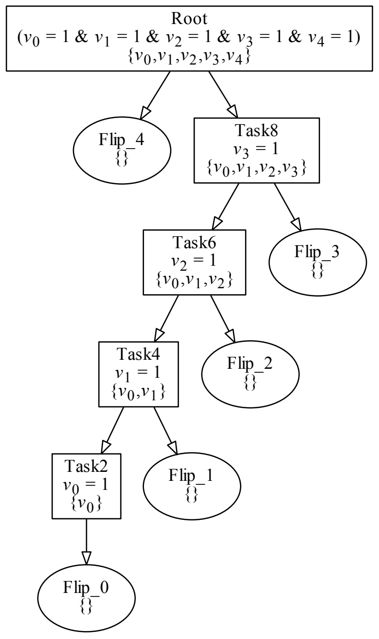
6.4.3 Wargus Domain

A random map generator is applied to construct two sets of 100 source and 100 target maps. The source maps are 20×20 square units in size with 1 peasant, and the target maps are scaled up in every dimension (number of peasants, resource quotas, etc.) but the size. This is because an increase in size entails new map regions, which in turn necessitates an increase in the number of primitive Goto actions. HI-MAT is able to use the supplied action models to match these new actions to the primitive actions already in its constructed hierarchy and supplement the hierarchy. HierGen does not assume action models a priori and has no facility to append the new primitive Goto actions to the hierarchy constructed from the source trajectories.

This domain presents significant challenges for HierGen to discover and transfer its hierarchies. The first challenge is that the peasant's coordinates do not generalize across the source maps, because the entities are placed randomly in every map. To address this, the annotation is restricted to only the variables that do transfer. Interestingly, neglecting the coordinate variables is not an issue when employing Goto actions as it is when employing single-step navigational actions; in the latter case, the navigation



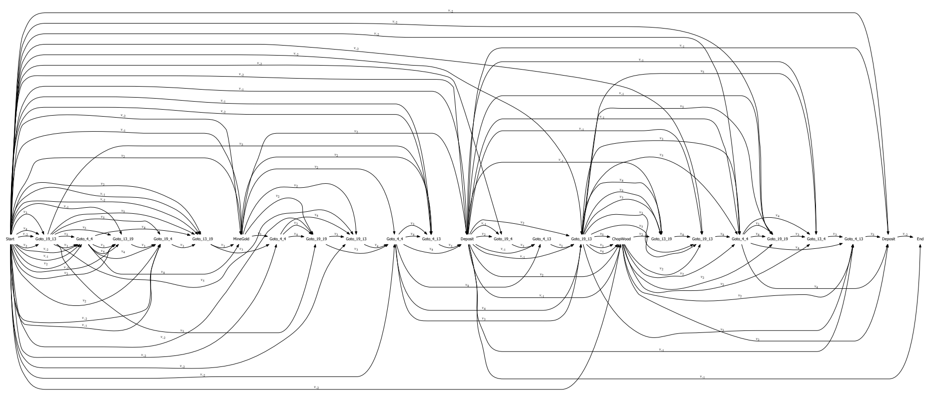
(a) A CAT.



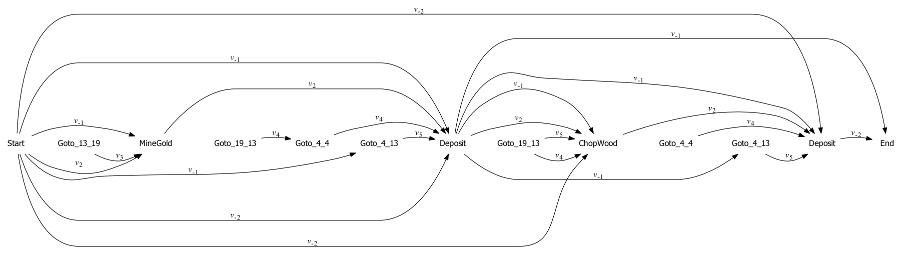
(b) HierGen's hierarchy based on a single random trajectory.

Figure 6.17: A CAT and the HierGen hierarchy for the 5-bit ModBitflip domain. Variable v_i corresponds to bit i.

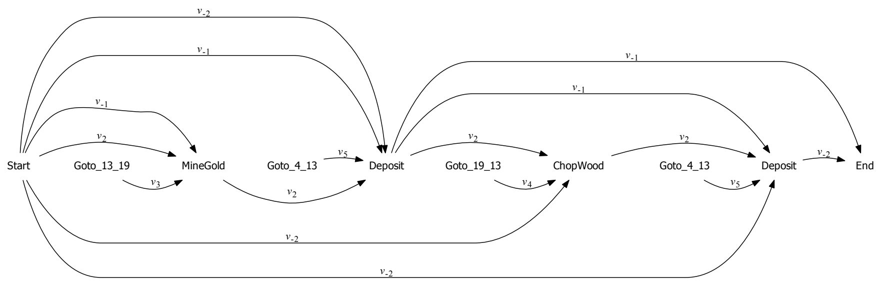
tasks in the discovered hierarchy do not have the correct abstraction, because neither the reward nor goal closures include the coordinate variables. The second challenge is that the final Deposit action affects $\{p.r, q.g\}$ in some trajectories and $\{p.r, q.w\}$ in others, which prevents unification of the termination conditions across the trajectories. To rectify this, the goal variables $q.g$ and $q.w$ are the only variables incident on End in the annotation; the goal conditions are still learned from the trajectories. The third challenge is that, with randomly generated source maps, exiting a gold mine can fortuitously place the peasant within reach of the townhall and a forest in one map but not another, again hindering unification across the trajectories. To circumvent this issue, the source maps are constrained to have their townhalls far enough away from the remaining map entities.



(a) Annotation based on learned action models.



(b) Annotation based on learned action models that are maximally pruned.



(c) Annotation based on hand-designed action models.

Figure 6.18: Causal annotations for a trajectory based on different action models from the Wargus domain. Variables: $v_{-2} = q.w$, $v_{-1} = q.g$, $v_0 = p.x$, $v_1 = p.y$, $v_2 = p.r$, $v_3 = r.g$, $v_4 = r.f$, $v_5 = r.t$. The agent-centric variables have positive indices, while global variables have negative indices.

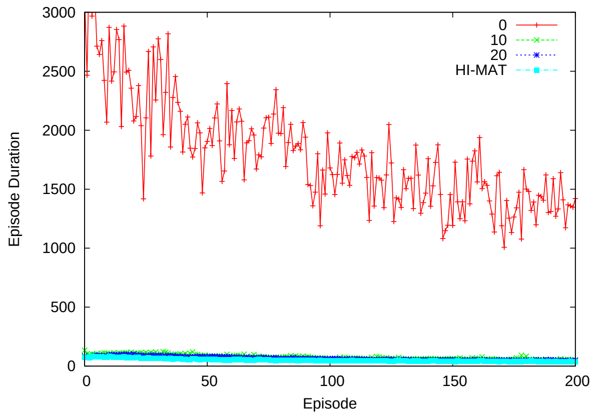
Finally, this domain is partially observable, because the topography of the map is only revealed through a limited set of location and indicator variables. Treating the domain as fully observable makes the actions appear very stochastic, which causes the learned action models (even with maximal pruning) to introduce redundant causal dependencies. To facilitate unification across trajectories, HierGen is provided with hand-designed models. For contrast, the annotations of a trajectory based on unpruned action models, models that are maximally pruned (pruning confidence = 1e-6), and manually-designed models, are shown in Figure 6.18. Just as with HI-MAT, all source trajectories are restricted to fetching every resource exactly once.

To generate a source trajectory, a source map is picked at random and a random policy executed therein. For the learning curves, one target map is picked at random for the entire trial to allow the value function (that is dependent on the peasants' coordinates) to converge. The performance of the HierGen hierarchies contrasted against that of the HI-MAT hierarchy is shown in Figure 6.19. Providing HierGen with fewer input trajectories results in either missing links between the navigation tasks and the Goto actions or missing Goto actions in the discovered hierarchy, which causes the hierarchy to fail in some trials. By 20 trajectories, the hierarchy (Figure 6.20) can successfully complete all trials. Figure 6.19b shows that the performance of this hierarchy is slightly worse than that of HI-MAT. This is because HI-MAT uses the information on the primary effects of the actions provided by the action model API to create additional constraints on the hierarchical policy, such as discovering the Put Gold and Put Wood tasks (Figure 5.7). HierGen is not afforded this luxury as it has to utilize declarative action models.

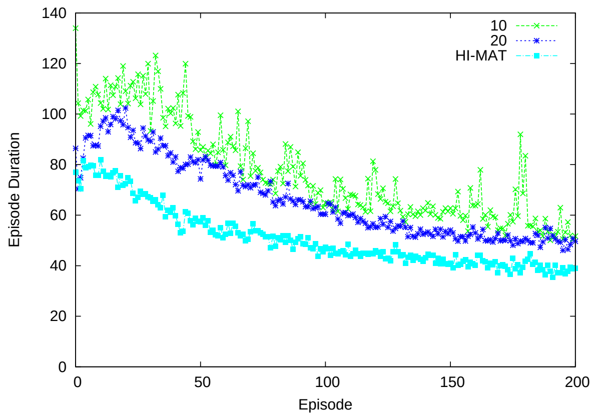
6.5 Conclusion

This chapter has described the HierGen framework that advances the HI-MAT framework in several dimensions including being able to deal with simpler models, a more robust annotation and parsing mechanism, and the simultaneous parsing of multiple trajectories.

While HierGen handles multiple trajectories, it makes strict assumptions about the nature of these trajectories. Complex real-world domains present a strong motivation for autonomous structure discovery and transfer, but the trajectories from these domains could be stochastic, suboptimal, or even unsuccessful; they could also show alternative ways of achieving the goals or involve the behaviors of multiple effectors acting in parallel.



(a) Learning curves. The numbers correspond to the number of input trajectories. The curve for 0 corresponds to the performance of the shallowest hierarchy, which is equivalent to Q-learning. The learning curves for 1 and 5 are only slightly worse than that for 10 and have been omitted for clarity.



(b) Learning curves: zoomed in.

Number of trajectories	1	5	10	20
Successful trials	6%	79%	94%	100%

(c) Success rate.

Figure 6.19: Performance of HierGen in the Wargus domain (across 100 trials).

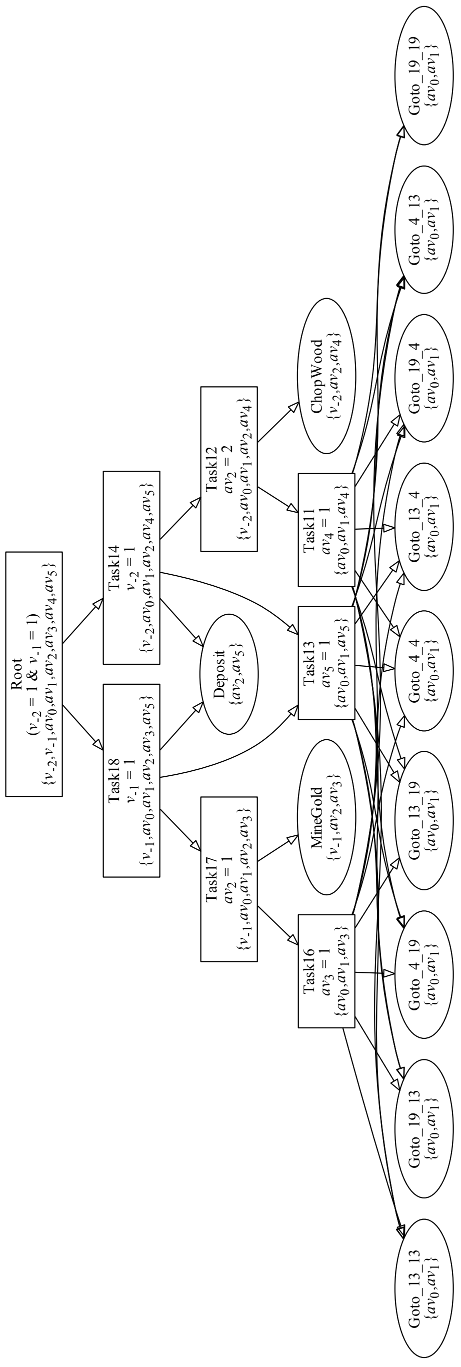


Figure 6.20: HierGen's hierarchy based on 20 trajectories from the Wargus domain.

Loosening HierGen's assumptions in the following ways is an important direction for future work:

Disjunctive subgoals: Many real-world domains exhibit disjunctive goals. Consider a simpler example of a grid-world with two goal locations g_1 and g_2; some trajectories will end at g_1 and the others at g_2, and a simple conjunctive unification is impossible. Instead, the parsing algorithm will have to recognize the disjunctive nature of the goal condition in this domain. Further, these disjunctive conditions could occur anywhere within the trajectories. For instance, if the agent starts off in a room with two doors, then two possible room exits will be demonstrated by the trajectories. The framework has to be able to cope with the vagaries of the real world and cannot assume that all the trajectories can be fully unified. Instead, it might need to cluster the trajectories based on their overall objectives before it tries to extract and generalize the essential behaviors in them. The extent of clustering will have to be balanced against generalization of the hierarchical structure based either on theoretical principles, empirical cross-validation, or an objective scoring function.

Failure conditions: Given access to only successful trajectories precludes the ability to analyze failure conditions with a domain. For instance, termination due to lack of fuel in the Fuel-Taxi domain (Dietterich, 2000) will never occur in the successful trajectory, and the agent will remain oblivious to the prevention of this failure. By analyzing both successful and unsuccessful trajectories, the parsing mechanism will have the opportunity to witness a mix of successes and failures, and assess the failure conditions. These failure conditions could then be incorporated into the termination conditions of the subtasks.

Goal maintenance: Real-world domains encompass both task-oriented and process-oriented problems (Boutilier et al., 1999). In previous chapters, we have seen that the MDP formalism is equally applicable to the problems of *goal maintenance* as it is to the problems of *goal satisfaction.* For example, situating the Wargus resource-collecting agent within the full-fledged game might require it to maintain a certain rate of resource production. In this case, the root task of this sub-hierarchy should never terminate, and the sub-hierarchy should operate in parallel with the decision-making of the overall agent. While a single trajectory certainly seems incapable of imparting goal maintenance information, it is not immediately clear how this can be learned via multiple finite-length trajectories.

Suboptimality: Although the robust parsing mechanism allows HierGen to deal with non-hierarchically generated trajectories, its decomposition algorithm is still susceptible to the suboptimality therein. For instance, when the taxi is allowed to drop the passenger off anywhere, HierGen fails to decompose because it is unable to unify termination conditions across the multiple trajectories. Being able to recognize and correct for the observed suboptimal behavior in the input trajectories is crucial to discovering more compact hierarchical structure.

Exogenous events: In this chapter, the action-model representation assumes that every change in the environment is due to some action. However, some domains have exogenous events that occur independent of the actions. For instance, in the multi-agent Taxi domain from Chapter 3, the appearance/disappearance of a passenger occurs stochastically, regardless of the actions that the agents are executing. Recognizing these exogenous events will result in more compact action models that more accurately capture the causal dependencies among the actions.

Multiple agents: The presence of multiple agents or effectors within a domain is a compelling reason to be able to parse multiple trajectories. Although the trajectories could be based in the joint state and actions spaces of the agents, this will preclude opportunities for discovering and consequently leveraging agent-centric task descriptions such as in the MASH framework. Instead, each agent might provide trajectories based on the agent-centric state and action spaces, and these trajectories have to be unified across agents.

Partial observability: Real-world domains might have state spaces that are not fully observable. For instance, a robot cannot know what objects exist in the next room until it actually navigates there. Such domains can be theoretically modeled as partially observable MDPs (POMDPs) (Kaelbling et al., 1998). However, a task hierarchy could solve a POMDP exactly if the state abstraction at a subtask matches up exactly with the set of observable state variables while that subtask is being executed. Leveraging this would be a significant boon of discovering hierarchical structure in partially observable domains.

The HierGen algorithm is essentially a batch procedure wherein the models are learned and then the trajectories are parsed. An effective agent should incrementally refine its action models as it gains more experience within a domain and use these improving action models, along with its experience so far, to refine the hierarchical structure. This would allow the agent to cope with learning and planning within a much larger domain where procuring complete trajectories might be infeasible. For instance, consider the full-blown game of Wargus. Initially, the agent has very poor knowledge of the domain and might only be able to construct the shallowest hierarchy (primitive tasks directly under the root). It could learn the action models incrementally as it gains more experience within the domain and simultaneously employ these models to refine the hierarchical structure. Arguably, learning a non-hierarchical policy could be less efficient than discovering structure and then learning the corresponding policy because of the possibility of reapplying hierarchical substructures elsewhere in the domain. Online structure discovery would delve into the following issues: ① actively learning the models required for analyzing partial trajectories; ② determining how changes in the models affect the hierarchical structure, that is, the gradient of the hierarchical structure with respect to the action models; ③ constructing higher-level state representations in parallel with structure refinement; ④ reusing the learned knowledge within a structure for future refinements of that structure; ⑤ analyzing the exploration/exploitation trade-off in the hierarchical structure space with the possible extension of existing Bayesian approaches to non-hierarchical spaces.

HierGen has dealt with the near-transfer setting where the source and target problems share the causal dependencies in their domain dynamics. Although we cannot expect positive transfer between arbitrary pairs of unrelated domains, people are able to effectively transfer between much more disparate domains than what computers are able to do at present. To be able to transfer between increasingly disparate domain pairs, the search for an effective mapping between the state variables and actions of the pair must be suitably constrained, by using analogical reasoning (Hinrichs and Forbus, 2007) for example. Moreover, transferring the entire structure of the hierarchy without alteration is less likely to be successful for disparate target tasks. Various adaptations might be necessary before the hierarchy is plugged into the target problem. For instance, the termination conditions or state abstraction might need to be changed, or the linkage structure might need some tweaking. We could even try transferring parts of the induced task hierarchy when the entire hierarchy fails to transfer. The online agent discussed earlier would conceivably broaden the scope of transfer beyond the qualitatively similar domains of HierGen by actively adapting the hierarchical structure to new situations and environments.

Just as Chapters 3 and 4 finessed the issue of hierarchy design, Chapter 5 finessed the issue of where the action models come from. This chapter partially addressed the latter issue by learning simple action models in batch from the trajectories. However, learning the action models incrementally while solving externally-provided problems or through directed exploration is still a seminal problem. Because exact action models are often difficult to learn and represent, learning approximate action models that are amenable to efficient planning also becomes necessary. The next chapter analyzes the autonomous learning of such action models in two general frameworks.

Chapter 7: Action Model Learning

Planning research typically assumes that the planning system is provided complete and correct models of the actions. However, truly autonomous agents must learn these models. Moreover, model learning, planning, and plan execution must be interleaved, because agents need to plan long before perfect models are learned. This chapter formulates and analyzes the learning of deterministic action models used in planning for goal achievement. The deterministic setting lets us impose stronger criteria for success, namely, a polynomial number of faulty plans or planning attempts before model convergence in the worst case. It has been shown that deterministic STRIPS actions with a constant number of preconditions can be learned from raw experience with at most a polynomial number of plan prediction mistakes (Walsh and Littman, 2008). In spite of this positive result, compact action models in fully observable, deterministic action models are not always efficiently learnable. For example, action models represented as arbitrary Boolean functions are not efficiently learnable under standard cryptographic assumptions such as the hardness of factoring (Kearns and Valiant, 1989).

Learning action models for planning is different from learning an arbitrary function from states and actions to next states, because one can ignore modeling the effects of some actions in certain contexts. For example, most people who drive do not ever learn a complete model of the dynamics of their vehicles; while they might accurately know the stopping distance or turning radius, they could be oblivious to many aspects that an expert auto mechanic is comfortable with. To capture this intuition, this chapter introduces the concept of an *adequate* model, that is, a model that is sound and sufficiently complete for planning for a given class of goals. When navigating a city, any spanning tree of the transportation network connecting the places of interest would be an adequate model.

This chapter defines two distinct frameworks for learning adequate models for planning and then characterizes sufficient conditions for success in these frameworks. In the *mistake-bounded planning* (MBP) framework, the goal is to continually solve user-generated planning problems while learning action models and guarantee at most a polynomial number of faulty plans or mistakes. While a polynomial number of mistakes is not always reasonable, when flying real helicopters to learn their dynamics for example, the idea here is to characterize the minimal structure that lends itself to autonomous learning when mistakes are relatively cheap. We assume that in addition to the problem generator, the learner has access to a sound and complete planner and a simulator (or the real world). This chapter also introduces a more demanding *planned exploration* (PLEX) framework, where the learner needs to generate its own problems to refine its action model. This requirement translates to an experiment-design problem, where the learner needs to design problems in a goal language to refine the action models.

The MBP and PLEX frameworks can be reduced to *over-general query* learning, concept learning with strictly one-sided error, where the learner is only allowed to make one kind of mistake (Natarajan, 1987). This is ideally suited for the autonomous learning setting in which there is no oracle who can provide positive examples of plans or demonstrations, but negative examples are observed when the agent's plans fail to achieve their goals. This chapter introduces mistake-bounded and exact learning versions of this learning framework and shows that they are strictly more powerful than the recently introduced KWIK framework (Li et al., 2008). It views an action model as a set of state-action-state transitions and ensures that the learner always maintains a hypothesis which includes all transitions in some adequate model. Thus, a sound plan is always in the learner's search space, while it may not always be generated. As the learner gains more experience in generating plans, executing them on the simulator, and receiving observations, the hypothesis is incrementally refined until an adequate model is discovered.

To ground the analysis, this chapter considers a general family of hypothesis spaces, each of which is learnable in polynomial time in the two frameworks for appropriate goal languages. This family includes a generalization of propositional STRIPS operators with conditional effects.

7.1 Over-General Query Learning

This section introduces a variant of a concept-learning framework that serves as formal underpinning of our model-learning frameworks. This variant is motivated by the principle of "optimism under uncertainty", which is at the root of several related algorithms in reinforcement learning (Brafman and Tennenholtz, 2002; Li, 2009).

Definition 7.1. A *concept* is a set of instances. An *hypothesis space* $\mathcal{H}$ is a set of strings or hypotheses, each of which represents a concept.

The size of the concept is the length of the smallest hypothesis that represents it. Without loss of generality, $\mathcal{H}$ can be structured as a *generalization graph*, where the nodes correspond to sets of equivalent hypotheses representing a concept and there is a directed edge from node n_1 to node n_2 if and only if the concept at n_1 is strictly more general than (a strict superset of) that at n_2.

Definition 7.2. The *height* of $\mathcal{H}$ is a function of n and is the length of the longest path from a root node to any node representing concepts of size n in the generalization graph of $\mathcal{H}$.

Definition 7.3. A hypothesis h is *consistent* with a set of negative examples Z if $h \cap Z = \varnothing$. Given a set of negative examples Z consistent with a target hypothesis h, the *version space* is the subset of all hypotheses in $\mathcal{H}$ that are consistent with Z and is denoted as $\mathcal{V}(Z)$.

Definition 7.4. $\mathcal{H}$ is *well-structured* if, for any negative example set Z which has a consistent target hypothesis in $\mathcal{H}$, the version space $\mathcal{V}(Z)$ contains a most general hypothesis $\mathrm{mgh}(Z)$. Further, $\mathcal{H}$ is *efficiently well-structured* if there exists an algorithm that can compute $\mathrm{mgh}(Z \cup \{z\})$ from $\mathrm{mgh}(Z)$ and a new example z in time polynomial in the size of $\mathrm{mgh}(Z)$ and z.

Definition 7.5. A hypothesis space $\mathcal{H}$ is *closed under union* if $h_1, h_2 \in \mathcal{H}$ implies that $h_1 \cup h_2 \in \mathcal{H}$.

If $\mathcal{H}$ is closed under union, then its generalization graph has a unique root node which corresponds to the most general hypothesis of $\mathcal{H}$.

Lemma 7.1. *Any finite hypothesis space $\mathcal{H}$ is well-structured if and only if it is closed under union.*

Proof. (If) Let Z be a set of negative examples and let $H_0 = \bigcup_{h \in \mathcal{V}(Z)} h$ represent the unique union of all concepts represented by hypotheses in $\mathcal{V}(Z)$. Because $\mathcal{H}$ is closed under union and finite, H_0 must be in $\mathcal{H}$. If $\exists z \in H_0 \cap Z$, then $z \in h \cap Z$ for some $h \in \mathcal{V}(Z)$. This is a contradiction, because all $h \in \mathcal{V}(Z)$ are consistent with Z. Consequently,

H_0 is consistent with Z, and is in $\mathcal{V}(Z)$. It is more general than (is a superset of) every other hypothesis in $\mathcal{V}(Z)$ because it is their union.
(Only if) Let h_1, h_2 be any two hypotheses in $\mathcal{H}$ and Z be the set of all instances not included in either h_1 and h_2. Both h_1 and h_2 are consistent with examples in Z. As $\mathcal{H}$ is well-structured, mgh(Z) must also be in the version space $\mathcal{V}(Z)$, and consequently in $\mathcal{H}$. However, mgh$(Z) = h_1 \cup h_2$ because it cannot include any element without $h_1 \cup h_2$ and must include all elements within. Hence, $h_1 \cup h_2$ is in $\mathcal{H}$, which implies that $\mathcal{H}$ is closed under union. □

In the *over-general query* (OGQ) framework, the teacher selects a target concept $c \in \mathcal{H}$. The learner outputs a query in the form of a hypothesis $h \in \mathcal{H}$, where h must be at least as general as c. The teacher responds with yes if $h \equiv c$ and the episode ends; otherwise, the teacher gives a counterexample $x \in h - c$. The learner then outputs a new query, and the cycle repeats.

Definition 7.6. A hypothesis space is *OGQ-learnable* if there exists a learning algorithm for the OGQ framework that identifies the target c with the number of queries and total running time that is polynomial in the size of c and the size of the largest counterexample.

Theorem 7.1. *$\mathcal{H}$ is learnable in the OGQ framework if and only if $\mathcal{H}$ is efficiently well-structured and its height is a polynomial function.*

Proof. (If) If $\mathcal{H}$ is efficiently well-structured, then the OGQ learner can always output the mgh, guaranteed to be more general than the target concept, in polynomial time. Because the maximum number of hypothesis refinements is bounded by the polynomial height of $\mathcal{H}$, it is learnable in the OGQ framework.
(Only if) If $\mathcal{H}$ is not well-structured, then $\exists h_1, h_2 \in \mathcal{H}, h_1 \cup h_2 \notin \mathcal{H}$. The teacher can delay picking its target concept, but always provide counterexamples from outside both h_1 and h_2. At some point, these counterexamples will force the learner to choose between h_1 or h_2, because their union is not in the hypothesis space. Once the learner makes its choice, the teacher can choose the other hypothesis as its target concept c, resulting in the learner's hypothesis not being more general than c. If $\mathcal{H}$ is not efficiently well-structured, then there exists Z and z such that computing mgh$(Z \cup \{z\})$ from mgh(Z) and a new example z cannot be done in polynomial time. If the teacher picks mgh$(Z \cup \{z\})$ as the target concept and only provides counterexamples from $Z \cup \{z\}$, then the learner cannot

have polynomial running time. Finally, the teacher can always provide counterexamples that forces the learner to take the longest path in $\mathcal{H}$'s generalization graph. Thus, if $\mathcal{H}$ does not have polynomial height, then the number of queries will not be polynomial. $\square$

7.1.1 Over-General Mistake-Bounded Learning

In order to facilitate a comparison of the OGQ framework to other learning frameworks, we first define the *over-general mistake-bounded* (OGMB) learning framework, in which the teacher selects a target concept c from $\mathcal{H}$ and presents an arbitrary instance x from the instance space to the learner for a prediction. An inclusion mistake is made when the learner predicts $x \in c$ although $x \notin c$; an exclusion mistake is made when the learner predicts $x \notin c$ although $x \in c$. The teacher presents the true label to the learner if a mistake is made, and then presents the next instance to the learner, and so on.

Definition 7.7. A hypothesis space is *OGMB-learnable* if there exists a learning algorithm for the OGMB framework that never makes any exclusion mistakes and its number of inclusion mistakes and the running time on each instance are both bounded by polynomial functions of the size of the target concept and the size of the largest instance seen by the learner.

Theorem 7.2. *OGQ* $\subsetneq$ *OGMB.*

Proof. We can construct an OGMB learner from the OGQ learner as follows. When the OGQ learner makes a query h, we use h to make predictions for the OGMB learner. As h is guaranteed to be over-general, it never makes an exclusion mistake. Any instance x on which it makes an inclusion mistake must be in $h - c$ and this is returned to the OGQ learner. The cycle repeats with the OGQ learner providing a new query. Because the OGQ learner makes only a polynomial number of queries and takes polynomial time for query generation, the simulated OGMB learner makes only a polynomial number of mistakes and runs in at most polynomial time per instance. The converse does not hold in general because the queries of the OGQ learner are restricted to be "proper", that is, they must belong to the given hypothesis space. While the OGMB learner can maintain the version space of all consistent hypotheses of a polynomially-sized hypothesis space, the OGQ learner can only query with a single hypothesis and there may not be any hypothesis that is guaranteed to be more general than the target concept. $\square$

If the learner is allowed to ask queries outside $\mathcal{H}$, such as queries of the form $h_1 \cup \cdots \cup h_n$ for all h_i in the version space, then over-general learning is possible. In general, if the learner is allowed to ask about any polynomially-sized, polynomial-time computable hypothesis, then it is as powerful as OGMB, because it can encode the computation of the OGMB learner inside a polynomial-sized circuit and query with that as the hypothesis. We call this the OGQ+ framework and claim the following theorem (the proof is straightforward).

Theorem 7.3. *OGQ+ = OGMB.*

7.1.2 A Comparison of Learning Frameworks

The Knows-What-It-Knows (KWIK) learning framework (Li et al., 2008) is similar to the OGMB framework with one key difference: it does not allow the learner to make any prediction when it does not know the correct answer. In other words, the learner either makes a correct prediction or simply abstains from making a prediction and gets the true label from the teacher. The number of abstentions is bounded by a polynomial in the target size and the largest instance size. The set of hypothesis spaces learnable in the mistake-bound (MB) framework is a strict subset of that learnable in the probably-approximately-correct (PAC) framework (Littlestone, 1989) leading to the following result.

Theorem 7.4. *KWIK $\subsetneq$ OGMB $\subsetneq$ MB $\subsetneq$ PAC.*

Proof. OGMB $\subsetneq$ MB: Every hypothesis space that is OGMB-learnable is MB-learnable because the OGMB learner is additionally constrained to not make an exclusion mistake. However, every MB-learnable hypothesis space is not OGMB-learnable. Consider the hypothesis space of conjunctions of n Boolean literals (positive or negative). A single exclusion mistake is sufficient for an MB learner to learn this hypothesis space. In contrast, after making an inclusion mistake, the OGMB learner can only exclude that example from the candidate set. As there is exactly one positive example, this could force the OGMB learner to make an exponential number of mistakes (similar to guessing an unknown password).

KWIK $\subsetneq$ OGMB: If a concept class is KWIK-learnable, it is also OGMB-learnable — when the KWIK learner does not know the true label, the OGMB learner simply predicts that the instance is positive and gets corrected if it is wrong. However, every OGMB-learnable hypothesis space is not KWIK-learnable. Consider the hypothesis space of disjunctions of n Boolean literals. The OGMB learner begins with a disjunction over all possible literals (both positive and negative) and hence predicts all instances as positive. A single inclusion mistake is sufficient for the OGMB learner to learn this hypothesis space. On the other hand, the teacher can supply the KWIK learner with an exponential number of positive examples, because the KWIK learner cannot ever know that the target does not include all possible instances; this implies that the number of abstentions is not polynomially bounded. $\square$

This theorem demonstrates that KWIK is too conservative a framework for model learning — any prediction that might be a mistake is disallowed. This makes it impossible to learn even simple concept classes such as pure disjunctions.

7.2 Planning Components

A factored planning domain $\mathcal{P}$ is the tuple $(\mathcal{X}, A, T)$, where $\mathcal{X} = \{v_1, \ldots, v_n\}$ is the set of state variables and A is the set of actions. $S = \times_i \mathcal{D}(v_i)$ represents the state space and $T \subset S \times A \times S$ is the transition relation, where $(s, a, s') \in T$ signifies that taking action a in state s results in state s'. As we only consider learning deterministic action models, the transition relation is in fact a function, although the learner's hypothesis space may include nondeterministic models. The domain parameters, $|A|, n$, and $d = \max_i |\mathcal{D}(v_i)|$, characterize the size of $\mathcal{P}$ and are *implicit in all claims of complexity* in the remainder of this chapter.

Definition 7.8. An *action model* is a relation $M \subseteq S \times A \times S$.

Definition 7.9. A *planning problem* is a pair (s_0, g), where $s_0 \in S$ and the goal condition g is an expression chosen from a goal language $\mathcal{G}$ and represents a set of states in which it evaluates to true. A state s *satisfies* a goal g if and only if g is true in s.

Definition 7.10. Given a planning problem (s_0, g), a *plan* is a sequence of states and actions $s_0, a_1, \ldots, a_p, s_p$, where the state s_p satisfies the goal g. The plan is *sound* with respect to (M, g) if $(s_{i-1}, a_i, s_i) \in M$ for $1 \leq i \leq p$.

Definition 7.11. A *planner* for the hypothesis-goal space $(\mathcal{H}, \mathcal{G})$ is an algorithm that takes $M \in \mathcal{H}$ and $(s_0, g \in \mathcal{G})$ as inputs and outputs a plan or signals failure. It is *sound* with respect to $(\mathcal{H}, \mathcal{G})$ if, given any M and (s_0, g), it produces a sound plan with respect to (M, g) or signals failure. It is *complete* with respect to $(\mathcal{H}, \mathcal{G})$ if, given any M and (s_0, g), it produces a sound plan whenever one exists with respect to (M, g).

We generalize the definition of soundness from its standard usage in the literature in order to apply to nondeterministic action models, where the nondeterminism is "angelic" — the planner can control the outcome of actions when multiple outcomes are possible according to its model (Marthi et al., 2007). One way to implement such a planner is to do forward search through all possible action and outcome sequences and return an action sequence if it leads to a goal under some outcome choices. Our analysis is agnostic to plan quality or plan length and applies equally well to suboptimal planners. This is motivated by the fact that optimal planning is hard for most domains, but suboptimal planning such as hierarchical planning can be quite efficient and practical.

Definition 7.12. A *planning mistake* occurs if either the planner signals failure when a sound plan exists with respect to the transition function T or when the plan output by the planner is not sound with respect to T.

We now describe the concept of an adequate action model for a class of goals.

Definition 7.13. Let $\mathcal{P}$ be a planning domain and $\mathcal{G}$ be a goal language. An action model M is *adequate* for $\mathcal{G}$ in $\mathcal{P}$ if $M \subseteq T$ and the existence of a sound plan with respect to $(T, g \in \mathcal{G})$ implies the existence of a sound plan with respect to (M, g). $\mathcal{H}$ is adequate for $\mathcal{G}$ if $\exists M \in \mathcal{H}$ such that M is adequate for $\mathcal{G}$.

An adequate model may be partial or incomplete in that it may not include every possible transition in the transition function T. However, the model is sufficient to produce a sound plan with respect to (T, g) for every goal g in the desired language. Thus, the more limited the goal language, the more incomplete the adequate model can be. In the example of a city map, if the goal language excludes certain locations, then the spanning tree would be able to exclude them as well, although not necessarily so.

Definition 7.14. A *simulator* of the domain is always situated in the current state s. It takes an action a as input, transitions to the state s' resulting from executing a in s, and returns the current state s'.

Definition 7.15. Given a goal language $\mathcal{G}$, a *problem generator* generates an arbitrary problem $(s_0, g \in \mathcal{G})$ and sets the state of the simulator to s_0.

7.3 Mistake-Bounded Planning Framework

This section constructs the MBP framework that allows learning and planning to be interleaved for user-generated problems. It actualizes the teacher of the OGQ framework by a combination of a problem generator, a planner, and a simulator, and interfaces with the OGQ learner to learn action models as hypotheses over the space of possible state transitions for each action. It turns out that the one-sided mistake property is needed for autonomous learning because the learner can only learn by generating plans and observing the results; if the learner ever makes an exclusion error, there is no guarantee of finding a sound plan even when one exists and the learner cannot recover from such mistakes.

Definition 7.16. Let $\mathcal{G}$ be a goal language such that $\mathcal{H}$ is adequate for it. $\mathcal{H}$ is *learnable in the MBP framework* if there exists an algorithm L that interacts with a problem generator over $\mathcal{G}$, a sound and complete planner with respect to $(\mathcal{H}, \mathcal{G})$, and a simulator of the planning domain $\mathcal{P}$, and outputs a plan or signals failure for each planning problem while guaranteeing at most a polynomial number of planning mistakes. Further, L must respond in time polynomial in the domain parameters and the length of the longest plan generated by the planner, assuming that a call to the planner, simulator, or problem generator takes $O(1)$ time.

The goal language is picked such that the hypothesis space is adequate for it. We cannot bound the time for the convergence of L, because there is no limit on when the mistakes are made.

Theorem 7.5. *$\mathcal{H}$ is learnable in the MBP framework if $\mathcal{H}$ is OGQ-learnable.*

Proof. Algorithm 7.1 is a general schema for action model learning in the MBP framework. The model M begins with the initial query from OGQ-LEARNER. PROBLEM-GENERATOR provides a planning problem and initializes the current state of SIMULATOR. Given M and the planning problem, PLANNER always outputs a plan if one exists because $\mathcal{H}$ is adequate for $\mathcal{G}$ (it contains a "target" adequate model) and M is at least

Algorithm 7.1 MBP Learning Schema

```
Input: Goal language $\mathcal{G}$
1:  $M \leftarrow$ OGQ-Learner()  // Initial query
2:  loop
3:     $(s, g) \leftarrow$ ProblemGenerator($\mathcal{G}$)
4:     $plan \leftarrow$ Planner($M, (s, g)$)
5:     if $plan \neq$ false then
6:        for $(\hat{s}, a, \hat{s}')$ in $plan$ do
7:           $s' \leftarrow$ Simulator($a$)
8:           if $s' \neq \hat{s}'$ then
9:              $M \leftarrow$ OGQ-Learner($(s, a, \hat{s}')$)
10:             print  mistake
11:             break
12:          $s \leftarrow s'$
13:       if no mistake then
14:          print  $plan$
```

as general as every adequate model. If Planner signals failure, then there is no plan for it. Otherwise, the plan is executed through Simulator until an observed transition conflicts with the predicted transition. If such a transition is found, it is supplied to OGQ-Learner and M is updated with the next query; otherwise, the plan is output. If $\mathcal{H}$ is OGQ-learnable, then OGQ-Learner will only be called a polynomial number of times, every call taking polynomial time. As the number of planning mistakes is polynomial and every response of Algorithm 7.1 is polynomial in the runtime of OGQ-Learner and the length of the longest plan, $\mathcal{H}$ is learnable in the MBP framework. $\square$

The above result generalizes the work on learning STRIPS operator models from raw experience (without a teacher) in (Walsh and Littman, 2008) to arbitrary hypotheses spaces by identifying sufficiency conditions. (A family of hypothesis spaces considered later in this chapter subsumes propositional STRIPS by capturing conditional effects.) It also clarifies the notion of an adequate model, which can be much simpler than the true transition model, and the influence of the goal language on the complexity of learning action models.

7.4 Planned Exploration Framework

The MBP framework is appropriate when mistakes are permissible on user-given problems as long as their total number is limited and not for cases where no mistakes are permitted after the training period. In the planned exploration (PLEX) framework, the agent seeks to learn an action model for the domain without an external problem generator by generating planning problems for itself. The key issue here is to generate a reasonably small number of planning problems such that solving them would identify a deterministic action model. Learning a model in the PLEX framework involves knowing where it is deficient and then planning to reach states that are informative, which entails formulating planning problems in a goal language. This framework provides a polynomial sample convergence guarantee which is stronger than a polynomial mistake bound of the MBP framework. Without a problem generator that can change the simulator's state, it is impossible for the simulator to transition freely between strongly connected components (SCCs) of the transition graph. Hence, we make the assumption that the transition graph is a disconnected union of SCCs and require only that the agent learn the model for a single SCC that contains the initial state of the simulator.

Definition 7.17. Let $\mathcal{P}$ be a planning domain whose transition graph is a union of SCCs. $(\mathcal{H}, \mathcal{G})$ is *learnable in the PLEX framework* if there exists an algorithm L that interacts with a sound and complete planner with respect to $(\mathcal{H}, \mathcal{G})$ and the simulator for $\mathcal{P}$ and outputs a model $M \in \mathcal{H}$ that is adequate for $\mathcal{G}$ within the SCC that contains the initial state s_0 of the simulator after a polynomial number of planning attempts. Further, L must run in polynomial time in the domain parameters and the length of the longest plan output by the planner, assuming that every call to the planner and the simulator takes $O(1)$ time.

A key step in planned exploration is designing appropriate planning problems. We call these *experiments* because the goal of solving these problems is to disambiguate nondeterministic action models. In particular, the agent tries to reach an *informative* state where the current model is nondeterministic.

Definition 7.18. Given a model M, the set of *informative states* is $I(M) = \{s : (s, a, s'), (s, a, s'') \in M \land s' \neq s''\}$, where a is said to be *informative* in s.

Definition 7.19. A set of goals G is a *cover* of a set of states R if

$$\bigcup_{g \in G} \{s : s \text{ satisfies } g\} = R.$$

Given the goal language $\mathcal{G}$ and a model M, the problem of experiment design is to find a set of goals $G \subseteq \mathcal{G}$ such that the sets of states that satisfy the goals in G collectively cover all informative states $I(M)$. If it is possible to plan to achieve one of these goals, then either the plan passes through a state where the model is nondeterministic or it executes successfully and the agent reaches the final goal state; in either case, an informative action can be executed and the observed transition is used to refine the model. If none of the goals in G can be successfully planned for, then no informative states for that action are reachable. We formalize these intuitions below.

Definition 7.20. The *width* of $(\mathcal{H}, \mathcal{G})$ is defined as

$$\max_{M \in \mathcal{H}} \min_{G \subseteq \mathcal{G} : G \text{ is a cover of } I(M)} |G|,$$

where $\min_G |G| = \infty$ if there is no $G \subseteq \mathcal{G}$ to cover a nonempty $I(M)$.

Definition 7.21. $(\mathcal{H}, \mathcal{G})$ permits *efficient* experiment design if, for any $M \in \mathcal{H}$, ① there exists an algorithm (ExperimentDesign) that takes M and $\mathcal{G}$ as input and outputs a polynomial-sized cover of $I(M)$ in polynomial time and ② there exists an algorithm (InfoActionStates) that takes M and a state s as input and outputs an informative action and two (distinct) predicted next states according to M in polynomial time.

If $(\mathcal{H}, \mathcal{G})$ permits efficient experiment design, then it has polynomial width because no algorithm can always guarantee to output a polynomial-sized cover otherwise.

Theorem 7.6. *$(\mathcal{H}, \mathcal{G})$ is learnable in the PLEX framework if it permits efficient experiment design, and $\mathcal{H}$ is adequate for $\mathcal{G}$ and is OGQ-learnable.*

Proof. Algorithm 7.2 is a general schema for action model learning in the PLEX framework. The model M begins with the initial query from OGQ-Learner. Given M and $\mathcal{G}$, ExperimentDesign computes a polynomial-sized cover G. If G is empty, then the model cannot be refined further; otherwise, given M and a goal $g \in G$, Planner

Algorithm 7.2 PLEX LEARNING SCHEMA

```
Input: Initial state $s$, goal language $\mathcal{G}$
Output: Model $M$
1:  $M \leftarrow$ OGQ-LEARNER()  // Initial query
2:  loop
3:    $G \leftarrow$ EXPERIMENTDESIGN($M, \mathcal{G}$)
4:    if $G = \varnothing$ then
5:      return $M$
6:    for $g \in G$ do
7:      $plan \leftarrow$ PLANNER($M, (s, g)$)
8:      if $plan \neq$ false then
9:        break
10:   if $plan =$ false then
11:     return $M$
12:   for $(\hat{s}, a, \hat{s}')$ in $plan$ do
13:     $s' \leftarrow$ SIMULATOR($a$)
14:     $s \leftarrow s'$
15:     if $s' \neq \hat{s}'$ then
16:       $M \leftarrow$ OGQ-LEARNER($(s, a, \hat{s}')$)
17:       break
18:   if $M$ has not been updated then
19:     $(a, \hat{S}') \leftarrow$ INFOACTIONSTATES($M, s$)
20:     $s' \leftarrow$ SIMULATOR($a$)
21:     $M \leftarrow$ OGQ-LEARNER($(s, a, \hat{s}' \in \hat{S}' - \{s'\})$)
22:     $s \leftarrow s'$
23: return $M$
```

may signal failure if either no state satisfies g or states satisfying g are not reachable from the current state of the simulator. If PLANNER signals failure on all of the goals, then none of the informative states are reachable and M cannot be refined further. If PLANNER does output a plan, then the plan is executed through SIMULATOR until an observed transition conflicts with the predicted transition. If such a transition is found, it is supplied to OGQ-LEARNER and M is updated with the next query. If the plan executes successfully, then INFOACTIONSTATES provides an informative action with the corresponding set of two resultant states according to M; OGQ-LEARNER is supplied with the transition of the goal state, the informative action, and the incorrectly predicted next state, and M is updated with the new query. A new cover is computed every time

Table 7.1: The principal differences between the MBP and PLEX frameworks.

	MBP	**PLEX**
Planning problems	Externally generated	Internally designed
Sample complexity	Poly. number of mistakes	Poly. number of planning attempts
Computational complexity	Polynomial per response	Polynomial overall

M is updated, and the process continues until all experiments are exhausted. If $(\mathcal{H}, \mathcal{G})$ permits efficient experiment design, then every cover can be computed in polynomial time and InfoActionStates is efficient. If $\mathcal{H}$ is OGQ-learnable, then OGQ-Learner will only be called a polynomial number of times and it can output a new query in polynomial time. As the number of failures per successful plan is bounded by a polynomial in the width w of $(\mathcal{H}, \mathcal{G})$, the total number of calls to Planner is polynomial. Further, as the innermost loop of Algorithm 7.2 is bounded by the longest length l of a plan, its running time is a polynomial in the domain parameters and l. Thus, $(\mathcal{H}, \mathcal{G})$ is learnable in the PLEX framework. □

The key differences between the MBP and PLEX frameworks are highlighted in Table 7.1.

7.5 A Hypothesis Family for Action Modeling

This section describes a family of hypothesis spaces for action modeling and proves its learnability in the MBP and PLEX frameworks. Let $\mathcal{U} = \{u_1, u_2, \ldots\}$ be a polynomial-sized set of polynomially computable basis hypotheses (polynomial in the relevant parameters), where u_i represents a deterministic set of transition tuples. Let Power$(\mathcal{U}) = \{\bigcup_{u \in H} u : H \subseteq \mathcal{U}\}$ and Pairs$(\mathcal{U}) = \{u_1 \cup u_2 : u_1, u_2 \in \mathcal{U}\}$.

Lemma 7.2. *Power($\mathcal{U}$) is OGQ-learnable.*

Proof. Power$(\mathcal{U})$ is efficiently well-structured, because it is closed under union by definition and the new mgh can be computed by removing any basis hypotheses that are not consistent with the counterexample; this takes polynomial time as $\mathcal{U}$ is of polynomial size. At the root of the generalization graph of Power$(\mathcal{U})$ is the hypothesis $\bigcup_{u \in \mathcal{U}} u$ and at the leaf is the empty hypothesis. Because $\mathcal{U}$ is of polynomial size and the longest path from the root to the leaf involves removing a single component at a time, the height of Power$(\mathcal{U})$ is polynomial. □

Lemma 7.3. *Power($\mathcal{U}$) is learnable in the MBP framework.*

Proof. This follows from Lemma 7.2 and Theorem 7.5. □

Lemma 7.4. *For any goal language $\mathcal{G}$, (Power($\mathcal{U}$), $\mathcal{G}$) permits efficient experiment design if (Pairs($\mathcal{U}$), $\mathcal{G}$) permits efficient experiment design.*

Proof. Any informative state for a hypothesis in Power($\mathcal{U}$) is an informative state for some hypothesis in Pairs($\mathcal{U}$), and vice versa. Hence, a cover for (Pairs($\mathcal{U}$), $\mathcal{G}$) would be a cover for $(Power(\mathcal{U}), \mathcal{G})$. Consequently, if (Pairs($\mathcal{U}$), $\mathcal{G}$) permits efficient experiment design, then the associated efficient algorithms ExperimentDesign and InfoActionStates are directly applicable to (Power($\mathcal{U}$), $\mathcal{G}$). □

Lemma 7.5. *For any goal language $\mathcal{G}$, (Power($\mathcal{U}$), $\mathcal{G}$) is learnable in the PLEX framework if (Pairs($\mathcal{U}$), $\mathcal{G}$) permits efficient experiment design and Power($\mathcal{U}$) is adequate for $\mathcal{G}$.*

Proof. This follows from Lemmas 7.2 and 7.4, and Theorem 7.6. □

7.5.1 Sets of Action Productions

We now define a hypothesis space that is a concrete member of the family. Let an *action production* r be defined as "act : pre $\rightarrow$ post", where act(r) is an action and the precondition pre(r) and postcondition post(r) are conjunctions of "variable = value" literals.

Definition 7.22. A production r is *triggered* by a transition (s, a, s') if s satisfies the precondition $pre(r)$ and $a = \text{act}(r)$. A production r is *consistent* with (s, a, s') if either ① r is not triggered by (s, a, s') or ② s' satisfies the post(r) and all variables not mentioned in post(r) have the same values in both s and s'.

A production represents the set of all consistent transitions that trigger it. All the variables in pre(r) must take their specified values in a state to trigger r; when r is triggered, post(r) defines the values in the next state. An example of an action production is "Do : $v_1 = 0, v_2 = 1 \rightarrow v_1 = 2, v_3 = 1$". It is triggered only when the Do action is executed in a state in which $v_1 = 0$ and $v_2 = 1$, and defines the value of v_1 to be 2 and v_3 to be 1 in the next state, with all other variables staying unchanged.

Let k-SAP be the hypothesis space of models represented by a set of action productions (SAP) with no more than k variables per production. If $\mathcal{U}$ is the set of productions, then $|\mathcal{U}| = O\big(|A| \sum_{i=1}^{k} \binom{n}{i} (d+1)^{2i}\big) = O(|A| n^k d^{2k})$, because a production can have one of $|A|$ actions, up to k relevant variables figuring on either side of the production, and each variable set to a value in its domain. As $\mathcal{U}$ is of polynomial size, k-SAP is an instance of the family of basis action models. Moreover, if Conj is the goal language consisting of all goals that can be expressed as conjunctions of "variable = value" literals, then (Pairs(k-SAP), Conj) permits efficient experiment design.

Lemma 7.6. *(k-SAP, Conj) is learnable in the PLEX framework if k-SAP is adequate for Conj.*

7.6 Conclusion

The first contribution of this chapter is the identification of adequate models for characterizing the complexity of learning. The second contribution is the development of the MBP and PLEX frameworks. This chapter clarifies the relationship between the expressiveness of the goal language and the complexity of learning the action models. The third contribution is the results on learning a family of hypothesis spaces that is, in some ways, more general than standard action modeling languages. For example, unlike propositional STRIPS operators, k-SAP captures the conditional effects of actions.

While STRIPS-like languages served us well in planning research by creating a common useful platform, they are not designed from the point of view of learnability or planning efficiency. Many domains such as robotics and real-time strategy games are not amenable to such clean and simple action specification languages. This suggests an approach in which the learner considers increasingly complex models as dictated by its planning needs. For example, the model learner might start with small values of k in k-SAP and then incrementally increase k until a value is found that is adequate for the goals encountered. In general, this motivates a more comprehensive framework in which planning and learning are tightly integrated, which is the premise of this chapter. Another direction is to investigate better exploration methods that go beyond using optimistic models to include Bayesian and utility-guided optimal exploration.

Chapter 8: Conclusion

This dissertation has analyzed the autonomous discovery and transfer of hierarchical structure in sequential decision problems. This chapter summarizes the contributions of this body of work and looks ahead to the future.

8.1 Summary of the Dissertation

The contributions of the dissertation can be summarized as follows:

- Chapter 3 develops the MASH framework that facilitates the transfer of knowledge across multiple cooperating agents in an environment. This framework allows agents coordinating in a domain to share their hierarchical value functions to be able to learn more effectively. With coordination, this framework can speed up learning in multi-agent domains as demonstrated by the empirical results.

- Chapter 4 describes the VRHRL framework in which vectorized value-function learning and the caching of hierarchical task policies leads to effective and accelerated transfer across variable-reward MDPs. In the model-based setting, this framework has the added advantage that the transition models need not be relearned from scratch when only the rewards change.

- Chapter 5 presents the HI-MAT approach to discovering hierarchical structure in a source problem and transferring it to target problems that share the same relevant structure of the system dynamics. Given action models and a single successful trajectory, HI-MAT analyzes the relevant dependencies among the actions to autonomously discover a compact and safe task hierarchy that is consistent with the trajectory. Empirical results validate that leveraging the observed trajectory allows HI-MAT to learn more compact hierarchies than algorithms that employ only action models. Further, in a transfer setting, HI-MAT hierarchies perform comparably to manually-designed hierarchies and provide more effective transfer than direct transfer of the value function.

- Chapter 6 develops the HierGen approach that advances HI-MAT in several significant ways, including employing simple action models that can be learned easily, a more robust trajectory parsing mechanism that facilitates generalization across multiple trajectories, and increased subtask sharing through explicit parameterization. In this work, the structure of the task hierarchy transfers in its entirety to target domains in which the actions share the same causal dependencies.
- Chapter 7 introduces a new notion of action model approximation, adequacy, and characterizes the complexity of learning such models in two new frameworks. In doing so, it clarifies the relationship between the expressivity of the model representation and that of the goal language. It also provides results for learning a concrete family of hypothesis spaces that is more general than common action-modeling representations.

While this dissertation has focused on hierarchical structure discovery and transfer in sequential decision problems, the broader implications of the research are extensive. In more grandiose terms, this research aims to analyze the analysis of problems — it tries to get at the heart of the divide-and-conquer strategy for problem-solving, an invaluable tool of human analysis and comprehension. It could be taking the first baby steps toward automated computer programming.

8.2 Future Work

I outline two general directions for future work.

8.2.1 Richer Hierarchical Structure

In order to function in real-world domains, the hierarchical structures must have the representational capacity to deal with the complexity within these domains. Two strategies to make hierarchical structure more expressive are as follows.

State abstraction is crucial to hierarchical structure. HierGen autonomously discovers the appropriate state abstraction for the hierarchical structure assuming that the discrete state variables and actions are designed with the potential for structure — poor representation will severely hamper its abilities to discover structure. For instance, domains using a factored state space and actions that affect as few factors as possible are most conducive to structure discovery. However, this assumption is a luxury in real-world domains where the state features are rudimentary sensor inputs and the actions are primitive actuators. Higher-level reasoning requires synthesizing richer and deeper state representations from the more primitive ones, culminating in a representational type hierarchy. The recent developments in deep learning and value function approximation (regularized approximate linear programming for instance) will be very relevant in serving this need. Moreover, autonomous hierarchical structure discovery can decompose a partially observable problem into a set of fully observable ones if it is possible to develop state representation for the subproblems such that only the set of observable variables are relevant during the task currently being executed. For instance, the exact situation in the room adjacent to the robot's location is unknown, but only becomes a part of the abstraction when the robot navigates there and assesses it. A related issue is evaluating the trade-off of the learning performance of deficient or inexact state abstraction versus the reduction in space and computational complexity.

The goal specification language is a succinct representation of the purpose of a task within the hierarchical structure and a richer language can express richer task functionality. While the language of conjunctions of equalities might suffice for simple discrete variables, those in real-world domains are likely to be thresholded (inequalities) or combined functionally and relationally. For instance, a robot might need to leverage deictic references for objects it does not model explicitly or it might have to regulate its speed as a function of the terrain. In such domains with continuous actions and variables and functional goals, parameterization will be essential because the size of the structure without parameterization could be potentially infinite. More expressive goal languages will enable new opportunities such as intention propagation for state abstraction, where a decision can be passed down the hierarchical structure via parameterization, obviating the need for any of the descendants to reassess those variables.

8.2.2 Increased Scope of Transfer

The transfer or generalization of knowledge begs the question: how related are the source and target domains? Through this dissertation, the assumptions about the relatedness of the domains for knowledge transfer has been weakened. The MASH framework begins with assuming that the agents are in exactly the same domain. The VRHRL framework progresses to domains that share the transition but not the reward dynamics. The HI-MAT and HierGen systems have assumed only that the system dynamics share the same qualitative dependencies.

Further loosening the assumptions for transfer presents an interesting avenue for future work. For instance, the domains could belong to a family in which the variables and actions map bijectively from one to another. This mapping could be relaxed to be injective or surjective. The mappings could also be partial, where elements of the domains might be undefined. In more ambitious transfer scenarios, various adaptations such as transferring only specific subtasks of the induced hierarchy might become necessary. Such transfer scenarios will also necessitate richer hierarchical structure.

Bibliography

P. Abbeel and A. Ng. Apprenticeship Learning via Inverse Reinforcement Learning. In *International Conference on Machine Learning*, 2004.

D. Andre and S. Russell. State Abstraction for Programmable Reinforcement Learning Agents. In *National Conference on Artificial Intelligence*, 2002.

R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller. Context-Specific Independence in Bayesian Networks. In *Conference on Uncertainty in Artificial Intelligence*, 1996.

C. Boutilier, T. Dean, and S. Hanks. Decision-Theoretic Planning: Structural Assumptions and Computational Leverage. *Journal of Artificial Intelligence Research*, 11: 1–94, 1999.

R. Brafman and M. Tennenholtz. R-MAX — A General Polynomial Time Algorithm for Near-Optimal Reinforcement Learning. *Journal of Machine Learning Research*, 3: 213–231, 2002.

Ö. Şimşek and A. Barto. Using Relative Novelty to Identify Useful Temporal Abstractions in Reinforcement Learning. In *International Conference on Machine Learning*, 2004.

T. Dean and K. Kanazawa. A Model for Reasoning about Persistence and Causation. *Computational Intelligence*, 5:142–150, 1990.

T. Dietterich. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *Journal of Artificial Intelligence Research*, 9:227–303, 2000.

C. Diuk, M. Littman, and A. Strehl. A Hierarchical Approach to Efficient Reinforcement Learning in Deterministic Domains. In *Conference on Autonomous Agents and Multi-Agent Systems*, 2006.

E. Feinberg and A. Schwartz. Constrained Markov Decision Models with Weighted Discounted Rewards. *Mathematics of Operations Research*, 20:302–320, 1995.

F. Fernández and M. Veloso. Reusing and Building a Policy Library. In *International Conference on Automated Planning and Scheduling*, 2006.

Z. Gabor, Z. Kalmar, and C. Szepesvari. Multi-Criteria Reinforcement Learning. In *International Conference on Machine Learning*, 1998.

M. Ghavamzadeh and S. Mahadevan. Continuous-Time Hierarchical Reinforcement Learning. In *International Conference on Machine Learning*, 2001.

M. Ghavamzadeh and S. Mahadevan. Learning to Communicate and Act Using Hierarchical Reinforcement Learning. In *Conference on Autonomous Agents and Multi-Agent Systems*, 2004.

C. Guestrin, D. Koller, and R. Parr. Multiagent Planning with Factored MDPs. In *Advances in Neural Information Processing Systems*, 2001.

C. Guestrin, D. Koller, C. Gearhart, and N. Kanodia. Generalizing Plans to New Environments in Relational MDPs. In *International Joint Conference on Artificial Intelligence*, 2003.

M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explorations*, 11, 2009.

B. Hengst. Discovering Hierarchy in Reinforcement Learning with HEXQ. In *International Conference on Machine Learning*, 2002.

B. Hengst. Safe State Abstraction and Discounting in Hierarchical Reinforcement Learning. Technical report, National ICT Australia, 2003.

T. Hinrichs and K. Forbus. Analogical Learning in a Turn-based Strategy Game. In *International Joint Conference on Artificial Intelligence*, 2007.

C. Hogg, U. Kuter, and H. Munoz-Avila. Learning Hierarchical Task Networks for Nondeterministic Planning Domains. In *International Joint Conference on Artificial Intelligence*, 2009.

G. Iba. A Heuristic Approach to the Discovery of Macro-Operators. *Machine Learning*, 3:285–317, 1989.

A. Jonsson and A. Barto. Causal Graph Based Decomposition of Factored MDPs. *Journal of Machine Learning Research*, 7:2259–2301, 2006.

L. Kaelbling. Hierarchical Reinforcement Learning: Preliminary Results. In *International Conference on Machine Learning*, 1993.

L. Kaelbling, M. Littman, and A. Cassandra. Planning and Acting in Partially Observable Stochastic Domains. *Artificial Intelligence*, 101:99–134, 1998.

S. Kambhampati and J. Hendler. Flexible Reuse of Plans via Annotation and Verification. In *International Conference on Artificial Intelligence for Applications*, 1989.

M. Kearns and L. Valiant. Cryptographic Limitations on Learning Boolean Formulae and Finite Automata. In *Annual ACM Symposium on Theory of Computing*, 1989.

G. Konidaris and A. Barto. Autonomous Shaping: Knowledge Transfer in Reinforcement Learning. In *International Conference on Machine Learning*, 2006.

P. Langley and D. Choi. Learning Recursive Control Programs from Problem Solving. *Journal of Machine Learning Research*, 7:493–518, 2006.

L. Li. *A Unifying Framework for Computational Reinforcement Learning Theory*. PhD thesis, Rutgers University, 2009.

L. Li, M. Littman, and T. Walsh. Knows What It Knows: A Framework for Self-Aware Learning. In *International Conference on Machine Learning*, 2008.

N. Littlestone. *Mistake Bounds and Logarithmic Linear-Threshold Learning Algorithms*. PhD thesis, U.C. Santa Cruz, 1989.

Y. Liu and P. Stone. Value-Function-Based Transfer for Reinforcement Learning using Structure Mapping. In *National Conference on Artificial Intelligence*, 2006.

R. Makar, S. Mahadevan, and M. Ghavamzadeh. Hierarchical Multi-agent Reinforcement Learning. In *Conference on Autonomous Agents and Multi-Agent Systems*, 2001.

B. Marthi, S. Russell, D. Latham, and C. Guestrin. Concurrent Hierarchical Reinforcement Learning. In *International Joint Conference on Artificial Intelligence*, 2005.

B. Marthi, S. Russell, and D. Andre. A Compact, Hierarchically Optimal Q-function Decomposition. In *Conference on Uncertainty in Artificial Intelligence*, 2006.

B. Marthi, S. Russell, and J. Wolfe. Angelic Semantics for High-Level Actions. In *International Conference on Planning and Scheduling*, 2007.

M. Mataric. Reinforcement Learning in the Multi-Robot Domain. *Autonomous Robots*, 4:73–83, 1997.

Mausam and D. Weld. Solving Relational MDPs with First-Order Machine Learning. In *International Conference on Automated Planning and Scheduling Workshop on Planning under Uncertainty and Incomplete Information*, 2003.

A. McGovern and A. Barto. Automatic Discovery of Subgoals in Reinforcement Learning using Diverse Density. In *International Conference on Machine Learning*, 2001.

I. Menache, S. Mannor, and N. Shimkin. Q-Cut – Dynamic Discovery of Sub-Goals in Reinforcement Learning. In *European Conference on Machine Learning*, 2001.

N. Meuleau, M. Hauskrecht, K. Kim, L. Peshkin, L. Kaelbling, T. Dean, and C. Boutilier. Solving Very Large Weakly Coupled Markov Decision Processes. In *National Conference on Artificial Intelligence*, 1998.

S. Minton. *Learning Search Control Knowledge: An Explanation-Based Approach*. Kluwer Academic Publishers, 1988.

B. K. Natarajan. On Learning Boolean Functions. In *Annual ACM Symposium on Theory of Computing*, 1987.

S. Natarajan and P. Tadepalli. Dynamic Preferences in Multi-Criteria Reinforcement Learning. In *International Conference on Machine Learning*, 2005.

D. Nau, T. Au, O. Ilghami, U. Kuter, J. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, 20:929–935, 2003.

N. Nejati, P. Langley, and T. Konik. Learning Hierarchical Task Networks by Observation. In *International Conference on Machine Learning*, 2006.

R. Parr. *Hierarchical Control and Learning for Markov Decision Processes*. PhD thesis, University of California at Berkeley, 1998a.

R. Parr. Flexible Decomposition Algorithms for Weakly Coupled Markov Decision Problems. In *Conference on Uncertainty in Artificial Intelligence*, 1998b.

M. Pickett and A. Barto. PolicyBlocks: An Algorithm for Creating Useful Macro-Actions in Reinforcement Learning. In *International Conference on Machine Learning*, 2002.

B. Price and C. Boutilier. Accelerating Reinforcement Learning Through Implicit Imitation. *Journal of Artificial Intelligence Research*, 19:569–629, 2003.

C. Reddy and P. Tadepalli. Learning Goal Decomposition Rules using Exercises. In *International Conference on Machine Learning*, 1997.

S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.

S. Russell and A. Zimdars. Q-Decomposition for Reinforcement Learning Agents. In *International Conference on Machine Learning*, 2003.

M. Ryan. Using Abstract Models of Behaviors to Automatically Generate Reinforcement Learning Hierarchies. In *International Conference on Machine Learning*, 2002.

A. Schwartz. A Reinforcement Learning Method for Maximizing Undiscounted Rewards. In *International Conference on Machine Learning*, 1993.

S. Seri and P. Tadepalli. Model-based Hierarchical Average Reward Reinforcement Learning. In *International Conference on Machine Learning*, 2002.

R. Sutton and A. Barto. *Reinforcement Learning.* The MIT Press, 1998.

R. Sutton, D. Precup, and S. Singh. Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artificial Intelligence*, 112(1-2): 181–211, 1999.

P. Tadepalli and T. Dietterich. Hierarchical Explanation-Based Reinforcement Learning. In *International Conference on Machine Learning*, 1997.

P. Tadepalli and D. Ok. Model-based Average Reward Reinforcement Learning. *Artificial Intelligence*, 100:177–224, 1998.

M. Tan. Multi-Agent Reinforcement Learning: Independent versus Cooperative Agents. In *International Conference on Machine Learning*, 1993.

M. Taylor and P. Stone. Transfer Learning for Reinforcement Learning Domains: A Survey. *Journal of Machine Learning Research*, 10:1633–1685, 2009.

M. Taylor, P. Stone, and Y. Liu. Value Functions for RL-Based Behavior Transfer: A Comparative Study. In *National Conference on Artificial Intelligence*, 2005.

S. Thrun and A. Schwartz. Finding Structure in Reinforcement Learning. In *Advances in Neural Information Processing Systems*, 1995.

L. Torrey, J. Shavlik, T. Walker, and R. Maclin. Relational Macros for Transfer in Reinforcement Learning. In *Conference on Inductive Logic Programming*, 2007.

T. Walsh and M. Littman. Efficient Learning of Action Schemas and Web-Service Descriptions. In *National Conference on Artificial Intelligence*, 2008.

C. Wang, S. Joshi, and R. Khardon. First Order Decision Diagrams for Relational MDPs. *Journal of Artificial Intelligence Research*, 31:431–472, 2008.

C. Watkins. *Learning from Delayed Rewards.* PhD thesis, King's College, Cambridge, 1989.

J. Weeks. *The Shape of Space: How to Visualize Surfaces and Three-dimensional Manifolds.* Routledge, 1985.

D. White. Multi-Objective Infinite-horizon Discounted Markov Decision Processes. *Journal of Mathematical Analysis and Applications*, 89:639–647, 1982.

M. Wynkoop and T. Dietterich. Learning MDP Action Models Via Discrete Mixture Trees. *Lecture Notes in Computer Science*, 5212:597–612, 2008.

S. Yamada and S. Tsuji. Selective Learning of Macro-Operators with Perfect Causality. In *International Joint Conference for Artificial Intelligence*, 1989.

APPENDICES

Appendix A: Space of Hierarchical Structures

The discovery of hierarchical structure can be framed as a search problem in the space of hierarchical structures. A loose upper bound for the *size* of a task hierarchy $H = \{T_i : 0 \leq i < m\}$ is $O(|H|) = \sum_{i=1}^{m} |S_{\Phi_i} - G_i||C_i|$, where S_{Φ_i} is the state space induced by the projection function Φ_i, G_i is the set of goal states, and C_i is the set of child tasks of T_i. This bound is based on the total number of Q values within H, which impacts the sample complexity during learning. $|H|$ can also be quantified based on the size of the induced space of hierarchical policies Π_H that has a loose upper bound of $O(|\Pi_H|) = \prod_{i=1}^{m} |C_i|^{|S_{\Phi_i} - G_i|}$. If H_0 represents the shallowest hierarchy, and $\pi^*_{H_0} \in \Pi_{H_0}$ is the optimal policy, then the objective of hierarchical structure discovery is to induce the most compact hierarchy H such that $|H| = \min_i |H_i|$ and $\pi^*_{H_0} \in \Pi_H$. Finding this global minimum exhaustively will be intractable, because an infinite number of possible composite tasks can be constructed (the termination condition is an arbitrary logical expression) and the space of unconstrained hierarchical structures is infinite. Instead, we seek to map out the space of constrained hierarchical structure.

The *depth* of a hierarchy is the length of the longest path from the root to a primitive leaf. The most elementary hierarchical structure is the composite root task with one primitive child, and has a depth of 1. We assume that if a composite task has a single child, then the child must be primitive; otherwise, the two composite tasks can be merged into one without changing the semantics of the hierarchy. Also, if a task T has a set of child tasks C, where $|C| > 1$, then every primitive task $c_i \in C$ can be *wrapped* by inserting a composite task T_i between T and c_i, allowing the hierarchy to impose admissibility conditions on primitive actions.

A.1 Tree-Structured Hierarchies

The number of non-empty partitions of a set of size n is $B_n = \sum_{k=0}^{n-1} \binom{n-1}{k} B_k$ where $B_0 = 1$. This Bell number is equal to the number of tree-structured hierarchies of depth ≤ 2 incorporating exactly n actions and ignoring all wrapping permutations. The number of tree-structured hierarchies of unconstrained depth, incorporating exactly n actions, and inclusive of all wrapping permutations is

$$H_n = 2\sum_{k=1}^{n-2}\binom{n-1}{k}H_k H_{n-k} + 4H_{n-1}, \text{ where } H_1 = 1, H_2 = 4.$$

As a hierarchy could incorporate 1 through n actions, the total number of hierarchies for n actions is

$$H_n^* = \sum_{k=1}^{n}\binom{n}{k}H_k.$$

A.2 Restricted-Termination Hierarchies

An arbitrary task hierarchy is a directed acyclic graph (DAG). However, the number of nodes of this graph could be infinite. If the termination condition of the composite tasks, excluding the root, is restricted to a single literal of the form $x \odot \mathcal{D}(x)$, where x is a state variable, $\odot \in \{=, \neq, <, \leq, >, \geq\}$, and $\mathcal{D}(x)$ is the domain of x, then the number of composite tasks is $n = 6\sum_{x\in\mathcal{X}}|\mathcal{D}(x)| + 1 + |\mathcal{A}|$. The number of hierarchical structures of n composite tasks is upper-bounded by the number of labeled DAGs on n nodes,

$$G_n = \sum_{k=1}^{n}(-1)^{k-1}\binom{n}{k}2^{k(n-k)}G_{n-k}, \text{ where } G_0 = 1.$$

This is a loose upper bound, because not all DAGs represent legal hierarchies, including ones in which ① there is an incoming edge at the root, ② there is an outgoing edge at a primitive task, ③ there is no directed path from the root to any primitive task, and ④ a composite task connected to the root either has no children or a single composite child. Also, all permutations of subgraphs unreachable from the root can be ignored.

A.3 Trajectory-Consistent Hierarchies

A trajectory τ of length l can be split into 2^{l-1} contiguous subtrajectories, because a split either does or does not exist between any two actions. Every subtrajectory can then be recursively split further. The number of hierarchies of unconstrained depth that include all wrapping permutations and are consistent with τ is

$$J_l = 2\sum_{k=1}^{l-2} J_k J_{l-k} + 4J_{l-1}, \text{ where } J_1 = 1, J_2 = 4.$$

This is similar to the number of tree-structured hierarchies except that the order of the primitive actions is constrained by the trajectory.