Economic approaches to hierarchical reinforcement learning

A thesis presented

by

Erik G. Schultink

to

Computer Science

in partial fulfillment of the honors requirements

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

April 3, 2007

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many real-world problems can be easily broken down into sub-problems. Often we do not even recognize that many everyday tasks decompose in this way. Consider the task of grocery shopping, which can be broken down into separate sub-tasks of going to the store, purchasing the necessary groceries, and driving home. The component of driving can itself be broken down into many subtasks - accelerating, decelerating, turning left, and turning right. Each of these can be further decomposed into physical actions that we hardly even notice in our conscious thoughts. When we think of the task of 'grocery shopping', we are using an abstraction of a task that consists of a series of thousands of muscular movements. Without such an abstraction, the task would seem overwhelming. We are able to understand that how to drive to the store is not closely tied to how to perform the actions of shopping inside the store. Additionally, we are able to rely on our experience solving past instances of the grocery shopping task's components - walking, driving, etc - even if we had never completed the grocery shopping task itself. Such task decomposition and abstract reasoning are done implicitly throughout our daily lives.

Hierarchical reinforcement learning (HRL) is a formalization of the use of such abstractions. A human programmer provides a task hierarchy to a HRL algorithm, specifying how

to decompose a problem into sub-problems. Using this information, the HRL algorithm can exploit the structure of a problem domain to converge to a solution policy more quickly than traditional reinforcement learning (Parr and Russell, 1998; Dietterich, 2000a; Andre and Russell, 2002). In general, HRL can be thought of as decomposing a problem into sub-problems derived from a hierarchy and using an agent at to find a solution for each sub-problem. Each of these agents can be thought of as corresponding to a particular node in the hierarchy, and solving the sub-problem rooted at that node. By piecing together the solution policies to all the sub-problems in the hierarchy, HRL algorithms construct a solution policy for the entire problem.

Not all HRL algorithms converge to optimal policies. Some algorithms, such as MAXQQ learning (Dietterich, 2000a), reason only about the effect of an action on the local sub-problem. This leads to a policy that is *recursively optimal* (Dietterich, 2000a), as the solution for each sub-problem can be thought of as locally optimal given the solutions to the other sub-problems. Such a policy is not necessarily globally optimal, as it may be necessary for a sub-problem to do something that is slightly sub-optimal in order to save a lot of utility in another sub-problem. This is only true when a sub-problem has multiple goal states that can be reached from a given initial state, such as a variable level of gas in your car when you arrive the store. For example, if you are passing a gas station on your way to the store but have enough gas to reach the store, it is not optimal to stop if you are considering only the sub-problem of 'driving to the store'. However, from the global perspective, if you will not have enough gas to reach a gas station from the store when you try to drive home, then it is globally optimal to stop. By only considering the local effects of actions, MAXQQ and similar algorithms are unable to understand such trade-offs. Although it may yield a lower quality solution policy, ignoring the external effects of local actions allows for faster learning.

HRL algorithms that represent and learn the impact of local actions on the entire problem can converge to *hierarchically optimal* policies (Dietterich, 2000a; Andre and Russell, 2002;

Marthi et al., 2006). A hierarchically optimal policy is not necessarily globally optimal, as a poor hierarchy may impose constraints on the policy. We follow the common assumption of a hierarchy provided by a programmer and look to learn an optimal policy given this hierarchy, exploiting the information it contains to do so as rapidly as possible. There are several HRL algorithms that explicitly represent the effect of the local solution on the quality of the solution to the entire problem (Andre and Russell, 2002), and thus may converge to hierarchically optimal solution policies. In this paper, we propose an algorithm that uses transfers of reward between agents solving different sub-problems in the hierarchy to alter local behavior to produce hierarchically optimal policies. The goal of this approach is to align the interests of the agent solving the sub-problem with those of the agent solving global problem (commonly referred to as the `Root` agent).

We introduce Economic Hierarchical Q-learning (EHQ) with the desire to show that market structures can effectively align local and global interests. Such a system should be robust and able to be extended to loosely-coupled, distributed settings. As such, we take the economic approach of modeling each sub-problem as being learned by an independent, self-interested agent. In HRL, the `Root` agent begins with control of the world. It can then choose an action from among its children (immediate descendents) in the hierarchy. If the child selected is a primitive action, that action is taken by the parent in the world. If the child is a subtask (has its own descendents) than control is passed to that agent, returning control to the parent after it has completed its subtask. Within this context, we consider each parent agent to select an action that is optimal based on its beliefs about its expected reward for the action in the current state. For each child action, EHQ has multiple agents that implement the action and place bids in an auction to win control of the world from the parent. Each child's bid is based on its beliefs about the value of the world in its current state. The winning child agent gets control of the world, with the obligation to satisfy the predicate that specifies the action and then return control of the world to the parent agent. In return, the child is paid the amount it bid for the world by the parent. If there are

several possible exit-states that the child could return control of the world in, the parent that invokes the action can define subsidies over those exit-states. These subsidies provide additional compensation to the child implementing the action in return for it producing what the parent sees as a more valuable state of the world. We show empirically that, using exit-state subsidies, EHQ can converge to a hierarchically optimal solution policy.

The EHQ algorithm has several advantages over the MAXQQ (Dietterich, 2000a), ALispQ (Andre and Russell, 2002), and HOCQ (Marthi et al., 2006) HRL algorithms. EHQ's exit-state subsidies, derived automatically though the market mechanisms, intuitively provide the necessary incentives to induce a hierarchically optimal solution policy. By precisely identifying the trade-off between local and global solution quality, they clearly capture the relevant information from the world and internalize it within an agent's value function. EHQ agents interact in simple, highly structured economic transactions, which allows for greater decoupling of agents than MAXQQ or HOCQ, both of which use extensive sharing of information amongst parts of the problem. These transactions are not only conceptually simple, but provide a clear possibility of extending the market-based ideas of EHQ to distributed or multi-agents architectures. In such systems, the auction and subsidies in EHQ can force agents to reveal the private information that the other agents need to behave hierarchically optimally. MAXQQ and the other HRL algorithms discussed in this paper require that the agents 1) have access to the decomposed value functions of all agents below them in the hierarchy, and 2) observe the intrinsic reward that is accrued by the system when any agent below them takes a primitive action. EHQ avoids such complex and extensive interactions, but pays a cost by requiring that each agent learn more about the costs faced by the other agents, which degrades its learning speed in practice. EHQ's system also results in the replication of information and learning in comparison to MAXQQ, ALispQ, and HOCQ.

This paper makes the following contributions:

- Adapts the concept of an artificial economy to a hierarchical context
- Demonstrates how agents in this artificial economy can generate appropriate subsidies which, through the market system, incentivize agents solving other sub-problems to produce better solutions
- Proposes a hierarchical reinforcement learning algorithm called Economic Hierarchical Q-Learning (EHQ) based upon this concept of a hierarchical artificial economy
- Demonstrates an innovative use of auctions in the economy to provide exploration amongst agents with different beliefs
- Provides a proof that a simplified version of the EHQ algorithm called rEHQ will converge to a recursively optimal policy
- Shows the empirical results that EHQ's simple system of subsidies, generated by the market system, can lead it to converge to a hierarhically optimal policy
- Contrasts the solution quality yielded by EHQ to the recursively optimal policy found by MAXQQ, and explains the underlying tension within HRL between hierarchical and recursive optimality in this regard
- Identifies the key difficulties in proving convergence to hierarchically optimal policies for algorithms such as EHQ, ALispQ, or HOCQ

1.1 Motivating Example Domains

To illuminate discussion throughout this paper, we provide two motivating examples that will help us to explain particular aspects of HRL, state abstraction, and related algorithms. The first is the Taxi domain, which closely replicates the domain used in Dietterich (2000a).

The Taxi domain takes place in a grid world as depicted in Figure 1.1. There are four locations, labeled R, B, G, and Y, where the passenger may appear. The goal is to pick the

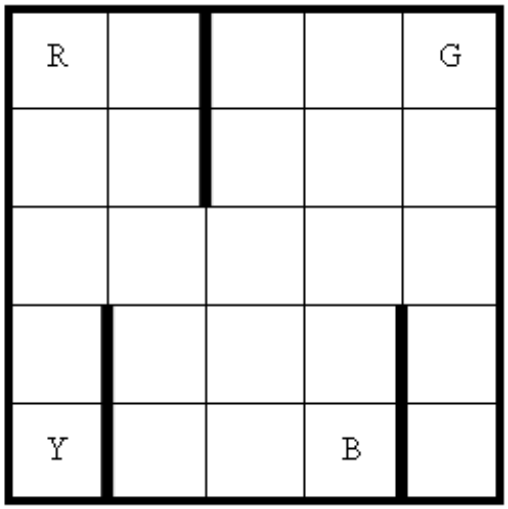

Figure 1.1: Grid world for the Taxi domain, based closely on the problem domain from Dietterich (2000a).

passenger up with the taxi and then drop the passenger off at the destination (also one of R, B, G, and Y). Equivalently to Dietterich (2000a), the taxi must pickup and drop-off the passenger to complete the episode, even if the passenger began in the destination. There are 25 possible locations for the taxi, 5 locations for the passenger, and 4 possible destinations for a total of 500 possible states in the world. There are 4 primitive actions for movement - `north`, `east`, `south`, and `west` - each of with a reward of -1. If movement in the specified direction would cause the taxi to run into a wall (indicated with bold), the action is a no-op and the reward is -10. There is also a `pickup` action, which puts the passenger in the taxi if executed when the passenger and the taxi are in the same marked cell. Otherwise, the `pickup` action is a no-op. Similarly, there is a `drop-off` primitive action which, if executed when the taxi is in one of the marked states and the passenger is in the taxi, removes the passenger from the taxi, leaving the passenger in the marked state that the taxi is in. When the taxi is not in a marked state, the `drop-off` action is a no-op. Both the `pickup` and `drop-off` primitives have a cost of -1. There is a reward of 100 for executing the `drop-off`

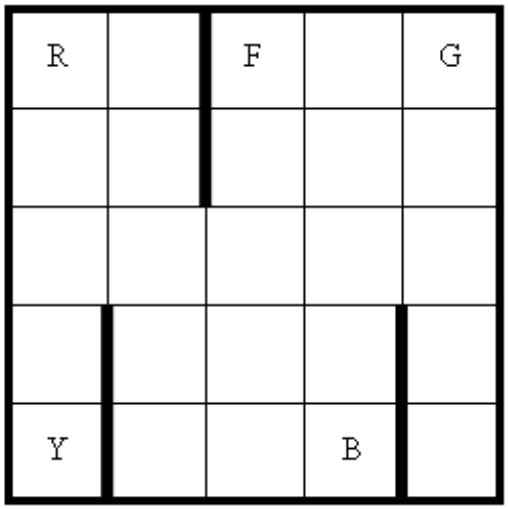

Figure 1.2: The grid world for TaxiFuel, a variation on the Taxi domain.

primitive when the passenger is in the taxi and the taxi is in the correct destination state.

In the episodic setting, executing the `drop-off` primitive such that there is a reward of 100 leads to a terminal absorbing state with probability 1. Each episode begins with the taxi positioned in a random location in the grid-world and a passenger at a random marked state waiting to be taken to a random destination. This setting is used for most of this paper's discussion.

This paper will also discussion the TaxiFuel domain, a variation on the Taxi domain that adds a fuel constraint. Each movement primitive decreases the taxi's fuel by 1, unless the taxi has 0 fuel remaining, in which case there is a reward of -10. The taxi can be refueled to its maximum capacity of 10 if the `fill-up` primitive is executed in location labeled F in Fig. 2. This variation increases the state space size to 5500 states and has distinct recursively optimal and hierarchically optimal solutions.

1.2 Outline

This paper focuses on the tension between hierarchically and recursive optimality, and the impact of hierarchies, value decomposition, and state abstraction on the convergence of HRL algorithms. Our goal is to use market structures to propogate information through our hiearchical artificial economy such that the system will converge to a hierarchically optimal policy. We ultimately develop the EHQ algorithm as such an artificial economy, which is capable of leveraging the same level of state abstraction as other hierarchically optimal HRL algorithms. We begin by provding background on Markov Decision Problems (MDPs), Semi-Markov Decision Problems (SMDPs), hierarchies, hierarchical policies, value decomposition, and state abstraction in Chapter 2. Chapter 3 discusses several known HRL algorithms, highlighting their particular convergence properties and concluding that all of these algorithms must reason about the value of exit-states to converge to hierarchically optimal solutions. In Chapter 4, we lay out the structure for a basic hierarchical artificial economy, based on Hayek (Baum and Durdanovich, 1998), and formalize this as the rEHQ algorithm. We prove the convergence of rEHQ to a recursively optimal policy. We then introduce an approach for determining the value of exit-states, introducing these values as subsidies in the artificial economy and discuss its economic motivation. This modification provides the Economic Hierarchical Q-Learning (EHQ) algorithm, a robust, decentralized system that will be shown to converge to hierarchically optimal policies in Chapter 5. In Chapter 5, we present empirical results comparing the convergence performance of traditional flat-Q learning, MAXQQ-learning, and our EHQ algorithm. In Chapter 6, we conclude by contrasting our approach to the other known HRL algorithms and discussing potential extensions to the EHQ algorithm.

Chapter 2

Background and formal definitions

The formal definitions used in this paper of MDPs, SMDPs, hierarchies, and hierarchical policies are largely based on Dietterich (2000a). This chapter presents those definitions, as well as briefly introduces Q-learning, value decomposition, and state abstraction. These concepts form the basis of the discussion of HRL algorithms presented in chapter 3.

2.1 Markov Decision Problems

Like other work in HRL, we model the world with which our system interacts as a fully-observable stochastic Markov Decision Problem (MDP). The following are standard definitions for MDP settings.

We define a MDP M as a 5-tuple, $M = \langle S, A, P, R, P_0 \rangle$, with:

- S : set of possible states of the environment. Each state $s \in S$ is a vector specifying a complete assignment of state variables and is fully observable by the agent.
- A : finite set of primitive actions.
- P : probabilistic transition function for a current state s and chosen action a, defining

a transition $s \rightarrow s'$ according to a probability distribution $P(s'|s,a)$.

- R : real-valued stochastic reward function, yielding a reward $r = R(s'|s,s)$. We consider only finite rewards.
- P_0 : probability distribution of initial states. The MDP begins in state s with probability $P_0(s)$.

By definition, $P(s'|s,a)$, $R(s'|s,s)$, and $P_0(s)$ must be stationary distributions.

A policy π for the MDP M is a mapping from states in S to actions in A.

The primary setting we consider is episodic MDPs, with a discount factor γ. To conform with the settings of Dietterich (2000a), an episode MDP must have at least one zero-cost absorbing terminal state. Like Dietterich (2000a), we consider only problems where all deterministic policies have some non-zero probability of reaching a terminal state when started in an arbitrary state. The goal of the agent in the episodic setting is to maximize expected cumulative reward.

The second setting we consider is infinite time horizon, with some discount factor $\gamma < 1$. In this setting, the agent's goal is to maximize expected discounted sum of future reward. For both the episodic and infinite horizon settings, the goal of learning is to find a policy π that maximizes the following value function, where E denotes the expectation over the possibly stochastic transition and reward functions:

$$V^{\pi}(s) = E\left\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t = t, \pi\right\}$$

A value function V^{π} defines the expected value for the state s following a policy π. In this equation, the random variable r_t represents the reward at time t following policy π from s. For most of this paper, we consider the episodic setting with $\gamma = 1$. In this case,

the value function can be expressed as follows:

$$V^{\pi}(s) = E\{r_t + r_{t+1} + r_{t+2} + \ldots | s_t = t, \pi\}$$

Note that in the infinite horizon setting with $\gamma = 1$, V will be an infinite sum that does not converge. By convention, we take $\gamma < 1$ in the infinite horizon setting to avoid such a case.

The value function V^{π} satisfies the Bellman equation for a fixed policy:

$$V^{\pi}(s) = \sum_{s'} P\left(s'|s,a\right)\left[R\left(s'|s,\pi\left(s\right)\right) + \gamma V^{\pi}\left(s'\right)\right]$$

This equation states that the value of state s when following policy π is equal to the sum over all possible successor states s' of the reward for s' from taking the action a when the world is in s, plus the discounted value of that s', weighted by the probability of that s' given s and $\pi(s)$. Note that $\pi(s)$ denotes the action specified by the policy π for the state s. Bellman (1957) proved that the optimal value function, denoted V^*, which simultaneously maximizes the expected cumulative reward in all states $s \in S$, is the unique solution to the Bellman equation:

$$V^{\pi}(s) = \max_{a'} \sum_{s'} P\left(s'|s,a\right)\left[R\left(s'|s,\pi\left(s\right)\right) + \gamma V^{\pi}\left(s'\right)\right]$$

Any policy which maps all s to the corresponding s that maximizes the right-hand side of the Bellman equation is an optimal policy denoted π^*. There may be many such policies.

To properly formalize the notion of HRL, it is necessary to introduce a generalized form of a Markov Decision Process, called a Semi-Markov Decision Process (SMDP). In a SMDP, actions may take a variable number of timesteps to complete. (For the purposes of this paper, we consider time to be discrete). This generalization requires us to modify

the definitions of P and R. It also complicates our conception of the value functions above for $\gamma \neq 1$. Let a random variable N represent the number of timesteps an action takes to complete. We extend the transition function to be $P(s', N|s, a)$, a joint distribution on s' and N given the current state s and a chosen action a. Similarly, we extend the reward function to be $R(s', N|s, a)$, again a joint distribution on s' and N given the current state s and a chosen action a. This formalization is taken from Dietterich (2000a). He notes that it differs slightly from standard formalizations of SMDPs, which define a separate cumalitive distribution function.

In accordance with this formalization of a SMDP, the value function for a fixed policy π is:

$$V^{\pi}(s) = \sum_{s',N} P\left(s', N|s, \pi(s)\right)\left[R\left(s', N|s, \pi(s)\right) + \gamma^{N} V^{\pi}\left(s'\right)\right]$$

The expected value must be taken over both the successor states and action durations, represented by s' and N, respectively. The discount factor is raised to the power of N to discount the value of the successor state by duration of the action.

In the episodic setting with $\gamma = 1$, the SMDP value function for a fixed policy is equivalent to the value function for an MDP.

2.2 Q-learning

The best known reinforcement learning algorithm is Q-learning. All the HRL algorithms discussed in this paper are based on the same principles as Q-learning. Q-learning is an approach to learning an action-value function, or Q-function, denoted $Q^{\pi}(s, a)$. The Q-function represents the expected cumulative reward of performing action a in state s and then following policy π thereafter for a given MDP. The Q-function satisfies a Bellman equation:

$$Q^{\pi}(s, a) = \sum_{s'} P\left(s'|s, a\right)\left[R\left(s'|s, a\right) + \gamma Q^{\pi}\left(s', \pi\left(s'\right)\right)\right]$$

Accordingly, the optimal Q-function, denoted $Q^*(s,a)$, satisfies the equation:

$$Q^*(s,a) = \sum_{s'} P\left(s'|s,a\right)\left[R\left(s'|s,a\right) + \gamma \max_{a'} Q^*\left(s',a'\right)\right]$$

Any policy that is greedy with respect to Q^* by mapping $s \rightarrow a$ such that $Q^*(s,a) = \max_{a'} Q^*(s,a')$ is satisfied is an optimal policy for an MDP M. There may be many policies that satisfy this condition and are therefore optimal for , differing only in how ties are broken among actions.

The traditional flat Q-learning algorithm stores a Q-value table, with an entry for every state-action pair. To learn the optimal Q-function, the algorithm must observe the current state of the world s, the action a (chosen according to an exploration policy), the received reward r, and the successor state s'. Based on these values, the algorithm must make the following update:

$$Q_t(s,a) := (1-\alpha_t)\, Q_{t-1}(s,a) + \alpha\left[r + \gamma \max_{a'} Q_{t-1}\left(s',a'\right)\right]$$

The first term in this update represents the agent's prior belief of $Q(s,a)$, the expected cumulative reward of taking action a in state s and following the believed optimal policy thereafter. The second term is the reward for the current observation plus the discounted expected cumulative reward for the following the believed optimal reward in the successor state s'. The parameter α_t is the learning rate, which, roughly speaking, controls the tradeoff between the prior experience and the current observation, as well as how rapidly reward can propagate upstream. It is a value between 0 and 1 that is updated at each time step such that $\lim_{T\rightarrow\infty} \alpha_t = 0$. Typically, this learning rate must be tuned for efficient learning in any particular MDP domain.

In general, this traditional model of flat Q-learning is inefficient in practice. This is at least partly due to the update rule only backing up reward one step per update. What

is appealing about Q-learning is its simplicity and proof of convergence. Bertsekas and Tsiksiklis (1996) prove that Q-learning will converge to the optimal action-value function with probability 1 under certain conditions. These conditions require that the Q-learning agent use an exploration policy that is greedy in the limit of infinite exploration (GLIE). Formally, a GLIE policy is defined as follows:

Definition 2.2.1. (Dietterich, 2000a) A GLIE (greedy in the limit of infinite exploration) policy is any policy is any policy satisfying

1. Each action is executed infinitely often in every state that is visited infinitely often.
2. In the limit, the policy is greedy with respect to the Q-value function with probability 1.

Additionally, the learning rate parameter must meet the following conditions:

$$\lim_{T\to\infty}\sum_{t=1}^{T}\alpha_t = \infty \quad and \quad \lim_{T\to\infty}\sum_{t=1}^{T}\alpha_t^2 < \infty$$

Q-learning is a *model-free* approach to learning, as it does not attempt to explicitly model a MDP by learning the transition and reward functions, but rather learns an action-value function. Q-learning is an *on-policy* learning algorithm, as it updates assume that a greedy policy based on its current values of $Q(s,a)$ is an optimal policy. Q-learning has become a common standard of comparison in reinforcement learning literature. In particular, many authors look to provide a proof of convergence for their algorithms under the same conditions as Q-learning.

2.3 Hierarchies

In Hierarchical Reinforcement Learning (HRL), hierarchies are used to provide structure and information about a domain, ultimately permitting both temporal abstractions and

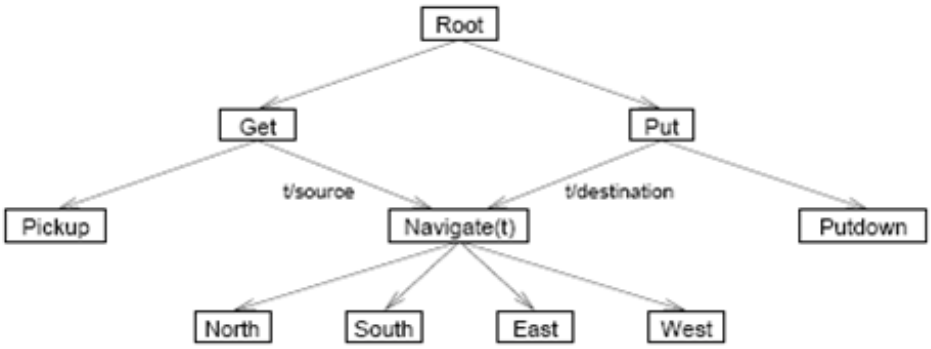

Figure 2.1: An example hierarchy from Dietterich (2000b).

state abstractions.

The hierarchy shown in Figure 2.1 depicts a possible for the Taxi domain described in chapter 1. The hierarchy decomposes the `Root` task into `Get` and `Put` tasks, which are further broken down into `pickup` and `Navigate(source)`, and `putdown` and `Navigate(destination)`, respectively.

Formally, we represent hierarchies as directed acyclic graphs that decompose an MDP M into a set of subtasks $\{M_0, M_1, ..., M_n\}$. For consistency with Dietterich (2000a), who our notation closely follows, we use the convention that M_0 is the `Root` subtask. All nodes in the hierarchy that have descendants represent subtasks. All nodes without descendents represent primitive actions in A. The nodes labeled `pickup` and `north` are examples of primitive actions shown in 2.1.

Definition 2.3.1. (Dietterich, 2000a) A *subtask* M_i is a 3-tuple, $\langle T_i, A_i, R_i \rangle$ defined as follows:

- T_i : termination predicate partitioning S into a set of active states S_i and a set of exit-states E_i.
- A_i : set of actions that can be performed to achieve M_i. In the graph representation

of a hierarchy, the set A_i contains all the immediate descendents of the node corresponding to M_i. As such, A_i may contain both primitive actions from A as well as other subtasks. The actions available for the subtask M_i may depend on the state, so the set of actions can be expressed as a function $A_i(s_i)$.

- R_i : a *pseudo-reward function* that can be defined by a programmer to provide a pseudo-reward for transitions from a state $s \in S_i$ to an exit state $e \in E_i$.

Like Dietterich, we allow for parameterized subtasks. The `Navigate(t)` subtask shown in 2.1 is an example of a parameterized subtask, with possible bindings of the formal parameter t to source and destination. Typically, parameterized subtasks are simply a notational convenience. In practice, all possible bindings of the parameters to actual values are treated as distinct subtasks, though there is often considerable opportunity to reuse common information between them Dietterich (2000a). For parameterized subtasks, we can extend the termination predicate and the pseudo-reward function to depend on the actual binding b, as $T_i(s, b)$ and $R_i(s'|s, a, b)$. Generally, the bindings will be omitted from the notation, as we could similar define distinct termination predicates and pseudo-reward functions for each possible binding since we treat each binding as specifying a distinct subtask. Primitive actions in the hierarchy can be considered to be subtasks that execute immediately after executing the action with no pseudo-reward.

Throughout this paper, we will also use the terms *macroaction* and *subroutine* to refer to a subtask, as they provide useful intuition in many contexts. For instance, any non-primitive subtask M_i can be modeled formally as a SMDP with its action space A_i consisting of both primitive actions and macroactions. This is a form of temporal abstraction (Dietterich, 2000c). The term macroaction in this context is useful, as it emphasizes that, from the perspective of a parent node, passing control to a particular subtask M_j represents a monotonic, stochastic transition to a successor state s', yielding reward and potentially taking an undetermined number of time steps. The actual sequence of primitive actions that are

executed, as well as the intermediate states of the world, can be ignored by the agent solving M_i; it needs only to reason about the resulting state, the reward, and the number of time steps that elapse.

2.4 Hierarchical policies

The notion of a policy can be extended for a MDP M that has been decomposed into a set of subtasks. Local solution to these subtasks can be combined to create a policy for the entire MDP (Dean and Lin, 1995).

Definition 2.4.1. (Dietterich, 2000a) A *hierarchical policy*, π, is a set containing a policy for each of the subtasks that compose the problem: $\pi = \{\pi_0, \pi_1, ..., \pi_n\}$. Each π_i is a mapping from a state s to either a primitive action a or a policy π_j, where π_j is the policy for an subtask M_j that is a child of the node corresponding to M_i in the hierarchy.

In conceptualizing the execution of a hierarchical policy, each policy π_i can be considered to be a particular subroutine. In this context, the hierarchy can be thought of as a recursive call graph. Initially, control is given to policy π_0, which controls the `Root` subtask. This policy maps each state in the `Root` SMDP's state space to either a primitive action a or a policy π_j for one of the `Root` node's child SMDPs. Formally, $\pi_0(s) = a$ or $\pi_0(s) = \pi_j$. While M_0 is not terminated, simply execute $\pi_0(s)$. The following pseudo-code generalizes the execution of any hierarchical policy:

```
EXECUTE(policy π for subtask i)
while T_i is not satisfied do
   s := the current state of the world
   if π(s) is a primitive action then
      execute the action π(s) in the world
   else
      EXECUTE(π(s))
   end if
end while
```

The execution of a hierarchical policy can be thought of passing control between subroutines, as constrained by the call graph of the hierarchy. This notion is compatible with the idea of passing control amongst agents, where each agent is responsible for solving the SMDP as seen from its particular perspective in the hierarchy.

2.5 Q-value decomposition

The two known forms of Q-value decomposition are the MAXQ value decomposition (Dietterich, 2000a) and the ALisp value decomposition (Andre and Russell, 2002). The ALisp decomposition decomposes the value $Q(s,a)$ into the sum of the following three terms:

- $Q_V(s,a)$: the expected sum of reward while a is executing. Note that when a is a macroaction (rather than a primitive), $Q_V(s,a)$ need not be represented explicitly but can be recursively decomposed into $\max_{a' \in A_j} [Q_V(s,a') + Q_C(j,s,a)]$ where j is the index for subtask M_j that implements a.
- $Q_C(i,s,a)$: the expected sum of rewards after a completes, until the current subtask M_i completes
- $Q_E(i,s,a)$: the expected sum of rewards after the current subtask M_i exits after taking the action a in state s

For a non-primitive action a, the MAXQ decomposition (Dietterich, 2000a) uses only the $Q_V(s,a)$ and $Q_C(i,s,a)$ terms, ignoring $Q_E(i,s,a)$, to approximate $Q(s,a)$. This amounts to ignoring the impact of the current subtask's policy on the rest of the subtasks in the hierarchy. When there is only one possible exit-state for the sub-task given the state that it is called in, $Q_E(i,s,a)$ can be safely ignored because it will be the same for all state-action pairs within a given subtask. Because it omits $Q_E(i,s,a)$, the MAXQ decomposition cannot represent hierarchically optimal value functions in general, although it can represent any recursively optimal value function (see 3.1). The ALisp decomposition, in contrast, is

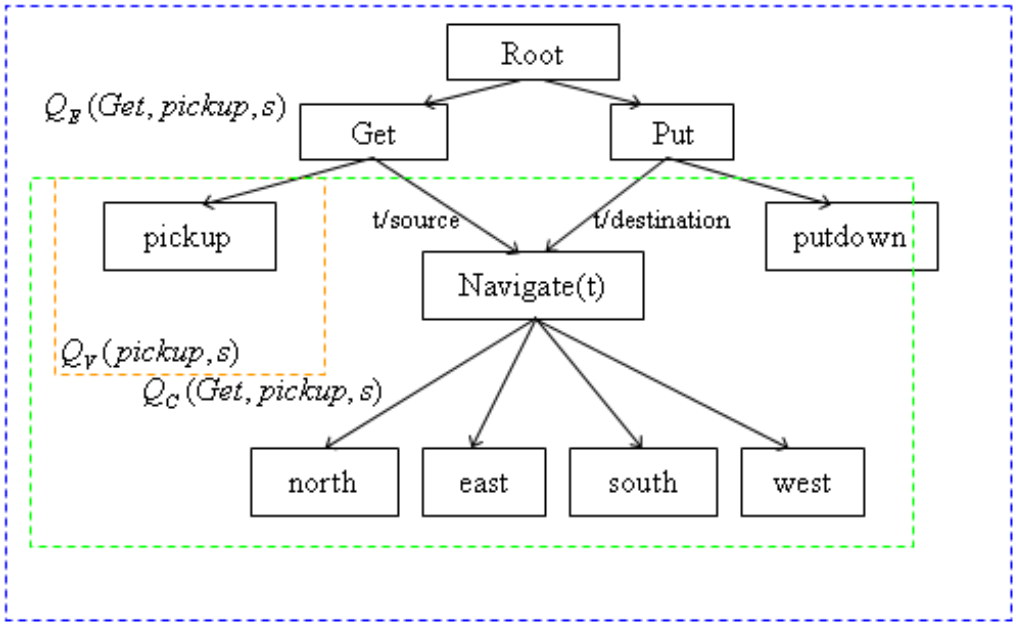

Figure 2.2: Depiction of value decomposition in the Taxi domain.

proven to be able to represent any hierarchically optimal policy. While ALisp may express a better quality policy as a result, it requires many more values to be explicitly stored.

Note that Dietterich uses the notation of $V(s,a)$ for $Q_V(s,a)$ and $C(i,s,a)$ for $Q_C(i,s,a)$. The notation used in this paper is more similar to the ALisp papers, which use $Q_r(s,a)$, $Q_c(i,s,a)$ and $Q_e(i,s,a)$.

The recursive decomposition of $Q(s,a)$ is an important aspect of value decomposition, as it decreases the number of values that must be explicitly represented. For emphasis, this decomposition is depicted in Figure 2.2 for the Taxi domain. Recursive decomposition requires a given node at the hierarchy to access the Q-tables for all its descendants in the hierarchy. As will be shown in Chapter 4, this is a troubling necessity if we consider each node in the hierarchy as being represented by independent, possibly distributed agents.

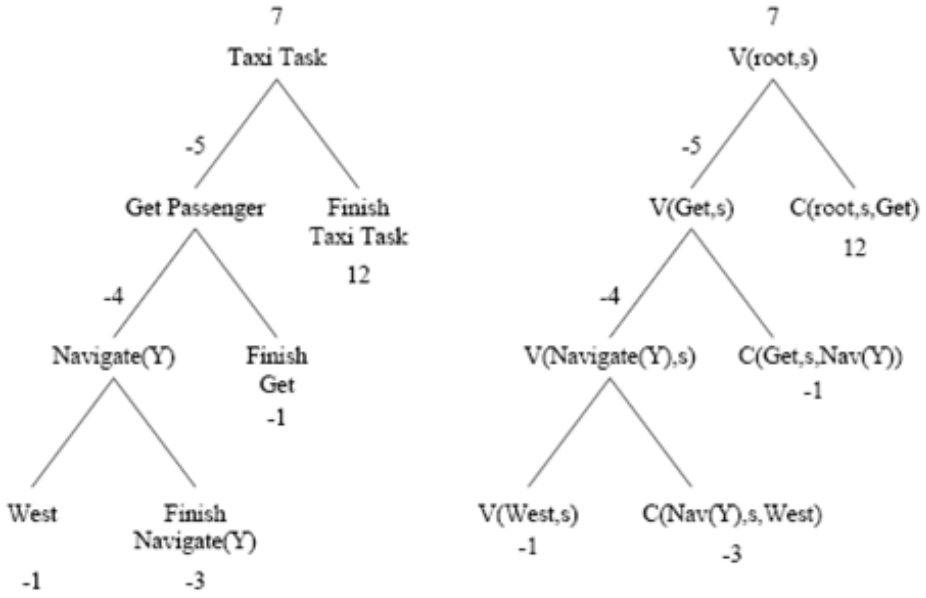

Figure 2.3: Depiction of the recursive decomposition of a value function, as used by the MAXQ decomposition (Dietterich, 2000a), based on the hierarchy given in Figure 2.1

2.6 State abstraction

Aside from the advantages of providing temporal abstractions, hierarchies can improve learning speed by allowing for state abstractions. The intuition for the usefulness of state abstractions is that not all state variables directly effect the value of a given state-action pair. For example, what you had for breakfast is not relevant to the decision of turning left or right at a particular intersection when you are driving to work. This notion can be formalized as a type of abstraction referred to as *irrelevant variables* (Dietterich, 2000a). Specifying irrelevant variables is one way to define an *abstract state*. In general, an abstract state is a region (or a set of regions) of the state space of a MDP (Dietterich, 2000a). The term *aggregate state* can also be used to describe this concept, as an abstract state represents a set of underlying states of the MDP.

Definition 2.6.1. An *abstract state* or *aggregate state* is a subset of the state space S of a

MDP M.

State abstraction is clearly very useful, as it allows for more succinct expression of policies and value functions (Dietterich, 2000c). While human agents intuitively use state abstractions in daily life, it is necessary to formalize how they can be used in HRL.

Definition 2.6.2. (Andre and Russell, 2002) State abstractions are *safe* if optimal solutions in the abstract space are also optimal in the original space.

Aside from irrelevant variables, Dietterich (2000a) also defines the notion of *funnel abstraction*. A funnel abstraction can be defined for any action which take a large set of input states and maps them into a small set of resultant states. An example is the `Navigate(source)` task from the Taxi domain, which takes the world from any state to a state where the taxi is in the source location. In practice, this allows an algorithm to using state abstraction to only store a small set of Q_C values for `Navigate(source)`.

Dietterich lays out five formal conditions for safe state abstraction in the MAXQ value decomposition (Dietterich, 2000c), which are included below. The first two are formal ways to define irrelevant variables, while the last three define funnel abstractions.

1. Subtask Irrelevance: For subtask M_i, a set of state variables Y is irrelevant to the subask if the state variables of can be partitioned into two sets X and Y such that for any stationary abstract hierarchical policy π executed for M_i and its descendents, the following two properties hold:

   (a) the state transition probability distribution $P^\pi(s', N|s, j)$ for each child action j of M_i can be factored into the product of two distributions:

   $$P^\pi\left(s', y', N|x, y, j\right) = P^\pi\left(x', N|x, j\right) \cdot P^\pi\left(y'|y, j\right)$$

   where x and x' give values for the variables in X, and y and y' give values for the variables in Y

(b) for any pair of states $s_1 = (x, y_1)$ and $s_2 = (x, y_2)$, and any child action j, $V^\pi(j, s_1) = V^\pi(j, s_2)$

2. Leaf Irrelevance: A set of state variables Y is irrelevant for a primitive action a if for any pair of states s_1 and s_2 that differ only in their values for the variables in Y,

$$\sum_{s'_1} P\left(s'_1|s_1, a\right) R\left(s'_1|s_1, a\right) = \sum_{s'_2} P\left(s'_2|s_2, a\right) R\left(s'_2|s_2, a\right)$$

3. Result Distribution Irrelevance: A set of state variables Y_j is irrelevant for the result distribution of action a if, for all abstract hierarchical policies π executed by M_j the following holds: for all pairs of states s_1 and s_2 that differ only in their values for the state variables in Y_j,

$$\forall s' \quad P^\pi\left(s'|s_1, j\right) = P^\pi\left(s'|s_2, j\right)$$

This form of abstraction is primarily useful in undiscounted settings ($\gamma = 1$).

4. Termination: For any child j of subtask M_i, if all exit-states of M_j are also exit-states of M_i, the completion value Q_C of j for M_i must be 0 for all states.

5. Shielding: Consider subtask M_i and let s be a state such that for all paths from the root of the hierarchy down to M_i, there exists a subtask that is terminated. Then no Q_C values need to be represented for M_i in s.

The above conditions provide safe state abstractions for the MAXQ value decomposition, ensuring that the state abstraction will not contrain it from representing the optimal policy. To explain them briefly, the first two conditions provide irrelevant variable abstractions for subtask nodes and primitive nodes, respectively. The first says the Y can be ignored if the transition function for subtask given the variables in X is a distribution that is independent of Y and the value function for any policy is unaffected by the value of variables in Y. The second says Y can be ignored for a primitive action if the one-step reward for that

primitive action does not depend on the value of variables in Y. These allow us to abstract the state space for Q_V values over the set Y for the action. The last three conditions specify funnel abstractions, which allow us to abstract the state space for Q_C. The first of these funnel abstractions allows a set Y_j to be ignored for the completion value of a if Y_j does not affect the transition function under any policy. The second funnel abstraction allows us to explicitly set the Q_C value for a subtask to 0 if all of its exit-states are also exit-states of the parent. The third funnel abstraction extends a similiar notion to allow us to not represent Q_C for a subtask M_i for states in which M_i can never be called, because some subtask above it will be terminated.

Andre and Russell (2002) provide similar conditions that provide safe abstraction for the ALisp decomposition. We discuss the nature of the difference between MAXQQ and ALisp's abstraction conditions in greater detail in the section 4.8.

For the remainder of this paper, our discussion and notation will refer to abstract policies, abstract value functions, and abstract states unless otherwise specified. We will differentiate between abstract states and completely exact states of the world, by explicitly referring to the later as an "underlying state", "concrete state", or "state of the entire MDP" where it is not obvious by context. Further use of the symbol s will refer to an abstract state unless explicitly stated otherwise.

Chapter 3

Hierarchical reinforcement learning algorithms

3.1 Optimal hierarchical policies

Three different concepts of an optimal hierarchical policy have been recognized (Dietterich, 2000a):

Definition 3.1.1. A *globally optimal* hierarchical policy is one that selects the same primitive action as the optimal policy in every underlying state.

Definition 3.1.2. A *hierarchically optimal* policy is one that selects the same primitive actions as the optimal policy in every underlying state, except where prevented from doing so by the constraints imposed by the hierarchy.

Definition 3.1.3. A policy is *recursively optimal* if, for each subtask M_i in the hierarchy, the policy is optimal given the policies for the subtasks in A_i and their descendents.

Ultimately, the difference in solution quality between a globally optimal policy and a hierarchically optimal policy is the fault of the designer of the hierarchy, whether it is

designed by a human or some computational system. Ideally, the globally optimal policy is equivalent to the hierarchically optimal policy.

In HRL, the burden for creating a hierarchy in which the hierarchically optimal policy is equivalent to the globally optimal policy is placed on the designer. In practice, this does not seem difficult to achieve. For the hierarchically optimal policy to be sub-optimal, the hierarchy must impose constraints on the hierarchical policy by limiting the states in which a primitive action can be called. For example, in the TaxiFuel domain, the hierarchy shown in Figure 3.1 would constrain the policy space such that the hierarchically optimal policy would be strictly worse for some initial states than that globally optimal policy. This is because the hierarchy constrains the policy to executing the `fill-up` primitive only under the `Navigate(destination)` subtask, which will be called in the hierarchically optimal policy only after the passenger has been picked up by the taxi. There are initial states where the optimal flat policy will call the `fill-up` predicate (via the `Refuel` subtask) in a state in which the passenger is in the taxi. In the hierarchy for TaxiFuel given in Figure 3.4 does not impose this constraint, as it allows `Navigate(source)` to call `Refuel`, and thus allows the `fill-up` primitive to be executed in a state in which the passenger is not in the taxi.

It is also possible to define a hierarchy such that the recursively optimal policy is equivalent to the hierarchically optimal policy. A sufficient condition for the hierarchically optimal policy and the recursively optimal policy to be equivalent is that all subtasks in the hierarchy have no more than 1 possible exit-state for all states in which the subtask can be called. When there is more than one possible exit state for a subtask, the exit state that the agent solving the subtask reaches when optimizing its own expected reward may be sub-optimal for its parent in the hierarchy.

Consider the simple grid world navigation task in Figure 3.2 and the associated hierarchy shown in Figure 3.3. The vehicle begins in the location labeled Start and must reach the location labeled Goal. The possible actions are `north`, `east`, `south`, and `west`, all result in

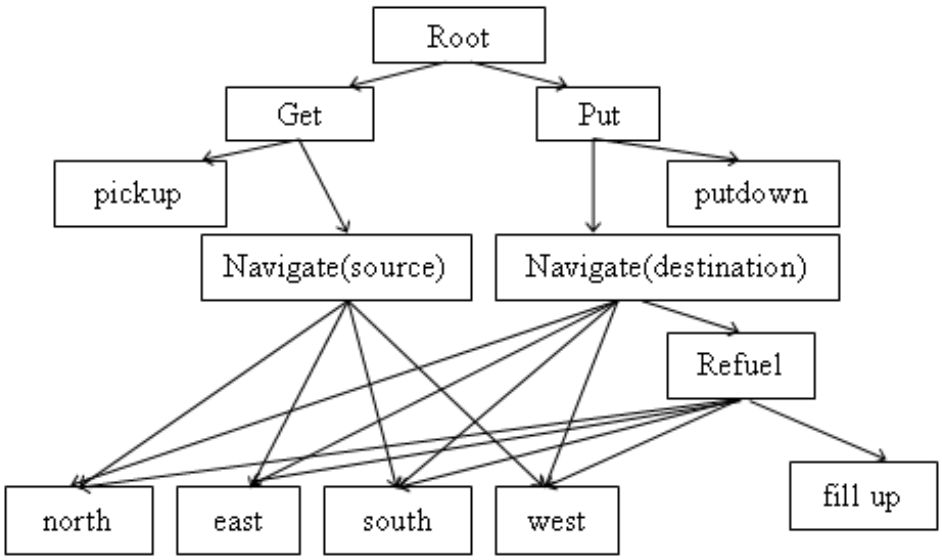

Figure 3.1: A hierarchy for TaxiFuel, constructed such that the hierarchically optimal policy consistent with this hierarchy is not equivalent to the globally optimal policy.

a reward of -1 and move the vehicle in the corresponding cardinal direction. If such a move is not possible because of the present of a wall, the action is a no-op, leaving the world in the same state and resulting in a reward of -10. We consider this as an episodic domain, with Goal as the only terminal state.

For the SMDP corresponding to the `Leave left room` subtask, the optimal policy is to leave via the lower door. This route can be achieved with a maximum total reward of -3. The other exit-state, through the upper door, can only be achieved with a maximum total reward of -4. The recursively optimal policy for this hierarchy would require that the policy for `Leave left room` be optimal from its own perspective, with reward of -3. This would result in a total cumulative reward of -8, as the optimal policy for `Reach goal` would receive reward of -5 to achieve its exit-state. In contrast, the hierarchically optimal, which is equivalent to the globally optimal policy in this case, would require the agent at `Leave left room` to use the upper door, with reward of -4. This would allow the agent at

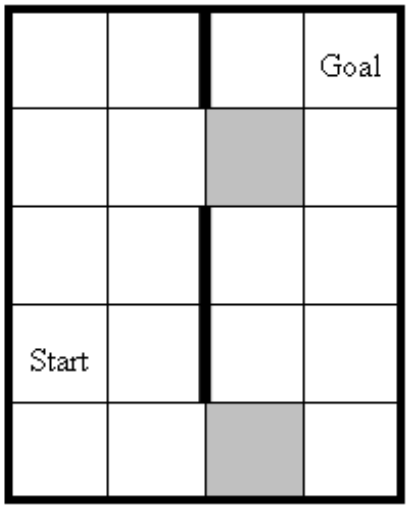

Figure 3.2: A simple grid world navigation problem, adapted from a similar example of Dietterich (2000a), with exit states shaded for the `Leave left room` sub-task, as depicted in Figure 3.3.

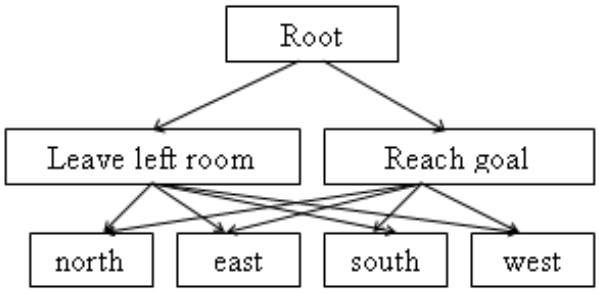

Figure 3.3: Simple hierarchy for the problem depicted in Figure 3.2.

"Reach goal" to achieve its goal with reward of -3, giving a total reward of -8. To find a hierarchically optimal policy rather than only a recursively optimal policy, a HRL algorithm must be able to recognize such trade-offs and handle them appropriately by allowing some subtasks to be solved in a sub-optimal manner.

For the Taxi domain (introduced in section 1.1) with the hierarchy given in Figure 2.1, the recursively optimal and hierarchically optimal policies are equivalent. For the TaxiFuel domain, there is no obvious hierarchy that is both useful and yields equivalent recursively optimal and hierarchically optimal policies. The hierarchy given in Figure 3.4 is a useful decomposition, but it potentially yields recursively optimal policies that may be considerably worse than the hierarchically optimal policies. For both domains, the hierarchically optimal policies are globally optimal.

Note that it is always possible to produce a trivial hierarchy under which all recursively optimal policies are also hierarchically and globally optimal: this is simply the hierarchy with only a `Root` node. It is also possible in many domains to transform a hierarchy such that all subtasks will have only 1 possible exit-state given the initial state in which they are called. For the example domain in Figure 3.2, this transformation would replace `Leave left room` with `Leave left room via upper door` and `Leave left room via lower door`. Such a hierarchy would yield a recursively optimal policy that is equivalent to its hierarchically optimal policy, and is also globally optimal. This is because, given the context in which each subtask is called, there is only one possible exit-state for each subtask. In general, such modifications can be done for any hierarchy by creating a separate subtask for each possible exit-state within a subtask. This approach undermines the motivation for HRL by placing more burden on the human programmer to reason about the possible exit-states for all possible states in which a subtask can be called. Additionally, introducing many nearly identical subtasks increases the size of the action space and severely diminishes the potential for reuse of learning compared to considering all these as one subtask, although some sharing may still be possible. Both of these factors will significantly slow convergence in practice.

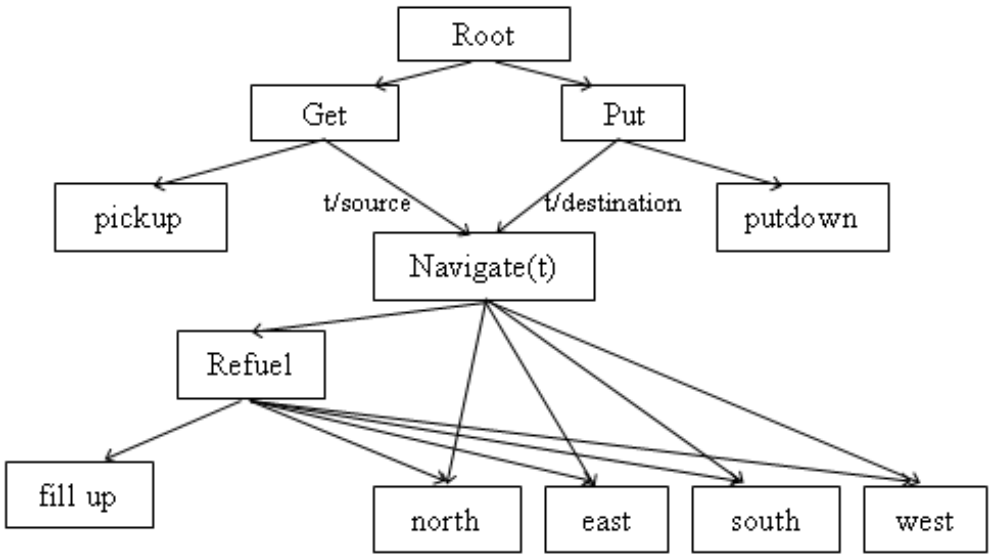

Figure 3.4: Hierarchy created for the TaxiFuel domain.

This paper and all the HRL work discussed in this paper assume the hierarchy is fixed and do not consider approaches to modify it automatically to afford better convergence results.

The issue of recursive vs. hierarchical optimality is a central tension in HRL work and a primary focus of this paper. The remainder of this chapter introduces several known approaches to hierarchical learning and planning, and places these approaches within the context of this tension. This discussion exposes trade-offs between recursive and hierarchical optimality in speed of convergence in practice, simplicity and use of state abstractions, and ability to provide formal proof of convergence. The hierarchical planning work of Dean and Lin (1995) introduced the concept of breaking apart an MDP into sub-problems and then iteratively solving these individual problems and re-combining them to plan an optimal policy. The HAMQ algorithm (Parr and Russell, 1998) applied a similar approach to reinforcement learning and provided a more structured concept of a hierarchy. The MAXQQ algorithm (Dietterich, 2000a) provided an alternative formalism of a hierarchy and introduced value decomposition, which in turn allowed for the introduction of sophisticated

state abstraction. ALispQ (Andre and Russell, 2002) introduced a third formalism for a hierarchy and a value decomposition that allowed for representation of hierarchically optimal policies. Finally, HOCQ (Marthi et al., 2006) introduced methods to learn more rapidly through greater state abstraction when using the ALisp value decomposition. This chapter discusses these approaches in this sequence in an effort to underscore the recursive-hierarchical optimality distinction and expose the associated difficulties.

3.2 Hierarchical decomposition for planning

There are a number of algorithms known that follow the same basic strategy of hierarchically decomposing a problem, solving the resulting sub-problems, and combining those local policies to create a hierarchical policy for the entire MDP. Dean and Lin (1995) introduced a form of planning algorithm that decomposes an MDP in this way. The approach they suggest makes initial guesses about the exit-state values for the local sub-problems, solves the local sub-problems given the exit-state values, constructs an abstract hierarchical learning problem from the costs and transition probabilities of these local solution policies, solves that abstract hierarchical problem, and updates the exit-state values for each sub-problem based on that solution. By repeatedly breaking apart the problem and re-combining solutions in this manner, the algorithm finds exit-state values such that the recursively optimal solution, given those values, is equivalent to the hierarchically optimal solution.

Dean and Lin (1995) prove that this off-line approach will yield a hierarchically optimal solution policy in a finite number of iterations. They formulate their iterative method as solving a large linear program and apply the Dantzig-Wolfe method of decomposition. This affords them strong results for rapid convergence to within 1-5% of quality of the optimal policy. Dietterich (2000a) points out that a drawback of this approach is that they must repeatedly solve the hierarchical learning problem, which he claims will not always yield faster convergence than solving the flat problem. Dean and Lin (1995) make no use of

value decomposition and state abstraction. They provide an illustration of the necessity of properly valuing exit-states for an optimal policy and prove that their off-line planning approach will converge to a hierarchically optimal policy.

3.3 Hierarchical Abstract Machines

Hierarchical abstract machines Q-learning (HAMQ-Learning), developed by Parr and Russell (1998), applies the general ideas introduced by Dean and Lin (1995) to reinforcement learning. In HAMQ, hierarchies are specified as *non-deterministic finite state machines* called hierarchically abstract machines (HAMs), which include choice states for which the appropriate transition must be learned. A HAM can have designated `Action` states, which execute primitive actions in the world, as well as `Call` states, which transfer control to another HAM as a subroutine. Control is returned to the calling HAM when the current HAM being executed reaches one of a set of designated `Stop` states. The HAMQ learning algorithm learns a solution policy for an MDP by executing an HAM. The HAM can constrain the possible actions that can be executed in a given state, effectively reducing the state space and the search space for the solution policy. This allows HAMQ to take advantage of the information encoded in the HAM structure to converge rapidly to a hierarchically optimal policy. Parr and Russell (1998) provide proof that HAMQ will converge to a hierarchically optimal policy with probability 1.

3.4 MAXQQ Learning

Dietterich (2000b) introduced HSMQ, which used a directed acyclic graph as a hierarchy, depicted in 2.1. (This is the same formalism of a hierarchy used in this paper and is discussed in detail in section 2.3. Each node in the hierarchy for HSMQ learned to solve the SMDP represented by its subtask by maintaining its own table of Q-values and updating them

to perform traditional Q-learning of that SMDP. In this sense, it is one of the simplest HRL algorithms, effectively performing flat RL at each node in the hierarchy. HSMQ will converge to a recursively optimal policy under similar conditions to traditional Q-learning. Dietterich (2000a) adapts HSMQ by introducing the MAXQ value decomposition. By decomposing the Q-values for the sub-problems at each node in a hierarchy into Q_V and Q_C values, MAXQ allows the use of the state abstractions as described in section 2.6. The MAXQ decomposition also allows for the re-use of these values throughout the hierarchy where appropriate. Dietterich (2000a) provides a proof that the MAXQQ-learning algorithm converges to a recursively optimal policy with a GLIE exploration policy, even when using state abstractions.

Pseudo-code for MAXQQ is given below for reference and comparison to the algorithms proposed later in the paper. It reflects the version of MAXQQ implemented to generate the experimental results presented in this paper. A more technically precise version of the pseudo-code can be found in Dietterich (2000a). The version presented below follows that presented in Dietterich (2000b), but is corrected to reflect to apparent errors. In particular, we correct an obvious typo in the indexing of Q_C and remove an else construct so that Q_C is updated whether or not the action a is a primitive. We justify the latter correction by noting that without this change, no Q_C value will ever be stored for a primitive action a under the subtask i. These corrections are consistent with the more technically precise version of MAXQQ presented by Dietterich (2000a). This pseudo-code does not expressly deal with pseudo-reward functions or discounting, which are handled by the more technical version of the pseudo-code.

```
MAXQQ(state s, subtask i)
let total_reward = 0
while T_i is not statisfied do
  choose action a according to the exploration policy π_x
  if a is primitive then
    execute a
```

observe one-step reward r
else
r = MAXQQ(s,a) { which invokes the subroutine for a and returns the total reward received while the subroutine executed}
$total\_reward := total\_reward + r$
observe resulting state s'
if a is primitive **then**
$Q_V(s,a) := (1-\alpha_t) \cdot Q_V(s,a) + \alpha \cdot r$
else

$$Q_C(i,s,a) := (1-\alpha_t) \cdot Q_C(i,s,a) + \alpha \cdot \max_{a \in A_i} \left[Q_V\left(s', a'\right) + Q_C\left(i, s', a'\right) \right]$$

end if
return $total\_reward$;
end if
end while

In the MAXQQ algorithm, each non-leaf node in the hierarchy can be thought of as being implemented by an agent. As the MAXQ decomposition indexes its Q_C values by macroaction, each non-leaf node in the MAXQQ algorithm can be thought of as storing its own Q_C table indexed by state-action pairs.

While MAXQQ is proven to converge to a recursively optimal policy under safe state abstractions and discounting, the authors claim that it will converge to a hierarchically optimal policy if the pseudo-reward function R_i is hand-coded to give the appropriate amount of reward. While hand coding reward in this manner seems possible for some of the trivial domains presented in the literature, we argue that in a discounted setting, it will be very difficult for a programmer to reason about the appropriate amount of exit-reward to assign. While the possibility of hierarchically optimal convergence for MAXQQ using exit reward should be highlighted, we claim that this approach is tedious at best and wholly impractical at worst. We seek to avoid relying on hand coded pseudo-reward functions in our approach.

3.5 ALisp-Q Learning

Andre and Russell (2002) built on HAMQ and MAXQQ by combining earlier notion from Parr and Russell (1998) of non-deterministic finite state machines with the idea of value decomposition. In this work, they specify non-deterministic finite state machines as partial programs written in a dialect of the Lisp programming language which they call ALisp. Within these programs are choice points (analogous to choice states in HAM), which give the program flexibility to choose over some set of primitive actions and other partial programs to call as subroutines. Andre and Russell (2002) introduce the ALisp value decomposition, which represents Q-values as the sum $Q_V + Q_C + Q_E$. Inclusion of the Q_E term allows the ALisp value decomposition to represent the value function for a hierarchically optimal policy. They propose the ALispQ algorithm to learn the values for Q_V, Q_C, and Q_E. The use of the ALisp decomposition allows for reuse and the introduction of state abstractions in a similar manner to MAXQ. Since ALispQ uses a value decomposition which represents Q_E, each agent solving a sub-problem can reason about its effect on the solutions to other sub-problems, allowing the algorithm to converge to a hierarchically optimal policy in practice. However, Q_E will often depend on a much larger number of state variables than Q_V and Q_C, thus limiting the possibility for state abstraction and leading to slower convergence than MAXQQ.

Marthi et al. (2006) propose a variation on the ALispQ algorithm that does not explicitly represent and learn Q_E, thus improving convergence speed while maintaining the possibility of convergence to a hierarchically optimal solution policy. Instead, Hierarchically Optimal Cascaded Q-Learning (HOCQ) uses an algorithm to learn the conditional probability distribution for the exit-state of each subtask M_j given that some subtask M_i calls M_j in state s. Using this probability, denoted as $P_E(e, j|s, a)$, the algorithm calculates a value for as follows:

$$\forall s' \quad Q_E\left(j, s', a\right) = \sum_{e \in E_j} P_E\left(e, j + s', a\right) \cdot V^*\left(i, e\right)$$

$V^*\left(i, e\right)$ is M_i's value for the state e, itself represented by a decomposed value function of $Q_V + Q_C + Q_E$. Using this value for Q_E, it is possible to optimally solve M_j with respect to the rest of the hierarchy, yielding a hierarchically optimal policy for the entire MDP. Through the use of state abstractions and approximations for P_E, HOCQ can perform significantly better than ALispQ.

3.6 Discussion

From the description provided above, it should be clear that algorithms which correctly model the value of different the exit-states for a subtask have the potential to converge to hierarchically optimal policies. In MAXQQ learning, this can be done by the programmer correctly guessing values for the possible exit states and constructing the pseudo-reward function $R_i\left(s'|s, a\right)$ to give the local subtasks exit reward that reflects these values (Dietterich, 2000a). However, in highly stochastic domains with discounting, estimating the appropriate exit-state rewards to yield hierarchically optimal convergence is an unreasonable burden to place on the programmer of the hierarchy.

ALispQ-learning is an improvement over MAXQQ in this regard, as it represents the exit-state values as Q_E and therefore can learn them. The Marthi et al. (2006) approach of learning P_E and using the value function of the parent is particularly informative. In the case where a subtask has only one exit-state given the context it is called in, in which case the hierarchically and recursively optimal policies are equivalent, $P_E = 1$ for that exit-state and 0 for all others. In such a case, Q_E is uniform across all local actions that are considered and therefore has no effect on the local decision. This approach can be viewed as considering the parent's value function to represent exit-state values or exit-state rewards.

The local agent learns to reason about these rewards by learning P_E, which is in effect the probability of getting a particular exit-state reward. The agent then acts to maximize its expected cumulative reward plus its expected exit-state value, even though the agent does not explicitly gain utility for exiting in a particular state. Our approach, presented in the following chapter, builds upon these intuitions.

Table 3.1 summarizes the convergence properties of the HRL algorithms introduced in this chapter.

Only the planning work of Dean and Lin (1995) and Dietterich's MAXQQ algorithm are supported with formal proofs of convergence. A detailed discussion of the difficulties associated with HRL convergences proofs, with particular focus on our own efforts with EHQ is provided in section 4.9. As a result of these theoretical difficulties, many authors offer empirical support for their claims.

Table 3.1: Summary of hierarchical reinforcement learning algorithms

	HPlan[9]	HAMQ	MAXQQ	ALispQ	HOCQ	rEHQ	EHQ	Q[10]
RO policy[1]	X	X	X*	X*	X*	X	X	X*
HO policy[2]	X	X		X*	X*		X	X*
RO convergence[3]	X*	X*	X*	X†	X†	X†	X†	X*
HO convergence[4]	X*	X*		X†	X†		X†	X*
Value decomp.[5]			X	X	X	X	X	
State abstraction[6]			X	X	X	X	X	
Decentralized[7]	X						X	
Models exit value[8]	X	X		X	X		X	X

* shown by formal proof
† shown empirically
[1] the value decomposition can represent a recursively optimal policy
[2] the value decomposition can represent a hierarchically optimal policy
[3] the algorithm will converge to a recursively optimal policy
[4] the algorithm will convergence to a hierarchically optimal policy
[5] uses some form of value decomposition
[6] has the capacity for substantial state abstraction
[7] multiple nodes in the hierarchy do not share the same Q-values
[8] maintains or calculates, implicitly or explicitly, a value for a given exit-state within a subtask
[9] the hierarchical planning approach of Dean and Lin (1995)
[10] traditional flat Q-learning algorithm, for comparison

Chapter 4

Economic Hierarchical Reinforcement Learning

The overview of HRL algorithms presented in Chapter 3 showed that for hierarchically optimal convergence, an algorithm must represent the impact of local actions on the global solution quality. All the hierarchically algorithms discussed in the previous section did this by assigning values to exit states. Rather than hand code these values, this chapter provides an automatic, conceptually simple approach to set these exit-state values. We begin by providing background on artificial economies. We follow by introducing the general outline of our system and specify a particular algorithm called Recursively Optimal Economic Hierarchical Q-learning (rEHQ). Following a proof of this algorithm's convergence to a recursively optimal solution policy, we present a number of modifications, including the introduction of exit-state subsidies, necessary to allow the system to represent hierarchically optimal policies. We discuss the numerous concerns that these subsidies can create but ultimately address them with modifications that produece a conceptually simple, decentralized HRL algorithm. This EHQ algorithm has the potential to converge to hierarchically optimal policies in practice, which will be shown in Chapter 5. We close the chapter with a discussion

of the difficulties of proving hierarchical optimal convergence of HRL algorithms in general, and the particular difficulties associated with EHQ.

4.1 Artificial Economies

Inspiration for our approach comes from the work of Baum and Durdanovich (1998) on the Hayek system, which itself is a variation on the Holland classifier system. In Hayek, an agent may buy control of the world in an auction, perform actions upon the world, and then auction off control of it again. Each agent is a program, specified as a set of expressions that can be evaluated to give the amount the agent bids and the specific primitive actions it performs, contingent upon the state of the world. The domains presented by Baum and Durdanovich are episodic, with a single terminal state with an associated reward. Only the agent that takes the final action to reach the terminal state receives reward directly from the world. All other agents receive reward only from the income they receive by selling control of the world. There is a small tax imposed on each agent for executing a primitive action, to avoid agents taking unnecessary actions. An agent is profitable if it is consistently able to buy the world, convert it to a different state, and then resell it at a higher price. Profitable agents produce offspring that are randomly mutated versions of themselves. Unprofitable agents die off. Over many generations, Baum and Durdanovich (1998) argue that this system will produce a population of agents than can collectively produce an optimal solution to a problem instance. If the language used to specify the agents is Turing complete, they empirically show that this system can produce a population capable of solving any arbitrary instance of particular problem domain. Furthermore, when the system reaches equilibrium, it will have solved the credit assignment problem, attributing to each agent a utility equivalent to that they are abstracting from the world.

The argument for this claim is to assume that the system is not in equilibrium. In this case, there must be an agent that is making too much profit. An agent can enter, bid

a higher amount for the world, win the auction, and sell the world for the same price as the previous agent. Such competition will drive agents making excessive profits out of the system. An agent which performs unnecessary actions will find that the state of the world it is producing will not be sufficiently valued by other agents in equilibrium for the agent to be profitable, and so it will be killed off. The random, evolutionary nature of the system makes it unpredictable and slow. In practice, the Hayek approach proves to be slow for many problem domains and requires careful tuning. It must be gradually scaled up from small, training examples of the domain to help it develop a desirable population of agents, although it has the potential to evolve a population of agents that can solve a general instance of the domain. Despite its undesirable practical aspects, Hayek demonstrates the power of a system of economically self-interested agents to cooperatively solve problems and distribute utility within the system to generate a globally optimal solution.

4.2 Recursively Optimal Economic Hierarchical Q-Learning

Following a notion similar to Hayek, we construct the framework for our system as a hierarchical artificial economy. We use the same value decomposition and formalization of a hierarchy as MAXQQ-learning. Unlike MAXQQ, we allow multiple agents to implement each subtask in the hierarchy. Like Hayek, agents buy and sell control of the world. In our system, the agents to which any given agent can transfer control of the world to is constrained according to the hierarchy. Additionally, we use the subtask predicates provided by the hierarchy to specify when control returns to the agent who formerly had control of the world. Only one agent in the system owns control of the world at a time. That agent can choose amongst its child actions in the hierarchy until it satisfies its predicate. When an agent satisfies its predicate, it returns control to the agent that called it. For completeness, the `Root` subtask has no predicate; in the episodic setting, it finishes only in when the world reaches a terminal state, and in the infinite horizon setting, it never completes. This

is generally analogous to how MAXQQ operates in the way it passes control between nodes in the hierarchy.

Unlike MAXQQ, agents do not return the sum of the primitive reward accrued below them up to the agent that called them. Instead, these transfers are governed by auctions. Recall that there are multiple agents representing each subtask in the hierarchy. The parent agent, currently with control of the world, must select action in accordance with its value function and exploration policy. For a macroaction, the parent must choose which agent implementing the macroaction to which to pass control. To select the agent for a given macroaction, we hold an auction in which all agents implementing the macroaction submit bids (see Figure 4.1). The agent submitting the highest bid is given control by the parent (see Figure 4.2). The macroaction, which has a defined predicate, functions as a contract as to what "product" this child agent must provide - namely, the child agent must return the world in an exit-state that satisfies the predicate. In return for providing this product, the child pays the amount of its bid (which may be negative). This transfer is the only reward that the parent agent receives for invoking a macroaction.

We argue that a rational, self-interested agent, with perfect knowledge of the world, should bid up to its V^* value for the current state of the world. This follows from the intuition that this behavior is the only Nash equilibrium in this system; with multiple identical agents and ties are broken randomly, no single agent will deviate from this behavior. We do not set out to prove this here, but simply define our agents to bid in this manner. See section 6.7.2 for discussion of an extension to this algorithm in which agents bid strategically.

An apt metaphor for this system is the notion of outsourcing - agents who must fulfill a contract either take primitive actions to fulfill that contract or else may offer subcontracts, as specified by the hierarchy, to other agents. The primitive actions are analogous to paying the costs of resources or labor directly. Note that the sub-contractor agents are bidding reward they will pay the parent for the right to fulfill the contract, so the highest bidder wins the contract.

Our auction policy does not strictly select the agent with the highest bid, but only does so in the limit. If it did so, some agents may never have the chance to act in the world and develop beliefs about the costs it faces. Without experience, an agent cannot learn a realistic representation of its value function and thus cannot bid appropriately. There would never be effective competition in such a system; this competition is necessary for the robustness of the system. To avoid this case, we define the auction policy to be epsilon-greedy. With probability $1-\epsilon$, the agent submitting the highest bid wins control of the world; otherwise, a winner is selected uniformly at random. We cool ϵ such that our auction policy satisfies the requirements of GLIE. In this manner, the auction policy enforces exploration. However, this makes our claim that agents should bid their V^* significantly more dubious and perhaps only valid in the limit when $\epsilon \to 0$.

Agents within this system update their Q_V and Q_C tables in an analogous manner to MAXQQ. The only difference is that the observed reward used for these updates of macroactions is not primitive reward, but the transfer from the child based on the child's belief about the world. Like MAXQQ, this system recursively decomposes Q_V for macroactions, modeling $Q_V(s,a)$ as the highest bid by an agent implementing a. Since agents bid their V^*, agents bids must be further decomposed in this manner. The consequences of this are discussed in section 4.4.

We will refer to the system described above as Recursively Optimal Economic Hierarhical Q-Learning (rEHQ), as we claim that it will converge to a recursively optimal solution policy for a given MDP and hierarchy. It is the basis for the hierarchically optimal EHQ algorithm that will be presented later in this chapter. Figures 4.1 and 4.2 illustrate the flow of control and transfer of reward in rEHQ. We provide the following pseudo-code for the rEHQ algorithm for further specification:

rEHQ(agent g implementing subtask i)
while T_i is not satisfied **do**
  select an action a from A_i according to exploration policy π_x

if a is a primitive action **then**
take action a
observe one-step reward r
observe resulting state s'
$Q_V(s,a) \Leftarrow (1-\alpha_t)\cdot Q_V(s,a) + \alpha_t \cdot r$

$$Q_C(g,s,a) \Leftarrow (1-\alpha_t)\cdot Q_C(g,s,a) + \alpha_t \cdot \gamma \cdot \max_{a' \in A_i} \left[Q_V\left(s',a'\right) + Q_C\left(g,s',a'\right) \right]$$

else
take bids from all agents that implement a
select winning agent *child* according to auction policy
rEHQ(*child*)
$r \Leftarrow BID(child, s)$
observe resulting state s'

$$Q_C(g,s,a) \Leftarrow (1-\alpha_t)\cdot Q_C(g,s,a) + \alpha_t \cdot \gamma \max_{a' \in A_i} \left[\max_{g' \in agents(a')} \left[bid\left(g',s'\right) + Q_C\left(g,s',a'\right) \right] \right]$$

end if
end while

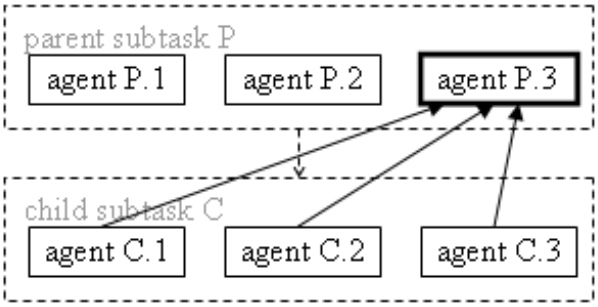

Figure 4.1: EHQ phase 1: The parent agent which currently has control of the world solicits bids from the agents implementing the selected child action.

In our implementation of rEHQ (and EHQ), we use an epsilon-greedy exploration policy, which satisfies the requirement of GLIE. This policy selects the action which maximizes the expression with probability $1-\epsilon$; otherwise it selects an action uniformly at random. In the experimental results presented later in this paper, both the exploration and the auction

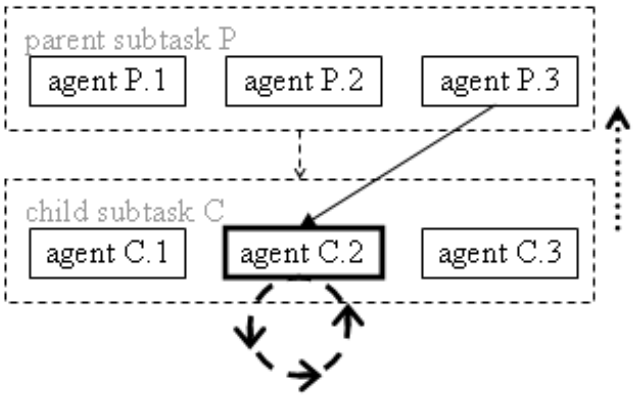

Figure 4.2: EHQ phase 2: The parent agent passes control to the winner of the auction in return for the transfer of bid amount. The child agent then acts by invoking other subactions or executing primitives, until its subtask is terminated.

policies used the same value for ϵ, although clearly distinct values may be used.

The system as described above will converge to a recursively optimal solution using these exploration and auction policies. Proof is by reduction to MAXQQ learning, which is shown to converge by Dietterich (2000a) with a GLIE exploration policy.

Theorem 1. *Let $M = \langle S, A, P, R, P_0 \rangle$ be an episodic MDP or a discounted infinite horizon MDP with discount factor γ. Let H be a hierarchy defined over subtasks $\{M_0, M_1, ..., M_k\}$, where all M_i have pseudo-reward functions $R_i(s'|s,a) = 0$ for all s'. Let $\alpha_t(i) > 0$ be a sequence of constants for each subtask i such that:*

$$\lim_{T\to\infty} \sum_{t=1}^{T} \alpha_t(i) = \infty \quad \textit{and} \quad \lim_{T\to\infty} \sum_{t=1}^{T} \alpha_t^2(i) < \infty$$

Let $\pi_x(i,s)$ be an ordered GLIE policy for each node i and state s and assume that the immediate rewards are bounded. Then with probability 1, the rEHQ algorithm converges to π_r^*, the unique recursively optimal policy for M consistent with H and π_x.

Proof: To show that rEHQ converges, it must be shown that 1) rEHQ updates its Q_V and Q_C tables as MAXQQ does and 2) these values are updated according to the conditions

of a GLIE policy. Given these two conditions, it will converge as MAXQQ does, yielding a recursively optimal policy.

For updating Q_V, rEHQ performs precisely the same update as MAXQQ, observing the implicit reward and using this value to update Q_V, weighted by the learning rate α. To show that rEHQ's update of Q_C is equivalent to MAXQQ, it must be shown that $Q_V(s', a')$ is equivalent to $\max_{g' \in agents(a')} bid(g', s')$ ($agents(a)$ is defined as the set of agents implementing the subtask a). The latter expression is the highest bid for control of the world in the current state by an agent implementing the subtask a'. In the version of rEHQ described here, each agent bids its estimate of V^* for the current state of the world by definition. In MAXQQ, $Q_V(s, a)$, where a is a non-primitive subtask M_j, is recursively decomposed into $\max_{a' \in A_j} [Q_V(s, a') + Q_C(j, s, a')]$. This value is equivalent to the bid value of an rEHQ agent implementing M_j, as the V^* of that agent is given by the expression $\max_{a' \in A_j} [Q_V(s, a') + Q_C(j, s, a')]$. Maximizing over the population of agents implementing M_j ensures that the expected value for M_j matches the likely cost that the parent agent will have to pay to execute M_j as $\epsilon \to 0$. Although the discount factor γ is omitted here, these claims hold for $\gamma \neq 1$.

To prove that the Q_V and Q_C values are update consistent with a GLIE policy, we must consider both the exploration policy by which an rEHQ agent selects which action to execute and the auction policy by which it selects which agent to implement a macroaction. By using a GLIE exploration policy, we ensure that all actions are executed an infinite number of times in all states. However, this is not enough to ensure that all agents execute all their actions an infinite number of times in all that states in their state space. To achieve that condition, we selected an auction policy such that all agents win the auction an infinite number of times. The auction policy can be thought of as greedy in the limit as for $\epsilon \to 0$ the agent with the highest bid wins the auction with probability 1. Therefore, all agents execute all actions in all states greedily in the limit of infinite exploration. Although we are storing multiple Q-tables that are trying to learn the same values, each of these tables

should converge to the appropriate values.

From these two conditions, we claim that rEHQ updates its Q-values and executes subtasks and primitives in an analogous manner to MAXQQ. The values used in these updates will be the same as the algorithm converges. We now rely on Dietterich's result for the convergence of MAXQQ, which we state without proof.

Lemma 2. *(based on Theorem 3 and Corollary 1 from Dietterich (2000a)) Let M, H, $\{M_0, ..., M_k\}$, $\pi_x(i,s)$, and $\alpha_t(i)$ be defined as specified in Theorem 1 under the same conditions. Then with probability 1, the MAXQQ learning algorithm will converge to the unique recursively optimal policy for the MDP M.*

We therefore claim that rEHQ, by reduction to MAXQQ, will converge to the unique recursively optimal policy π_r^*. Naturally, we require the same conditions as MAXQQ. **Q.E.D.**

For details, we refer to the proof given by Dietterich, which is based on the stochastic approximation argument presented in Bertsekas and Tsiksiklis (1996) to prove convergence of Q-learning and SARSA.

4.3 Exit-state subsidies

Having shown that rEHQ will yield convergence to a recursively optimal policy, we now suggest a natural modification to the rEHQ algorithm that will lead it to converge to a hierarchically optimal policy. For an algorithm to converge to a hierarchically optimal policy, it is necessary to propagate information about the local solutions effect on the entire problem into the local agent. Once the appropriate exit-state values are assigned, the locally optimal solution to a given sub-problem is also hierarchically optimal. In ALisp, this was done by calculating a value for Q_E, which was then used by the local agent to reason about the optimal actions.

Rather than represent Q_E, we propose propagating information about the effects of the

solution to the local sub-problem by having the parent agent promise an explicit subsidy to a child for each of the child's underlying exit-states. Intuitively, the parent agent should be willing to pay a child agent for returning the world in exit-state e' up to the value of the parents preference for e' over the exit-state e in which the child would otherwise leave the world without the subsidy. Consider an agent *parent* implementing a subtask M_i and another agent *child* implementing a subtask M_j that is a child of M_i. Define $SUBSIDY\,(parent, e)$, where e is a concrete exit-state of subtask M_j ($e \in E_j$), to be the expected marginal value of e to the agent *parent*. The following equation defines the value of $SUBSIDY\,(parent, e)$:

$$SUBSIDY\,(parent, e) = V^*_{parent}\,(e) - \min_{e' \in E_j} \left[V^*_{parent}\left(e'\right)\right]$$

For an example of how this approach is applied in practice, consider again the example from Figure 3.2, shown in Figure 4.3 with subsidies assigned to the exit-states of `Leave left room` according to the equations given above.

In 4.3, the exit-state subsidies change the locally optimal policy for `Leave left room` to exit through the upper door. Leave left room receives primitive reward of -4 plus exit-state reward of 3 for leaving through upper door for a total of 1. Therefore, it is optimal to exit by the upper door rather than to exit by the lower door, for which it will receive primitive reward of -3 and exit-state reward of 0. Note that a consequence of assigning subsidies is that the `Root` subtask is indifferent between receiving the world in either of `Leave left room`'s exit-states, as both solutions will have the same cost to the parent if the child has perfect knowledge. In general, assigning subsidies to be equal to be the parent's expected marginal value for the exit-state will make the parent indifferent over all the child's exit-states.

In general, we must price over possible underlying states of the world for which the child is terminated. From the perspective of the child, the concept of an *abstract exit-state* is not useful, as the child does not represent a value function for states in which it is terminated. In

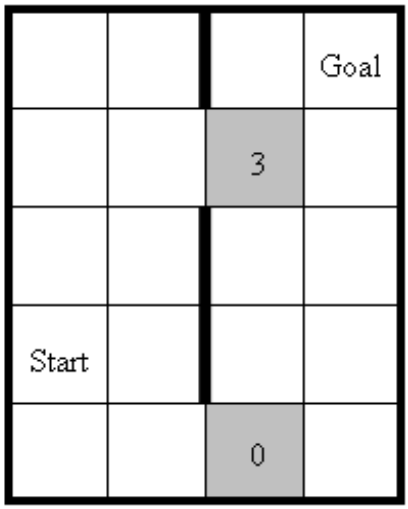

Figure 4.3: The simple grid world navigation task with subsidies assigned to the exit states of `Leave left room`.

practice, we can make use the parent's abstraction to transfer a more concise representation of the exit-state subsidies for the child over the underlying states in which the child may terminate.

The use of exit-state subsidies adds an additional stochastic component to the reward of the child for every state-action pair. The subsidy values propagate back into the child's value function of the state-action pairs that can lead to an exit-state. While the parent is learning its value function, the expected subsidy for an exit-state will be non-stationary. More discussion of this aspect and its implication on convergence is provided in section 4.9.

As the agents we are considering all use forms of Q-learning, they are slow to learn and incorporate exit-state subsidies into their reasoning. For the subsidies to have their correct influence, they must propagate back through all of an agent's Q_C values. Notably, only then will they be incorporating into the child's bids.

It is possible in this scheme for the parent to pay a much larger subsidy than the reward the parent expects to receive in the future. This is only rational as the subsidy will propagate

back through the child's Q_C values and into its bid, less whatever amount is necessary to change the child's behavior (assuming the subsidy is sufficiently large to affect a change). The excessive amount of the subsidy will be incorporated into the bid, so the parent will not have a net loss once the system converges. The amount of the subsidy that does not come back in the bid should be equivalent to the additional value to the parent of the exit-state that the subsidy incentivized the child to reach. Therefore, the parent cannot lose, as any amount that is returned in the bid is causing the world to be left in an exit-state that has exactly that same amount of additional value to the parent. This can be demonstrated through simple algebra. In practice, the subsidies make the parent indifferent between the possible exit-states of the child, passing the relevant information about its own values for these states to the child and relying on the child to incorporate this information into its value function to reach the optimal policy.

4.4 Recursive decomposition

The MAXQ value decomposition, as described in section 2.5, minimizes the number of Q-values it needs to explicitly represent by decomposing the value for a macroaction Q_V into $\max_{a' \in A_j} [Q_V(s,a) + Q_C(j,s,a')]$, where M_j is the subtask that implements a. If EHQ were to use this recursive decomposition, it would raise the same concerns about the independent, self-interested agents with private information as described in the previous section. In the recursively optimal version of EHQ introduced in the beginning of this chapter, we relied on the fact that a rational agent should bid its V^* value, which is equal to $\max_{a' \in A_j} [Q_V(s,a) + Q_C(j,s,a')]$. When bids are defined to be the agent's V^*, soliciting a bid is equivalent to calling a recursive function that performs the appropriate decomposition. However, since Q_C values are used in decision making, it is not clear that the child agent should be truthful. It seems plausible that there may be a domain for which the agent would want to make its action appear less costly, thereby leading the decision making of

the parent towards invoking it, allowing it to extract more utility. While an appeal to the competitive structure of a multi-agent market may eliminate this in equilibrium, it still seems to be a dangerous approach to have agents submit bids in contexts when these bids will never be used. The capacity of an agent to selectively lie seems very troubling in such a case.

An additional problem with using bids for recursive decomposition occurs when we introduce exit-state subsidies. The bid amount incorporates not only the reward that the child expects to accrue by executing its actions, but also its belief about the expected exit-state reward that the parent will pay it. From the perspective of the parent, having learned nothing about the exit-state distribution of the child (which is implicit in the child's private Q_C values), it is impossible to decompose the expected exit-state reward component out of the child's bid.

While there may be a number of ways to handle the two recursive decomposition problems described above, both are most easily dealt with by eliminating the recursive aspect of the decomposition. Therefore, we explicitly represent $Q_V(s, a)$ for all macroactions as well as primitives, avoiding the recursion entirely. This approach also resolves addition issues with the EHQ algorithm, which are discussed in detail in next section. After we discuss these issues, we will provide further discussion of the implications of explicitly representing and learning $Q_V(s, a)$ for macroactions.

4.5 Decentralization

Recall from Chapter 3 that the Q_V value in the MAXQ decomposition (as well as ALisp) is not indexed by subtask, but rather is shared by all nodes in the hierarchy that execute primitive actions. This has clear benefits to learning speed under state abstraction, as once the `Navigate(source)` subtask has learned that taking the primitive action `west` in a given location gives a penalty for hitting a wall, `Navigate(destination)` will see this same

penalty. When `Navigate(source)` is given a particular exit-state subsidy for reaching a location, this reward must be learned as part of Q_V, making this sharing inappropriate. To converge to hierarchically optimal policies, such rewards may need to be defined for MAXQQ-learning. Therefore, MAXQQ-learning must either use a state abstraction that differentiates between Q_V values in `Navigate(source)` and `Navigate(destination)` (for example, including a variable indicating whether the passenger is in the taxi or not), or else use Q_V values that are locally stored in each of these nodes. Only Q_V values for the exit-states of a subtask must be stored locally, so most Q_V values could still be shared. Since Q_C values are indexed by subtask, these can already be thought of as stored locally at each subtask.

By introducing exit-state subsidies to EHQ, we create an analogous problem. A primitive action may have a different expected reward depending on which subtask executes it. Again, the solution is to store Q_V locally for all the exit-states of a subtask.

While reuse of Q_V values for non-exit-states may speed convergence, it undermines the motivation for viewing this system as an artificial economy; namely, such an artificial economy should be a multi-agent system in which the agents are independent, self-interested economic actors. Obviously, such agents should not share their private knowledge of the costs of an action in the world, represented by Q_V, with the agents they are competing against. This would in turn undermine our desire to have a robust system that could be used in loosely coupled settings. As Q_V is updated by the agent acting in the world, we could avoid problem by making the somewhat plausible assumption that the other agents in the world see all the primitives that are taken. However, we would still need to make the extremely troubling assumption that all agents can see the intrinsic reward that accrues to any agent. Even with these assumptions, it remains problematic in the multi-agent setting to require that agents access some central Q_V table, particularly as agents may be widely distributed and transmissions costly.

Rather than make these assumptions, which would diminish the systems viability in a

distributed setting, we propose two alternatives that are more compatible with our goals. The first and most straightforward approach is to simply index Q_V with a tuple for agent-state-action. MAXQQ, ALispQ, and the pseudo-code given for EHQ all index Q_C in this manner (Dietterich, 2000a; Andre and Russell, 2002). This amounts to each agent learning private values for Q_V and Q_C. The second approach to consider is sharing non-exit-state Q_V values among agents which implement different macroactions. For example, consider an implementation of EHQ with 2 agents implementing `Navigate(source)` and 2 agents implementing `Navigate(destination)`. The first `Navigate(source)` agent and the first `Navigate(destination)` agent both use the same Q_V table, just as the second implementor of each subtask would similarly share Q_V tables. From the economic perspective, this could be justified as firms that offer multiple related products and use knowledge learned when building one product to more effectively produce the other products. In effect, the same firm is competing in both the market for `Navigate(source)` solutions and `Navigate(destination)` solutions. Both of these approaches would resolve the problems of recursive decomposition described earlier in this chapter.

We select the first approach, leaving the second as a possible future extension to this work, as it may yield faster convergence in practice. The EHQ algorithm therefore stores Q_V locally for *all* states, using the notation $Q_V\left(agent, s, a\right)$ in contrast to $Q_V\left(s, a\right)$. Since EHQ similarly indexes $Q_C\left(agent, s, a\right)$, each agent can be considered to have private knowledge of its own Q_V and Q_C tables. The difference between agents' private knowledge, derived from their experiences acting in the world, leads to agents implementing the same subtask to bid different amounts until the system converges.

4.6 Hierarchically Optimal Economic Q-Learning

This section presents the EHQ algorithm, which closely follows rEHQ with the addition of the modifications of exit-state subsidies, and the additional modifications that are necessary

to give hierarchical optimal convergence using these subsidies. Figure 4.4 provides a diagram of the subsidy transfer in EHQ, along with the 4.1. We provide the following pseudo-code for EHQ algorithm:

EHQ(agent g implementing subtask i,$EXIT\_STATE\_SUBSIDIES$)
while T_i is not satisfied **do**
  observe current state s
  select an action a from A_i according to exploration policy π_x
  if a is a primitive action **then**
    take action a and observe one-step reward r
    $N \Leftarrow 1$
  else
    take bids form all agents that implement a
    select winning agent *child* according to auction policy
    EHQ(*child*, DEFINE_SUBSIDIES(g,*child*))
    observe underlying exit-state e
    observe the number of time steps N elapsed during g
    $r \Leftarrow BID(child, s) - \gamma^N \cdot SUBSIDY(g, e)$
  end if
  observe resulting state s'
  if T_i is satsified in s' **then**
    $r \Leftarrow r + EXIT\_STATE\_SUBSIDIES(s')$
  end if
  $Q_V(g, s, a) \Leftarrow (1 - \alpha_t) \cdot Q_V(g, s, a) + \alpha_t \cdot r$

$$Q_C(g, s, a) \Leftarrow (1 - \alpha_t) \cdot Q_C(g, s, a) + \alpha_t \cdot \gamma^N \cdot \max_{a' \in A_i} \left[Q_V\left(s', a'\right) + Q_C\left(g, s', a'\right) \right]$$

end while

$DEFINE\_SUBSIDIES(parent, child)$ is a sub-routine that assigns subsidies to the exit-states, in accordance with the equations provided in the section 4.3. A detailed discussion of how subsidies are defined over the child's set of concrete exit-states can be found in that section.

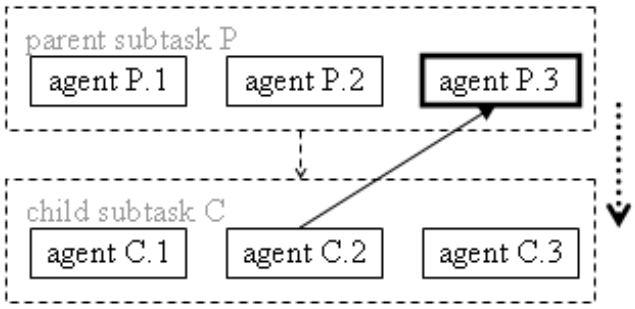

Figure 4.4: EHQ phase 3: The child agent passes control back to the agent that invoked it and receives the appropriate exit-state subsidy.

4.7 Economic motivation

In economic terms, we describe the interaction between parent and child as the child bidding for a contract. The child's bid is a promise to return the world in a state of the world that satisfies its termination predicate if it is selected as the winner of the auction. In exchange, the parent transfers the amount of bid (which may be negative) to the child. Additionally, the parent can provide a subsidy transfer to the child to alter the child's perception of the cost of its exit-states. The amount of this subsidy is promised when control is transferred to the child, but the actual transfer of the subsidy occurs when the child returns.

In this light, the child's V^* should not be thought of as its expected cumulative reward, but rather its expected cumulative reward of all the actions below it, plus any subsidy transfer it receives. When all agents' have converged, the transfer of the bid amount between the child and the parent cancels out the the expected gain or loss of the child, leaving it with a net reward of 0. Even if their V^* is negative, the children do not necessarily have a net loss of utility, as the parent transfers exactly the amount that will compensate the child for any losses in equilibrium. Similarly, in equilibrium the children are forced bid their true V^*, so if they expect to gain utility, this excess utility is transferred to the parent, again leaving them with a net gain of 0.

This is necessary, as since we consider agents in this setting to be independent and self-interested, they should be voluntarily participating in the system. Thus, if they have an expected utility of 0 for participating in the system, the `Root` node, which is the only agent that can realize a non-zero net utility, can distribute some small amount ϵ to them to get them to participate. This requires us to assume that the agents to have no opportunity costs, so their expected reward of not participating in the system is 0 in every time step. If the agents do have alternatives which have positive expected value, the `Root` node would have to ensure that utility is distributed such that that participating in the system has a higher expected utility than not participating. This may require a more substantial redistribution of utility from the `Root` to its descendents, which is a non-trivial proposition. We do not make claims about such a setting, leaving it for future work.

Each agent in the EHQ algorithm requires only a limited view of the hierarchy - specifically, its own termination predicate, its own action space, including non-primitive children, and the termination predicates of each of its non-primitive children. Depending on how the system is specified, EHQ may also require knowledge of which agents implement its particular child subtasks, but this is rather trivial. Knowledge of the agents own predicate and action space is also quite reasonable. The only point of concern it that knowledge termination predicates of the children is required to subsidize exit-states appropriately. We claim that such knowledge is quite reasonable in the economic context of this paper. In effect, a termination predicate is part of the contract between parent and child, in which the child agrees to take control of the world for a price, and return control of the world in a state that satisfies the predicate. The parent agent solicits bids to determine which agent to give this contract to. Therefore, it seems quite reasonable that an agent should have complete knowledge of what contracts it can offer and the specification of those contracts.

The way EHQ incorporates subsidies into the contract is somewhat problematic. While EHQ requires that the parent give the child subsidies it promises when it invokes the child, the child directly incorporates these subsidies into its belief about its value function. The

child agent therefore does not know what the subsidies should be, but only has some sense of their expected value. It also must be possible for the subsidies to change from one instance to the next. Therefore, the child agent cannot reason about how the new set of subsidies have changed from the previous instance or how the change should effect its bid. This creates a situation that a strategic parent agent could exploit when it sets subsidies.

During the convergence of the algorithm, the agents may gain or lose utility in practice, even though they will have a net of zero once the algorithm has converged to equilibrium. The direction of their net utility during the convergence depends on the difference between their beliefs about the costs they face and the true values of those costs. If their prior beliefs are an underestimate, they will continually bid too low and lose money to the parent until they converge to the correct value. Conversely, if their prior beliefs are an overestimate, they will continually bid too much and receive a net gain from the parent. The competitive nature of the system will minimize the ability of agents to exploit these conditions - any agent that is clearly over estimating its costs will generally lose the auction to agents with more reasonable beliefs.

4.8 Safe hierarchical state abstraction

The precise rules that allow for safe state abstraction depend on the particular value decomposition used and the form of the hierarchy. Dietterich (2000c) and Marthi et al. (2006) each lay out conditions for safe state abstraction under their respective decompositions. In general, some state abstractions might be sufficient to allow an algorithm to converge to a recursively optimal policy, but not for a hierarchically optimal policy.

Definition 2.6.2 provides a rather strong notion of safe abstraction, as it is not dependent on either the hierarchy or the value decomposition. In practice there may be abstractions that are not safe under definition 2.6.2 but will do no harm to the convergence of MAXQQ or HOCQ. To formalize this claim, we provide the following definitions:

Definition 4.8.1. A state abstraction is *hierarchically safe* given a hierarchy H if the hierarchically optimal policy in the original state space is hierarchically optimal in the abstract space.

Definition 4.8.2. A state abstraction is *recursively safe* given a hierarchy H if the recursively optimal policy in the original state space is recursively optimal in the abstract space.

These three notions of safe abstractions parallel the three definitions of optimality from section 3.1. These definitions would be equivalent if three types optimal policies were also equivalent. We mention the distinction only because when using an algorithm that will not converge to a hierarchically optimal policy (MAXQQ or rEHQ) it makes little sense to use a hierarhically safe abstraction. The more concise recursively safe abstraction will allow faster convergence without degrading the quality of the learned policy. Hierarchically safe is a stronger definition than recursively safe, and safe abstraction as defined in 2.6.2 is stronger still.

If MAXQQ is using pseudo-reward (which could allow a hierarchically optimal convergence), a recursively safe abstraction may no longer be sufficient, as the pseudo-reward may depend on variables that were abstracted away. For instance, if the taxi in the TaxiFuel domain was only penalized for running out of fuel at when it drops off the passenger, the fuel cost would be irrelevant for recursive optimal policy in the `Navigate(n)` subtasks - neither the Q_V or Q_C values of either subtask would depend on the amount of fuel. This would be sufficient for the recursively optimal policy, as that policy would dictate that `Navigate(n)` agents should never invoke the `Refuel` subtask; this is a recursively safe abstraction. If such an abstraction were used by these nodes under the EHQ algorithm (or MAXQQ with pseudo-reward), the agents implementing `Navigate(n)` would not understand the difference between a full gas tank and an empty gas tank. Therefore, even if the parent assigned a higher subsidy for a full gas tank, this subsidy would be a purely stochastic exit-reward

from the child's perspective, independent of any variable of which it is aware. Therefore, the child would be unable to learn and reason about the higher reward exit-state and EHQ would be doomed to recursive optimality.

To get hierarhical optimality in EHQ or MAXQQ, it is necessary to give a subsidy or pseudo-reward to `Navigate(source)` for a high fuel level in some states, as it will be hierarchically optimal for `(Navigate(source)` to refuel when it passes right by the refueling point. The abstractions that were safe without this subsidy or pseudo-reward are no longer safe once it is introduced. The distribution pseudo-reward or subsidy must be considered along with $R(s'|s,a)$ in the safe abstraction conditions given in section 2.6 in order to give a hierarchically safe abstraction. In practice, EHQ can use the same abstraction that is safe for MAXQQ with pseudo-rewards that will induce hierarchically optimal convergence. The need to find a hierarchically safe abstraction is somewhat troubling, as it requires reasoning about what aspects of the state the subsidies will depend on. However, a hierarchically safe abstraction is clearly necessary for any algorithm to converge in general to a hierachically optimal policy.

Note the the concept of recursively safe abstraction makes little sense for the ALisp decomposition, as such an abstraction could completely abstract the statespace for Q_E, effectively turning the ALisp decomposition into the MAXQ decomposition. Since the algorithms using the ALisp decomposition do not rely on pseudo-reward or subsidies, they do not introduce additional dependencies to $R(s'|s,a)$, allowing them to have coarser abstractions over Q_V and Q_C than MAXQ. Furthermore, the abstractions that are recursively safe for ALisp for Q_V and Q_C are also hierarchically safe. The difference between the two concepts is entirely in the number of states for which Q_E must be representing - from 1 for recurisvely safe to much of the state space for hierarchically safe.

4.9 Theoretical convergence of HRL algorithms

If the state abstractions used are recursively safe for the hierarchy, Dietterich (2000a) provides proof that MAXQQ will converge to a recursively optimal policy. That proof requires an inductive structure, proving the claim that if a node's descendents have converged, the transition distribution for the resulting state and elapsed number of time steps, $P(s', N|s, j)$ and the reward distribution $R(s', N|s, j)$ for each descendant M_j will be stationary. If $P(s', N|s, j)$ is stationary, $R(s', N|s, j)$ must be stationary given our framework. If these distributions are stationary, they can be learned through the standard stochastic approximation algorithms, allowing MAXQ decomposition values to converge to the optimal value function given $P(s', N|s, j)$ and $R(s', N|s, j)$ for all its descendents. Since $P(s', N|s, j)$ must have converged, any learning the parent does about the effects of j on the rest of the problem will not alter $P(s', N|s, j)$ or $R(s', N|s, j)$. This limits MAXQQ to recursively optimal convergence.

The base case for the MAXQQ convergence proof is the case of a subtask i that has only primitive nodes as children. By the definition of a MDP, primitive actions must have stationary transition and reward distributions, which can therefore be learned. More formally, if j is a primitive node, $P(s', N|s, j)$ and $R(s', N|s, j)$ must be stationary distributions, as they are defined to be the stationary distributions $P(s'|s, a)$ and $R(s'|s, a)$ in the MDP. A node that has all primitive children will therefore converge. Note that the MAXQ framework allows for pseudo-reward functions $R_i(s'|s, a)$ for any subtask M_i, but these functions must be stationary.

All of these claims of stationarity hold for rEHQ, in which the parent's learning cannot effect the transition or reward distributions. This allows for the formal proof of convergence given in section 4.2. In EHQ, there is no such analogous base case, due to the non-stationary distribution of exit-state subsidies. Therefore, proving the convergence of EHQ to a hierarchically optimal policy by induction up the hierarchy in this manner does

not seem promising. In EHQ, $R(s', N|s, j) = BID(child, s) - SUBSIDY(parent, s')$, where g is an agent that implements subtask M_j. Since the amount of the subsidy must be propagated back into the bid and the subsidy depends on the value function of the parent, $P(s', N|s, j)$ is not stationary until both the parent and g have converged. It follows that since the exploration policy used by g relies on g's value function, $P(s', N|s, j)$ is not stationary until the parent and g have both converged. We are therefore left with a disturbing circularity that does not bode well for a formal proof of convergence.

The various HRL algorithms that use the ALisp decomposition also lack convergence proofs. A convergence proof for HOCQ may encounter the similar difficulties to EHQ - namely, that the expected exit value is non-stationary. The HOCQ case seems a little better, as while in EHQ the non-stationary propagates back into the Q_C values, HOCQ isolates it in the Q_E values. It thus seems possible to prove that the Q_V and Q_C values of HOCQ will converge appropriately for the base case in a similar manner to MAXQQ. A typical convergence proof would require that the exploration policy of the child must be greedy with respect to the child's value function in the limit. Therefore, the child must use Q_E in its reasoning. Since $Q_E(j, a, s') = \sum_{e \in E_j} P_E(j, e|s, a) \cdot V^*(i, e)$, Q_E of the child is dependent on the value function of the parent, and so the value function of the child is dependent on the value function of the parent. The child cannot converge until the parent has converged, preventing us from making an inductive argument from the bottom of the hierarchy up. Since the child has not converged, $P(s', N|s, j)$ for the parent is not stationary, so the parent cannot converge. This circularity seems to preclude this form of inductive proof of convergence for any algorithm using the ALisp decomposition.

The only possibility for a formal convergence proof of EHQ or an Alisp algorithm would seem to be some form of induction on the history of subtasks, rather than induction on the hierarchy. This follows from the observation that, if the child is never invoked again, ALisp's Q_E or EHQ's subsidies are independent of the child's value function. We now provide a sketch of how such a proof might work, but do not claim it as a formal result or even to

have completely explored its technical requirements.

The base case is the last subtask in the typical history, for which Q_E is 0 and the subsidies would be 0. For ALisp, Q_E is defined to be the expected reward after exiting the current subtask, which is quite clearly 0 for the last subtask. For EHQ subsidies, the last subtask would lead to a terminal state for the `Root` node and every intermediate node up the hierarchy. No macroaction has a value function defined for states in which it is terminated, so the subsidies are all 0. Therefore, both of these are stationary, and so the last subtask in the history will converge.

To prove the recursive case, it must be argued that if all the subtasks invoked after a given subtask have converged, the given subtask will converge as well. Since the subtasks that follow it have converged, it can be argued that the relevant portion of the value functions for the subtasks above the given subtask in the hierarchy, in which these subtasks will operate after the given subtask exits, will converge as well. This implies that exit-state subsidies will have converged, which would be sufficient for EHQ to converge. It also implies that the $V^*(i, e)$ component of Q_E will have converged, and thus ALisp can converge.

The obviously troubling requirement is the necessity that a subtask never be called twice in the history. This requirement avoids the general case of $V^*(i, e)$ for one of the subtask's exit-states e would circularly depend on the subtask's own value function. Although troubling, this requirement is met in practice by the Taxi and HO1 domains presented in this paper. It can even be argued that it is not a problem for TaxiFuel, which although it may invoke `Refuel` multiple times, the `Refuel` task's transition probability $P(s', N|s, Refuel)$ does not depend any the exit-state value (Q_E or an exit-state subsidy). This follows from the observation that for any state in which `Refuel` is invoked, there is only one possible exit-state `Refuel` can reach. This is sufficient for HOCQ using ALisp to claim that $R(s', N|s, Refuel)$ will converge as well, but is more difficult to argue for EHQ, as the non-stationary exit-reward must be propagated through into the bid for $R(s', N|s, Refuel)$ to converge. While the requirement of not invoking a subtask twice may seem reasonable

in many episodic domains, it seems wholly unlikely to be reasonable for any discounted, infinite horizon domain. At best, a proof of the type described in this section would allow a claim of convergence to a hierarchically optimal policy for HOCQ and EHQ in episodic domains that meet particular requirements on the subtask history of optimal policies.

Notably, HAMQ is proven to converge to a hierarchically optimal policy (Parr and Russell, 1998). However, HAMQ does not use value decomposition (and therefore has very limited capacity for state abstraction). The proof of HAMQ's convergence relies on this fact by using the reasoning that the HAM H acts as a contraint on the available actions and reachable states in the MDP M. Therefore, the HAM H is effectively composed with the MDP to create a new MDP $H \circ M$. This composed MDP completely fits the necessities of the definition of an MDP, with stationary reward and transition distributions. Therefore, simply applying Q-learning to it, which is essentially what HAMQ does, will yield convergence to an optimal policy for $H \circ M$ just as Q-learning would for any MDP. Since this policy will be optimal for $H \circ M$ rather than M, it is hierarchically optimal rather than globally optimal.

4.10 Summary

This chapter provided an overview of the rEHQ algorithm and proved its convergence to a recursively optimal solution policy under the same conditions as MAXQQ. The rEHQ proof was followed by the extension of exit-state subsidies, showing how the use of these subsidies might lead to hierarchically optimal solution policies. With respect to these subsidies, we discussed the issues of recursive decomposition and decentralization and proposed necessary modifications to the rEHQ algorithm to make it well-motivated in this paper's economic context. We then provide a specification of the complete EHQ algorithm, following by a detailed discussion of its economic motivation and properties. We introduced the concepts of hierarchically and recursively safe state abstractions, showing how different abstractions can

limit the potential quality of a policy that the EHQ algorithm might converge to. Finally, we discussed the difficulties of providing a formal convergence proof for EHQ, arguing that other HRL algorithms using value decomposition have not been proven to converge to hierarchically optimal policies because of the same circularity issue faced by EHQ.

In this chapter, we demonstrated how EHQ might allow a hierarchically optimal value function to be represented with the MAXQ value decomposition. In the following chapter, we provide evidence to show that EHQ can converge to a hierarchically optimal policy in practice.

Chapter 5

Experimental results

This chapter presents a comparison of various versions of the EHQ algorithm with MAXQQ learning and flat Q-learning. The empirical results represent data from simulations on the two motivating examples provided in section 1.1. HRL literature evaluates convergence speed in terms of number of primitive actions, and this convention is followed in the results of this work. This metric can be thought of as which algorithms derive the most knowledge about the world per unit of knowledge taken from it. Dietterich (2000a), Andre and Russell (2002), and Marthi et al. (2006) all present results evaluating their algorithms by this metric. The justification for this is that in practice, HRL algorithms are designed to be used to learn real-world problems, for which it is impractical or expensive to experiment on by trying millions of primitive actions.

Note that most HRL algorithms, like traditional Q-learning, have learning rates and exploration policies that must be tuned to maximize convergence speed for a given domain. All results presented in this chapter were tuned to improve convergence speed, although we do not claim that the particular parameters we used maximize convergence speed for any particular algorithm - we merely made a reasonable effort to achieve that end.

Our implementation was written in C++ and included versions of flat-Q learning, MAXQQ

learning, and EHQ learning. The learning rate α_t for flat-Q, MAXQQ, and EHQ is cooled according to the expression $1/\left(1 + t/C_\alpha\right)$, where t is the number of time steps elapsed since the algorithm was initiated and C_α is a tuning parameter.

All three algorithms used epsilon-greedy exploration policies. We defined the exploration policies in our implementations of MAXQQ and EHQ to consider only *applicable* subtasks - those which are not terminated in the in present state. We similarly limit the set of actions over which we maximize the value function when calculating V^* to only applicable actions, to account for the fact that these actions cannot be taken in the particular state. In our experiments, we define the MDP such that all primitives are can executed in every state; it is left to the hierarchy restrict the available actions, if necessary.

Similarly to α, the value ϵ used by is updated in to the value of the expression $1/\left(1 + t/C_\epsilon\right)$, where again t is the number of time steps and C_ϵ is a tuning parameter. The particular values of the parameters will be noted where we present specific experimental results. Both and α are updated in each time-step. The EHQ algorithm also uses an epsilon-greedy policy to select the winner of the auction, as noted in section 4.2. The ϵ value in our implementation is shared by both the exploration and auctions policies, those this certainly need not be the case.

Our implementation used significant state abstraction, but did not use all state abstractions possible for a given domain. In practice, state abstraction is difficult to express and encode for large problem instances - far more difficult than creating the hierarchy itself.

Our implementation initializes all Q_V and Q_C values to be 0. As with general Q-learning, this is a somewhat arbitrary choice. However, we believe that it is important that they be initialized uniformly, rather than randomized. Since the subsidies are defined as the marginal value over the set of exit-states, this convention will lead to the subsidies starting low (at 0) and growing as the algorithm converges. While we provide no formal claim or evidence to support the benefit of this convention, it seems that it should be preferable to

having subsidies begin at arbitrarily high values.

5.1 Hierarchically optimal convergence

To show the capacity of EHQ to yield a hierarchically optimal solution policy, we created the HO1 domain for experimental simulation. The domain is a variation on the earlier example of a domain with distinct hierarchically optimal and recursively optimal policies (see section 3.1). The grid-world for this domain is depicted in Figure 5.1. We use the same hierarchy given in Figure 3.3. To exaggerate the difference between the recursively optimal and hierarchically optimal policies, each movement primitive gives reward of -1 in the left room and -2 in the right room. Successfully reaching the goal gives a reward of 10. Colliding with a wall is a no-op with a reward of -10. Additionally, we treat the doors as one-way passages so that `Reach goal` will only terminate in the goal state; this prevents the recursively optimal policy from exiting the right room back out the lower door. Such behavior would create an infinite loop, as it would be recursively optimal for `Reach goal` to exit through the lower door, resulting in `Root` calling `Leave goal`, which would leave through the lower door, returning to the original state. If `Reach goal` was allowed to operate in the left room, the recursively optimal and hierarchically optimal policies would be equivalent: `Root` would invoke `Reach goal` in every state, which would effectively operate as a flat Q-learner.

As the state space for this domain is extremely small, convergence speed is uninteresting - even flat Q-learning is reasonably effective. We instead present experimental results in Figure 5.2 that clearly show the superior solution quality yielded by EHQ ($C_\epsilon = 50$, $C_\alpha = 100$) over MAXQQ ($C_\epsilon = 50$, $C_\alpha = 100$). Solution quality is averaged over the last 10 episodes. MAXQQ converges to an average of -6, the cumulative reward of the recursively optimal policy. EHQ converges to an average of 3, the cumulative reward of the hierarchically optimal policy. The noise seen in the plots is caused by the exploration policy.

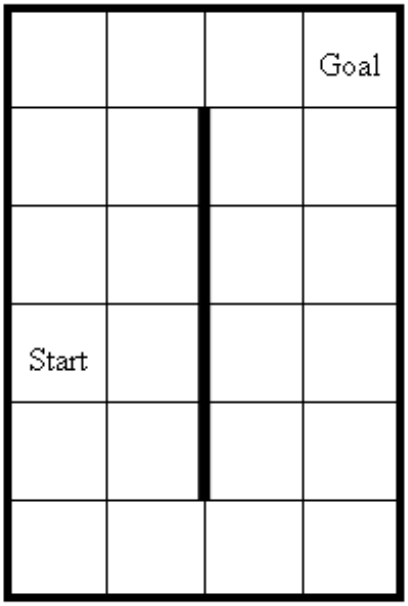

Figure 5.1: The grid world for HO1 domain, a variation of Dietterich's hierarchically optimal example domain from Figure 3.2.

Negligible state abstraction was used. EHQ's slower convergence can be attributed to the replication of the Q-values, as the implementation uses two EHQ agents implementing each of the `Leave left room` and `Reach goal` subtasks. EHQ therefore faced an effectively larger state-space.

EHQ's convergence to a higher quality policy can be attributed to its ability to propagate information about the right room to the agents acting in the left room using exit-state subsidies. The plot of convergence of the values of exit-state subsidies is shown in Figure 5.3 ($C_\epsilon = 50$, $C_\alpha = 100$). We graph both the exit-state that can be reached through the upper door and the one reached through the lower door. While these are the only two exit-states that `Leave left room` can reach in practice, its termination predicate is defined such that all the states in the right-hand room can be considered to be exit-states, and thus must be assigned subsidies. We plotted the subsidies of two unreachable exit-states, to underscore their presence and show that they are handled appropriately.

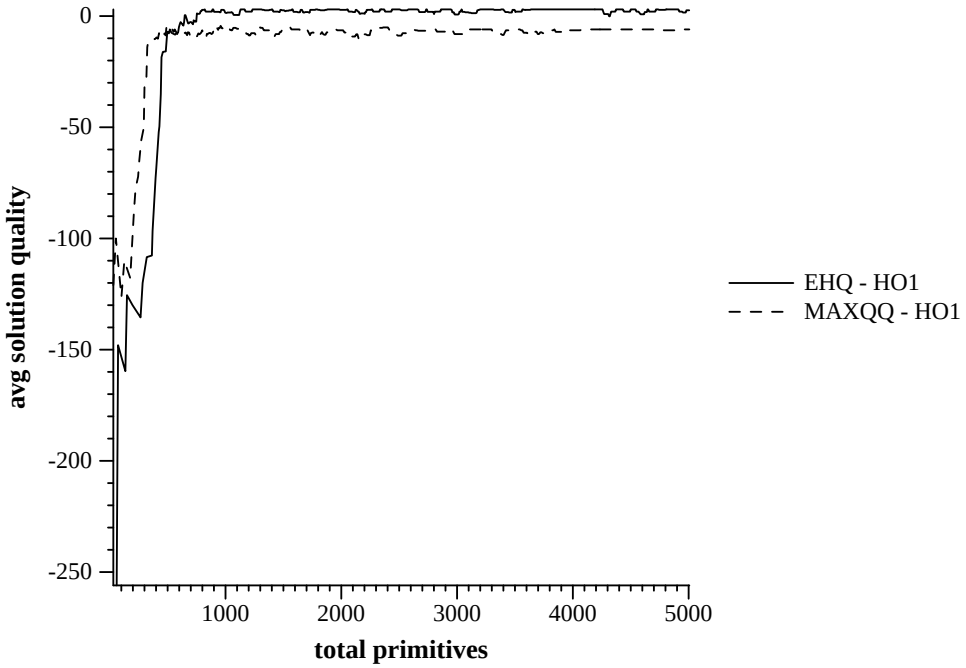

Figure 5.2: Results of experimental trials on the HO1 domain, showing EHQ's convergence to a hierarchically optimal policy.

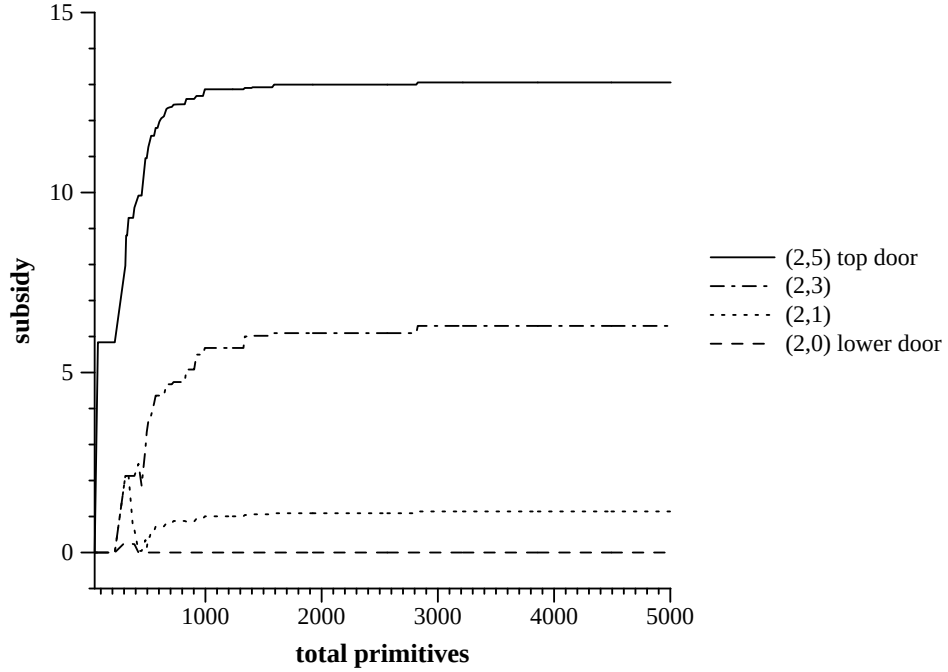

Figure 5.3: Exit-state subsidy convergence in EHQ algorithm on HO1 domain for selected exit-states of the `Leave left room` subtask.

Note that the exit-state subsidy for the upper door is larger than the cumulative intrinsic reward that the `Root` agent can expect to get. While this seems unusual, it is does not introduce a problem, as the subsidy value propagates back through the Q-values of the agents implementing `Leave left room`. The value is therefore canceled out once the algorithm converges as the subsidy is subtracted from the bid that the parent receives.

It is also worth noting that the subsidy has not converged to quite the correct values, which should be as shown in Figure 5.4. If `Root` had perfect knowledge, the marginal value of the upper door over the lower door would be 10, as shown in Figure 5.1. This is likely caused by the learning parameters being optimized for policy convergence speed, not for convergence to the optimal subsidies. Regions of the state space that the algorithm quickly learned were unpromising were not fully-explored. The algorithm learned that the lower door was not worth exploring and therefore the surrounding region of the state space was not experienced properly, causing the Q-values to underestimate the values of these exit-states.

Having demonstrated hierarchical convergence in a very simple domain (with only 24 states), we now introduce a slightly more complex version of the domain called HOFuel. We use the same grid world from Figure 5.1, but introduce a fuel constraint. The taxi begins with a fuel level of 5. Each move decreases the fuel level by 1, unless the fuel level is already 0. If the fuel level is 0, the move occurs but there is an additional penalty of -10. HOFuel has an additional `fill up` primitive that returns the fuel level to 5, but this may only be executed in the left room. All primitive actions have the usual reward of -1.

The recursively optimal policy for this domain is to leave through the lower door without refueling. This dooms the taxi to receiving considerable penalties in the right room. The hierarchically optimal policy, in addition to exiting through the upper door, should refuel if there are less than 2 units of fuel remaining. This avoids the `Reach goal` agent from incurring the large penalties. The reward of 10 for reaching the goal is still given. There is clearly a substantial difference between the quality of these solutions. HOFuel has 144

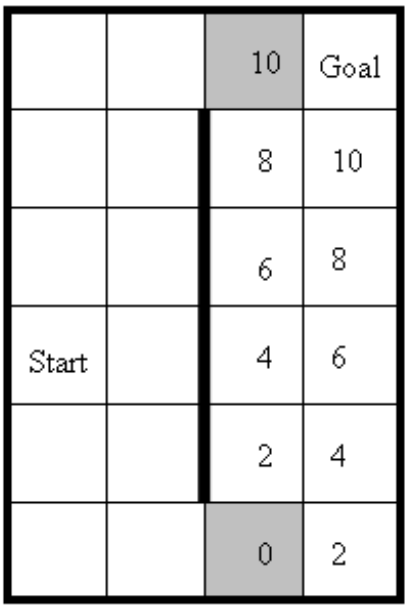

Figure 5.4: The grid world for HO1 domain, with exit-states subsidies marked for all exit-states of `Leave left room` and the two reachable exit-states shaded.

states and 5 actions. A hierarchy for HOFuel is depicted in Figure 5.5. Results shown in Figure 5.6 clearly show that flat-Q ($C_\epsilon = 400$, $C_\alpha = 1500$) and EHQ ($C_\epsilon = 400$, $C_\alpha = 2200$) converge to hierarchically optimal policies, while MAXQQ ($C_\epsilon = 300$, $C_\alpha = 1800$) and rEHQ ($C_\epsilon = 400$, $C_\alpha = 2200$) converge to recursively optimal policies. The plot shows average reward over the last 30 trials.

The exit-state subsidy plot for HOFuel ($C_\epsilon = 400$, $C_\alpha = 2200$) shown as Figure 5.7 again provides evidence of the value of exit-state subsidies. The values of the subsidies are quite large, reflecting how comparitively poor the worst possible exit-state is. These large values are not a problem, as the `Root` subtracts the exit-state subsidy it gives to `Leave left room` from its value for the reward. In this example, it appears that these values must be propogating through into the bid of the `Leave left room` agents, allowing for them to be canceled out appropriately so `Root` can reason about the true costs of its actions. The exit-state subsidy plots also show that the driving force is not the algorithm learning that

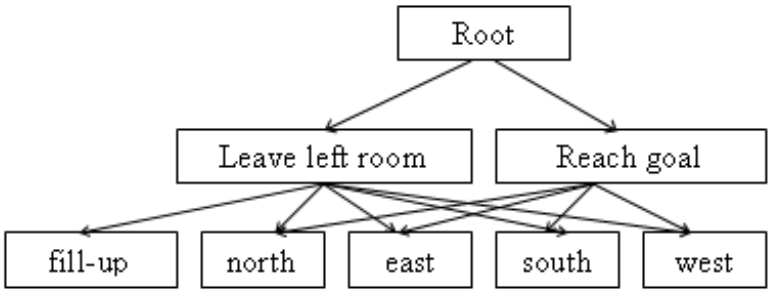

Figure 5.5: A hierarchy for the HOFuel domain. Note that the fill-up primitive is only available in the Leave left room subtask.

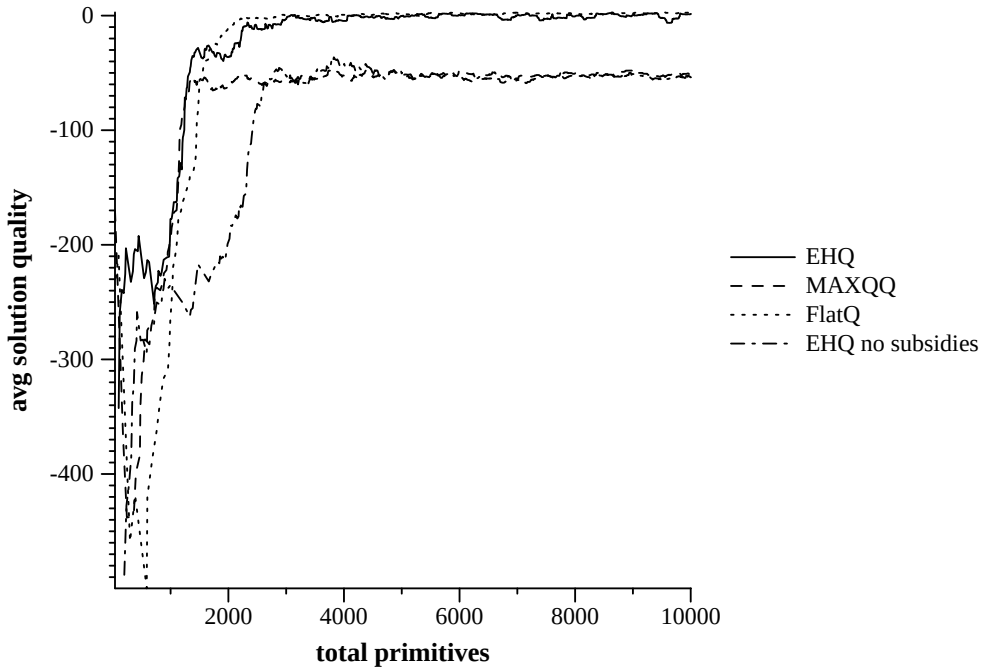

Figure 5.6: Results on HOFuel, showing EHQ and FlatQ achieving significantly better policies than MAXQQ and EHQ without subsidies.

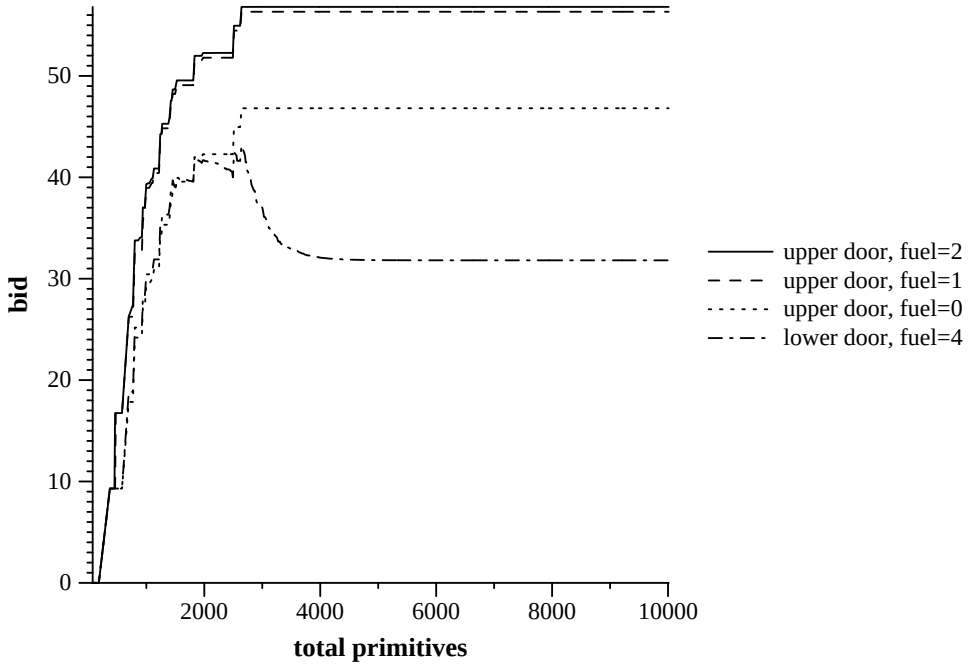

Figure 5.7: HOFuel exit-state subsidy convergence for EHQ.

certain exit-states are better, but that the worst case is worse. This is why we see a high degree of coordination in the movement of the exit-state subsidies.

Finally, we present several plots showing EHQ on the Taxi domain from section 1.1. This domains larger state space (500 states) proved difficult for EHQ to handle, although MAXQQ was effective. The growth in values of the tuning parameters from the HO1 domain of 24 states to the HOFuel domain illustrates the difficulty involved - both C_α and C_ϵ increased substantially. Figure 5.8 ($C_\epsilon = 400$, $C_\alpha = 2200$) shows the difficulty in convergence for EHQ. As time runs longer, oscillations such as those seen in Figure 5.8 get larger, eventually dragging down the average despite a seeming upward trend. While averaging over a larger window of previous trials can hide these oscillations and expose a general upward trend, numerous attempts to do this showed that the oscillations simply continued to grow until the troughs had sufficient magnitude to dominate and depress the average. The period of these oscillations seems to grow progressively longer. Reasons for

this are not entirely clear, but it may be related to the epsilon greedy exploration and auction policies. When ϵ gets very low, the odds of control reaching a poorly explored part of the state space or under-experienced agent are also very low. However, when such a rare event does occur, it may be difficult escape. With a low learning rate, an unexperienced agent may get stuck taking the same pointless actions thousands of times before it learns that it must use an alternative. Randomly breaking out of such a loop is also difficult, as ϵ is low. The result of such a situation would be a few extremely bad episodes, which would get progressively more rare yet progressively more catastrophic. Something along these lines may be happening in Figure 5.8.

Larger values for the learning rates, while clearly necessary, did not greatly improve the situation. Figure 5.9 illustrates this on a much longer time scale, with $C_\epsilon = 800$ and $C_\alpha = 15000$, averaged over the last 50 episodes. Larger values of C_ϵ appeared to be very determental in practice. Since there is no noise in the Taxi domain, the only reason to have a large exploration rate is the potential to discover lucrative exit subsidies that have only recently appear. From Figure 5.9, it seems clear that one very bad, very long episode can severely impact the average reward. On a long time scale, the random occurence of such an episode is quite likely. Cooling the ϵ for the auction policy more slowly might improve the situation, as such a jump is most likely responsible for landing the system in a place where it has little experience yet already low exploration and learning rates.

Clear modifications to EHQ are necessary for it to scale effectively. One option that may specifically address the difficulty in Figure 5.8 would be to cool α_t and ϵ_t on a per agent basis, using the number of time steps experienced by the agent rather than the system overall. This is discussed further in section 6.6. Due to the large amount of replication and the need to propogate significant amounts of information through the system, far more work must be done to improve EHQ's convergence in practice.

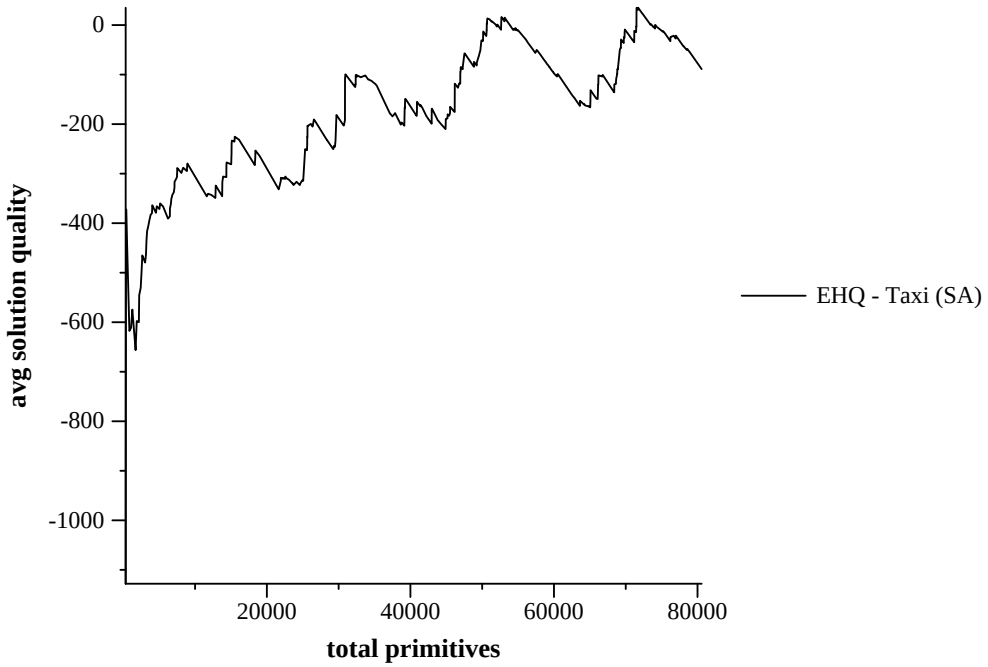

Figure 5.8: EHQ plot for the Taxi domain.

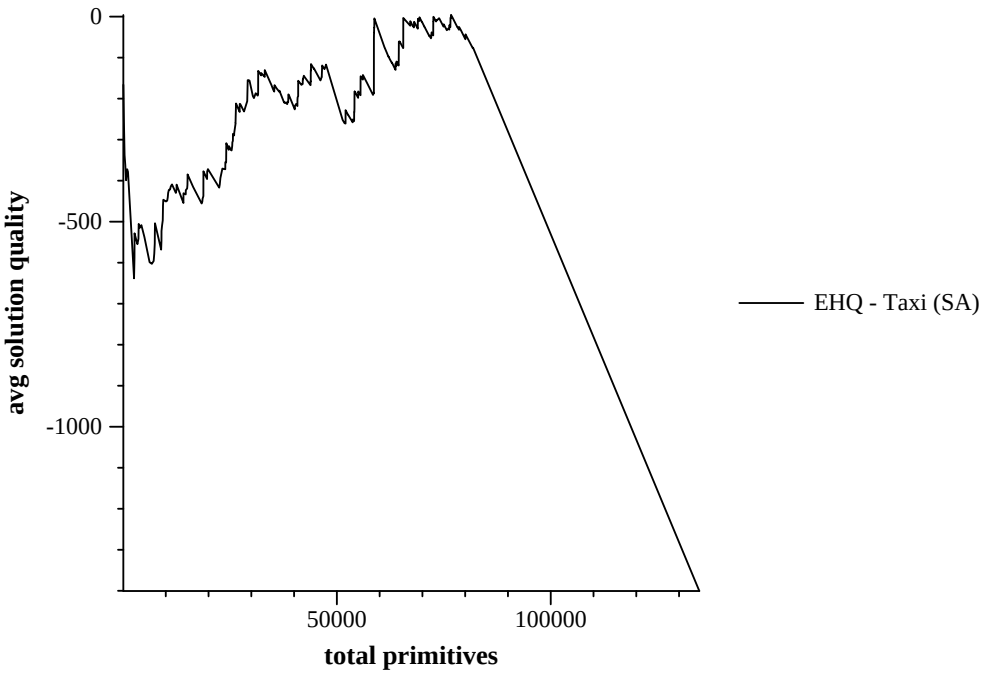

Figure 5.9: EHQ plot for the Taxi domain, with larger values for learning parameters.

5.2 Delayed use of subsidies

Initially, we speculated that allowing EHQ to learn as rEHQ does and then turn subsidies on after some number of episodes might improve convergence. In practice, this approach does quite poorly. This is likely because turning subsidies on later, when the learning rate is lower, makes them slow to propagate back through the system. Turning subsidies on causes a sudden shift in the distribution of both the exit-state reward function for macroactions and the value function of a parent for its non-primitive children. This is because of the sudden subtraction of the exit-state reward the parent gives the child from the child's bid. The later this is done, the lower the learning rate is, and the slower that these shifts can be learned and propagate back through the completion values. In the version of EHQ presented in this paper, it seems best in general to always use subsidies. Experimental results that illustrate the effects of turning on subsidies after some number of time steps are shown in Figure 5.10 for HO1($C_\epsilon = 50$, $C_\alpha = 100$) and Figure 5.11 for HOFuel ($C_\epsilon = 400$, $C_\alpha = 2200$). The deteriment caused by turning on subsidies later does not appear extremely strong for either of these small domains, but did seem to cause the system to tend closer to the recursively optimal policy in HOFuel. Once subsidies are turned on, it moves slowly towards the hierarchically optimal policy.

A somewhat analogous problem can occur in practice when the parent introduces a large exit-state subsidy relatively late in the trial. The slow propagation of exit-state rewards back through the system requires that the exploration policy of the child take it to towards the highly valued exit-state several times. If the exploration rate has cooled substantially, then it will take a very long time for the child to have explored sufficiently to have propagated the information about the higher exit-state subsidy all the way back through its table. Furthermore, the lower the learning rate the more visits to an exit-state and the intermediate states that will be required before the new exit-state subsidy will back up all the way into the agent's bid. Situations such as these appear to greatly harm EHQ's

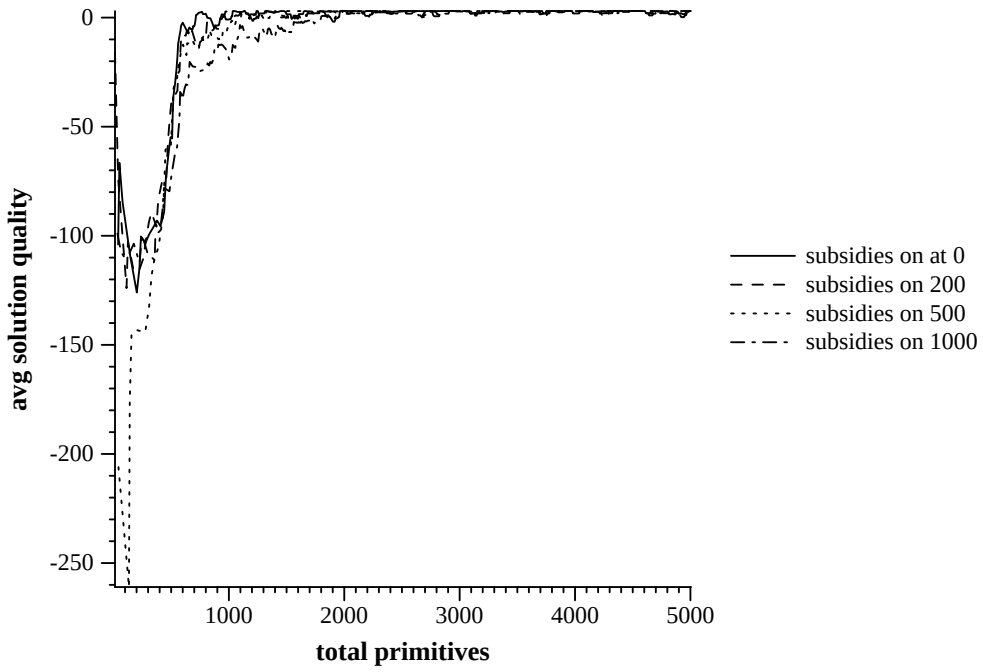

Figure 5.10: Results on HO1, turning subsidies on after various numbers of time steps.

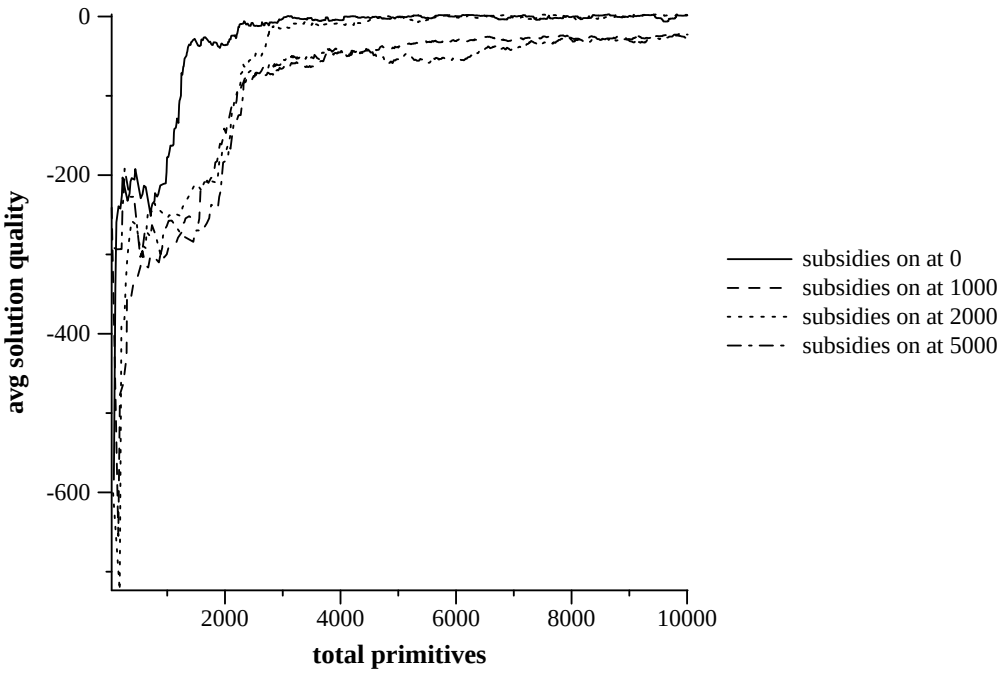

Figure 5.11: Results on HOFuel, turning subsidies on after various numbers of time steps.

practical convergence for large domains.

5.3 Summary

It is clear from the experimental results presented in this section that, through the use of exit-state subsidies, EHQ can achieve hierarchically optimal policies superior to those obtained by MAXQQ. The experimental domains presented in this chapter were too simple to give sufficient insight to the relative convergence speed of these algorithms. In particular, they afforded essentially no opportunity for state abstraction. In further experiments on larger domains, we were unable to get similar convergence for EHQ, but highlighted several aspects of our implementation that may partially account for this.

For future work, it remains to be shown that EHQ can be effectively applied to large domains. This will undoubtedly require significant optimization of the algorithm and introduction of techniques to provide faster convergence in practice. In particular, EHQ agents seem to have difficulty propagating information about changing exit-state values back through their Q-tables, resulting in value information moving very slowly from one side of the system to the other. Similar optimization for practical convergence has been done for MAXQQ and HOCQ, affording significant speed-ups to those algorithms (see section 6.6).

Chapter 6

Discussion and future directions

6.1 Conclusions

The EHQ algorithm provides several innovative contributions, including the use of a hierarchical artificial economy, using market structures to align local and global interests, and use of auctions coupled with differential experiences of agents to provide exploration. Using these technigues, EHQ can converge to hiearchically optimal solution policies in practice. Although the issues surrounding value decomposition, state abstraction, and exit-state subsidies are complex, the EHQ algorithm is conceptually simple. Agents bid their expected reward for solving a sub-problem and the parent passes them control in return for a transfer of the bid amount. The parent can subsidize the possible exit-states of the child in order to get the child to produce an exit-state it prefers. The child will produce such a state if the parent pays it enough to exceed the additional cost the child will incur.

EHQ's simplicity also allows any one agent in the system to require only a limited view of the system. In particular, beyond its own predicate, action space, and state abstraction, an agent need only know the predicates of the subtasks directly below it in the hierarchy. This gives great potential to use this approach in highly distributed or multi-agent settings,

for which HOCQ and MAXQQ are poorly motivated. The market structures make EHQ more robust in such loosely coupled settings, as we model agents as self-interested. Further work with strategic agents (see section 6.7.2) should provide more weight for this claim.

Additional work remains to be done on EHQ to improve its convergence speed and scalability. We were unable to effectively scale the algorithm to large state spaces - one of the motivating goals of HRL. We also failed to provide formal proof of EHQ's convergence to a hierarchically optimal policy. However, we identified the fundamental circularity that undermines the possibility of an inductive proof of convergence for either HOCQ or EHQ of the form used by Dietterich (2000a) for MAXQQ. It seems that EHQ is in fair company in this regard. With further work, EHQ can represent a promising approach for applying HRL tehcnigues to loosely-coupled, distributed systems.

For EHQ to be truly practical in loosely-coupled systems, we must find a way to futher improve independence of an agent from its calling context. This will facillitate the type of subtask reuse that is critical effectively scaling HRL to large domains. We discussthis particular aspect of EHQ in section 6.3 and propose future work to address it in the form of indirect exit-state subsidies (section 6.4) and explicitly modeling the expected exit-state subsidies (section 6.5). The remainder of this paper discusses a number of particular aspects of EHQ in contrast to MAXQQ and HOCQ, with particular attention paid to why EHQ struggles to scale effectively. We suggest a number of extensions that may improve EHQ's performance or would provide interesting insights into the system.

6.2 Comparison Discussion

In ALispQ and HOCQ, all agents act in the interest of the whole system, showing no preference for reward extracted by their own actions over reward extracted somewhere else. These agents effectively behave as self-less automatons, subservient to the `Root` and willing to sacrifice local reward for the gain of the whole system. In MAXQQ, the agents can be

considered to be self-interested, as they act to maximize local reward, but not independent, as they access each others beliefs and value functions. In EHQ's hierarchical artificial economy, subtasks are implemented by independent, self-interested agents.

This leads to another important characteristic of EHQ's value decomposition. Strictly speaking, EHQ follows the value decomposition definitions laid out in section 2.5 from the local perspective of an agent. However, it differs conceptually from both, as Q_V is no longer the expected sum of primitive and pseudo-reward, but includes the expected exit-state subsidy. In this sense, Q_V is no longer strictly local, but may depend on value derived elsewhere in the hierarchy and transferred into this local context. By extension, Q_C incorporates the expected exit-state subsidies as well. In the MAXQQ and ALisp decompositions, these values incorporate only expected reward within the current subtask and its descendents. More discussion of the implications of these observations is provided in section 6.5.

The subsidies used by EHQ serve the same purpose as the subtask's Q_E values would be under the ALisp decomposition, although subsidies are normalized while ALisp's Q_E is not. ALisp's Q_E value incorporates the expected reward after exiting the current node, whereas the EHQ subsidies are actual reward for a particular exit-state.

In ALispQ, HOCQ, and MAXQQ, nodes accumulate primitive reward and simply pass it up the hierarchy. In the hierarchical economy of EHQ, reward moves both up and down the hierarchy and is never a simple aggregation of the accrued primitive reward. This bi-directional flow causes mutual dependence between parent and child on each others' value functions. We found that this causes difficulty in providing a formal proof of convergence (section 4.9).

It should be noted that while MAXQQ is proven to converge to recursively optimal policy, Dietterich (2000a) describes using the hierarchal, decomposed value function learned by MAXQ as a basis for creating a non-hierarchical policy for the MDP. Furthermore, this

value function can be used as the basis of an iterative algorithm for further improving the policy. The results of that algorithm can be used to set new pseudo-reward functions for the MAXQ hierarchy before performing additional rounds of MAXQQ-learning. Dietterich (2000a) empirically demonstrates the rapid convergence of this system towards to a solution policy that is significantly better than the recursively optimal policy in several domains. Such non-hierarchical execution, while compelling, is not compatible with our goals of having a simple, decentralized HRL system or the economic motivations of the EHQ system.

To accommodate pseudo-reward, whether hand-coded or produced through this non-hierarchical technique, Dietterich's technical version of MAXQQ stores an additional table to hold the expected reward for within the current subtask plus the expected pseudo-reward for completing this subtask. We will denote these values as $\tilde{Q}_C$. This modification allows MAXQQ to use $\tilde{Q}_C$ values in its exploration policy but use Q_C values for the recursively decomposed expected value of a macroaction. This avoids having the local subtask's pseudo-reward function incorporated into the value function of the subtasks above it. MAXQQ's use of $\tilde{Q}_C$ also avoids an analogous problem to recursive decomposition issue with bids that will contain the expected exit-state subsidy (see section 4.4). It is also similar to the motivation behind updating a subtask's Q-tables with the value $BID\,(child, s) - SUBSIDY\,(parent, e)$ instead of simply the bid amount.

Use of pseudo-reward is very helpful in MAXQQ, as well-defined pseudo-reward may allow MAXQQ to yield a policy that is better than the recursively optimal policy and potentially equivalent to the hierarchically optimal policy. Rather than rely on pseudo-rewards, EHQ can be thought of as behaving like MAXQQ with the addition of learning what the appropriate pseudo-rewards should be. Given an MDP, hierarchy, and state abstraction for which MAXQQ will converge to a hierarchically optimal policy if it was provided with appropriately designed pseudo-reward functions, EHQ may find these pseudo-rewards as subsidies within its economic system and converge to an equivalent hierarchically optimal policy.

6.3 Subtask context

EHQ's propogation of non-local information into its Q_V and Q_C tables causes these values to have a strong dependence on the calling context of the subtask. Marthi et al. (2006) strongly emphasize the importance of separating a subtask from its context. In the MAXQ decomposition, where only reward local to the subtask and its descendents is represented, the subtask completely ignores the context in which it was called. Such separation allows for a subtask to be used by multiple nodes in the hierarchy. It also allows the value function for a subtask to be reused in other variations of the domain, if that variation effects some other portion of the hierarchy, rather than any of the values under the subtask. MAXQQ's limit of recursive optimality is in part the price for completely ignoring calling context. Using pseudo-reward to get practical convergence to a hierarchical policy can create context dependencies (and will if and only if EHQ's subsidies will also create such dependencies). In the case of MAXQQ, the pseudo-reward function may have to be defined for a particular context and these rewards will propagate back into the subtask's value function.

HOCQ and ALispQ use non-local reward to make decisions within a subtask, allowing some cases in which subtasks are not well separated from their context. One example is when the choices in the subtask affect the number of time steps that elapse before exiting the subtask, which can affect the exit value of some of the state variables (Marthi et al., 2006). To deal with this issue, Marthi et al. (2006) developed several techniques for accounting for this dependence in practice.

In EHQ, subtasks are less separated from context, as EHQ incorporates external reward into both Q_V and Q_C. By definition, the exit-state subsidies a child agent receives depend on the parent and the state, not the child. Specifically, this non-local information is passed through EHQ's exit-state subsidies, which are defined with respect to the parent's value function and the set of exit-states of the child. As the set of exit-states is a property of the child, the meaningful aspect of the calling context is the parent's value function. Multiple

parent nodes will clearly present a problem for EHQ in this regard, as the agents implementing these nodes will likely converge to different value functions and thus offer different subsidies. Additionally, it potentially undermines practical convergence, as since there may be multiple agents implementing each parent subtask in EHQ, each with individual beliefs about the correct value function, the differences in their value functions until they converge will add non-stationary noise to the exit-state subsidies received by the child agent. We must count on all the agents for a given parent eventually converging to the same value function or for ϵ to be sufficiently near 0 such that the same (highest bidding) agent wins the auction nearly every time. In these two ways, EHQ's lack of complete subtask independence may be detrimental.

One approach to alleviating the subtask context issue for EHQ is to represent the expected subsidy as a separate value. This approach is discussed in the section 6.5. Note that subtask context is not a problem for EHQ in the case when, given the state in which a subtask is called, there is only one exit-state in which the subtask can terminate. A subtask meeting this requirement, such as `Refuel` in the TaxiFuel domain, can be called by multiple parent subtasks. The apparent stochastic exit-state subsidy it receives cannot influence its policy.

6.4 Indirect exit-state subsidies

In the EHQ algorithm, exit-state subsidies are used to incentivize agents to consider the impact of the solution policy for their local sub-problem on the entire MDP. The EHQ algorithm only allows subsidies to be given from a parent to a child. The approach described in this section is a more generalized form of subsidy. Rather than merely allow a parent to subsidize its child, we suggest allowing any agent to subsidize the agents that appear to be acting before it in the system and impacting its reward. This approach is more difficult to motivate economically, but may allow faster propagation of value information from between

otherwise distant parts of the hierarchy. If this value information had to be passed through direct subsidies, it would have to propagate through numerous agents' Q-values along the way, which may be quite slow. Indirect subsidies may speed convergence in practice, allow for greater state abstraction, and reduce subtask context dependence.

For informal intuition of why this might be the right approach, we refer to the TaxiFuel domain and the hierarchy of 3.4. Many nodes in this hierarchy must see the fuel level, but only `Refuel` and `Navigate(n)` take primitives that either affect fuel level or receive reward dependent on the fuel level. `Get` and `Put` must therefore see the fuel level to be able to model Q_V for those macroactions. Since `Put` depends on the fuel level, `Root` must similarly see the fuel level to represent Q_V appropriately for `Put`. To find a hierarchically optimal policy for TaxiFuel, `Navigate(destination)` must learn its dependence on fuel and what its value for different levels of fuel are. Once it has learned this information, it will propagate up to `Put` through `Navigate(destination)`'s bid, and then up to `Root` through `Put`'s bid. Then `Root` must appropriately subsidize `Get`, which in turn must propagate this information into its subsidies for `Navigate(source)`. Only after this information has been propagated into the value functions of the `Navigate(source)` actions can it potentially act to improve the situation for `Navigate(destination)` and push the system towards hierarchical optimality. Clearly, it would be preferable to avoid the middlemen and just allow `Navigate(destination)` to learn that the agents at `Navigate(source)` are affecting fuel and allow `Navigate(destination)` to directly give those agents subsidies to incentivize them to act in `Navigate(destination)`'s interest.

Formalizing this notion is potentially very complicated, as it is not clear which agents should pay subsidies, to whom they can pay subsidies, and how they might learn to subsidize `Navigate(source)` rather than `Refuel`, to mention just a few areas of concern. We start with the observation that, in the episodic setting, hierarchical solutions consist of an ordered set of actions and macroactions of finite length. In the infinite horizon setting, a solution must cycle, and therefore we can consider one cycle to be a solution for our immediate

purposes here. As we are discussing subsidies, we ignore the primitives in the solution, and view a solution as an ordered sequence of macroactions. We argue that for many domains, HRL algorithms will quickly converge to an ordering of macroactions that is consistent with the optimal policy. For the Taxi domain, this ordering is always { `Root`, `Get`, `Navigate(source)`, `Put`, `Navigate(destination)` }, and seems stable in practice after only a few episodes of experience. We suggest that an agent should be able to subsidize the agents of any macroaction that precedes it in the ordering. The question of how to determine which of the preceding macroactions to subsidize remains open.

With multiple agents at each node, indirect subsidies of this nature are difficult to motivate economically. It is clear that the agent that acts to implement the subsidized subtask should receive the subsidy, but it is not clear how the amount of the subsidy should be set or who should pay it. In the current version of EHQ, it is unknown which implementer of the subtask from which the subsidy is coming will actually get control of the world and thus potentially benefit from the subsidy. Some manner of pooling of the agents' knowledge in setting the subsidy and collectively paying it is incompatible with the notion of independent, self-interested agents but it still worth exploring.

Ultimately, if this approach would work, it would cause the EHQ system to function significantly more like MAXQQ with well-designed pseudo-reward functions. Rather than propogate values through multiple agents that do not have a direct effect on the key aspects of the state, indirect subsidies could act like a short cut to transfer reward to the points where it is needed to affect change immediately. This also may improve the subtask context issue, as if the parent is not the agent providing the exit-state subsidies, having multiple parents is no longer a problem. This is not clear-cut, as context may simply become a high level notion of what other agents are present in the system and where in the ordering they are being invoked. This may allow for greater state abstraction, as intermediate nodes can abstract some parts of their state space if they are no longer being subsidized for those aspects of the state space (their reward is no longer dependent on those aspects of their

state variables). Only the nodes which affect particular aspects of the state space need the capacity to be subsidized for those aspects. While potentially beneficial, indirect exit-state subsidies appears to be a very challenging approach to formalize.

6.5 Explicitly modeling the expected exit-state subsidies

Incorporating the exit-state subsidies into the Q_V and Q_C tables of an EHQ agent creates several problems including subtask context dependence, as noted above, and bids including the expected exit-state subsidy, both of which ultimately lead to slower convergence in practice. A potential remedy for these problems is to explicitly model an agent's expected exit-state reward for each possible state-action pair. We will denote an agent's expected exit-state reward as $Q_S\left(agent, s, a\right)$ to differentiate it from Q_E, although it serves essentially the same purpose and would likely be updated in a similar manner to Q_E in ALispQ-learning.

Without the expected subsidy incorporated in its Q_V and Q_C values, an EHQ agent can bid $Q_V + Q_C$, which will be a conservative estimate of its V^* since exit-state subsidies must be positive. This avoids the problematic delay of changes in the exit-subsidy propagating back into the bid, preventing any potential for the agent to be exploited by a parent. However, if we consider strategic bidding, it is not clear why the child would only bid $Q_V + Q_C$. It would seem that a competing agent would want to bid $Q_V + Q_C + \epsilon$ if it had some belief that it would receive an exit-state subsidy. This competition should drive up bids until the agents are bidding $Q_V + Q_C + Q_S$. This situation is unfortunate, as it may allow the parent to exploit its children in the short run by varying its subsidies strategically.

Explicitly storing Q_S avoids the practical problems of recursive decomposition, as the parent's Q_V for a macroaction would be the highest bid of the children below, as is done in rEHQ. Such a recursive approach would be poorly motivated in our economic context, as it would completely violate any notion of independence, decentralization, and potentially would be intractable in a strategic setting.

This approach would allow for greater subtask independence, as agents could store one set of Q_S values for every context in which they are called. This modification would not necessarily eliminate the subsidy propagation problem, but it may open the door for some iterative approach that could allow subsidy information to propagate at a faster rate, to reflect its tendency to be more volatile than other values in some domains.

As an alternative to explicitly storing Q_S, it is possibly to further decompose Q_S into the product of the probability of exiting in a given state and the subsidy for that state:

$$\forall s' Q_S\left(agent, s', a\right) = \sum_{e \in E_j} P_E\left(agent, e | s', a\right) \cdot SUBSIDY\left(parent, e\right)$$

It would then be necessary to learn P_E, essentially creating an economic version of HOCQ. This modification would separate the subtask from its context without the need for storing a Q_S table for each context. More importantly, it would allow the bids to reflect the subsidy values without relying on propagation, making the algorithm more robust to significant changes in the parent's value function.

If Q_V values do not incorporate expected exit-state reward, it would be practical to share them between subtasks, as done by MAXQQ and discussed in the context of EHQ in section 4.5. This is still troubling from an economic perspective, but as discussed in section 4.5, there are settings in which it may be appropriate.

6.6 Techniques for faster convergence

Beyond the two general approaches described above, there are a number of minor technical modifications that may improve EHQ convergence. We describe two in this section. The first is aimed at speeding up propagation of subsidy information through the agents' Q-tables. The technical version of MAXQQ given in Dietterich (2000a) provided for updates to Q_C of the parent for all intermediate states that the child passes through while it is

executing; these updates will potentially speed convergence in practice by backing up value information more rapidly. We omitted an analogous extension from EHQ for simplicity, but see no obvious reason why similar techniques could not be used. As we are storing Q_V for macroactions, a similar technique could speed convergence of the Q_V values for macroactions as well by soliciting bids for every intermediate state. Such techniques may help EHQ to scale, by allowing it to better propagate information about evolving exit-state subsidies back into the bid amounts. This would improve convergence and also lessen the potential for a parent to exploit a child, as mentioned in section 4.7.

The primary objection to incorporating such techniques into EHQ is that it requires that each agent, when it invokes a child, keep track of the history of states through which the child and its descendents pass until they return control to the agent. In the economic context, it requires the agent to be able to see the world even when the agent does not have control of the world. Fortunately, the agent needs only to see the world from the perspective of its own abstraction, so it need not watch aspects of the world of which it would not otherwise be aware. Updating Q_C values, requires no knowledge of any reward, but is effectively an iteration that relies only on the sequence of abstract states. The level of visibility of the world required by such an extension is not desirable in an economic context, but does not significantly undermine the motivations of EHQ.

A second technical modification that may provide better convergence results in practice is to cool the learning rate α and exploration rate ϵ on a per agent basis, rather than maintain these as universal values as our implementation does (as mentioned in section 5.1). MAXQQ cooled rates on a per node basis, cooling the nodes lower in the hierarchy faster to help them converge first (Dietterich, 2000a). Due to the same circularity problem that precludes a formal convergence result (see section 4.9), it is doubtful that this approach would work well for EHQ. However, even if the rates are not cooled progressively up the hierarchy, it still would seem preferable to cool based on the number of actions executed by the particular agent, rather than the total number of actions executed in the world. This

would would allow the learning and exploration rates to better reflect the experience of the individual agent, rather than the entire system.

6.7 Future extensions

This section briefly introduces some potential applications of the EHQ system to other settings. First, we look to leverage EHQ's robust, decentralized properties to apply it in multi-agent settings. Second, we suggest viewing the agents as strategic, instead of directly defining their bids as V^*. Such an extentsion may enhance our claims about the robustness of the market structures in a loosely coupled system. Lastly, we briefly discuss using EHQ as a potential method to compare different state abstractions.

6.7.1 Concurrent EHQ

Marthi et al. (2005b) presents a variation of ALisp to solve HRL problems with multiple effectors. A *multiple-effector* problem, in the sense discussed in this work, is one in which there are multiple agents in the world working cooperatively to accomplish a common goal. The Resource domain used in Marthi et al. (2005b) is from the game Stratagus. The goal of the Resource domain is to efficiently use peasants (effectors) to collect wood and gold. An obvious difficultly arises as if each peasant attempts to behave optimally considering only their own state and actions, peasants will inevitably collide with each other. Learning a centralized policy for the agents is inefficient, as the joint state space and joint action space require that an extremely large number of Q-values be learned. This is obvious motivation for an approach similar to HRL to deal with the state space consideration. To decompose the joint action space, Marthi et al. (2005b) introduces Concurrent ALisp, augmenting ALisp (Andre and Russell, 2002) with constructs that allow new partial programs to be spawned as separate threads and effectors to be assigned and reassigned between the threads currently running. They call this a *multi-threaded partial program*. Note that threads may still control

more than one effector at a time.

We propose that using a framework such as EHQ, with independent, self-interested agents might be better motivated for multiple-effector problems. Potentially, transfers could be used, along the lines of the exit-state subsidies developed for EHQ, that would allow one agent to provide incentive to the other agents in the world to do what that agent wanted. For instance, an agent implementing a particular peasant might be willing to pay the agent implementing another peasant who would otherwise go collect gold to collect wood instead. Formalizing this concept into a well-defined framework would seem to be a natural extension to the economic motivations provided in this paper.

6.7.2 Strategic agents

Introducing strategic bidding and second price auctions would be an interesting extension. While formalizing how an EHQ agent might bid strategically is challenging, it remains clear that in equilibrium, competition will force the agent to bid its V^*. Introduction of opportunity costs and allowing agents to enter and exit the system will complicate this picture. EHQ does not presently solve the hierarchical credit assignment problem, as only the `Root` node will make a net profit in equilibrium, and thus the `Root` may be extracting additional reward from the system than it should receive. If other `Root`s were allowed to enter and bid on having initial control of the world, this profit might be driven out.

Another possible extension would allow agents to bid with respect to the subsidies the parent is offering. As noted in the previous chapters, in the current version of EHQ subsidies will change as the system converges and there will be some delay before the bids of reflect such a change. If the parent can strategically set subsidies, a child agent must be able to bid with respect to the subsidies being offered in the present instance, or else it will be exploited. Potentially, such modifications will help EHQ more accurately solve the hierarchical credit assignment problem, just as Hayek showed the potential to solve the credit assignment

problem in general domains (Baum and Durdanovich, 1998).

6.7.3 Competing abstractions

One of the clear drawbacks of HRL is the reliance on a programmer to design a hierarchy. The argument can be made for many domains, particularly those that are highly stochastic, that it is easy for a human to infer some sense of the structure of the optimal policy while still being prohibitively difficult to exactly specify the optimal policy. State abstractions are even more challenging to specify, even though ALisp (Andre and Russell, 2002) provides a relatively simply language for expressing them. In particular, effective state abstractions must be specified for each action in the action space of a given subtask. The major work cited for this paper take the hierarchy and state abstraction to be fixed, but there have been a number of papers examining automated methods for determining hierarchies and state abstractions.

We suggest that a version of EHQ in which the agents implementing a node in a hierarchy each had different abstract views of the state space. In such a setting, agents would compete not only in the amount of their bid (based on their experience) but also on their ability to see the necessary aspects of the state space. For this approach to be useful, agents that bid too high, and thus run a loss over many episodes, would have to be penalized or eliminated entirely. Such a system may be interesting if we allow for agents to enter and exit the system, with the possibility for the abstractions of new agents entering the system to be some novel permutation. This would give the system a Hayek flavor (Baum and Durdanovich, 1998) as the population would over time produce agents that were specialized for the relevant aspects of the state space.

6.8 Concluding remarks

This paper has closely examined issued surrounding hierarhicical reinforcement learning, including primarily the distinction between hierarchical optimality and recursive optimality. The EHQ algorithm uses subsidies to align interests such that an agent which would otherwise converge to a recursively optimal policy will instead act hiearchically optimally, assuming the agent has a hierachically safe state abstraction. This significant contribution underscores the value of market approaches and provides a number of promising avenues for future work. Our empirical results also demonstrated, however, that further optimization of EHQ is required to help it scale effectively to large domains. With such work, there are numerous potential applications of the EHQ approach to large, multi-agent settings.

Bibliography

Andre, D. and Russell, S. (2002). State abstraction for programmable reinforcement learning agents. In *AAAI-02*, Edmonton, Alberta. AAAI Press.

Baum, E. B. and Durdanovich, I. (1998). Evolution of cooperative problem-solving in an artificial economy. *Journal of Artificial Intelligence Research*.

Bellman, R. E. (1957). *Dynamic Programming*. Princeton University Press.

Bertsekas, D. P. and Tsiksiklis, J. N. (1996). *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA.

Dean, T. and Lin, S.-H. (1995). Decomposition techniques for planning in stochastic domains. In *IJCAI-95*, pages 1121–1127, San Francisco, CA. Morgan Kaufmann Publishers.

Dietterich, T. G. (2000a). Hierarchical reinforcement learning with maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303.

Dietterich, T. G. (2000b). An overview of maxq hierarchical reinforcemnet learning. In *Symposium on Abstraction, Reformulation and Approximation SARA, Lecture Notes in Artficial Intelligence*, pages 26–44, New York. Springer Verlag.

Dietterich, T. G. (2000c). State abstraction in maxq hierarchical reinforcement learning. *Advances in Neural Information Processing Systems*, 12:994–1000.

Marthi, B., Russell, S., and Andre, D. (2006). A compact, hierarchically optimal q-function decomposition. In *UAI-06*, Cambridge, MA.

Marthi, B., Russell, S., and Latham, D. (2005a). Writing stratagus-playing agents in concurrent alisp. In *IJCAI-05 Workshop on Reasoning, Representation, and Learning in Computer Games*, Edinburgh, Scotland.

Marthi, B., Russell, S., Latham, D., and Guestrin, C. (2005b). Concurrent hierarchical reinforcement learning. In *IJCAI-05*, Edinburgh, Scotland.

Mehta, N. and Tadepalli, P. (2005). Multi-agent shared hierarchy reinforcement learning. In *ICML Workshop on Richer Representations in Reinforcement Learning*.

Parr, R. and Russell, S. (1998). Reinforcement learning with hierarchies of machines. *Advances in Neural Information Processing Systems*, 10.

Russell, S. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, New Jersey, 2 edition.

Russell, S. and Zimdars, A. (2003). Q-decomposition for reinforcement learning agents. In *ICML-03*, Washington, DC.