Optimal Planning with Approximate Model-Based Reinforcement Learning

by

Hai Feng Kao

M.Sc., National Taiwan University, 2006

B.Sc., National Taiwan University, 2004

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

The University Of British Columbia

(Vancouver)

December 2011

Abstract

Model-based reinforcement learning methods make efficient use of samples by building a model of the environment and planning with it. Compared to model-free methods, they usually take fewer samples to converge to the optimal policy. Despite that efficiency, model-based methods may not learn the optimal policy due to structural modeling assumptions. In this thesis, we show that by combining model-based methods with hierarchically optimal recursive Q-learning (HORDQ) under a hierarchical reinforcement learning framework, the proposed approach learns the optimal policy even when the assumptions of the model are not all satisfied. The effectiveness of our approach is demonstrated with the Bus domain and Infinite Mario – a Java implementation of Nintendo's Super Mario Brothers.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I am very fortunate to have had the opportunity to study and do research at the University of British Columbia. I am most grateful to my supervisor, Alan Mackworth. When I proposed the idea of developing a generic game AI for video games, he accepted and allowed me to work on a problem which I am really passionate about. Besides, I would like to thank David Buchman, John Chia and Bruno Norberto da Silva for sharing their knowledge and expertise in reinforcement learning. I would also like to thank Tzu-Kuo Huang for the discussion of recent work at CMU. One of the works actually inspired the idea of this thesis. A special thank goes to Prof. Poole, who gave me many precious ideas to make this thesis better. And most importantly, I would like to thank my parents, who support me whenever I need it. Without them, it would have been impossible to finish this work.

Chapter 1

Introduction

Reinforcement learning (RL) addresses the problem of finding an optimal policy in a stochastic environment. In the RL setting, an agent interacts with the environment, optimizing its behaviour to maximize the received rewards. In many applications, it is expensive to acquire samples from the environment, so it is important to for the agent to learn an effective policy in as few samples as possible.

RL methods can be broadly classified into two classes: model-based and model-free. Model-based methods learn an effective policy by constructing the model from samples and simulating experiences from the model. It generally requires fewer samples to learn the optimal policy.

R-MAX [20] learns the model by exploring parts of the domain. To learn an accurate model, the agent needs to explore every state m times. Thus, it is impractical to apply such method to large domains.

For large domains, it is necessary to apply machine learning algorithms to generalize the knowledge to unvisited states. Learning the model is a supervised learning problem: given the agent's current state and action, the learning algorithm needs to predict the next state and reward. Many supervised learning algorithms can generalize over unvisited states.

Methods based on factored Markov decision processes (FMDPs) assume the problem has some factored structure, and use specialized algorithms that exploit the structure [15, 18, 38, 40]. However, not all of problems have the factored structures. Thus, the applicability of these methods are limited.

Hester and Stone [17] observed that modeling relative transition effects of actions is more generalizable than modeling their absolute values. They proposed RL-DT to model relative effects with decision trees.

Walsh et al. [38] proposed KWIK (Know What It Knows) linear regression to learn the transition probabilities and applied it to Stochastic STRIPS domains while assuming the preconditions and effects of each action are known in advance.

Degris and Sigaud [10] proposed SDYNA, which extended Dyna [34] by learning the structure of a problem with incremental decision trees. Sutton et al. [36] introduced linear Dyna – a combination of Dyna and linear function approximation. Instead of enumerating all states, which is not feasible for large problems, their methods predict the features of the next state and reward, using function approximation techniques.

Model-based methods are powerful techniques. That allows us to predict the outcome of the agent behaviour and plan over it. They can effectively reduce the number of samples which are required to find a good policy. With the learned model, it is easy for model-based methods to generalize the knowledge to novel scenarios. However, it is difficult for model-based methods to learn optimal policies. The possible reasons are:

- Inaccuracy of the underlying supervised learning algorithms: most of model-based methods rely on supervised learning algorithms to learn the model. When these algorithms predict incorrectly, a suboptimal policy might be learned.

- Structure assumption of the model: most model-based methods have some assumptions on the structure of model. For example, the methods based on FMDPs assume the problem has some factored structure; Linear Dyna [36] assumes the features of next state can be predicted with linear function approximation. When the assumptions are incorrect, a suboptimal policy will be learned.

- Impossibility to learn all the effects: as indicated in [38], to learn the effects in stochastic STRIPS domains is NP-Hard. If the number of all possible effects is small, it is possible to enumerate all of them and exclude the effects

with small probability. However, since the problem is NP-Hard, there are no known efficient solutions to resolve it.

- Computational constraints: even if we can learn all the effects, it may be too expensive to consider all of them during the planning process. Due to the stochastic nature of MDP, each effect may result in several possible outcomes given the same state and action. If we consider too many effects during the planning process, the number of states in a planning envelope might become too large to compute.

On the other hand, model-free methods learn the Q-function directly. There are no simulation steps for model-free methods and modeling is unnecessary. The existing linear function approximation algorithms for model-free methods have been successfully applied to large domains [7, 32]. However, these methods may have slower learning rates since they cannot predict the outcome of the agent's behaviour and are unable to use planning techniques to search for a better policy. Instead, they need to collect the samples by trial-and-error. This makes it more difficult for these methods to generalize to novel scenarios.

In this thesis, we investigate the possibility of combining both model-based and model-free methods and get the best of each - simulate the experiences from samples to increase the learning rate and learn an optimal policy even when the assumption of model is not satisfied.

To combine these two methods, we need a framework which can incorporate different RL algorithms. One possible choice is hierarchical reinforcement learning (HRL). We show that by combining model-based methods with hierarchically optimal recursive Q-learning (HORDQ) [1], which is a model-free method, we can guarantee that the overall policy will converge to the optimal one even when our model fails to approximate the problem. Our approach assumes the task hierarchy for an MDP is given. The hierarchy can either be designed manually or learned by automatic hierarchy discovery techniques [16].

We are not the first to try to combine different RL methods within the HRL framework. Ghavamzadeh and Mahadevan [13] combined value function-based RL and policy gradient RL to handle continuous MDP problems. Cora [8] incorporated model-based, model-free and Bayesian active learning into the MAXQ

framework. Nevertheless, these methods seek recursive optimality in the learning process, thus they fail to satisfy any optimality condition when one of the subtasks fails to find its optimal policy. In contrast, our method learns the optimal policy without the requirement that all of the policies of the subtasks need to be optimal. It is more robust and allows us to incorporate approximate approaches into the same framework.

1.1 Contributions

We are the first to address the problem of learning the optimal policy when the structural assumption of the model-based methods is not satisfied.

Our contribution are:

- Deriving the condition of a task hierarchy that the hierarchically optimal policy is equal to the optimal policy when some of the policy of subtasks in the task hierarchy are suboptimal.
- Developing a hierarchical reinforcement learning framework which allows unsafe state abstraction without the loss of optimality guarantee
- Introducing pseudo-rewards to the HORDQ algorithm and show that it can improve the learning rate of the HORDQ algorithm.

1.2 Thesis organization

We provide a brief review of reinforcement learning in Chapter 2. That includes Markov decision processes (MDPs), semi-Markov decision processes (SMDPs), and hierarchical reinforcement learning (HRL). Our main theory is presented in Chapter 3. We provide the experimental results on the Bus domain and Infinite Mario in Chapter 4. Finally, conclusions, limitations, and directions for future work are discussed in Chapter 5.

Chapter 2

Background

In this chapter, we introduce the basic concept of reinforcement learning (RL) , the Markov decision process (MDP) and semi-Markov decision process (SMDP) formalisms. We describe the solution methods of MDPs, that include model-free and model-based methods. Then we review the concepts and algorithms in the hierarchical reinforcement learning (HRL) framework. We present a brief introduction of previous work in the RL field. For more comprehensive introductions of MDP, SMDP and RL, please refer to the standard texts [19, 27, 30, 35]. A more theoretic treatment of RL is provided by Bertsekas and Tsitsiklis [6]. Barto and Mahadevan provide an excellent review on HRL [4].

2.1 Reinforcement learning

Reinforcement learning (RL) is a collection of methods which allow an agent to execute a sequence of actions and improve its actions by the rewards which are provided by the environment. There are two kinds of RL learning tasks – episodic and continuing tasks. In an episodic task, there are some terminal states that will terminate the current task (episode) when one of them is encountered. For the episodic task, an action only affects the subsequently received rewards within the current episode. There are no terminal states for a continuing task. Therefore, the sequence of actions which are executed by the agent will be infinite.

In the RL setting, the environment can be modeled in several formalisms. One

of the most commonly adopted formalisms is that of Markov decision processes (MDPs). An MDP assumes that the state of the environment is fully observable, and each action takes a single step to finish. Semi-Markov decision processes (SMDPs) remove the latter assumption and allow the action to take several time steps. Partially observable Markov decision processes (POMDPs) remove the former assumption and the agent needs to decide the actions without access to the full state of the environment. MDPs and SMDPs are introduced in Sections 2.2 and 2.4. POMDPs are orthogonal to our work, thus they will not be covered.

2.2 Markov decision processes

Definition 1 *A Markov decision process is formalized as a tuple* $< S,A,P,R >$*, where:*

- S *is a finite set of states of the environment.*
- A *is a finite set of actions.*
- $P : S \times A \times S \to [0,1]$ *is the transition function which defines a probability distribution over the possible next states.*
- $R : S \times A \to \mathbb{R}$ *is the reward function which defines the reward after executing a certain action at a certain state.*

Given a state of the environment, a policy $\pi : S \to A$ dictates what action should be performed at that state. The value function $V^\pi : S \to \mathbb{R}$ represents the expected cumulative reward when policy π is followed from state s.

The value function is defined as:

$$V^\pi(s) = E[R(s_0, \pi(s_0)) + \gamma R(s_1, \pi(s_1)) + \gamma^2 R(s_2, \pi(s_2)) + \ldots | s_0 = s, \pi], \quad (2.1)$$

where $\gamma \in [0,1]$ is the discount factor which discounts the future reward to the present value.

The value function satisfies the Bellman equation [5]:

$$V^\pi(s) = \sum_{s'} P(s'|s, \pi(s))[R(s, \pi(s)) + \gamma V^\pi(s')] \quad (2.2)$$

Similarly, the action-value function (or Q-function) represents the expected cumulative reward after action a is executed in state s and policy π is followed thereafter. The Q-function is defined as:

$$Q^{\pi}(s,a) = E[\sum_{t=0}^{\infty} \gamma^t R(s_t,a_t)|s_0 = s, a_0 = a, \pi] \tag{2.3}$$

The Bellman equation for the Q-function is:

$$Q^{\pi}(s,a) = \sum_{s'} P(s'|s,a)[R(s,a) + \gamma Q^{\pi}(s',\pi(s'))] \tag{2.4}$$

We are interesting in finding the optimal policy, which is the policy that yields a value that is as high as the value of any other policies in all states. It is defined as:

$$\pi^*(s) = \underset{a \in A}{\operatorname{argmax}}[R(s,a) + \gamma \sum_{s'} P(s'|s,a)V^*(s')], \tag{2.5}$$

where V^* is the optimal value function. It satisfies the following equation:

$$V^*(s) = \max_{\pi} V^{\pi}(s) = \max_{a \in A}[R(s,a) + \gamma \sum_{s'} P(s'|s,a)V^*(s')] \tag{2.6}$$

Likewise, the optimal Q-function is the solution of the following equation:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a' \in A} Q^*(s',a') \tag{2.7}$$

2.3 Solution methods for MDPs

There are several types of methods to solve an MDP problem. If the transition and reward functions are known as a priori, the optimal policy can be computed with dynamic programming (DP) algorithms. These methods are considered offline algorithms, since they are performed without the interaction with the environment.

In the field of RL, people are more interested in solving problems where the underlying MDP is unknown in the beginning. Due to the lack of a priori model, it is necessary for the agent to interact with the environment and collect samples to acquire the statistical knowledge of the MDP. Therefore, the methods to solve

these problems are considered online algorithms.

Depending on how samples are processed, a distinction is made between model-based and model-free methods. Model-based methods use samples to construct a model of environment and use the techniques such as dynamic programming to compute the optimal policy from it. On the other hand, model-free methods do not learn the model but directly learn a policy or value function.

For both types of methods, an exploration policy is required to guide the sampling process. The simplest strategy is to select an action which leads to the highest value. However, this strategy does not allow the agent to explore the states which are not visited before. A better approach is to use ε-greedy method. Such a method allows the agent to abandon the best action and choose a random action with a very small probability ε. The higher the probability, the more likely that the agent would explore the new actions. However, if the exploration probability is too high, it will increase the time to converge.

2.3.1 Model-free methods

Most of the model-free methods fall into the category of the temporal difference (TD) learning. The main idea of temporal difference learning methods is to iteratively estimate the new values based on the old estimates. After the agent selects an action, the value of the previous state is updated based on the immediate reward and the value of current state.

The equation to update the value function in TD learning is:

$$V(S_t) \leftarrow V(S_t) + \alpha[r_{t+1} + \gamma V(S_{t+1}) - V(S_t)],$$

where $V(s_t)$ is the value function of the state s_t. $V(s_t)$ is the expected reward when the agent reaches state s_t. r_{t+1} is the reward given to the agent when it chooses the action at state s_t.

SARSA[29] and Q-learning[39] are the most popular TD methods. SARSA is an on-policy TD approach, which indicates that it learns from the current policy. Different from other TD approaches, SARSA updates the Q-value from the value of the next state-action pair. The Q-value is updated by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)],$$

where $Q(s,a)$ is the expected reward when the agent takes the action a at the state s. α is a constant step-size parameter. γ is the discount factor.

Algorithm: SARSA

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize s
    Choose a based on s using policy derived from Q (e.g., ε-greedy method)
    Repeat (for each step of episode):
        Take action a, obtain reward r and next state s' from the environment
        Choose a' based on s' using policy derived from Q (e.g., ε-greedy method)
        $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$
        $a \leftarrow a'$
    Until s is terminal

Q-Learning is an off-policy TD approach. Compared to SARSA, Q-Learning updates the Q value by the highest value of the next possible state-action, rather than the next state-action executed by the agent. The Q value is updated by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)],$$

where $\max_a Q(s_{t+1}, a)$ is the highest value of the next possible state-action.

Algorithm: Q-Learning

Initialize $Q(s,a)$ arbitrarily

Repeat (for each episode):

  Initialize s

  Repeat (for each step of episode):

    Choose a based on s using policy derived from Q (e.g., ε-greedy method)

    Take action a, obtain reward r and next state s' from the environment

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

    $s \leftarrow s'$

  Until s is terminal

2.3.2 Model-based methods

Model-based methods construct a model of the environment from the samples and conduct a planning process over it. With the constructed model, these methods can predict the outcome of the agent's action, and choose an optimal action that can achieve the highest predictive reward. Since this type of methods makes efficient use of samples, they often require fewer samples to learn a good policy than model-free methods. They are important to the applications where computation is cheap and the interactions of the environment are expensive.

A straightforward approach is to build the model by learning the transition function and reward function, then use dynamic programming (DP) to compute the solution of the Bellman equations (Equation 2.7).

The solution of the Bellman equation is unique, but there might be several policies which can have the same value function. The most well-known dynamic programming algorithms are value iteration [5] and policy iteration [19].

Value iteration iteratively applies the Bellman equation to update the values for all possible states. In theory, value iteration requires an unbounded number of iterations to converge to the optimal value function V^*. In practice, we stop when the maximal update is smaller than certain amount.

Algorithm: Value Iteration

Initialize $V(s)$ arbitrarily
Repeat:
  for $s \in S$ do
    $V(s) \leftarrow \max_a [R(s,a) + \gamma \sum_{s'} P(s'|s,a)] V(s')$
  end for
Until maximal update smaller than δ

Policy iteration separates the iterative process into two stages. The first stage is policy evaluation, which computes the value function for current policy. The second stage is policy improvement, which improves the policy by choosing a better action based on the value function computed in previous stage. Policy iteration often quickly converges in a few iterations.

Algorithm: Policy Iteration

Initialize π_1 arbitrarily
$k \leftarrow 1$
Repeat:
  Repeat:
    for $s \in S$ do
      $Q^{\pi_k}(s,a) \leftarrow [R(s,\pi_k(s)) + \gamma \sum_{s'} P(s'|s,\pi_k(s)) Q^{\pi_k}(s',\pi_k(s))]$
    end for
  Until maximal update smaller than δ
  for $s \in S$ do
    $\pi_{k+1}(s) \leftarrow \text{argmax}_a \, Q^{\pi_k}(s',a)$
  end for
  $k \leftarrow k+1$
Until $\pi_{k+1} = \pi_k$

Another popular class of model-based methods is the Dyna architecture [34]. Dyna [34] uses the learned model to generate extra experiences through the simulating process, but the underlying learning algorithm is still a TD method.

2.4 Semi-Markov decision processes

Semi-Markov decision processes (SMDPs) [27] extend MDPs to handle the actions which may take several steps to finish. It allows us to handle the problem which the agent cannot make a decision for every step. The state may be updated several times between each decision made by the agent. SMDPs provide a way to model these temporal-extended actions, which make them an important tool to describe the environment of hierarchical reinforcement learning.

Definition 2 *A semi-Markov decision process (SMDP) is formalized as a tuple* $< S, A, P, R >$*, where:*

- S *is a finite set of states of the environment.*
- A *is a finite set of actions.*
- $P : S \times \mathbb{N} \times A \times S \to [0,1]$ *is a multi-step transition function.* $P(s', N|s, a)$ *denotes the probability that state is transited from* s *to* s' *after action* a *is taken at state* s *and finished in exact* N *steps.*
- $R : S \times \mathbb{N} \times A \to \mathbb{R}$ *is the reward function which defines the reward after executing a certain action at a certain state.*

The formalism of SMDP is similar to an MDP, except that the transition function and reward function now include the duration of actions.

The definition of value function for SMDP is:

$$V^{\pi}(s) = E[R(s_0, \pi(s_0)) + \gamma^{N_0} R(s_1, \pi(s_1)) + \gamma^{N_1} R(s_2, \pi(s_2)) + \ldots | s_0 = s, \pi] \quad (2.8)$$

The definition is similar to equation 2.1, except that the reward is discounted by the duration of actions.

The Bellman equation for SMDP can be described as:

$$V^{\pi}(s) = \sum_{s', N} P(s', N|s, \pi(s))[R(s', N|s, \pi(s)) + \gamma^{N} V^{\pi}(s')], \quad (2.9)$$

Similarly, the Q-function for SMDP is defined as:

$$Q^{\pi}(s,a) = \sum_{s',N} P(s',N|s,a)[R(s',N|s,a) + \gamma^{N} Q^{\pi}(s',\pi(s'))]. \tag{2.10}$$

The optimal value function and Q-function satisfy the following equations:

$$V^{*}(s) = \max_{a \in A}[R(s,a) + \sum_{s',N} \gamma^{N} P(s',N|s,a) V^{*}(s')] \tag{2.11}$$

$$Q^{*}(s,a) = R(s,a) + \sum_{s',N} \gamma^{N} P(s',N|s,a) \max_{a' \in A} Q^{*}(s',a') \tag{2.12}$$

2.5 Hierarchical reinforcement learning

The idea of Hierarchical reinforcement learning (HRL) framework is to decompose a large MDP into several smaller subtasks. Each subtask is responsible to learn a policy of part of the original state space. It allows us to adopt a divide-and-conquer strategy and focus on finding the optimal policy for a part of the original problem (state decomposition). It also allows us to group a sequence of actions together and share it for several subtasks (temporal abstraction). Finally, it allows us to ignore the features which are irrelevant to the subtask (spatial abstraction).

Singh [31] presented Hierarchical DYNA (H-DYNA) which extends Dyna to learn a hierarchy of abstract models. Singh proposed a gating architecture which switches between temporal-extended actions and adopted H-DYNA to solve the problem. Dayan and Hinton [9] introduced Feudal Q-learning, a hierarchical method that recursively partitions the problem in both spatial and temporal scales.

The above methods depend on the MDP formulation. However, they are not suitable for the problem of HRL. In HRL, the agent makes decisions when each temporal-extended actions are finished, thus the state may change several times between each decision. The SMDP model which we have described in section 2.4 is a better alternative. The application of SMDP model to HRL led to developing powerful HRL methods such as options [33], MAXQ [12], hierarchies of abstract machines (HAMs) [25], and HORDQ [1].

The options framework introduced by Sutton et al. [33] extends Q-learning

to include temporal-extended actions with hard-coded policies. In options framework, the temporal-extended actions are indivisible and opaque units which are called "options". Because the MDP combined with the options is an SMDP, they proposed SMDP Q-learning, which is the SMDP version of one-step Q-learning. It updates the Q-function after each option is terminated.

Parr and Russell [25] introduced HAMs. They observed that the execution of a hierarchical policy resembles finite-state machines. The state of a machine includes the internal execution states as well as the current state of the agent. They showed that the whole hierarchical learning process can be reduced to learning the optimal policy for a flat SMDP with each state of the SMDP, which corresponds to the finite-state machine.

The MAXQ framework is one of the early methods that combine temporal abstraction with state abstraction. Unlike the options framework or HAMs which reduce the problem to a single SMDP, MAXQ decomposes the problem into several MDPs. Each MDP corresponds to a subtask in the hierarchy. The main idea of MAXQ is to decompose the value function into the expected reward that the agent will receive after an action is finished and the expected reward that the agent will receive on the completion of a subtask. The problem of MAXQ framework is that the learning process of each subtask is isolated from the rest of hierarchy. Since each subtask is unaware of the consequence of its decision after the subtask is completed, the agent will learn a recursively optimal policy, which is worse than a hierarchically optimal policy, which can be learned by the options, HAMs, or HORDQ.

To address the problem, Andre and Russell [1, 2] introduced hierarchically optimal recursive decomposed Bellman equations Their method decomposes the value function into three parts. The first two parts are identical to the MAXQ decomposition. The additional part is the expected reward after the subtask is completed. With the additional part, they are able to show that a hierarchical optimal policy can be learned with hierarchically optimal recursive decomposed Q-learning (HORDQ).

Since the MAXQ framework and HORDQ are more relevant to our work, we provide more details about their formalisms and algorithms in the following sections.

2.5.1 The MAXQ decomposition

Definition 3 *Given an MDP M, the MAXQ framework decomposes M into a finite set of subtasks* $M' = \{M_0, M_1, \ldots, M_n\}$*, where* M_0 *is the root subtask. A non-primitive subtask* M_i *is defined as a tuple* $< U_i, A_i, R_i >$*, where:*

- U_i *is a termination predicate. It partitions the state space into active states* S_i *and terminal states* T_i*. If subtask* M_i *enters any terminal state, it terminates immediately and returns control to the parent subtask.*
- A_i *is a set of actions which are accessible to subtask* M_i*. An action can be either primitive or composite. If it is composite, it simply invokes the corresponding subtask. No recursive calls are allowed in the hierarchy.*
- R_i *is the reward function for subtask* M_i*. The reward function is defined for transitions to terminal states, and the rewards are zero elsewhere. Note that the reward function might be different from the one of MDP M. The reward function is served to encourage the agent to achieve the subgoal of each subtask rather than achieve the optimal performance in the original MDP M. It is the programmer's responsibility to design the hierarchy in a way that the optimal performance of the original MDP can be achieved.*

Definition 4 *A hierarchical policy* $\pi = \{\pi_0, \pi_1, \ldots, \pi_n\}$ *is a set which contains all subtask policies.*

The subtask policy $\pi_i : S_i \rightarrow A_i$ maps an active state to an action to execute.

In the MAXQ framework, the Q-function is decomposed as:

$$Q^{\pi}(i,s,a) = E[\sum_{t=0}^{\infty} \gamma^t r_t] = E[\sum_{t=0}^{N-1} \gamma^t r_t] + E[\sum_{t=N}^{\infty} \gamma^t r_t] \tag{2.13}$$

$$= Q_r^{\pi}(i,s,a) + Q_c^{\pi}(i,s,a), \tag{2.14}$$

where r_t is the random variable of the reward that the agent receives at step t and N is the number of primitive actions to finish action a. Q_r^{π} represents the expected cumulative reward for executing action a at state s. Q_c^{π} is the expected cumulative reward after action a is finished.

Q_r^π and Q_c^π satisfy the following equations:

$$Q_r^\pi(i,s,a) = \begin{cases} Q_r^\pi(a,s,\pi_a(s)) + Q_c^\pi(a,s,\pi_a(s)) & \text{if } a \text{ is composite} \\ \Sigma_{s'} P(s'|s,a) R(s'|s,a) & \text{if } a \text{ is primitive} \end{cases} \tag{2.15}$$

$$Q_c^\pi(i,s,a) = \sum_{s',N} P^\pi(i,s',N|s,a)\gamma^N[Q_r^\pi(i,s',\pi_i(s')) + Q_c^\pi(i,s',\pi(s'))], \tag{2.16}$$

To learn the recursively optimal policy, MAXQ-Q learning iteratively updates the estimates of Q_r and Q_c:

$$Q_r^{t+1}(i,s,a) \leftarrow (1-\alpha_t)Q_r^t(i,s,a) + \alpha_t R_t(s'|s,a) \text{ if } a \text{ is primitive} \tag{2.17}$$

$$Q_c^{t+1}(i,s,a) \leftarrow (1-\alpha_t)Q_c^t(i,s,a) + \alpha_t \gamma^N[Q_r^t(i',s',a') + Q_c^t(i',s',a')] \tag{2.18}$$

2.5.2 The HORDQ learning

Andre and Russell [1] introduced hierarchically optimal recursive decomposed Bellman equations which extend the decomposition of MAXQ in a way that hierarchical optimality can be guaranteed.

The Q-value is decomposed as:

$$Q^\pi(i,s,a) = E[\sum_{t=0}^{\infty} \gamma^t r_t] = E[\sum_{t=0}^{N_1-1} \gamma^t r_t] + E[\sum_{t=N_1}^{N_2-1} \gamma^t r_t] + E[\sum_{t=N_2}^{\infty} \gamma^t r_t] \tag{2.19}$$

$$= Q_r^\pi(i,s,a) + Q_c^\pi(i,s,a) + Q_e^\pi(i,s,a), \tag{2.20}$$

where r_t is the random variable of the reward that the agent receives at step t, N_1 is the number of primitive actions to finish action a, and N_2 is the number of primitive actions to finish subtask M_i. Q_r^π is the expected cumulative reward for executing action a. Q_c^π is the expected cumulative reward when subtask M_i finishes after the execution of action a. Q_e^π is the expected cumulative reward when the episode ends after the execution of subtask M_i .

They show that a hierarchically optimal policy can be learned with hierarchically optimal recursive Q-learning (HORDQ):

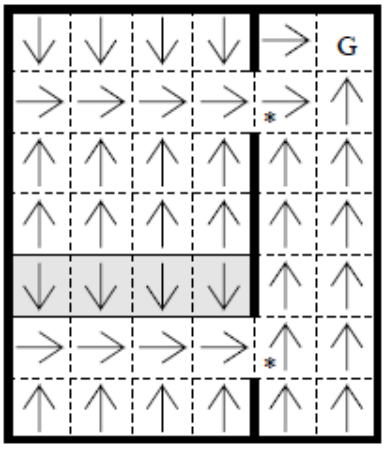


Figure 2.1: A simple maze problem [12].

$$Q_r^{t+1}(i,s,a) \leftarrow (1-\alpha_t)Q_r^t(i,s,a) + \alpha_t R_t(s'|s,a) \text{ if } a \text{ is primitive} \tag{2.21}$$

$$Q_c^{t+1}(i,s,a) \leftarrow (1-\alpha_t)Q_c^t(i,s,a) + \alpha_t \gamma^N [Q_r^t(i',s',a') + Q_c^t(i',s',a')] \tag{2.22}$$

$$Q_e^{t+1}(i,s,a) \leftarrow \begin{cases} (1-\alpha_t)Q_e^t(i,s,a) + \alpha_t \gamma^N [Q_e^t(i',s',a')] \text{ if } s' \in S_i \\ (1-\alpha_t)Q_e^t(i,s,a) + \alpha_t \gamma^N [Q^t(i',s',a')] \text{ if } s' \in T_i \end{cases} \tag{2.23}$$

2.5.3 Recursive optimality vs hierarchical optimality

There are three definitions of optimality in HRL [12]:

Definition 5 ***Optimality:*** *An optimal policy π^* for MDP M is a policy that achieves the highest cumulative reward among all policies for the MDP.*

Definition 6 ***Recursive Optimality:*** *A recursively optimal policy for MDP M with hierarchical decomposition $M' = \{M_0, M_1, \ldots, M_n\}$ is a policy $\pi = \{\pi_0, \pi_1, \ldots, \pi_n\}$ such that each policy π_i achieves the highest cumulative pseudo-reward for the corresponding subtask M_i, assuming fixed policies for its child subtasks.*

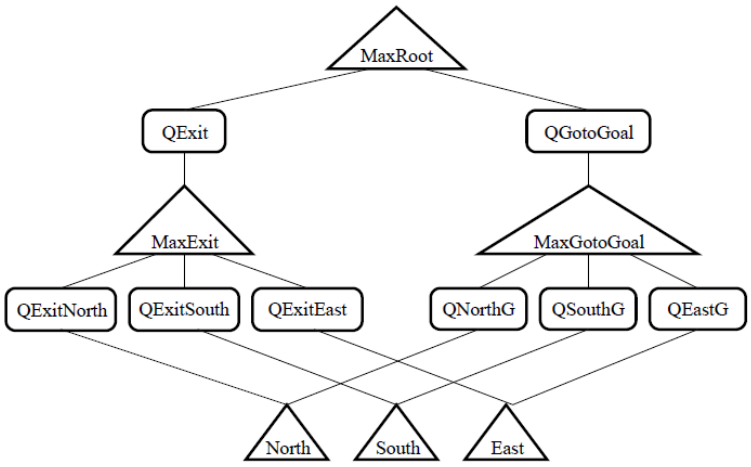


Figure 2.2: The task hierarchy of the simple maze problem.

Definition 7 ***Hierarchical Optimality:*** *Given Pi_H, which is the set of all policies consistent with hierarchy H, then a hierarchically optimal policy for MDP M is a policy $\pi^* \in Pi_H$ that achieves the highest cumulative reward among all policies $\pi \in Pi_H$.*

An optimal policy is what we want to find for any MDP M. However, it may not be possible due to the constraints imposed by the hierarchy. Instead of seeking optimality, we can seek hierarchical optimality or recursive optimality, which are two important optimality guarantees in HRL.

Recursive optimality guarantees that the policy of each subtask is optimal given the policies of its child subtasks. In this form of optimality, each subtask learns the optimal policy while ignoring the policy from its ancestors and the all subsequent rewards after the agent arrives the terminal states of the subtask. Recursive optimality allows the policy of subtask to be reused for different hierarchies. Since each subtask only needs to seek the optimality within its own subproblem, it is also possible to adopt state abstraction by ignoring the state variables which are irrelevant to the subproblem. The MAXQ-Q [12] algorithm converges to a recursively optimal policy.

A hierarchically optimal policy is the policy which achieves the highest cumulative reward given the hierarchical constraints. The hierarchically optimal policy is a stronger form of optimality. It may achieve a higher cumulative reward than a recursively optimal policy. In hierarchical optimality, the policy of each subtask may not be optimal within its own subproblem, but the overall policy of the entire hierarchy is optimal. The HAMQ [25], SDMP [33], tracked Q and HORDQ [1] algorithms learn a hierarchically optimal policy.

Dietterich [12] demonstrates the difference of recursively and hierarchically optimal policies with a simple Maze problem (Figure 2.1). A robot starts at left room and it needs to reach the goal G in the right room. It has three primitive actions, North, South and East. The robot receives a reward of -1 for each move. There are two subtasks, Exit and GotoGoal. Subtask Exit terminates when the robot exits the left room, and GotoGoal terminates when the robot reaches the goal. The arrows in Figure 2.1 show the recursively optimal policy. The arrows in the left room indicate a policy which seeks to exit the left room with minimum steps. The arrows in the right room seek a shortest path to the goal. Note that the policy in the shaded area is recursively optimal but not hierarchically optimal nor optimal. A hierarchically optimal policy should exit the left room by the upper door, but a recursively optimal policy always exits the room with minimum moves since a recursively optimal policy ignores the consequences after the subgoal is achieved.

Despite the fact that a hierarchically optimal policy is better than a recursively optimal policy, the algorithms which learn the hierarchically optimal policy do not always yield a speedup over the flat algorithms such as Q-learning or SARSA [2, 12].

Chapter 3

Learning the optimal policy with HORDQ

In this chapter, we show how to exploit the property of hierarchical optimal reinforcement learning to help model-based methods learn the optimal policy by combining them with model-free methods. The main idea is to exploit the fact that a hierarchically optimal reinforcement learning method allows a subtask to know the ”consequence” of its own action, thus it will pursue a subgoal only when it leads to the optimal policy. It allows the subtask to act optimally regardless of which subgoal is set by its parent subtask.

However, a hierarchically optimal method alone cannot guarantee that the optimal policy can be always learned. To ensure that the optimal policy can be learned, we need to ”patch” the hierarchy to allow some subtasks to be able to solve the whole MDP on their own. The result is presented in section 3.1.

It is worth noting that what we are seeking is optimality, not recursive optimality nor hierarchical optimality. Since we allow some of subtasks to learn a suboptimal policy, the recursive optimality cannot be achieved. The hierarchical optimality cannot be achieved as well before the hierarchy is patched. In Theorem 2, we show that the hierarchically optimal policy is equal to optimal policy with a patched hierarchy. In other words, seeking the optimality or hierarchical optimality are the same with a patched hierarchy.

In this work, we follow a hierarchical formulation that is similar to the MAXQ

framework [12]. An MDP M is decomposed into a finite set of subtasks $M' = \{M_0, M_1, \ldots, M_n\}$, where M_0 is the root subtask. Each subtask is defined by 3 tuples $< U_i, A_i, R_i >$, where U_i is a termination predicate that partitions the state space into active states S_i and terminal states T_i. A_i is a set of actions for subtask M_i, and R_i is the reward function for subtask M_i. There are two types of actions: primitive and composite. A primitive action corresponds to one of the actions in MDP M, which takes one step to finish. A composite action corresponds to a subtask in the hierarchy, which will not finish before the subtask arrives one of its terminal states.

Different from MAXQ, the reward function R_i is separated into internal reward function and external reward function. The internal reward function is equivalent to the reward function of MAXQ. It defines the pseudo-reward that encourages the agent to pursue the subgoal of each subtask. The internal reward function is defined for terminal states, and zero elsewhere. The external reward function is identical to the reward function of MDP M. It encourages the agent to pursue the goal defined by the original MDP M. The analysis in section 3.1 assumes the pseudo-reward is zero everywhere. The issues with the pseudo-reward are discussed in section 3.2.

3.1 Optimal planning with task hierarchy

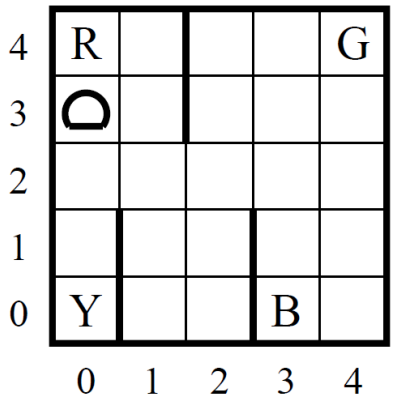


Figure 3.1: The Taxi Domain.

We illustrate our idea with the task hierarchy of Taxi domain [12] (Figure 3.2 and 3.1). The taxi problem is an episodic task. For each episode, the taxi starts at a

random location. To finish the task, the taxi needs to go to the passenger's location, pick up the passenger, go to the destination, and put down the passenger. The task can be further decomposed two subtasks: *Get* and *Put*. The goal of subtask *Get* is to move the taxi to the passenger's location and pick up the passenger. The goal of subtask *Put* is to put down the passenger at the destination.

Assume the policy of subtask *Root* is suboptimal and always invokes *Get* even when the passenger is already in the taxi. Optimality can be guaranteed if subtask *Get* learns to deliver the passenger to his destination and put him down. Or suppose the policy of *Root* always chooses *Put*. If subtask *Put* learns to navigate to the passenger's location and pick him up when he is not in the taxi, we will have the optimal policy because it does not matter which decision is made by *Root*, the passenger can always be picked up and delivered to the destination.

The above example provides two observations. First, in order to guarantee optimality, subtasks *Get* and *Put* need to act optimally in regards to the goal of whole problem, not the subgoal of each subtask. It implies that we need to seek hierarchical optimality rather than recursive optimality.

Second, optimality cannot be guaranteed without modification of the hierarchy. In the original hierarchy, subtask *Get* has no access to action *Putdown*. Even though subtask *Get* delivers the passenger to the destination, it cannot put him down. We need to modify the hierarchy to let subtask *Get* be able to solve the problem on its own. A way to achieve that is to let subtask *Get* have access to action *Putdown*.

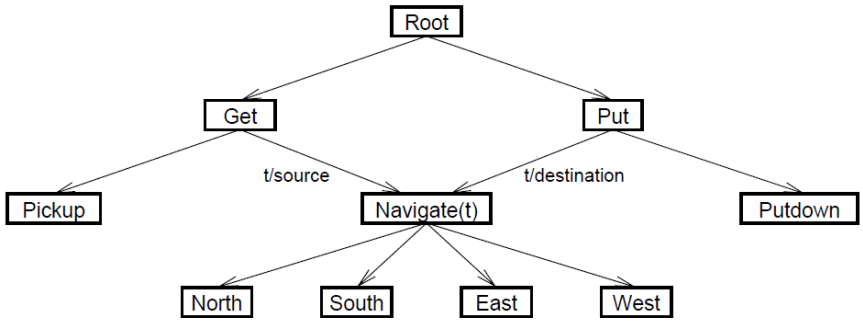


Figure 3.2: The task hierarchy of Taxi domain.

We define which subtasks shall act optimally with the following definition:

Definition 8 *$C(H) = \{M_{j_1}, M_{j_2}, \ldots, M_{j_k}\}$ is a leaf cover of hierarchy H if there is no subtask $M_i \notin C(H)$ which has access to a primitive action. Furthermore, $TC(H)$ is a total leaf cover if it is a leaf cover and all primitive actions are directly or indirectly (through child subtasks) accessible for every subtask $M_i \in TC(H)$.*

We can always find a leaf cover for a hierarchy by including all subtasks which have access to primitive actions. The total leaf cover can be constructed from the leaf cover by adding the missing primitive actions. Consider the task hierarchy in the Taxi domain in Figure 3.2, if we add action *Pickup* and *Putdown* to subtask *Navigate*, we get a total leaf cover which consists of subtasks *Get*, *Put*, and *Navigate*.

This conversion increases the exploration space because each subtask needs to explore more actions. It may increase the time to learn the optimal policy. However, as we show in our experiment, a good approximate model can effectively increase the learning rate, so the time to learn the optimal policy might still decrease overall.

Andre and Russell [1, 2] introduced hierarchical optimal recursive decomposed Bellman equations which extend the decomposition of MAXQ in a way that hierarchical optimality can be guaranteed.

The Q-value is decomposed as:

$$Q^{\pi}(i,s,a) = E[\sum_{t=0}^{\infty} \gamma^t r_t] = E[\sum_{t=0}^{N_1-1} \gamma^t r_t] + E[\sum_{t=N_1}^{N_2-1} \gamma^t r_t] + E[\sum_{t=N_2}^{\infty} \gamma^t r_t] \tag{3.1}$$

$$= Q_r^{\pi}(i,s,a) + Q_c^{\pi}(i,s,a) + Q_e^{\pi}(i,s,a), \tag{3.2}$$

where r_t is the random variable of the reward that the agent receives at step t, N_1 is the number of primitive actions to finish action a, and N_2 is the number of primitive actions to finish subtask M_i. Q_r^{π} is the expected cumulative reward for executing action a. Q_c^{π} is the expected cumulative reward when subtask M_i finishes after the execution of action a. Q_e^{π} is the expected cumulative reward when the episode ends after the execution of subtask M_i .

Q_r^π can be computed as:

$$Q_r^\pi(i,s,a) = \begin{cases} Q_r^\pi(a,s,\pi_a(s)) + Q_c^\pi(a,s,\pi_a(s)) & \text{if } a \text{ is composite} \\ \Sigma_{s'} P(s'|s,a) R(s'|s,a) & \text{if } a \text{ is primitive} \end{cases} \tag{3.3}$$

Q_c^π can be computed as:

$$Q_c^\pi(i,s,a) = \sum_{s',N} P_{S_i}^\pi(i,s',N|s,a)\gamma^N[Q_r^\pi(i,s',\pi_i(s')) + Q_c^\pi(i,s',\pi(s'))], \tag{3.4}$$

where $P_{S_i}^\pi(i,s',N|s,a)$ is the probability that s' is the first state in S_i which is encountered after the execution of action a which takes exactly N steps to finish.

And Q_e^π:

$$Q_e^\pi(i,s,a) = \sum_{s',N} P_{T_i}^\pi(k,s',N|s,a)\gamma^N[Q^\pi(k,s',\pi_k(s'))], \tag{3.5}$$

where k is the index of parent subtask which invoked subtask M_i.

To guarantee the optimality, we cannot use the Q-value of parent subtask M_k to update Q_e^π of subtask M_i because the Q-value might not be correct due to the biased model. Instead, we update Q_e^π with the Q-value of next subtask in $TC(H)$. Hence, we modify Equation 3.5 as:

$$Q_e^\pi(i,s,a) = \sum_{s',N} P_{T_i}(i',s',N|s,a)\gamma^N[Q^\pi(i',s',\pi_{i'}(s')], \tag{3.6}$$

where i' is the next subtask in $TC(H)$ that will be invoked, $P_{T_i}^\pi(i',s',N|s,a)$ is the probability that s' is the first state in T_i which is encountered after the execution of action a which takes exactly N steps to finish. Note that the property of leaf cover ensures that we can always find such subtask $M_{i'}$ before any primitive action is executed.

Theorem 1 *Let $C(H)$ be a leaf cover of hierarchy H, if $Q^\pi = Q_r^\pi + Q_c^\pi + Q_e^\pi$ and Q_r^π, Q_c^π,and Q_e^π follow Equations (3.3-3.4) and (3.6), we have Q^π satisfies:*

$$Q^\pi(i,s,a) = \begin{cases} \sum_{s'} P(s'|s,a)[R(s'|s,a) + \gamma Q^\pi(i',s',\pi_{i'}(s'))], & \textit{if a: primitive} \\ Q^\pi(a,s',\pi_a(s')), & \textit{if a:composite} \end{cases}$$

Although we changed the formula for Q_e^π, the argument of Theorem 10 in [1] still holds. We do not repeat the proof here.

With Theorem 1, we can prove that the optimal policy can be learned if there exists a total leaf cover for the hierarchy:

Theorem 2 *Given an MDP M and a hierarchy H that decomposes M into a finite set of subtasks $M' = \{M_0, M_1, \ldots, M_n\}$, let A_p denote the set of primitive actions for M, $Q^*(i,s,a)$ be the optimal Q-function for subtask M_i, and $Q^*(s,a)$ be the optimal Q-function for M. If $TC(H)$ is a total leaf cover of H, we have $Q^*(i,s,a) = Q^*(s,a), \forall s \in S_i, a \in A_p, M_i \in TC(H)$*

Proof: Let $\pi_f : S \to A_p$ be a policy for M. We can construct a hierarchical policy π, such that $\pi_i(s) = \pi_f(s) = a$, if $a \in A_p$, $\forall M_i \in TC(H)$ (if a is not directly accessible by M_i, we can let $\pi_i(s)$ be one of its composite actions). From Theorem 1, we know:

$$Q^\pi(i,s,a) = \sum_{s'} P(s'|s,a)[R(s'|s,a) + \gamma Q^\pi(i',s',\pi_{i'}(s'))]. \tag{3.7}$$

If $a_2 = \pi_{i'}(s')$ is a composite action, we have $Q^\pi(i',s',a_2) = Q^\pi(a_2,s',\pi_{a_2}(s'))$. We can keep applying the substitution until $\pi_{a_k}(s') \in A_p$, for some k. Since there are no indirect or direct recursive calls allowed in the hierarchy, the substitution can be done in finite steps. Now we have:

$$Q^\pi(i,s,a) = \sum_{s'} P(s'|s,a)[R(s'|s,a) + \gamma Q^\pi(a_k,s',\pi_{a_k}(s'))]. \tag{3.8}$$

Note that $a_k \in TC(H)$ because all subtasks which have direct access to primitive actions belong to $TC(H)$. By the construction of π, we have $\pi_{a_k}(s') = \pi_f(s')$.

Compare (3.8) to the Bellman equation of the flat MDP:

$$Q^{\pi_f}(s,a) = \sum_{s'} P(s'|s,a)[R(s'|s,a) + \gamma Q^{\pi_f}(s',\pi_f(s'))]. \tag{3.9}$$

Equations (3.8) and (3.9) are identical except for the Q values. Due to the uniqueness of the Bellman equation, we have $Q^{\pi_f}(s,a) = Q^\pi(i,s,a), \forall s \in S_i, a \in A_p, i \in TC(H)$. If $\pi_f(s) = \pi_f^*(s)$, $Q^*(s,a)$ is a solution to equation (3.8). So we

have $Q^*(i,s,a) \geq Q^*(s,a)$. Since a hierarchical policy cannot be better than an optimal policy, we have $Q^*(s,a) \geq Q^*(i,s,a)$. Thus we have $Q^*(i,s,a) = Q^*(s,a)$. **Q.E.D.**

Note that the above proof does not pose any constraints on the policy of subtasks $M_i \notin TC(H)$. If we adopt any learning algorithms for such subtasks, we still have the same optimality guarantee. We can compute the optimal policy using either policy iteration or value iteration algorithms. The arguments of Theorem 11 and 13 of [1] hold in our case. We do not repeat the arguments here.

The previous equations assume we have complete knowledge about the problem and can compute the Q-value with dynamic programming. If not, we can estimate the Q-value with the hierarchically optimal recursive Q-learning (HORDQ) [1] update rules:

$$Q_r^{t+1}(i,s,a) \leftarrow (1-\alpha_t)Q_r^t(i,s,a) + \alpha_t R_t(s'|s,a) \text{ if } a \text{ is primitive} \tag{3.10}$$

$$Q_c^{t+1}(i,s,a) \leftarrow (1-\alpha_t)Q_c^t(i,s,a) + \alpha_t \gamma^N [Q_r^t(i',s',a') + Q_c^t(i',s',a')] \tag{3.11}$$

$$Q_e^{t+1}(i,s,a) \leftarrow \begin{cases} (1-\alpha_t)Q_e^t(i,s,a) + \alpha_t \gamma^N [Q_e^t(i',s',a')] \text{ if } s' \in S_i \\ (1-\alpha_t)Q_e^t(i,s,a) + \alpha_t \gamma^N [Q^t(i',s',a')] \text{ if } s' \in T_i \end{cases} \tag{3.12}$$

where i' is the index of next subtask in $TC(H)$ that will be invoked and $a' = argmax_b Q^t(i',s',b)$.

Unfortunately, the convergence of HORDQ is an open problem (Conjecture 1 of [1]). However, if we let all subtasks in $TC(H)$ have access to all primitive actions directly, the convergence to the optimal policy can be guaranteed.

Theorem 3 *Let $TC(H)$ be a total leaf cover for a hierarchy H, and $Q^t = Q_r^t + Q_c^t + Q_e^t$. If we use equations (3.10-3.12) to update the Q-values for all subtasks in $TC(H)$, we have $lim_{t\to\infty} Q^t(i,s,a) = Q^*(i,s,a)$ if the following conditions hold:*

- *Every subtask in $TC(H)$ has access to all primitive actions and can only execute primitive actions*

[1] Also called ALispQ-learning in [1, 2]

- *A greedy in the limit with infinite exploration (GLIE) exploration policy is followed by every subtask in* $TC(H)$
- $Var\{R_t(s'|s,a)\}$ *is finite*
- $\sum_t \alpha_t = \infty$ *and* $\sum_t \alpha_t^2 \leq \infty$
- $0 < \gamma < 1$ *or* $\gamma = 1$ *and all policies* π_i *are proper*

Proof:

$$\begin{aligned}
Q^{t+1}(i,s,a) =& Q_r^{t+1}(i,s,a) + Q_c^{t+1}(i,s,a) + Q_e^{t+1}(i,s,a) \\
=& (1-\alpha_t) Q_r^t(i,s,a) + \alpha_t R_t(s'|s,a) + \\
& (1-\alpha_t) Q_c^t(i,s,a) + \alpha_t \gamma^N [Q_r^t(i',s',a') + Q_c^t(i',s',a')] + \\
& (1-\alpha_t) Q_e^t(i,s,a) + \alpha_t \gamma^N [Q_e^t(i',s',a')] \\
=& (1-\alpha_t)[Q_r^t(i,s,a) + Q_c^t(i,s,a) + Q_e^t(i,s,a)] + \\
& \alpha_t [R_t(s'|s,a) + \gamma^N [Q_r^t(i',s',a') + Q_c^t(i',s',a') + Q_e^t(i',s',a')]] \\
=& (1-\alpha_t) Q^t(i,s,a) + \alpha_t [R_t(s'|s,a) + \gamma^N Q^t(i',s',a')]
\end{aligned}$$

Since a and a' are primitive actions, the HORDQ rule identical to the standard Q-learning update rule. Therefore, it converges under the same condition as Q-learning. **Q.E.D.**

3.2 Pseudo-reward

One of the problem of hierarchically optimal RL is that policy π_i of subtask M_i is determined by the reward of the original MDP. There are no pseudo-rewards which are allowed as in MAXQ. Due to the lack of pseudo reward, each subtask M_i lacks the motivation to pursue the subgoal defined by the hierarchy, thus it makes the hierarchical design useless. It is necessary to add some pseudo-rewards to encourage each subtask to pursue the subgoal. However, we cannot guarantee optimality with a nonzero pseudo-reward.

In fact, the above argument illustrates the difference between the hierarchically optimal policy and the recursively optimal policy. As shown in the experiment of

[2], hierarchically optimal RL learns a better policy than the one of recursively optimal RL. However, its learning rate is slower than the recursively optimal one.

Since neither approach is better than the other in terms of learning rate and the quality of policy, there are no reason to stick to one of them. With the introduction of pseudo-reward, we can alternate the agent's behavior between the hierarchically optimal RL and the recursively optimal RL.

An approach is to use positive reward to encourage the effective exploration in the early stage, and gradually decrease it to 0 to learn the hierarchically optimal policy. Since the hierarchically optimal policy is equal to the optimal policy when the hierarchy contains a leaf cover, the optimal policy will be learned with a pseudo-reward equal to 0. The result in presented in Chapter 4.

Chapter 4

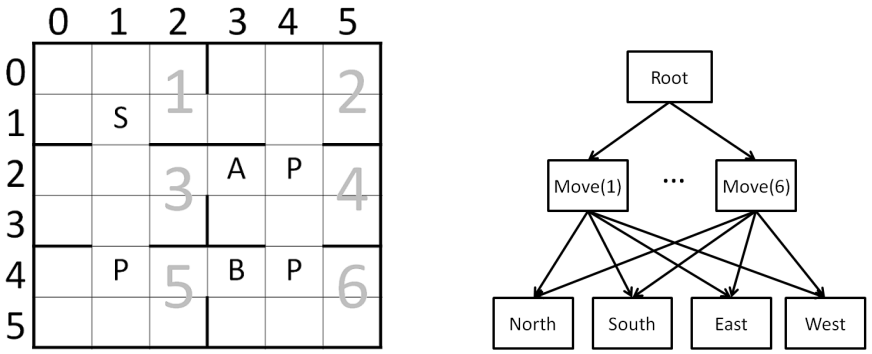
Empirical study

In this chapter, we demonstrate the effectiveness of our approach in a Bus domain and the Infinite Mario domain. The Bus domain is a 6 by 6 grid world problem. We have 2304 states in this domain, so it is small enough to apply table-lookup methods to learn the optimal policy. In this experiment, we combine table-lookup HORDQ and an approximate model-based method to illustrate how to increase the learning rate and learn a near-optimal policy with our approach.

For large problems with more than millions of states, it is not possible to use table-lookup methods to learn the optimal policy. Instead, it is necessary to adopt function approximation techniques to learn the policy. However, the optimality guarantee of Theorem 2 relies on table-lookup methods and does not hold for function approximation techniques. With function approximation, it is not possible to learn the optimal policy anymore. We illustrate our work with Infinite Mario to show that despite the fact that the optimality guarantee is lost in large problems, we can still use model-free methods to help model-based methods handle the effects which are not included in the model.

4.1 Bus domain

The Bus domain is introduced in this section. We use it as an example to show that model-based methods can have faster learning rates, even with a biased model. We also use it to illustrate how to combine them with model-free methods to improve



(a) (b)

Figure 4.1: (a) The Bus domain (b) A task graph for the Bus domain.

the learned policy.

Figure 4.1(a) shows the Bus domain. The bus starts at S. Its task is to pickup all passengers marked as P and return to S. The bus can move North, South, East, or West. There is a reward of -1 applied for each action. There is a 0.1 probability for the bus to move in a random direction. It stays put when trying to cross a wall. The passengers are picked up automatically when the bus moves into the location of a passenger. The passengers can be picked up in any order. The episode ends when it finishes the task. There are two possible locations for road construction. They are marked as A and B. If the bus passes a construction site, it will get damaged with probability 1 and has the probability of 0.25 of breaking down for each step afterwards. There is a reward of -50 if the bus breaks down and the episode ends immediately. At the beginning of an episode, the status of the road is randomly chosen from no construction, A is under construction, or B is under construction. There is a 0.05 probability for the road status to change for each step. The world is divided into six areas. There are six subtasks $Move(1), \ldots,$ and $Move(6)$ which move the bus to the corresponding area. Subtask $Move(t)$ can only be invoked if t is the adjacent area. The subtask terminates if the bus exits the current area. When it terminates, a pseudo-reward is applied if the bus arrives at designated area t and 0 otherwise. The task hierarchy is shown in Figure 4.1(b).

A state can be described by a 8-tuple $(x, y, h, p_1, p_2, p_3, a, b)$, where (x, y) is the location of the bus, h shows if the bus is damaged, p_i indicates if the correspond-

ing passenger has been picked up or not, and a and b are binary variables which indicate the status of the construction sites.

Since the six subtasks $Move(1), \ldots,$ and $Move(6)$ cover all primitive actions, they form a total leaf cover of the hierarchy. We used HORDQ in the subtasks to guarantee the convergence to the optimal policy. Subtask *Root* adopted our approximate model-based method.

4.1.1 Planning with static assumptions

Suppose the variables of our model are reduced to (x, y, p_1, p_2, p_3) and assume that the damage status and road conditions are static during the planning process, our model can learn that a large penalty will be received when the bus is damaged. However, our model cannot learn that the bus will get damaged if it passes through a construction site. The model is certainly biased in this case, thus it cannot learn the optimal policy. The objective of the experiment is to show that the optimal policy can still be learned if we combine the model-based approach with HORDQ.

Let state $s = (x, y)$, where x consists of planning variables and y consists of environment variables. Following the MAXQ approach, we decompose the Q-function as:

$$Q^{\pi}(i, x, j) = Q_r^{\pi}(i, x, j) + Q_c^{\pi}(i, x, j), \tag{4.1}$$

where $Q_r^{\pi}(i, x, j)$ is provided by the child subtask M_j.

The task of subtask M_i is to compute $Q_c^{\pi}(i, x, j)$ by:

$$Q_c^{\pi}(i, x, j) = \sum_{x'} P_m^{\pi}(x'|s, j)[Q_r^{\pi}(i, x', \pi_i(x')) + Q_c^{\pi}(i, x', \pi_i(x'))], \tag{4.2}$$

With the formula above, the Q-values can be computed by dynamic programming.

For simplicity, we use the multi-time model [33] to model the transition function:

$$P_m(x|s, j) = \sum_{N=1}^{\infty} \gamma^N P(x, N|s, j). \tag{4.3}$$

$P_m(x|s,j)$ can be estimated by:

$$\tilde{P}_m(x|s,j) = (1-\alpha)\tilde{P}_m(x|s,j) + \alpha[\gamma^N \delta_{x'x}], \tag{4.4}$$

for all $x \in S_i$, where $\delta_{x'x} = 1$ if $x' = x$ and is 0 otherwise.

4.1.2 Empirical results in the Bus domain

Figure 4.2(a) shows the learning curves with different levels of pseudo-rewards. With pseudo-reward +60, it learned a suboptimal policy because the pseudo-reward is too large to make subtask $Move(t)$ ignore the penalty of breakdown. As a result, the subtask followed the instruction of its parent too strictly.

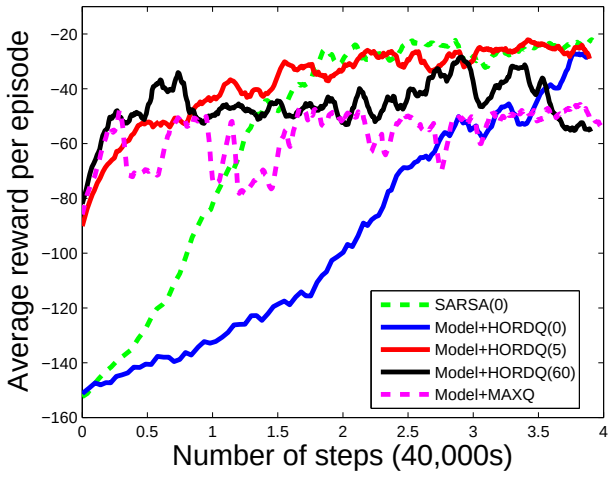
On the other hand, if we do not impose any pseudo-reward, the optimal policy can be learned, but the learning rate is slower than SARSA(0) learning. Since subtask $Move(t)$ has no incentive to follow the instruction of its parent subtask, the learning process is similar to SARSA(0) learning except it has six different Q-functions to learn (one for each subtask) instead of one. Thus it takes longer to learn the optimal policy.

With an appropriate pseudo-reward, we can get a near-optimal policy while the learning rate is faster than SARSA(0). Our experiment shows that a pseudo-reward of +5 is enough to make subtask $Move(t)$ follow the order of *Root* in most of the times, but it is not enough for the subtask to ignore the breakdown penalty. For example, when *Root* executes $Move(4)$ to move the bus from area 3 to area 4 and the road at location *A* is under construction, $Move(4)$ subtask will learn it is a bad decision with HORDQ. Instead of moving to area 4, $Move(4)$ may move to area 1 or 5 to avoid the breakdown penalty. In turn, *Root* learns that $Move(4)$ cannot be executed in such a scenario, thus it will seek an alternative plan if the same scenario is encountered.

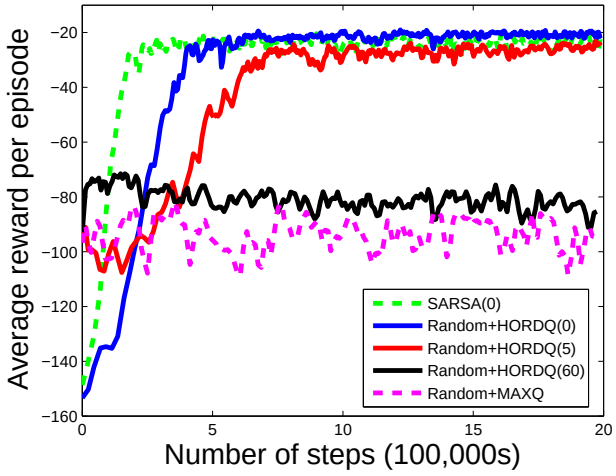
To illustrate the undesirable result when some suboptimal subtasks exist in the hierarchy and the MAXQ framework is adopted, we combine our approximate model-based approach and MAXQ-Q learning in our experiment. The combination learns a suboptimal policy similar to HORDQ with high pseudo-reward. Since MAXQ does not estimate the consequence of its action outside its own subtask, $Move(t)$ will move to area t at any cost. It leads to the frequent damage of the bus.

To simulate the performance of the combination of a poorly-approximated model-based method and HORDQ, we replaced our model-based approach with a random policy. The result is shown in Figure 4.2(b). In this case, SARSA(0) has the fastest learning rate. It takes more time for $Move(t)$ to realize that the policy of *Root* is bad with higher pseudo rewards. Nevertheless, it will eventually learn a near-optimal policy. The combination of random policy and MAXQ presented the worst result.

This result shows that a good approximate model can help increase the learning rate with the combination of HORDQ. If the model is poor, HORDQ serves as a fail-safe mechanism to keep the agent from repeating the same poor policy over and over again. On the other hand, MAXQ learned a poor policy in both cases. This evidence suggests that in order to construct a robust HRL algorithm, it is beneficial to incorporate HORDQ in the hierarchy.



(a)



(b)

Figure 4.2: The learning curve for the Bus domain, averaged over 400 episodes. (a) With our model-based approach. (b) With random policy. The pseudo-reward is shown in parentheses. The parameters are $\alpha = 0.1$ and $\gamma = 1$. All algorithms follow an ε-greedy exploration policy with $\varepsilon = 0.1$.

4.2 Infinite Mario

We use Infinite Mario to show the effectiveness of our approach in large domains. Large domains contain more than millions of states, so it makes table-lookup methods not applicable due to the curse of dimensionality. Approximation techniques are required to handle these problems, but the optimality guarantee will be lost. It is interesting to see how our work will perform with function approximation techniques, especially when model-free methods cannot learn the optimal policy.

4.2.1 Previous work

Infinite Mario is an open source Java implementation of Nintendo's Super Mario Brothers game. It received much attention in the AI community possibly due to the two AI competitions – RL 2009 competition[1] and Mario AI competition[2], which were held in 2009. The objective of these competitions is to build the best agent that can play this classic side-scrolling arcade-style game.

The RL 2009 competition required competitors to use RL algorithms to build their agents. On the other hand, the Mario AI 2009 competition [37] did not pose any restrictions on the underlying technique. It encouraged competitors to use neural networks, genetic genetic programming, fuzzy logic, temporal difference learning, and human ingenuity.

The observation of Mario AI in RL 2009 competition contains the location and speed of Mario and other monsters, as well as the 22×16 tiles on the screen. The agent receives +100 reward when it finishes a level, +1 reward when collects a coin, and -10 reward when it dies.

Mohan and Laird [21] combined HRL and Soar-RL [24] to build a Mario agent. In their hierarchy, the task is divided to "Grab Coin", "Search Question", "Tackle Monster", "Avoid Pit", and "Move to Goal". The root subtask chooses one of these subtasks to execute based on the learned preferences. Each subtask deals with at most one object. For example, the "Grab Coin" subtask is considered successful only when the specified coin is collected. Thus, the number of subtasks available for the root task to choose depends on the current objects in the screen.

[1] http://2009.rl-competition.org/

[2] http://julian.togelius.com/mariocompetition2009/

Gibson and Risk [14] adopts the options framework. The master agent can execute 3 options – a $SARSA(\lambda)$ agent, a "pit specialist" agent, and a rule-based agent. The "pit specialist" concerns only with the pits, and is available only when there is a pit within 4 tiles of Mario. The master agent uses SMDP Q-learning to learn which option to execute given a state.

Ringstad et al. [41] used a modified linear SARSA algorithm to build the agent. The input features are the locations of the 3 nearest monsters or pits. Their idea is to design an agent that can finish the level as fast as possible. To encourage the agent to finish the level, they rewarded the agent when it traverses intervals of 10 tiles away from the start. Their method won the Mario AI championship of RL competition 2009.

The state observation of Mario AI 2009 competition [37] includes 22×22 tiles on the screen, the location and the speed of monsters, and the status of Mario such as "isMarioOnGround" or "mayMarioJump".

The techniques which are adopted by the competitors of Mario AI competition include A^* search, genetic programming, hand-coded policy, a hybrid method of neural network and A^* search, and Cyberneurons. In general, A^* search achieves the best result, and hand-coded policy falls the second. Robin Baumgarten [37] won the championship of the Mario AI competition in 2009. His idea is to create a physics engine that can accurately predict the next state of Mario, and use A^* search to find a path to the right border of the screen as fast as possible. Since his approach requires the agent to have the complete knowledge of the environment, the approach falls into the category of classical planning.

Ross et al. [28] used supervised learning to learn the direct policy mapping between input features and the primitive actions. The input features are 22×22 tiles around Mario in previous 4 frames, the state of Mario and the last 6 actions. The tiles include the types of the ground, blocks, and monsters. The state of Mario includes the types (small, big and fire Mario) as well as a binary feature to indicate if Mario touches the ground or not. The training data is obtained through search-based methods similar to Baumgarten's work [37].

The above approaches either depend on game-specific information ("isMarioOnGround") or depend on the information that is difficult to be retrieved from image features (the location and size of pits). In this section, we introduce an ap-

proach of building an agent for Infinite Mario without using such information. We only restrict the agent to use the features provided by the simulator of Infinite Mario from RL competition 2009. The features can be retrieved from the screen directly, therefore it allow us to generalize our approach to other video games without the need to redesign specialized features for each individual problem.

4.2.2 Infinite Mario domain

We use the 2009 RL competition environment to conduct our experiment. The action space of Infinite Mario consists of 4 buttons which correspond to the original Nintendo controller. These buttons are:

- Direction pad: left or right
- A button: jump
- B button: speed

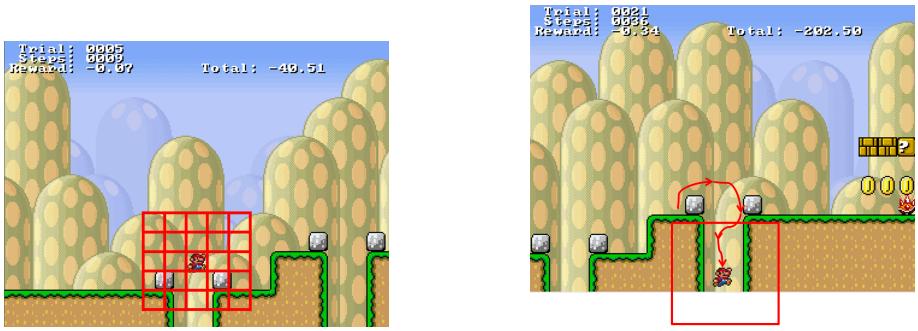
Mario can choose to press these buttons or not, so the number of possible actions are $3 \times 2 \times 2 = 12$ actions.

We exclude action "speed", "jump speed" and "no op" from the action space, since they do not seem to be relevant to the optimal policy. The agent can execute 9 actions in our experiment.

The screen of Infinite Mario is comprised of a matrix of 22×16 tiles. The matrix is an array of characters, with each element representing the type of tile. The types of tile are brick, question-block, coin, pipe, empty tile, the finish line and Mario.

Besides the tile information, the information of moving objects are also provided. The moving objects are Mario, Red Koopa, Green Koopa, Goomba, Spikey, Piranha Plant, Mushroom, Shell, Fire Flower and Fireball. The information of each object includes x- and y-positions, x- and y-velocities, and type of object. Note that the positions and velocities are continuous, so a quantization technique might be required.

The tile information can be captured from screen with basic image processing techniques [23]. The location and speed of monsters can be obtained by applying computer vision techniques such as object tracking or optical flow.



(a) (b)

Figure 4.3: (a) A screenshot of Infinite Mario (b) A planning process conducted by the search-based method.

The levels are generated with 3 parameters: random seed, type, and difficulty. The difficulty ranges from 0 (easiest) to 9 (hardest).

The agent will receive the following rewards:

- +100: finishing a level
- +1: collecting a coin
- +1 : hitting a question block
- +1 : killing a monster
- -0.01 : step cost
- -50 : getting killed

Besides, we apply a reward equal to the displacement of x-position to encourage the agent to move as right as possible and penalize the agent if it moves to left.

4.2.3 The model-based method for Mario domain

We adopt the model-based method to learn the transition function of Mario. The state of Mario can be described by a 4-tuple (x, y, dx, dy), where are the location and velocities of Mario. The ranges of x, y, dx and dy are $[0, 318]$, $[0, 15]$, $[-2.5, 2.5]$,

and $[-2.5, 2.5]$. These variable are continuous, so it is not possible to enumerate all possible states of Mario with dynamic programming techniques. It is possible to discretize the variables, but the resulting state space would still be too large if we want to predict the dynamics of Mario precisely.

Instead of discretizing the variables, we used the regression tree algorithm in the Orange package [11] to predict the future position and speed of Mario. The features are the current speed of Mario and the tiles within 5 by 5 area around Mario (Fig. 4.3(a)). There are 27 variables. Since each action has different dynamics, we build different trees for different actions. Given the current state and action, the regression trees should predict the speed and position of the following state. The regression trees predict the relative changes of positions $\varepsilon_x = x' - x$ and $\varepsilon_y = y' - y$ since modeling relative change might generalize better across states [17]. However, the relative changes of speeds do not generalize well, so we predict the value of speeds directly. The x-position, y-position, x-speed and y-speed are the class variables for the regression trees. For simplicity, we separately build regression trees for different class variables and actions. In our current implementation, there are 9 actions and 4 class variables, so we have 36 regression trees to model the dynamics of Mario.

We borrowed the idea of Baumgarten [37], using the search-based method to find a sequence of actions that moves Mario to the right edge of the screen as fast as possible. Instead of hard-coding the objective of the search process, we add a possible reward to the agent if it moves to the right and a negative reward otherwise. We adopt a simple k-step lookahead greedy search in our planning process. It begins with the current state of Mario. Then it expands one node in the search tree by applying all possible actions. To avoid fully expanding the whole search tree to the maximum depth, which is 6 in our experiment, we use greedy search and expand the node with the largest predicted reward. The process stops after the number of nodes exceeds 300, then the search algorithm returns a sequence of actions which achieves the highest predicted reward. To reduce the computation time spent on searching, the search algorithm will return immediately if there is a sequence of action which gets more than +6 reward. There are $9^6 = 531,441$ nodes in the fully-expanded tree. Since we only search a very small part of it, it is possible for the search algorithm to return a suboptimal sequence. The locations of

monsters and other objects are assumed to be the same during the planning process.

It is crucial for the search-based method to learn when Mario is going to be killed. However, the number of samples right before Mario gets killed is limited by the number of episodes due to the fact that the episode terminates immediately after the death of Mario. To efficiently use the available samples, we use a single regression tree to learn the reward function. The input features of the regression tree are the features for the transition function plus the action of the agent.

The strength of the above method is that it has the terrain knowledge and can move efficiently to finish a level.

Since we only model the dynamics of Mario, effects such as the dynamics of other objects or the interactions between objects are ignored.

Here is the list of effects which are not included in the model:

- Monsters may appear at the right edge of the screen
- Monsters disappear at the left edge of the screen
- Monsters can be killed by Mario, a fireball, or a moving shell
- Monsters can be killed by falling into a pit
- Koopma can be turned into a shell
- Mario can kick a shell to make it move
- Jump on top of a moving shell will make it stop
- Fire Mario can attack with fireballs
- Small Mario can be turned into Big Mario by consuming a mushroom
- Big Mario can be turned into Fire Mario by consuming a fire flower
- Coins, mushrooms and fire flowers can be consumed by Mario

To learn the transition function perfectly, it is necessary to learn all the effects correctly. However, learning the effects for a stochastic problem is NP-Hard [38] and heuristic solutions are required to solve it [26].

Our work provides an alternative approach to this problem–instead of learning all possible effects, we only learn part of them and let model-free methods handle the scenarios associated with the effects which are not included in the model.

The 5 by 5 tiles around Mario include the monsters as well, so it is possible for the model-based method to learn the imminent death caused by moving Mario directly to monsters. What it cannot handle is the delayed death cases, which happen when Mario moves very close to monsters, but does not touch it. In such a scenario, it doesn't matter which action Mario is going to take, the subsequent death is guaranteed. This also happens when Mario moves to a position which will be surrounded by monsters. After Mario moves to such a position, Mario will be killed inevitably. The supervised learning can only learn the reward function with immediate reward. When the death is actually caused by a decision made few steps before, it is difficult for a supervised learning algorithm to figure out such a relationship. On the other hand, such a delayed feedback will propagate back to previous states with model-free methods such as $SARSA(\lambda)$, so it is not a problem for these methods.

4.2.4 The model-free method for Mario domain

Since the model-based method cannot deal with the interaction between objects, we use a model-free method to handle it.

The features of model-free methods include the types and locations of moving objects other than Mario itself. To reduce the number of features, we do not include the speed of objects. As noted in [14], it is more generalizable if we use "egocentric representation". That is, we use the relative positions between Mario and objects as the features instead of the absolute positions.

Since the number of monsters can be any arbitrary number, we cannot use linear SARSA which depends on a fixed feature size. Instead, we use relational approaches here. We incorporated relational temporal difference learning (RTDL) [3] with HORDQ. RTDL is a relational extension to linear SARSA, thus it does not work for continuous variables. To discretize them, we simply round each variable to the nearest integer.

Unlike previous approaches [14, 21, 22, 41], we do not include the location of

pits in our features. Since the pit is not a moving object nor does it occupy a single tile, it is not available in the input features. Previous approaches relied on prior knowledge of the shape and size of pits to parse the tile information and extract the location of a pit. We argue that this would not contribute to the generality of the method. It would be possible to include the pit information by using the whole screen (22×16) as the feature. But the potential huge state space makes it inapplicable in practice (14^{352} states with 14 different types of tiles) Moreover, it is not necessary to include the pit information in model-free methods, since the pit can be handled by the model-based method.

4.2.5 The hybrid approach

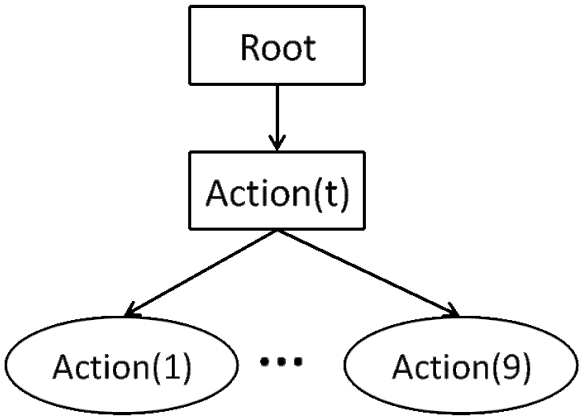


Figure 4.4: A task hierarchy for Infinite Mario

We combine the model-based method and model-free method with the task hierarchy shown in Fig. 4.4. The model-based method is responsible for subtask *Root* and the model-free method is for subtask $Action(t)$. For each step, subtask *Root* selects one of the actions to execute. Subtask $Action(t)$ then decide if it is going to follow the action suggested by *Root* or not. If it does, $Action(t)$ will receive a pseudo-reward. Since $Action(t)$ is the only subtask that executes the primitive action, the behaviour of the agent solely depends on its decision. Subtask *Root* can only influence its decision with the pseudo-reward. Note that $Action(t)$ will terminate immediately after executing a single primitive action. There is no

Method Name	# of finishing a level
SARSA	1
Model+HORDQ(10)	277
Model+HORDQ(20)	64
Model	0

Table 4.1: The number of times for Mario to finish the level within 1000 runs

temporal and spatial abstraction in this hierarchy. That means that we do not enjoy any benefits from adopting the HRL framework. In fact, the reason we adopted the HRL framework is to combine different methods. And the benefit of doing so does not come from the HRL framework, but from the power of combining model-free and model-based methods.

Unlike our approach with the Bus domain, we do not have individual subtask $Action(t)$ defined for each primitive action. Since subtask $Action(t)$ knows that it will be rewarded if it follows the action of *Root*, there is no need for us to separately learn the Q-function for every possible action. Instead, we use a single subtask $Action(t)$ to learn the Q-function for the original MDP. And the decision of $Action(t)$ is biased with the pseudo-reward when it is going to select the best action to execute. Thus, the method presented here does not have the overhead of maintaining multiple Q-functions as in our method for Bus domain.

For this special hierarchy, the pseudo-reward imposes a very interesting property: the policy is identical to the policy of model-free method when a pseudo-reward is equal to zero and it is identical to the one of model-based method given a sufficiently large pseudo-reward. We can adjust the value of pseudo-reward to alternate the agent's behaviour between two different methods.

4.2.6 The result of Infinite Mario

The level is generated with Infinite Mario simulator of RL competition 2009. The random seed is 1247 with type 0 and the difficulty is 3. The maximum distance for Mario to move to the right is 315. Beyond that, the level is finished with +100 reward.

Table 1 shows the number of times for Mario to finish the level within 1000

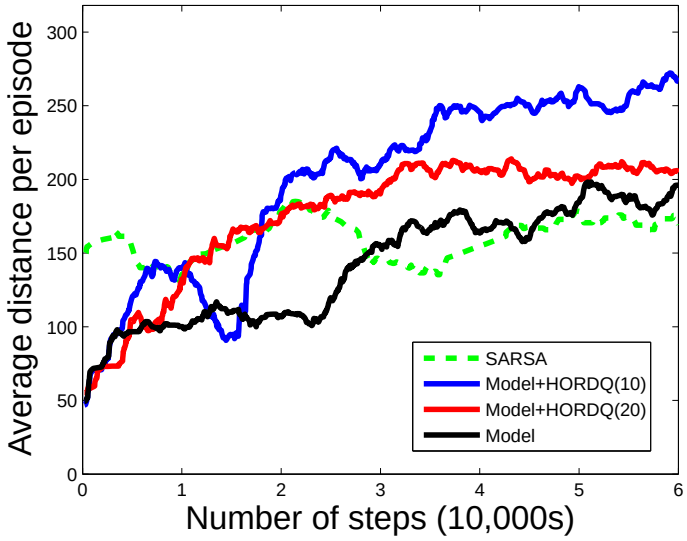


Figure 4.5: The distance that Mario moves for each episode, averaged over 20 episodes. The parameters are $\alpha = 0.05$ and $\gamma = 0.7$. The exploration policy ε-greedy with $\varepsilon = 0.01$.

episodes. Unsurprisingly, the model-based and model-free method are difficult to finish a level because they may fail to handle either monsters or pits. The model-based method only includes the dynamics of Mario, and ignores the rest of effects in game. Thus, the learned policy can get Mario killed. On the other hand, the features of model-free method do not include the information of pits, so it may fall into a pit.

Figure 4.5 shows the distance travelled by Mario with different pseudo-rewards. Unlike the experiment in the Bus domain, the model-based method does not have a better learning rate then the model-free method. The problem is that the model-based method may lead Mario to be killed by a monster in an early state, thus it suffers a decreased learning rate.

If we combine both methods, we can see that Mario now can learn a better policy since it can avoid both pits and monsters. If we increase the pseudo-reward, the chances for Mario to be killed will increase, as it follows the action of model-based method more often. If we decrease the pseudo-reward, the chances to fall

into a pit will increase, as the model-free method ignores the pit completely.

Our approach actually suggests a way to mix different methods and learn a better policy.

Chapter 5

Conclusions

Model-based methods are powerful tools. They allow us to predict the outcome of the agent behaviour and plan over it. They can effectively reduce the number of samples which are required to find the optimal policy. However, model-based methods may not be able to learn the optimal policy due to the structural assumptions. In this work, we propose an approach to combine the approximate model-based method with the model-free method (HORDQ) under the HRL framework. We are able to show that our approach can learn the optimal policy even when the assumptions of model are not satisfied. Furthermore, we show that optimality is guaranteed for any subtask policy as long as the conditions of Theorem 2 are satisfied.

In this chapter, we share our experiences about how to apply our theory to design a system. Since the performance of a system highly depends on the value of pseudo-rewards, we will also introduce some heuristics for choosing an appropriate pseudo-reward. Finally, we discuss limitations and possible directions for future works.

5.1 System design

An important design principle of the system is to design the model-based method first, and the model-free method later. We need to decide the features used by the model-based methods, the underlying supervised learning algorithms and most

importantly, the effects which we would like to include in our model. Then we run the experiments, and observe the scenarios where the model-based method fails. Based on the observations, we design a set of features for model-free methods to handle these scenarios. Note that we don't need to design a set of features to handle the whole problem, but only part of it. We only need the model-free method to take control when the model-based method fails. Therefore, we can reduce the number of features for model-free methods and let the overall system successfully handle all scenarios.

Since our work is about how to use model-free methods to improve the learned policy of model-based methods, it is not necessary to adopt our method if model-based methods can learn the optimal policy on their own.

Our work is not the only solution when model-based methods fail. Another alternative is to improve the quality of the model by including more domain knowledge. For example, in our Infinite Mario experiment, we did not include any effects of the interactions between monsters and Mario. It is possible to hand-code the preconditions and postconditions of these effects, as Walsh et al. proposed in [38]. In fact, the source code of Infinite Mario is publicly available. There is no need to use model-based methods to learn the model. Instead, we can simulate the experiences of the agent with the simulator of Infinite Mario. Since the environment and the model are identical, there are no biases which will be introduced during the simulation process. This is what Baumgarten, the champion of Mario AI competition 2009, did with his A^* method for Mario AI [37]. With the complete domain knowledge, it is unlikely for any RL methods to outperform his work.

However, the key idea of RL is to build an adaptive agent. Not only do we want the agent to perform well in a problem which we know very well, but we also want the agent to adapt itself to novel problems which we cannot foresee when we design the agent. If we put too much domain knowledge into the agent's design, we forbid it from adapting itself when the prior knowledge does not hold anymore. In this work, we introduce an alternative – instead of designing an omnipotent model-based agent, we divide the learning task into different parts and let model-free methods handle the parts which model-based methods cannot do.

5.2 Choosing an appropriate pseudo-reward

It is important to choose an appropriate pseudo-reward. If we choose a pseudo-reward which is too small, the policy of the agent will be similar to the policy of a model-free method. Therefore, we may lose the benefit of the faster learning rate. On the other hand, if the pseudo-reward is too large, the policy will be similar to the model-model method, which may be suboptimal when the assumptions of the model are not all satisfied.

It is easy to determine when a pseudo-reward is too large by looking at the difference between the expected reward of the optimal policy and the policy of model-based method. If a pseudo-reward is larger than this difference, the model-free method will follow the policy of the model-based method strictly, and the combined policy will be the same as the policy of model-based method.

In our experiments (Sections 4.1.2 and 4.2.6), a pseudo-reward larger than the expected death penalty is considered ”too large”, since it will let the model-free method follow the instruction of model-based method even when it will result in the death of the agent. If we choose a pseudo-reward which is smaller than it, the model-free method will choose an action that avoids the death of the agent.

It is more difficult to decide if the pseudo-reward is large enough. For small problems, if we adopt table-lookup HORDQ as the model-free method, the optimal policy will be learned when the pseudo-reward is decreased to zero. So a viable strategy is to choose some pseudo-reward, which is not too large, and gradually decrease it to zero. For large problems, we have adopted model-free methods with function approximation techniques , therefore the optimal policy might not be learned when the pseudo-reward is zero. Instead, we need to find out an optimal pseudo-reward which can maximize the expected reward. A way to decide it is to conduct the experiment with the model-free method, and choose a pseudo-reward which is large enough to encourage the model-free method to follow the policy of model-based one.

5.3 Limitations and future work

The quality of learned policy depends on the chosen model-free method, the model-based method and the pseudo-reward. Since we can control the pseudo-reward to

decide if the combined policy should be similar to model-free or model-based one, the combined policy can never be worse than any of them.

Since our work is a combination of the two, it will fail in scenarios where both methods fail. We could only improve on this scenario if we apply better model-free or model-based methods.

Nevertheless, our work is not useful when one method outperforms another. In general, model-based methods learn faster than model-free methods because of the efficient use of samples, but it may not be true for some problems. If the chosen model-based method is worse than the model-free method in both learning rate and the learned policy, it is pointless to combine both methods. Similarly, if the model-free method fails to handle the scenarios where the model-based method fails or the model-based method can learn the optimal policy, it is not necessary to apply our work. Our work has its edge when the model-based method learns faster than the model-free method but learns a worse policy compared to the model-free one. For small problems, if the model-based method fails to learn the optimal policy, we learn the optimal policy by combining it with table-lookup HORDQ as we prove in Theorem 2. For large problems, it is difficult since approximated model-free RL may not learn the optimal policy. It is necessary to have the knowledge about the domain and apply the knowledge to choose some good features.

We introduced the theory of improving the quality of the policy of model-based methods. However, we don't have any theory regarding the learning rate. It is true that if the model-based method is "approximately good", we can enjoy the faster learning rate, as we showed in the Bus domain experiment. Nevertheless, there is no theory to tell if a model-based method is approximately good or not. A possible direction of future work is to investigate what kind of properties of model-based methods are necessary to increase the learning rate and how much they can increase when they are combined with model-free methods.

Bibliography

[1] D. Andre. *Programmable Reinforcement Learning Agents*. PhD thesis, University of California at Berkeley, 2003. URL http://www.davidandre.com/diss.html. → pages 3, 13, 14, 16, 19, 23, 25, 26

[2] D. Andre and S. J. Russell. State abstraction for programmable reinforcement learning agents. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 119–125. AAAI Press, 2002. → pages 14, 19, 23, 26, 28

[3] N. Asgharbeygi, D. Stracuzzi, and P. Langley. Relational temporal difference learning. In *Proceedings of the Twenty-Third International Conference on Machine Learning*, pages 49–56, New York, NY, USA, 2006. ACM. → pages 41

[4] A. G. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13:341–379, 2003. → pages 5

[5] R. Bellman. *Dynamic programming*. Princeton University Press, Princeton, NJ, 1957. → pages 6, 10

[6] D. Bertsekas and J. Tsitsiklis. *Neuro-dynamic programming*. Athena Scientific, 1996. → pages 5

[7] J. A. Boyan. Least-squares temporal difference learning. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 49–56. Morgan Kaufmann, 1999. → pages 3

[8] V. M. Cora. Model-based active learning in hierarchical policies. Master's thesis, University of British Columbia, 2008. URL https://circle.ubc.ca/handle/2429/737. → pages 3

[9] P. Dayan and G. E. Hinton. Feudal reinforcement learning. In *Advances in Neural Information Processing Systems 5*, pages 271–278, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc. → pages 13

[10] T. Degris and O. Sigaud. Learning the structure of factored Markov decision processes in reinforcement learning problems. In *Proceedings of the Twenty-Third international conference on Machine learning*, pages 257–264, 2006. → pages 2

[11] J. Demšar, B. Zupan, G. Leban, and T. Curk. Orange: From experimental machine learning to interactive data mining. *Knowledge discovery in databases: PKDD 2004*, pages 537–539, 2004. → pages 39

[12] T. G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13: 227–303, 2000. → pages vi, 13, 17, 18, 19, 21

[13] M. Ghavamzadeh and S. Mahadevan. Hierarchical policy gradient algorithms. In *Proceedings of the Twentieth International Conference on Machine Learning*, pages 226–233. AAAI Press, 2003. → pages 3

[14] R. Gibson and N. A. Risk. Mario, state abstractions and the wonderful world of options. https://sites.google.com/site/richardggibson/publications-and-presentations, 2009. → pages 36, 41

[15] C. Guestrin, R. Patrascu, and D. Schuurmans. Algorithm-directed exploration for model-based reinforcement learning in factored MDPs. In *Proceedings of the Nineteenth International Conference on Machine Learning*, pages 235–242. Morgan Kaufmann Publishers Inc, 2002. → pages 1

[16] B. Hengst. Discovering hierarchy in reinforcement learning with HEXQ. In *Proceedings of the Nineteenth International Conference on Machine Learning*, pages 243–250. Morgan Kaufmann, 2002. → pages 3

[17] T. Hester and P. Stone. An empirical comparison of abstraction in models of Markov decision processes. In *Proceedings of the ICML/UAI/COLT Workshop on Abstraction in Reinforcement Learning*, June 2009. → pages 2, 39

[18] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the Fifteenth Conference on*

Uncertainty in Artificial Intelligence, pages 279–288. Morgan Kaufmann, 1999. → pages 1

[19] R. Howard. *Dynamic programming and Markov processes*. The MIT Press, 1960. → pages 5, 10

[20] N. K. Jong and P. Stone. Hierarchical model-based reinforcement learning: Rmax + MAXQ. In *Proceedings of the Twenty-Fifth International Conference on Machine Learning*, pages 432–439, New York, NY, USA, 2008. ACM. → pages 1

[21] S. Mohan and J. E. Laird. Learning to play Mario. Technical Report CCA-TR-2009-03, Center for Cognitive Architecture, University of Michigan, 2009. → pages 35, 41

[22] S. Mohan and J. E. Laird. Relational reinforcement learning in Infinite Mario. In *Proceedings of the Twenty-Fourth National Conference on Artificial Intelligence*, AAAI'10, pages 1953–1954. AAAI Press, 2010. → pages 41

[23] Y. Naddaf. Game-independent AI agents for playing Atari 2600 console games. Master's thesis, University of Alberta, 2010. URL http://gradworks.umi.com/MR/60/MR60546.html. → pages 37

[24] S. Nason and J. Laird. Soar-RL: Integrating reinforcement learning with Soar. *Cognitive Systems Research*, 6(1):51–59, 2005. → pages 35

[25] R. Parr and S. Russell. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems 10*, pages 1043–1049. MIT Press, 1997. → pages 13, 14, 19

[26] H. Pasula, L. Zettlemoyer, and L. Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29(1): 309–352, 2007. → pages 40

[27] M. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, Inc., 1994. → pages 5, 12

[28] S. Ross, G. Gordon, and D. Bagnell. Reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, 2011. → pages 36

[29] G. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report CUED/FINFENG/TR 166, Cambridge University Engineering Department, 1994. → pages 8

[30] Y. Shoham and K. Leyton-Brown. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, December 2008. → pages 5

[31] S. P. Singh. Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, AAAI'92, pages 202–207. AAAI Press, 1992. → pages 13

[32] P. Stone, R. S. Sutton, and G. Kuhlmann. Reinforcement learning for RoboCup soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005. → pages 3

[33] R. Sutton, D. Precup, and S. Singh. Between MDPs and Semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, August 1999. ISSN 0004-3702. → pages 13, 19, 31

[34] R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224. Morgan Kaufmann, 1990. → pages 2, 11

[35] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, March 1998. → pages 5

[36] R. S. Sutton, C. Szepesvri, A. Geramifard, and M. Bowling. Dyna-style planning with linear function approximation and prioritized sweeping. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, 2008. → pages 2

[37] J. Togelius, S. Karakovskiy, and R. Baumgarten. The 2009 Mario AI competition. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE, 2010. → pages 35, 36, 39, 47

[38] T. J. Walsh, I. Szita, C. Diuk, and M. L. Littman. Exploring compact reinforcement-learning representations with linear regression. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, UAI '09, pages 591–598, Arlington, Virginia, United States, 2009. AUAI Press. → pages 1, 2, 40, 47

[39] C. Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989. URL http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf. → pages 8

[40] M. Wynkoop and T. Dietterich. Learning MDP action models via discrete mixture trees. In *Proceedings of the European conference on Machine Learning and Knowledge Discovery in Databases - Part II*, pages 597–612, Berlin, Heidelberg, 2008. Springer-Verlag. → pages 1

[41] S. Yi, P. Ringstad, and F. Wang. An approach to Infinite Mario. http://2009.rl-competition.org/results/ringstad-mario.pdf, 2009. → pages 36, 41