LEARNING STATE AND ACTION SPACE HIERARCHIES FOR REINFORCEMENT LEARNING USING ACTION-DEPENDENT PARTITIONING

by

MEHRAN ASADI

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2006

Dedicated to

my mother

and

to the memory of my father

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor, Professor Manfred Huber, for his support, patience, and encouragement throughout my graduate studies. It is not often that one finds an advisor and colleague that always finds the time for listening to the little problems and roadblocks. His technical and editorial advices were essential to the completion of this dissertation and has taught me innumerable lessons and insights on the workings of academic research in general.

My thanks also go to the members of my committee, Prof. Diane Cook, Prof. Lawrence Holder, Professor Farhad Kamangar and Prof. Andrzej Korzeniowski for reading previous drafts of this dissertation and providing many valuable comments that improved the presentation and contents of this dissertation.

I must also thank the department of Computer Science and Engineering at the University of Texas at Arlington, the National Science Foundation (Grant ITR-0121297) and DARPA (Grant FA8750-05-2-0283) for the financial support.

Most importantly, I would like to thank my wife, Afrand Agah, for her understanding and love during the past few years. Her support and encouragement were in the end what made this dissertation possible.

Last, but not least, My parents and my brother, Mehrdad, receive my deepest gratitude and love for their dedication and the many years of support during my undergraduate studies that provided the foundation for this work.

July 25, 2006

ABSTRACT

LEARNING STATE AND ACTION SPACE HIERARCHIES FOR REINFORCEMENT LEARNING USING ACTION-DEPENDENT PARTITIONING

Publication No. ______

Mehran Asadi, Ph.D.

The University of Texas at Arlington, 2006

Supervising Professor: Dr. Manfred Huber

Autonomous systems are often difficult to program. Reinforcement learning (RL) is an attractive alternative, as it allows the agent to learn behavior on the basis of sparse, delayed reward signals provided only when the agent reaches desired goals.

Recent attempts to address the dimensionality of RL have turned to principled ways of exploiting temporal abstraction where decisions are not required at each step but rather invoke the execution of temporally-extended activities which follow their own policies until termination. This leads naturally to hierarchical control architectures and associated learning algorithms. This dissertation reviews several approaches to temporal abstraction and hierarchical organization that machine learning researchers have recently developed and presents a new method for the autonomous construction of hierarchical action and state representations in reinforcement learning, aimed at accelerating learning and extending the scope of such systems.

In this approach, the agent uses information acquired while learning one task to discover subgoals for similar tasks. The agent is able to transfer knowledge to subsequent tasks and to accelerate learning by creating useful new subgoals and by off-line learning of corresponding subtask policies as abstract actions (options). At the same time, the subgoal actions are used to construct a more abstract state representation using action-dependent state space partitioning. This representation forms a new level in the state space hierarchy and serves as the initial representation for new learning tasks (the decision layer). In order to ensure that tasks are learnable, value functions are built simultaneously at different levels of the hierarchy and inconsistencies are used to identify actions to be used to refine relevant portions of the abstract state space.

This representation serves as a first layer of the hierarchy. In order to estimate the structure of the state space for learning future tasks, the decision layer is constructed based on an estimate of the expected time to learn a new task and the system's experience with previously learned tasks. Together, these techniques permit the agent to form more abstract action and state representations over time. Experiments in deterministic and stochastic domains show that the presented method can significantly outperform learning on a flat state space representation.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Markov decision processes (MDPs) are useful ways to model stochastic environments as there are well established algorithms to solve these models. Even though these algorithms find an optimal solution for the model, they suffer from high time complexity when the number of decision points is large.

To address increasingly complex problems, it is necessary to find representations that are sufficient to address the task while remaining sufficiently compact to permit learning in an efficient manner. The importance here is put on the state space representation used in the decision-making process rather than on the one used for sensing and memory purposes. The idea is that a reduced representation for decision making combined with the use of increasingly competent actions in the form of policies can dramatically reduce the number of decision points and can lead to a much more efficient transfer of learning experiences across situations and tasks.

A number of learning approaches have used specially designed state space representations to increase the efficiency of learning [18, 20]. Here, particular features are hand-designed based on the task domain and the capabilities of the learning agent. In autonomous systems, however, this is generally a difficult task since it is hard to anticipate which parts of the underlying physical state are important for the given decision-making problem. Moreover, in hierarchical learning approaches the required information might change over time as increasingly competent actions become available. The same can be observed in biological systems where information about all muscle fibers is initially instrumental to generate strategies for coordinated movement. However, as such strategies become established and ready to be used, this low-level information does no longer have to be consciously taken into account when learning policies for new tasks.

To achieve similar capabilities in artificial agents, state and knowledge representations should depend on the action set that is currently available, and become increasingly abstract as more higher-level policies become available as actions and less of the low-level action primitives are required.

A small number of techniques for generating more compact state representations based on the actions and the reward function have been developed [18, 20]. The work presented here builds on the ϵ-reduction technique developed by Dean et al. [18] to derive representations in the form of state space partitions that ensure that the utility of a policy learned in the reduced state space is within a fixed bound of the optimal policy. The work presented here extends the ϵ-reduction technique by including policies as actions and thus using it to find approximate SMDP reductions. Furthermore it derives partitions for individual actions and composes them into representations for any given subset of the action space. This is further extended by permitting the definition of reward independent partitions that can be refined once the reward function is known.

To further enhance its capabilities, the agent, in this approach, uses information acquired while learning one task to improve its performance in similar tasks. The agent is able to transfer knowledge to subsequent tasks and to accelerate learning by discovering useful new subgoals and by off-line learning of corresponding subtask policies as abstract actions (options). At the same time, the subgoal actions are used to construct a more abstract state representation using action-dependent state space partitioning. This representation forms a new level in the state space hierarchy and serves as the initial representation (the decision layer) for new learning tasks. In order to ensure that tasks are learnable, value functions are built simultaneously at different levels of the hierarchy and inconsistencies are used to identify actions to be used to refine relevant portions of the abstract state space. Furthermore, this approach introduces a new method for estimating the structure of the state space for learning future tasks. This approach constructs the decision layer based on an estimate of the expected time to learn a new task based on previously learned tasks and builds a partition such as to optimize the utility of each action and the expected time to learn a new task.

1.1 Decision Making Under Uncertainty

One of the basic concepts in stochastic processes is a control process in an environment in which there is uncertainty. Solving a control process [16] is considered from the perspective of an agent that acts in the environment. An agent can be a robot which navigates a house, a human executing a strategy, or a program which controls traffic lights.

The goal of decision-making is to find a plan or a policy that maximizes the total benefit of acting in an environment over a period of time. Decision-making has broad applications in operations research, artificial intelligence, control theory, management and scheduling [15].

An example in artificial intelligence is a robot that moves through a grid world like the mouse and maze problem. The robot has the ability of performing actions such as moving forward and turning by an angle, and the maze is an environment with different states as shown in Figure 1.1. The purpose of this representation is to provide the information necessary to construct a navigation strategy for a given goal location.

Uncertainty is present at all time in this environment. For example, when the robot's motors do not function as expected, moving the robot in a wrong direction or moving it too far. Furthermore the sensors of the robot can be unreliable and provide incorrect readings from the environment.

The Markov property can be observed in many stochastic systems such as smart homes,

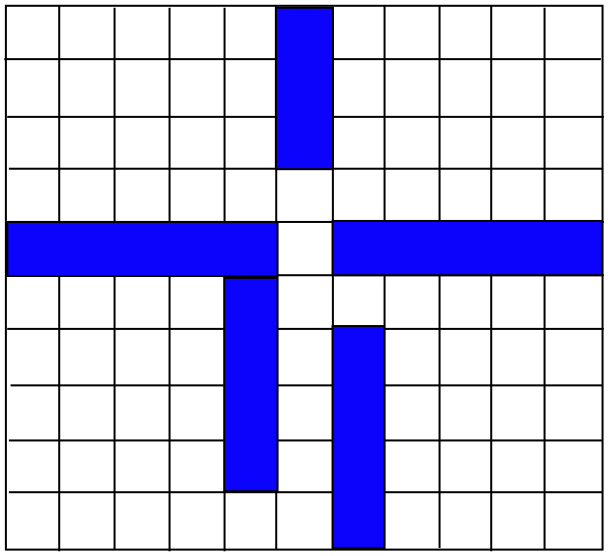


Figure 1.1. A sample robot navigation environment.

robotics, and control systems and they can thus be modeled as a MDP. While well-known algorithms exist to solve MDPs and find an optimal solution (policy), the state representation of these problems are often so large that these algorithms require a large amount of memory and time.

1.2 Models of Planning

The relationship between the time spent on planning and the time spent on executing a plan is a way to distinguish different planning models from each other. Usually, finding an optimal plan is time consuming. For this reason some planning methods construct solutions in off-line mode, which permits them to be performed on powerful computers. When a plan is constructed, it can be relocated to a smaller computer to be executed on-line. For example, the smaller computer can be a robot with less memory and a slower processor.

Off-line planning often assumes that complete knowledge of the environment is available and it considers all outcomes, even those that have a very small chance of occurring.

Thus, if the number of states is large this process has a high time complexity.

Unlike off-line planning, on-line planning does often not assume complete knowledge of the environment and the agent tries to construct and refine a plan while acting in the world. In the extreme case, the agent starts to act with no initial plan and no model of the environment. This is particularly useful when the state space is large, as the agent often only needs the information for its next act and does not require complete knowledge of the environment. Reinforcement Learning [39, 3] is an example of such methods.

1.3 Reinforcement Learning

Reinforcement Learning (RL) [5, 34] is an active area of machine learning research that is also receiving attention from the fields of decision theory, operations research, and control engineering. RL algorithms address the problem of how an agent can learn to approximate an optimal behavioral strategy while interacting directly with its environment. In control terms, this involves the on-line approximation of solutions to stochastic optimal control problems, usually under conditions of incomplete knowledge of the system being controlled. Most RL algorithms adapt standard methods of stochastic dynamic programming (DP) so that they can be used on-line for problems with large state spaces. By focusing computational effort along behavioral trajectories and by using function approximation methods for accumulating value function information, RL algorithms have produced good results on problems that pose significant challenges for standard methods [7, 18, 10, 36]. However, current RL methods by no means completely circumvent the curse of dimensionality, i.e. the exponential growth of the number of parameters to be learned with the size of any compact encoding of system state.

Recent attempts to combat the curse of dimensionality have turned to principled ways of exploiting temporal abstraction, where decisions are not required at each step, but rather invoke the execution of temporally extended activities which follow their own policies until termination [35]. This leads naturally to hierarchical control architectures and hierarchical learning algorithms.

Artificial intelligence researchers have addressed the need for large-scale planning and problem solving by introducing various forms of abstraction into problem solving and planning systems [28, 19, 9]. Abstraction allows a system to ignore details that are irrelevant for the task at hand. One of the simplest types of abstraction is the idea of a "macro operator," or just a "macro," which is a sequence of operators or actions that can be invoked by name as if it was a primitive operator or action. Macros form the basis of hierarchical specifications of operator or action sequences because macros can include other macros in their definitions: a macro can "call" other macros. Also familiar is the idea of a subroutine that can call other subroutines as well as execute primitive commands. Most of the current research on hierarchical Reinforcement Learning (HRL) focuses on action hierarchies that follow roughly the same semantics as hierarchies of macros or subroutines.

One of the fundamental steps toward HRL is to automatically establish subgoals, since subgoals can be treated as termination states for macro actions. Methods for automatically introducing subgoals have been studied in the context of adaptive production systems, where subgoals are created based on examinations of problem-solving protocols. For RL systems, several researchers have proposed methods by which policies learned for a set of related tasks are examined for commonalities or are probabilistically combined to form new policies [35]. Subgoal discovery has been addressed by several researchers [23, 12, 13]. The most closely related research to the subgoal discovery technique introduced here is that of Digney [12] where states that are visited frequently or states where the reward gradient is high are chosen as subgoals.

The next section describes the contribution of this dissertation. We start by introducing a method for abstracting the state and action representations using subgoal discovery and action dependent decomposition and then introduce a method for ensuring the learnability of the tasks that are learnable in the original state and action space. Furthermore, we introduce a measure for estimating the learning time for new tasks and we use this measure to automatically construct a representation for the decision layer for learning of future tasks.

1.4 Contribution of This Research

This research presents a new method for the autonomous construction of hierarchical action and state representations in reinforcement learning, aimed at accelerating learning and extending the scope of such systems. In this approach, the agent uses information acquired while learning one task to discover subgoals for similar tasks. The agent is able to transfer knowledge to subsequent tasks and to accelerate learning by creating useful new subgoals and by off-line learning of corresponding subtask policies as abstract actions (options) which can be used in subsequent tasks. At the same time, the subgoal actions are used to construct a more abstract state representation using action-dependent state space partitioning. This representation forms a new level in the state space hierarchy and serves as the initial representation (the decision layer) for new learning tasks. In order to ensure that tasks are learnable, value functions are built simultaneously at different levels of the hierarchy and inconsistencies are used to identify actions to be used to refine relevant portions of the abstract state space. Together, these techniques permit the agent to form more abstract action and state representations over time. Experiments in deterministic and stochastic domains show that the presented method can significantly outperform learning on a flat state space representation. The main contribution of the presented work can be described as a system that consists of the following four components:

- Sampling-based subgoal discovery.

- Autonomous state space abstraction.
- Hierarchical learning and refinement method.
- Autonomous hierarchy construction.

1.4.1 Temporal and State Abstraction

To make complex problems more tractable, a number of techniques have been developed that can be used to reduce a problem's overall learning complexity. Two of the most powerful tools in this can be found in temporal and state abstraction techniques that attempt to reduce the size of the problem by decreasing the number of decision points or by reducing the representation that has to be considered during learning.

Semi Markov decision processes [35] were originally introduced in order to address the representation of a hierarchical action space in a MDP by considering the execution of sequences of actions, i.e. policies. In this framework actions can be primitive or high-level. A primitive (low-level) action usually takes a constant amount of time while a high-level action corresponds to the execution of a policy and thus takes varying amounts of time. Using these different action types, SMDPs facilitate temporal abstraction by invoking high-level actions such as opening a door, which consists of several primitive actions like unlock, move and release, and require only one decision to be made for their entire execution. Based on this concept, the SMDP framework can optimal solutions in shorter amounts of time than standard MDP techniques.

State abstraction is a group of methods where a single state represents a large group of states. State abstraction often involves a trade-off between optimality and compactness and one of the questions that needs to be answered in abstracting the sate space is the relationship between a solution on the abstract model and a solution on the original model [7, 18, 10, 36]. One of the main problems in many of the existing techniques is that they require prior knowledge of the new task (or at least of the corresponding reward) to be able to construct appropriate state abstractions. This, in turn can significantly limit their applicability in real-world domains.

The work presented here presents a novel approach to hierarchical learning that tightly integrates autonomous temporal and state abstraction to form a system that can profit from both. To do so, it combines the ability of using the multi-step actions with the concepts of ϵ-reduction, leading to a system which can learn bounded optimal policies on significantly more compact state spaces without requiring prior knowledge of the reward function of the new task. Here, action and representational knowledge learned in previous tasks is transferred to new tasks and utilized to significantly improve learning performance.

1.4.2 Subgoal Discovery

An example that shows the importance of a subgoal, is a room to room navigation task where the agent should discover the utility of doorways. If the agent discovers that a doorway is a subgoal it can learn an option to reach the doorway which, in turn, can accelerate learning of new navigation tasks. The idea of using subgoals is not, however, limited to grid worlds. For example, for a robot arm to pick up an object, an important subtask is the recognition of the object and thus being aware of its presence would be a subgoal.

The main goal of automatic subgoal discovery is to find useful subgoals in the agent's state space. Once they are found, options to those subgoals can be learned and added to the behavioral repertoire of the agent. The approach presented here introduce a novel approach to subgoal discovery where subgoals are identified as states with particular structural properties in the context of a given policy. In particular, we define subgoals as states that, under a given policy, lie on a substantially larger number of paths than would be expected by looking at its successor states. To efficiently identify such states, this approach samples trajectories from a learned policy using Monte Carlo sampling and uses a statistical significance criterion to select relevant subgoals.

1.4.3 Action Dependent Partitioning

One of the aspects of hierarchical learning is to construct a tree type structure on the action space, in which actions in higher level sets can be considered as policies in lower level sets. A second aspect of hierarchical learning approaches is that as new and more complex actions become available, low-level actions are no longer required to learn a task and thus can be ignored. The intuition here is that such a policy will involve fewer decision points and as a result, new tasks can be learned substantially faster.

To take full advantage of this limitation of the action space, it should also be reflected in the state representation. In particular, once low-level actions are ignored, much more abstract state representations should be sufficient to address the same tasks. To address this, the approach presented here introduces a new method for state space partitioning which allows the efficient construction of an abstract state space which represents the aspects of the environment that are important for a given set of actions.

1.4.4 Reward Independent Decomposition

In many real world situations the reward information of a new task is not known beforehand. In this case many of the standard state reduction techniques for MDPs can not be applied. To address this, this research introduces a new approach which solves the state decomposition problem in multiple stages. The first stages here assume that the agent has no knowledge of rewards associated with the new task, and construct an initial state representation that captures the important aspects of the available actions independent of the new task. On this representation the system then attempts to learn a policy and it can be proven that under certain conditions (which basically characterize the degree to which the available actions are suitable for the new task) it will succeed at learning the task, albeit without any guarantees as to the optimality of the learned solution. On the other hand, the abstract, reward-independent state space promises to dramatically reduce the learning time as compared to the original state space. Moreover, the approach developed here includes a final phase which performs reward-specific refinement once reward information becomes available.

1.4.5 Learning Method

The presented work introduces a new method for the autonomous construction of hierarchical actions and state representations in Reinforcement Learning. In this approach, the agent uses information acquired while learning one task to discover subgoals for similar tasks by analyzing the learned policy using Monte Carlo sampling. The agent is able to transfer knowledge and to accelerate learning of subsequent tasks by creating new subgoals and by off-line learning corresponding subtask policies as abstract actions (options). At the same time, the subgoal actions are used to construct a more abstract state representation using action-dependent state space partitioning. This representation forms a new level in the state space hierarchy and serves as the initial representation for new learning tasks. However, this initial representation ensures that a new task is learnable only if it is addressable using the same subgoals as the initially learned tasks. To ensure that all new tasks can be learned, this work introduces a new learning method for hierarchical state representations where value functions are built at different levels of hierarchy and inconsistencies are used to identify actions to be used to refine relevant portions of the abstract state space.

1.4.6 Autonomous Hierarchical Construction

In the previous sections we have introduced the action dependent method for decomposition of the state space and a new technique for learning on a hierarchical state representation. So far this approach constructs the abstract space based on a simple heuristic, in particular all subgoal options are used to build an initial representation which might later be refined with some of the original actions. To make this more flexible and efficient, we extend this approach by introducing autonomous hierarchy construction, a new method for estimating the structure of the state space for learning future tasks by constructing the decision layer based on an estimate of the expected time to learn a new task according to the set of previously learned tasks and based on the utility of each action.

1.5 Outline

This dissertation presents powerful new algorithms for solving large MDPs. The next chapter describes the theoretical aspects related to this work, including reinforcement learning and the algorithms associated to it. Chapter 3 presents the related work for the research presented in this dissertation, such as temporal and state abstraction, $\epsilon - reduction$, and subgoal discovery. Chapter 4 presents the main technical contributions of this dissertation and Chapter 5 shows the experimental results of this work. Chapter 6 finally presents a comparison of the technique with MAXQ, one of the most popular hierarchical Reinforcement learning techniques, before Chapter 7 concludes this dissertation.

CHAPTER 2

FORMALISM

Most RL research is based on the formalism of Markov decision processes (MDPs). Although RL is by no means restricted to MDPs, this discrete-time, countable (in fact, usually finite) state and action formalism provides the simplest framework in which to study basic algorithms and their properties. Here we briefly describe this well-known framework, with a few twists characteristic of how it is used in RL research; additional details can be found in many references (e.g., Bertsekas and Tsitsiklis [5], Puterman [27], Ross [29] and Sutton and Barto [34]). A finite MDP models the following type of problem. At each stage in a sequence of stages, an agent (the controller) observes a system's state s, contained in a finite set S, and executes an action (control) a selected from a finite, non-empty set, A_s, of admissible actions. The agent receives an immediate reward having expected value $R(s, a)$, and the state at the next stage is s' with probability $P(s'|s, a)$. The expected immediate rewards, $R(s, a)$, and the state transition probabilities, $P(s'|s, a)$, $s, s' \in S$, together comprise what RL researchers often call the one-step model of action a.

Figure 2.1 illustrates a graphical representation of an MDP with state space $\{s_1, \ldots, s_8\}$ and action space $\{a, b\}$. The reward at each state is 0 except for state s_7 which has a reward of 10. The arrows represent the transition function.

A (stationary, stochastic) policy $\pi : S \times A \rightarrow [0, 1]$, specifies that the agent executes action a with probability $\pi(s, a)$ whenever it observes state s.

For any policy π and state $s \in S$, the function $V^\pi(s)$ denotes the expected infinite-horizon discounted return from state s given that the agent uses policy π. Letting s_t and

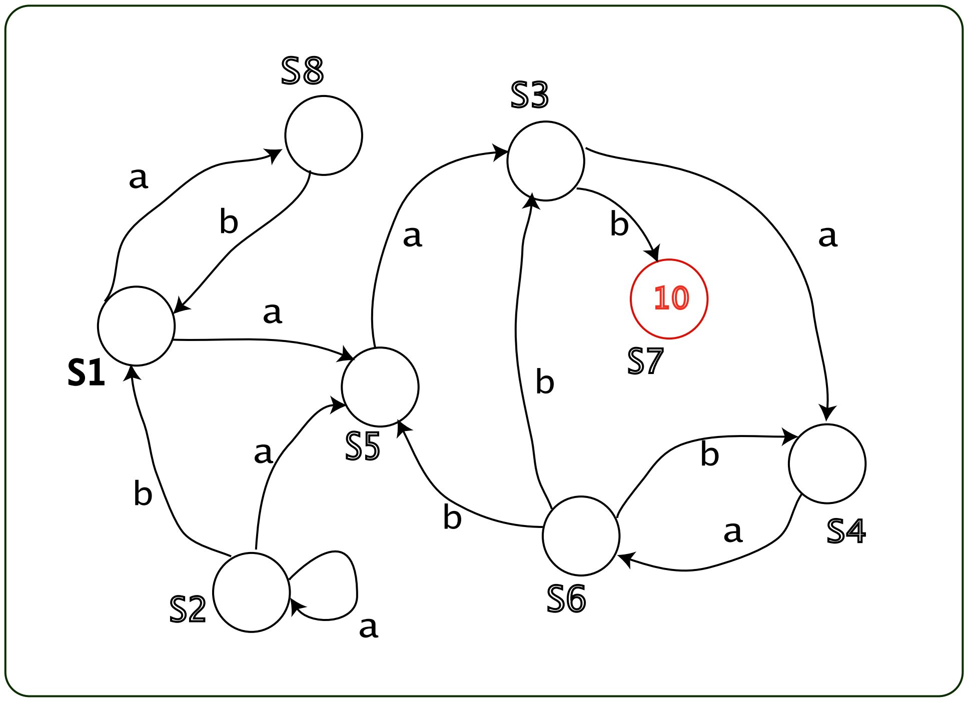


Figure 2.1. A graph representation of an MDP.

r_{t+1} denote the state at stage t and the immediate reward for acting at stage $t+1$ this is defined as:

$$V^{\pi}(s) = E\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | s_t = s, \pi\}$$

where γ, $0 \leq \gamma \leq 1$, is a discount factor. $V^{\pi}(s)$ is the value of state s under policy π, and V^{π} is the value function corresponding to π.

This is a discounted MDP. The objective is to find an optimal policy, i.e. a policy, π^*, that maximizes the value of each state. The unique optimal value function, V^*, is the value function corresponding to any of the optimal policies.

Most RL research addresses discounted MDPs since they comprise the simplest class of MDPs, and here we restrict our attention to discounted problems. However, RL algorithms have also been developed for MDPs with other definitions of return, such as average reward MDPs [4].

Playing important roles in many RL algorithms are action-value functions which assign values to admissible state-action pairs. Given a policy π, the value of state-action pair, (s, a), $a \in A_s$, denoted by $Q^\pi(s, a)$, is the expected infinite-horizon discounted return for executing action a in state s and thereafter following π:

$$Q^\pi(s, a) = E\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | s_t = s, a_t = a\pi\}$$

The optimal action-value function, Q^*, assigns to each admissible state-action pair (s, a) the expected infinite-horizon discounted return for executing action a in state s and thereafter following an optimal policy. Action-value functions for other definitions of return are defined analogously.

Dynamic programming (DP) algorithms exploit the fact that value functions satisfy various Bellman equations, such as [4]:

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

and

$$V^*(s) = \sum_{a \in A} \pi^*(s, a) \left[R(s, a) + \gamma \sum_{s'} P(s'|s, a) V^*(s') \right] \tag{2.1}$$

for all $s \in S$. Analogous equations exists for Q^π and Q^*. For example, the Bellman equation for Q^* is:

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s'} P(s'|s, a) max_{a' \in A} Q^*(s', a') \tag{2.2}$$

for all $s \in S$ and $a \in A$.

As an example DP algorithm, consider value iteration which successively approximates V^* as follows. At each iteration k, it updates an approximation V_k of V^* by applying the following operation for each state s:

$$V_{k+1}(s) = \max_{a \in A_s} \left[R(s, a) + \gamma \sum_{s'} P(s'|s, a) V_k(s') \right] \tag{2.3}$$

This operation is called a backup because it updates a state's value by transferring to it the information about the approximate values of its possible successor states. Applying this backup operation once to each state is often called a sweep. Starting with an arbitrary initial function V_0, the sequence $V_{\{k\}}$ produced by repeated sweeps converges to V^*. A similar algorithm exists for successively approximating Q* using the following backup [4]:

$$Q_{k+1}(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a' \in A_{s'}} Q_k(s',a')$$

Given V^*, an optimal policy is any policy that for each state s assigns non-zero probability only to those actions that realize the maximum on the right-hand side of Equation 2.3. Similarly, given Q^*, an optimal policy is any policy that for each state s assigns non-zero probability only to the actions that maximize $Q^*(s,a)$. These maximizing actions are often called greedy actions, and a policy defined in this way is a (stochastic) greedy policy. Given sufficiently close approximations of V^* and Q^* obtained by value iteration, optimal policies are taken to be the corresponding greedy policies. Note that finding greedy actions via Q^* does not require access to the one-step action models (the $R(s,a)$ and $P(s'|s,a)$) as it does when only V^* is available, where the right-hand side of Equation 2.3 has to be evaluated. This is one of the reasons that action-value functions play significant roles in RL.

In an MDP only the sequential nature of the decision process is relevant, not the amount of time that passes between decision stages. A generalization of this is the semi-Markov decision process (SMDP) in which the amount of time between one decision and the next is a random variable, either real- or integer-valued. In the real-valued case, SMDPs model continuous-time discrete-event systems (e.g. [7, 20, 32, 36]).

In a discrete-time SMDP decisions can be made only at (positive) integer multiples of an underlying time step. In either case, it is usual to treat the system as remaining in each state for a random waiting time at the end of which an instantaneous transition occurs to the next state. Due to its relative simplicity, the discrete-time SMDP formulation underlies most approaches to hierarchical RL, but there are no significant obstacles to extending these approaches to the continuous-time case.

Let the random variable t denote the (positive) waiting time for state s when action a is executed. The transition probabilities generalize to give the joint probability that a transition from state s to state s' occurs after t time steps when action a is executed. This joint probability can be written as $P(s', t|s, a)$. The expected immediate rewards, $R(s, a)$, (which must be bounded) now give the amount of discounted reward expected to accumulate over the waiting time in state s given action a. The Bellman equations for V^* and Q^* are

$$V^*(s) = \max_{a \in A_s} \left[R(s, a) + \sum_{t=1}^{\infty} \gamma^t \sum_{s'} P(s', t|s, a) V^*(s') \right]$$

and

$$Q^*(s, a) = R(s, a) + \sum_{t=1}^{\infty} \gamma^t \sum_{s'} P(s', t|s, a) max_{a' \in A} Q^*(s', a')$$

for all $s \in S$ [4].

2.1 Reinforcement Learning

DP algorithms have complexity polynomial in the number of states, but they still require prohibitive amounts of computation for large state sets, such as those that result from discretizing multi-dimensional continuous spaces or from representing state sets consisting of all possible configurations of a finite set of structural elements (e.g. possible configurations of a backgammon board [36]). Many methods have been proposed for approximating MDP solutions, in particular RL methods that use Monte Carlo, stochastic approximation, and function approximation techniques. Specifically, RL algorithms combine some, or all, of the following features [4]:

- Avoid the exhaustive sweeps of DP by restricting computation to states on, or in the neighborhood of, multiple sample trajectories, either real or simulated. Because computation is guided along sample trajectories, this approach can exploit situations in which many states have low probabilities of occurring in actual experience.
- Simplify the basic DP backup by sampling. Instead of generating and evaluating all of a state's possible immediate successors, estimate a backup's effect by sampling from the appropriate distribution.
- Represent value functions and/or policies more compactly than lookup-table representations by using function approximation methods, such as linear combinations of basis functions, neural networks, or other methods.

The first two features reflect the nature of the approximations usually sought when RL is used. Instead of policies that are close to optimal uniformly over the entire state space, RL methods arrive at non-uniform approximations that reflect the behavior of the agent. The agent's policy does not need high precision in states that are rarely visited. The last feature is the least understood aspect of RL, but results exist for the linear case (notably in Tsitsiklis and Van Roy [38]) and numerous examples illustrate how function approximation schemes that are nonlinear in the adjustable parameters (e.g. multilayer neural networks) can be effective for difficult problems (e.g. [7, 20, 32, 36]).

Of the many RL algorithms, perhaps the most widely used are Q-learning [39, 40] and Sarsa [30] .

Q-learning is based on the DP backup but with the expected immediate reward and the expected maximum action-value of the successor state respectively replaced by a sample reward and the maximum action-value for a sample next state. The most common way to obtain these samples is to generate sample trajectories by simulation or by observing the actual decision process over time. Suppose the agent observes a current state s, executes action a, receives immediate reward r, and then observes a next state s'. The Q-learning algorithm updates the current estimate, $Q_k(a, s)$, of $Q^*(s, a)$ using the following update:

$$Q_{k+1}(s, a) = (1 - \alpha_k)Q_k(s, a) + \alpha_k \left[r + \gamma \max_{a' \in A_{s'}} Q_k(s', a') \right] \tag{2.4}$$

where α_k is a time-varying learning-rate parameter. The values of all the other state-action pairs remain unchanged at this update. If in the limit the action-values of all admissible state-action pairs are updated infinitely often, and α_k decays with increasing k while obeying the usual stochastic approximation conditions, then $Q_{\{k\}}$ converges to Q^* with probability 1 [5]. As long as these conditions are satisfied, the policy followed by the agent during learning is irrelevant. Of course, when Q-learning is being used, the agent's policy does matter since one is usually interested in the agent's performance throughout the learning process, not just asymptotically. It is usual practice to let the agent select actions using a policy that is greedy with respect to the current estimate of Q^*, while also introducing non-greedy "exploratory actions" in an attempt to widely sample state-action pairs.

Another approach is Sarsa which is similar to Q-learning except that the maximum action-value for the next state on the right-hand side of 2.4 is replaced by the action-value of the actual next state-action pair:

$$Q_{k+1}(s, a) = (1 - \alpha_k)Q_k(s, a) + \alpha_k \left[r + \gamma Q_k(s', a') \right] \tag{2.5}$$

where a' is the action executed in state s'. (Sutton, [34], called this algorithm Sarsa due to its dependence on s, a, r, s', and a'. Equation 2.5 is actually a special case called Sarsa(0).)

2.2 Reinforcement Learning Algorithms

Two common methods for finding an optimal policy are value iteration and policy iteration. This section provides an algorithm for each of these methods.

2.2.1 Value Iteration

One way to find an optimal policy is to find the optimal value function. It can be determined by a simple iterative algorithm called value iteration that can be shown to converge to the correct V^* values [5]. One important result bounds the performance of the current greedy policy as a function of the Bellman residual of the current value function. it says that if the maximum difference between two successive value functions is less than ϵ , then the value of the greedy policy, (the policy obtained by choosing, in every state, the action that maximizes the estimated discounted reward using the current estimate of the value function) differs from the value function of the optimal policy by no more than $\frac{2\epsilon\gamma}{1-\gamma}$ in any state [27]. This provides an effective stopping criterion for the algorithm. Puterman [27] discusses another stopping criterion, based on the span semi-norm, which may result in earlier termination. Another important result is that the greedy policy is guaranteed to be optimal in some finite number of steps even though the value function may not have converged. In practice, the greedy policy is often optimal long before the value function has converged. The value iteration method starts with an arbitrary value function, such as $V_0(s) = R(s,a)$ for some $a \in A$, and uses this value function to find the next value function using the following equation:

$$V_{k+1}(s) = \max_{a \in A_s} \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V_k(s') \right]$$

the optimal value function V^* is formed when the value of $V_{k+1}(s) - V_k(s)$ is small enough for all $s \in S$. The corresponding optimal policy is:

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

Algorithm 1 describes this procedure.

Algorithm 1 Value Iteration

```
Require: Initialize $V$ arbitrary, e.g, $V(s) = 0, \forall s \in S$
  repeat
    $\Delta \leftarrow 0$
    for each $s \in S$ do
      $v \leftarrow V(s)$
      $V(s) \leftarrow \max_{a \in A_s} [R(s,a) + \gamma \sum_{s'} P(s'|s,a)V(s')]$
      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    end for
  until $\Delta < \theta$ (a small positive number)
  Output Policy $\pi(s) = argmax_a [R(s,a) + \gamma \sum_{s'} P(s'|s,a)V^*(s')]$
```

2.2.2 Policy Iteration

While value iteration updates the value and the policy at each step, policy iteration finds a policy and tries to improve it until the policy can not be improved. The policy iteration algorithm manipulates the policy directly, rather than finding it indirectly via the optimal value function. The value function of a policy is just the expected infinite discounted reward that will be gained, at each state, by executing that policy. It can be determined by solving a set of linear equations. Once we know the value of each state under the current policy, we consider whether the value could be improved by changing the first action taken. If it can, we change the policy to take the new action whenever it is in that situation. This step is guaranteed to strictly improve the performance of the policy. When no improvements are possible, then the policy is guaranteed to be optimal. Since there are at most $|A|^{|S|}$ distinct policies, and the sequence of policies improves at each step, this algorithm terminates in at most an exponential number of iterations [27]. Algorithm 2 describes a pseudo-code by which policy iteration can be implemented.

Algorithm 2 Policy Iteration

```
Require: Initialize $\pi_0$ an arbitrary policy
  $j \leftarrow 0$
  $Continue \leftarrow true$
  while Continue do
    Compute $V^\pi$
    $\pi_{j+1} \leftarrow argmax_a(V_{\pi_j})$
    if $\pi_j = \pi_{j+1}$ then
      $Continue \leftarrow false$
    else
      $j \leftarrow j + 1$
    end if
  end while
  return $\pi_j$
```

CHAPTER 3

RELATED WORK

Hierarchical approaches to RL generalize the macro idea to closed-loop policies or, more precisely, closed-loop partial policies because they are generally defined for a subset of the state set. The partial policies must also have well-defined termination conditions. These partial policies are sometimes called temporally-extended actions or options [35]. When option policies are learned, they usually are policies for efficiently achieving subgoals, where a subgoal is often a state, or a region of the state space, such that reaching that state or region is assumed to facilitate achieving the overall goal of the task. The canonical example of a useful subgoal is a doorway in a robot navigation scenario: the doorway has to be passed through to reach any goal outside the room. Figure 3.1 shows a SMDP that is derived from a MDP [18].
In this figure the top panel shows the state trajectory over discrete time in the MDP and the lower panel shows the larger state changes in a SMDP. The filled circles indicate decision points when a new action has to be selected while the empty circles in the SMDP represent states in which the previously selected multi-step action is still active. As can be seen, a smaller number of decisions have to be made in the SMDP which should accelerate learning.

3.1 Temporal Abstraction of SMDPs

Artificial intelligence researchers have addressed the need for large-scale planning and problem solving by introducing various forms of abstraction into problem solving and planning systems, e.g., Fikes et al. [28] and Korf [19]. Abstraction allows a system

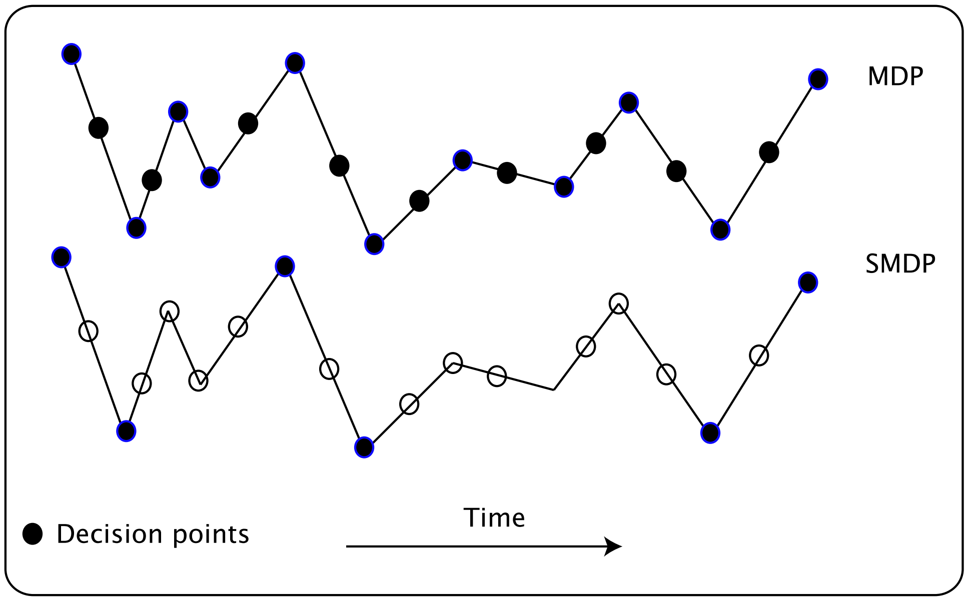


Figure 3.1. Comparison between MDP and SMDP.

to ignore details that are irrelevant for the task at hand. One of the simplest types of abstraction is the idea of a "macro-operator," or just a "macro," which is a sequence of operators or actions that can be invoked by name as if it were a primitive operator or action. Macros form the basis of hierarchical specifications of operator or action sequences because macros can include other macros in their definition: a macro can "call" other macros. Also familiar is the idea of a subroutine that can call other subroutines as well as execute primitive commands. Most of the current research on hierarchical RL focuses on action hierarchies that follow roughly the same semantics as hierarchies of macros or subroutines.

From a control perspective, a macro is an open-loop control policy and, as such, is inappropriate for most interesting control purposes, especially the control of stochastic systems. Hierarchical approaches to RL generalize the macro idea to closed-loop policies, or more precisely, closed-loop partial policies because they are generally defined for a subset of the state set. The partial policies must also have well-defined termination conditions. These partial policies are sometimes called temporally-extended actions, options [35], skills [37] or behaviors [6, 16].

For MDPs, this extension adds to the sets of admissible actions, $a \in A$, sets of activities, each of which can itself invoke other activities, thus allowing a hierarchical specification of an overall policy. The original one-step actions, called the “primitive actions,” may or may not remain admissible. Extensions along these general lines result in decision processes modeled as Semi Markov Decision Processes (SMDP), where the waiting time in a state corresponds to the duration of the selected activity. If t is the waiting time in state s upon execution of activity a, then action a takes t steps to complete when initiated in state s, where the distribution of the random variable t now depends on the policies and termination conditions of all of the lower-level activities that comprise action a.

Sutton et al. [35] formalize this approach to include activities in RL with their notion of an option. Starting from a finite MDP, which we call the core MDP, the simplest kind of option consists of a (stationary, stochastic) policy $\pi : S \times A \rightarrow [0, 1]$, a termination condition $\beta : S \rightarrow [0, 1]$, and an input set $I \subset S$. The option (I, π, β) is available in state s if and only if $s \in I$. If the option is executed, then actions are selected according to policy π until the option terminates stochastically according to β. For example, if the current state is s, the next action is a with probability $\pi(s, a)$, the environment makes a transition to state s', where the option either terminates with probability $\beta(s')$ or else continues, determining the next action a' with probability $\pi(s', a')$, and so on. When the option terminates, the agent can select another option.

It is usual to assume that for any state in which an option can continue, it can also be initiated, that is, $\{s : \beta(s) < 1\} \subset I$. This implies that an option's policy only needs to be defined over its input set I. Note that any action of the core MDP, a primitive action $a \in A$, is also an option, and it is called a one-step option, with $I = \{s : a \in A_s\}$

and $\beta(s) = 1$ for all $s \in S$. Sutton et al. [35] give the example of an option named open-the-door for a hypothetical robot control system. This option consists of a policy for reaching, grasping and turning the door knob, a termination condition for recognizing that the door has been opened, and an input set restricting execution of open-the-door to states in which a door is within reach.

To allow more flexibility, especially with respect to hierarchical architectures, one must include semi-Markov options whose policies can set action probabilities based on the entire history of states, actions, and rewards since the option was initiated [35].

Semi-Markov options include options that terminate after a pre-specified number of time steps and, most importantly, they are needed when policies over options are considered, i.e. policies $\pi : S \times O_s \rightarrow [0,1]$, where O_s is the set of admissible options for state s (which can include all the one-step options corresponding to the admissible primitive actions in A_s, i.e. the set of admissible actions for state s).

A policy π over options selects option o in state s with probability $\pi(s,o)$; o's policy in turn selects other options until o terminates. The policy of each of these selected options selects other options, and so on. Expanding each option down to primitive actions, we see that any policy π over options determines a conventional policy of the core MDP which Sutton et al. [35] call the flat (i.e. non-hierarchical) policy corresponding to π, denoted as $flat(\pi)$. Flat policies corresponding to policies over options are generally not Markov even if all the options are Markov. The probability of a primitive action at any time step depends on the current core state plus the policies of all the options currently involved in the hierarchical specification. This dependence is made more explicit in the work of Parr [24] and Dietterich [10]. Using this machinery, one can define hierarchical options as triples (I, π, β), where I and β are the same as for Markov options but π is a semi-Markov policy over options.

Value functions for option policies can be defined in terms of value functions of semi-Markov flat policies. For a semi-Markov flat policy π:

$$V^{\pi}(s) = E\{r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{\tau-1} r_{t+\tau} + \ldots | \varepsilon(\pi, s, t)\}$$

where $\varepsilon(\pi, s, t)$ is the event of π being initiated at time t in state s. Note that this value can depend on the complete history from t onwards, but not on events earlier than t since π is semi-Markov. Given this definition for flat policies, $V^{flat(\pi)}(s)$, the value of s for a policy π over options is defined to be $V^{\pi}(s)$. Similarly, one can define the option-value function for π as follows:

$$Q^{\pi}(s, o) = E\{r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{\tau-1} r_{t+\tau} + \ldots | \varepsilon(o_{\pi}, s, t)\}$$

where o_{π} is the semi-Markov policy that follows o until it terminates after τ time steps and then continues according to o_{π}.

Adding any set of semi-Markov options to a core finite MDP yields a well-defined discrete-time SMDP whose actions are the options and whose rewards are the returns delivered over the course of an option's execution. Since the policy of each option is semi-Markov, the distributions defining the next state (the state at an option's termination), waiting time, and rewards depend only on the option executed and the state in which its execution was initiated. Thus, all of the theory and algorithms applicable to SMDPs can be appropriated for decision making with options.

In their effort to treat options as much as possible as if they were conventional single-step actions, Sutton et al. [35] introduced the interesting concept of a multi-time model of an option that generalizes the single-step model consisting of $R(s, a)$ and $P(s'|s, a)$, $s, s' \in S$, of a conventional action a. For any option o, let $\varepsilon(o, s, t)$ denote the event of o being initiated in state s at time t. Then the reward part of the multi-time model of o for any $s \in S$ is:

$$R(s, o) = E\{r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{\tau-1} r_{t+\tau} | \varepsilon(o, s, t)\} \tag{3.1}$$

where $t+\tau$ is the random time at which o terminates. The state-prediction part of the model of o for s is:

$$F(s'|s,o) = \sum_{k=1}^{\infty} P(s_{t+k} = s', k|s_t = s, o)\gamma^k \tag{3.2}$$

for all $s \in S$. Though not itself a probability, $F(s'|s,o)$ is a combination of the probability that s' is the state in which o terminates together with a measure of how delayed that outcome is in terms of γ. The quantities $R(s,o)$ and $F(s'|s,o)$ respectively generalize the reward and transition probabilities, $R(s,a)$ and $P(s'|s,a)$, of the usual MDP in such a way that one can write a generalized form of the Bellman optimality equation. If V_O^* denotes the optimal value function over an option set o, then

$$V_O^*(s) = \max_{o \in O}[R(s,o) + \sum_{s'} F(s'|s,o)V_O^*(s')]$$

which reduces to the usual Bellman optimality equation (Equation 2.1), if all the options are one-step options, i.e. $\beta(s) = 1, s \in S$. A Bellman equation can be written for Q_o^*:

$$Q_O^*(s,o) = R(s,o) + \sum_{s'} F(s'|s,o) \max_{o' \in O} Q_O^*(s',o')$$

for all $s \in S$ and $o \in O$.

The DP backup analogous to Equation 2.2 for computing option-values is:

$$Q_{k+1}(s,o) = R(s,o) + \sum_{s' \in S} F(s'|s,o) \max_{o' \in O} Q_k(s',o')$$

and the corresponding Q-learning update is:

$$Q_{k+1}(s,o) = (1-\alpha_k)Q_k(s,o) + \alpha_k[r + \gamma^\tau \max_{o' \in O} Q_k(s',o')]$$

This update is applied upon the termination of o at state s' after executing for τ time steps, and r is the return accumulated during o's execution.

The primary motivation for the options framework is to permit one to add temporally extended activities to the repertoire of choices available to an RL agent, while at the same time not precluding planning and learning at the finer grain of the core MDP. The emphasis is therefore on augmentation rather than simplification of the core MDP. If all the primitive actions remain in the option set as one-step options, then clearly the space of realizable policies is unrestricted so that the optimal policies over options are the same as the optimal policies for the core MDP. But since finding optimal policies in this case takes more computation via conventional DP than does just solving the core MDP, one is tempted to ask what one gains from this augmentation of the core MDP. One answer is to be found in the use of RL methods. For RL, the availability of temporally-extended activities can dramatically improve the agent's performance while it is learning, especially in the initial stages of learning. Options also can facilitate transfer of learning to related tasks. Of course, only some options can facilitate learning in this way, and a key question is how does a system designer decide on what options to provide.

On the other hand, if the set of options does not include the one-step options corresponding to all of the primitive actions, then the space of policies over options is a proper subset of the set of all policies of the core MDP. In this case, the resulting SMDP can be much easier to solve than the core MDP [4].

3.1.1 Example of a SMDP

Consider the rooms problem, a grid world environment with five rooms and one hallway illustrated in Figure 3.2. The cells of the grid show the states of the environment. From any state the robot can perform one of the four actions up, down, left or right, which have a stochastic effect and will fail 50 percent of the time i.e. with probability $\frac{1}{2}$ they are successful, and the agent moves in any of the other three directions with probability $\frac{1}{6}$. The reward for each state is zero.

There is a hallway in this environment, designed to let the agent reach other rooms and

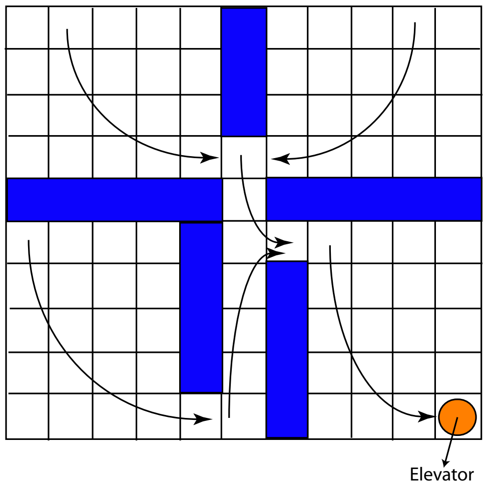


Figure 3.2. The Rooms example as a grid world environment.

the elevator. For each room there are policies π_i which move the robot along the shortest path to the hallway or the other rooms.

For example, the policy for one room is shown in Figure 3.3. The termination condition

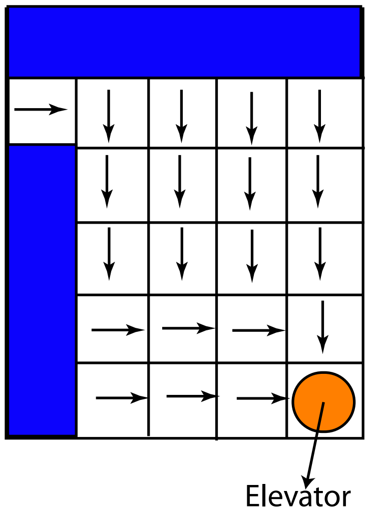


Figure 3.3. The policy of one of the four rooms.

for this policy is zero for states within the room and 1 for states out of the room. The initiation set I consists of the states in the room.

This describes a SMDP for the rooms environment with options that move the agent to a doorway with a single decision. As a result of this reduced number of decision points to navigate the environment, the SMDP converges significantly faster to a solution for a navigation task than the flat MDP using only primitive actions.

3.2 State Abstraction

State abstraction or state aggregation refers to a general class of methods where a single state is used to represent a large group of states. This produces a reduced state space which is easier to solve using the standard MDP algorithms discussed in this chapter. Note that state abstraction is common in traditional planning where only the relevant features of any state, typically some set of state variables, are used to represent a large class of states where the other variables have "don't care" values [24].
Difficult questions for state abstraction in MDPs arise because of the complex value relationships that can exist between seemingly unrelated states. Where traditional planning assumes a deterministic model with a goal of achievement, the stochasticity of MDPs and optimality criteria for MDPs can easily induce an optimal value function that assigns different values to every state and an optimal policy that implies some form of utility relationship between any two states. In some cases it is possible to show that MDP states can be aggregated without any effect on solution quality [24, 9], but state abstraction generally involves a tradeoff between optimality and compactness. Difficult issues that must be resolved are:

- The manner in which a transition model and reward function for the abstract model are derived from the original model.
- The relationship between the solution to the abstract model and the solution to the original model.

Different decisions about these two questions can lead to different tradeoffs between efficiency and solution quality. Conservative approaches [24, 9] aggregate states if they have similar reward and transition functions. This approach permits reasonable bounds on the relationship between the optimal solution to the aggregated model and the optimal solution to the original model. Unfortunately, such methods also fail to capture some of the intuitive notion of an aggregated state. For example, it would seem that all states within a particular room of a house should, at some level of abstraction, be grouped together as a single ”room” state. The fact that actions might have different effects in different parts of the room would prevent this. While state abstraction has many pitfalls, it ultimately must play a role in MDP methods. Since no two situations are ever truly the same, some implicit temporal abstraction is performed whenever a model is constructed and irrelevant features are discarded [24].

3.2.1 Bounded Parameter MDPs

The most strongly related work to the abstraction method described in this work is the method introduced by Dean et al. [9] as a mechanism to derive state space partitions of a MDP that ensure approximately optimal policies to be learned. These partitions depend on the action space and the particular reward function of the task. Kim and Dean, [18] introduced an algorithm to derive a set of such partitions and used it to learn a policy for the task indicated by the reward function. The resulting policy is ensured to be within an ϵ-dependent quality bound. Two shortcomings of this algorithm are that it does not address temporally abstract actions and that it requires a complete re-computation of the partitions when a new action is introduced. Moreover, it requires knowledge of the reward function prior to partitioning, and thus no part of the partitioning transfers across tasks.

The reduction technique is based on the framework of Bounded Parameter MDP (BP-

MDP) [18]. A BPMDP is a four tuple $\hat{M} = (\hat{S}, \hat{A}, \hat{P}, \hat{R})$ where $\hat{S}$ and $\hat{A}$ are defined as for MDPs, and $\hat{P}, \hat{R}$ are analogous to P and R in MDPs but assign closed intervals rather than single values to each state-action pair. That is, for any action a and states $s, s' \in S$, the values of $\hat{R}(s,a)$ and $\hat{P}(s'|s,a)$ are both closed intervals $[l,u]$ where l,u are both real numbers with $l \leq u$ and in the case of $\hat{P}$ we require $0 \leq l \leq u \leq 1$. To ensure that $\hat{P}$ is well-defined we require that for any action a and state s, the sum of the lower bounds of $\hat{P}(s'|s,a)$ over all states s' must be less than or equal to 1 while the upper bounds must sum to a value greater than or equal to 1. Figure 3.4 illustrates the state-transition diagram for a simple BPMDP with three states and one action.

An interval value function $\hat{V}$ is a map from states to closed intervals. A BPMDP

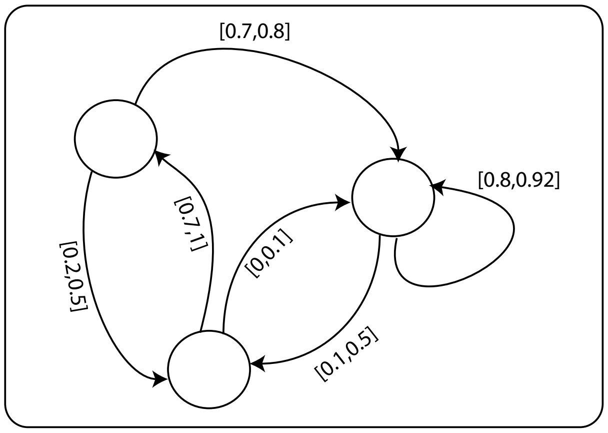


Figure 3.4. The state transition diagram for BPMDP.

$\hat{M} = (\hat{S}, \hat{A}, \hat{P}, \hat{R})$ induces an exact MDP $M = (S, A, P, R)$ where $S = \hat{S}$ and $A = \hat{A}$, and for any action a and states $s, s' \in S$, $R(s,a)$ and $P(s'|s,a)$ are in the range of $\hat{R}(s,a)$ and $\hat{P}(s'|s,a)$ respectively. In a BPMDP $\hat{M}$, the interval value $\hat{V}_\pi$ for state s is defined by the interval:

$$\hat{V}_\pi(s) = \left[\min_{\hat{P},\hat{R}} V_\pi(s), \max_{\hat{P},\hat{R}} V_\pi(s)\right]$$

3.2.2 ϵ-Reduction Method

Dean et al. [8] introduced a family of algorithms that take a MDP and a real value $0 \leq \epsilon \leq 1$ as an input and compute a Bounded Parameter MDP where each closed interval has a scope less than ϵ. The states in this MDP correspond to blocks of a partition of the state space in which the states in the same block have the same proprieties in terms of transitions and rewards. Let $P = \{B_1, \ldots, B_n\}$ be a partition of the state space [8] .

Definition 3.1 [8]: A partition $P = \{B1, \ldots, B_n\}$ of the state space of a MDP M has the property of ϵ-approximate stochastic bisimulation homogeneity with respect to M for $0 \leq \epsilon \leq 1$ if and only if for each $B_i, B_j \in P$, for each $a \in A$ and for each $s, s' \in B_i$:

$$|R(s,a) - R(s',a)| \leq \epsilon$$

and

$$\left| \sum_{s'' \in B_j} P(s''|s,a) - \sum_{s'' \in B_j} P(s''|s',a) \right| \leq \epsilon$$

Definition 3.2 [8]: A partition P' is a refinement of a partition P if and only if each block of P' is a subset of some block of P. In this case we say that P is coarser than P'.

Definition 3.3 [8]: The immediate reward partition is the partition in which two states $s, s' \in S$, are in the same block if they have the same rewards.

Definition 3.4 [8]: The block B_i of a partition P is ϵ-stable with respect to block B_j if and only if for all actions $a \in A$ and all states $s, s' \in B_i$:

$$\left| \sum_{s'' \in B_j} P(s''|s,a) - \sum_{s'' \in B_j} P(s''|s',a) \right| \leq \epsilon$$

The ϵ-model reduction algorithm first uses the immediate reward partition as an initial partition and checks the ϵ-stability for each block of this partition until there are no unstable blocks left. For example, when block B_i happens to be unstable with respect to block B_j, block B_i will be replaced by a set of sub-blocks $B_{i_1}, \ldots, B_{i_k}$ such that each B_{i_m} is a maximal sub-block of B_i that is ϵ-stable with respect to B_j.

Theorem 3.1 [8]: For $\epsilon > 0$, the partition P founded by the ϵ-reduction model algorithm from the MDP M,is coarser than, and thus no larger than M.

Once the ϵ-stable blocks of the partition have been constructed, the transition and reward function between blocks can be defined. The transition of each block by definition is the interval with the bounds of maximum and minimum probabilities of all possible transitions from all states of a block to the states of another block.

$$\hat{P}(B_i|B_j, a) = \left[\min_{s\in B_j} \sum_{s'\in B_i} P(s'|s,a), \max_{s\in B_j} \sum_{s'\in B_i} P(s'|s,a)\right]$$

and

$$\hat{R}(B_j, a) = \left[\min_{s\in B_j} R(s,a), \max_{s\in B_j} R(s,a)\right]$$

3.3 Subgoal Discovery

In the current state-of-the-art, the designer of an RL system typically uses prior knowledge about the task to add a specific set of options to the set of primitive actions available to the agent. In some case, complete option policies can be provided; in other case, option policies can be learned using, for example, intra-option learning methods together with option-specific reward functions that are provided by the designer [23, 12, 13].

Providing options and their policies a priori is an opportunity to use background knowledge about the task to try to accelerate learning and/or provide guarantees about system performance during learning. Perkins and Barto [26], for example, consider collections of options each of which descends a Lyapunov function. Not only is learning accelerated, but the goal state is reached on every learning trial while the agent learns to reach the goal more quickly by approximating a minimum-time policy over these options.

When option policies are learned, they usually are policies for efficiently achieving subgoals, where a subgoal is often a state or a region of the state space, such that reaching that state or region is assumed to facilitate achieving the overall goal of the task. The canonical example of a useful subgoal is a doorway in a robot navigation scenario: the doorway has to be passed through to reach any goal outside the room. Given a collection of subgoals, one can define subgoal-specific reward functions that positively reward the agent for achieving the subgoal (while possibly penalizing it until the subgoal is achieved). Options are then defined which terminate upon achieving a subgoal, and their policies can be learned using the subgoal-specific reward function and standard RL methods. Precup [35] discusses one way to do this by introducing subgoal values, and Dietterich [10], proposes a similar scheme using pseudo-reward functions.

A natural question then is how are useful subgoals determined? McGovern and Barto [23] developed a method for automatically identifying potentially useful subgoals by detecting regions that the agent visits frequently on successful trajectories but not on unsuccessful trajectories. An agent using this method selects such regions that appear early in learning and persist throughout learning, creates options for achieving them, learns their policies, and at the same time learns a higher-level policy that invokes these options appropriately to solve the overall task. Experiments with this method suggest that it can be useful for accelerating learning on single tasks, and that it can facilitate knowledge transfer as previously-discovered options are reused in related tasks. This approach builds on previous work in artificial intelligence that addresses abstraction, particularly that of Iba [17], who proposed a method for discovering macro-operators in problem solving. Related ideas have been studied by Digney [12] where states that are visited frequently or states where the reward gradient is high are chosen as subgoals.

3.3.1 Subgoal Discovery by Using Learned Policies

An example that shows the importance of a subgoal is a room to room navigation task where the agent should discover the utility of doorways as subgoals. If the agent discovers that a doorway is a subgoal, then it can learn a policy(option) to reach the doorway in order to accelerate the task of navigation. The idea of using subgoals is not, however, limited to grid worlds. For example, for a robot arm attempting to pick up certain objects, a subtask can be recognizing the objects and thus states in which the object are found and recognized can be a subgoal [14].

The main goal of automatic subgoal discovery is finding useful subgoals that can be defined in the agent's state space. Options to those subgoals are learned and added as actions. One of the problems with the subgoal discovery techniques presented in the previous section is that they heavily rely on the identification of successful trajectories or on the reward. Both of them, however, might be very specific to the already learned task and might thus not transfer well to other tasks. For this reason, [14] introduced a method for subgoal discovery which identifies subgoals as states which show a specific structural property under the learned policy, namely that they form local "funnels" for state space trajectories under the learned policy. To determine such states, this approach calculates the number of (potentially distant) predecessors and analyzes them for statistically significant outliers along policy-derived trajectories. The method here relies on the tact that in a uniformly connected space, where states have approximately the same expected number of direct predecessors under a given policy, every state will have an approximately equal number of direct predecessors under a given policy, except for regions near the goal state or close to the boundaries.

Definition 3.5 [14] : A state s is a direct predecessor to state s' if under the learned policy the action in state s can lead to s', i.e. $P(s'|s,a) > 0$.

Definition 3.6 [14] : Let $s_1, \ldots, s_n$ be the direct predecessor of state s. The Count metric for a state s under a learned policy is given by the sum of the count metrics of all the direct predecessors of s weighted by the transition probability plus the sum of the probabilities of all direct predecessors to lead to s.

Let $C(s)$ represents the count of predecessors for a state s under a given policy o, and $C_t(s)$ be the count of predecessors that can reach s in exactly t steps. then:

$$C_1(s) = \sum_{s' \neq s} P(s|s', o(s')) \tag{3.3}$$

and

$$C_{t+1}(s) = \sum_{s' \neq s} P(s|s', o(s')) C_t(s') \tag{3.4}$$

$$C(s) = \sum_{i=1}^{n} C_i(s) \tag{3.5}$$

where n is such that $C_{n+1} = C_n$ or $n = |S|$, whichever is smaller. The condition $s' \neq s$ prevents the counting of self loops and $P(s|s', o(s'))$ is the probability of reaching state s from state s' by executing action $o(s')$. In order to calculate the ratio along a path under a given policy, let $C(s_1)$ be the predecessor count for the initial state of the path and $C(s_t)$ be the count for the state that the agent will be in after t steps. Then we can compute:

$$\Delta_t = C(s_t) - C(s_{t-1}) \tag{3.6}$$

In order to identify the subgoals, the gradient ratio Δ_t/Δ_{t+1} is computed. If $\Delta_t < \Delta_{t+1}$ then the ratio is less than 1 and the state does not fit the criterion. However, if $\Delta_t > \Delta_{t+1}$ and the ratio is greater than a specified threshold, then state s' is a potential candidate

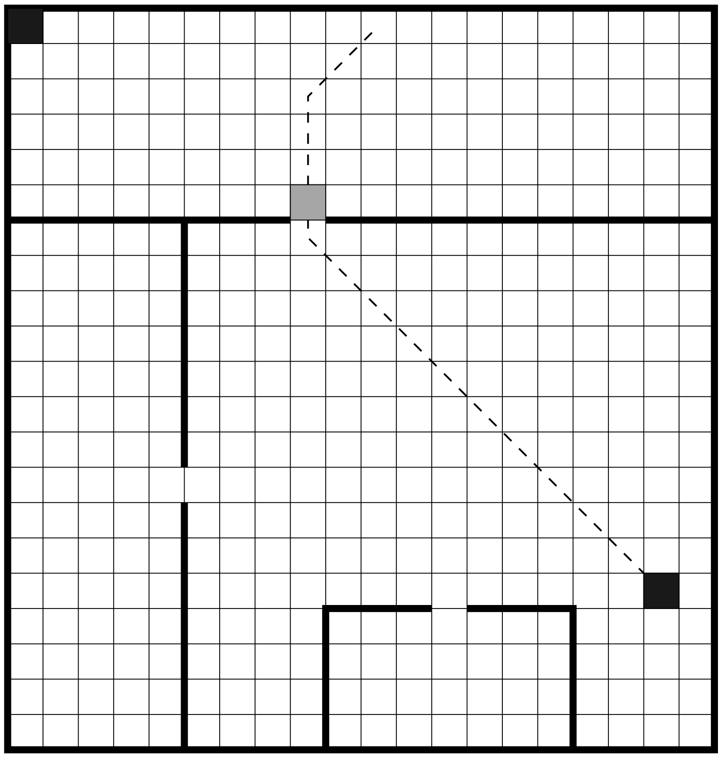


Figure 3.5. The grid world for the example.

for a subgoal. The threshold here depends largely on the characteristics of the state space but can often be computed independent of the particular environment.

3.3.2 Example

Figure 3.5 [14] shows a four-room example environment on a 20×20 grid. For this experiment, the goal state was placed in the lower right portion and each trial started from the same state in the left upper corner as shown in Figure 3.5.

The action space consists of eight primitive actions (North, East, South, West, Northwest, Northeast, Southwest and Southeast). The world is deterministic and each action succeeds in moving the agent in the chosen direction. With every action the agent receives a negative reward of -1 for a straight action and -1.2 for a diagonal action. In addition, the agent gets a reward of $+10$ when it reaches the goal state. Here, the learn-

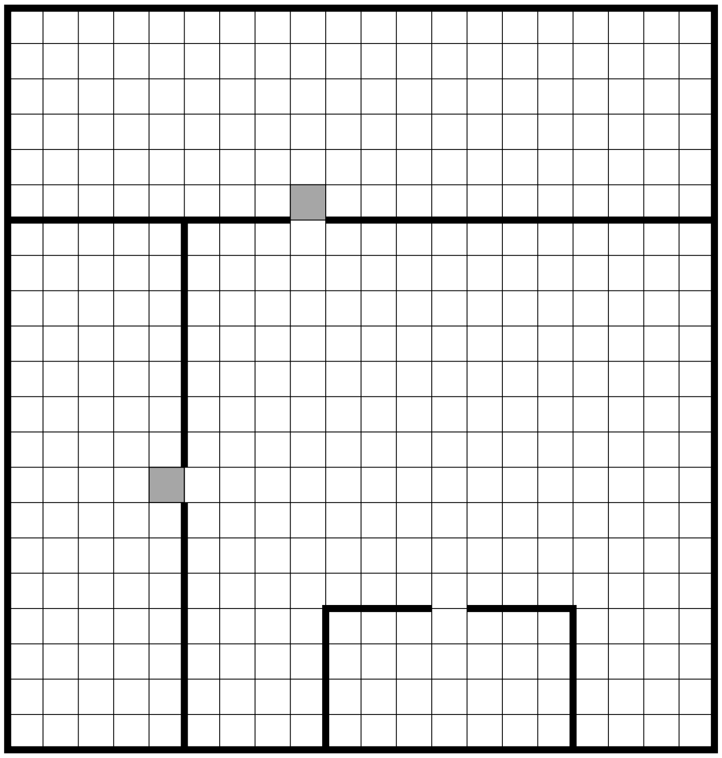


Figure 3.6. Extracted subgoals in the example.

ing method is Q-learning and the predecessor count for every state is computed using Equations 3.3, 3.4 and 3.5. The agent then evaluates the ratio of gradients along the count curve by choosing random paths, and picks the states in which the ratio is higher than a specified threshold as a subgoal states. For example, the count curve is calculated along one path illustrated in Figure 3.5. The value for the gradient ratio at step 4 along this trajectories is 1.444 while it is 95.0 at step 6 (which is a subgoal). Extracted subgoals are illustrated in Figure 3.6.

3.4 Approaches to Hierarchical Reinforcement Learning

Parr [24] and Parr and Russell [25] developed an approach to hierarchically structuring MDP policies called Hierarchies of Abstract Machines or HAMs. Like the options formalism, HAMs exploit the theory of SMDPs, but the emphasis is on simplifying complex MDPs by restricting the class of realizable policies rather than expanding the action choices. In this respect, as pointed out by Parr [24], it has much in common with the multilayer approach for controlling large Markov chains that considered a two-layer structure in which the lower level controls the plant via one of a set of pre-defined regulators. The higher level, the supervisor, monitors the behavior of the plant and intervenes when its state enters a set of boundary states. In the options framework, each option corresponds to a low-level regulator, and when the option set does not contain the one-step options corresponding to all primitive actions, the same simplification results. HAMs extend this idea by allowing policies to be specified as hierarchies of stochastic finite-state machines. Dietterich [11] developed another approach to hierarchical RL called the MAXQ Value Function Decomposition. Like options and HAMs, this approach also relies on the theory of SMDPs. Unlike options and HAMs, however, the MAXQ approach does not rely directly on reducing the entire problem to a single SMDP. Instead, a hierarchy of SMDPs is created whose solutions can be learned simultaneously [4].

3.4.1 MAXQ Value Function Decomposition

Dietterich [11] developed an approach to hierarchical RL called the MAXQ Value Function Decomposition, which we call simply MAXQ. The MAXQ approach starts with a decomposition of a core MDP M into a set of subtasks $\{M_0, \ldots, M_n\}$. The subtasks form a hierarchy with M_0 being the root subtask, which means that solving M_0 solves M. Actions taken in solving M_0 consist of either executing primitive actions or policies that solve other subtasks, which can in turn invoke primitive actions or policies of other subtasks, etc.

The structure of the hierarchy is summarized in a task graph, an example of which is given in Figure 3.7 for a Taxi problem that Dietterich used as an illustration. Each

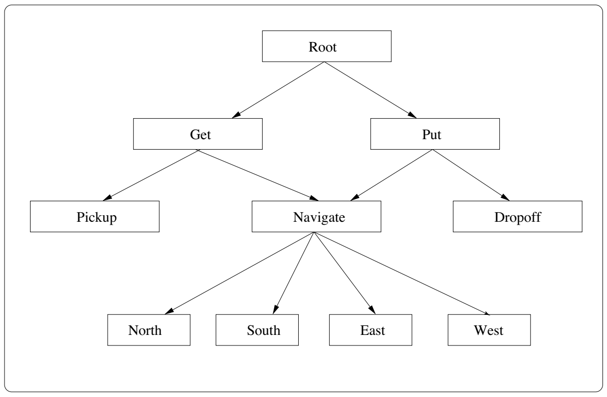


Figure 3.7. A task graph for the taxi problem [11].

episode of the overall task consists of picking up, transporting, and dropping off a passenger. The overall problem, corresponding to the root node of the graph, is decomposed into the subtask *Get*, which is the subtask of going to the passenger's location and picking them up, and the subtask *Put*, which is the subtask of going to the passenger's destination and dropping them off. These subtasks, in turn, are respectively decomposed into the primitive actions *Pickup* or *Dropoff* which ,respectively, pick up and drop off a passenger, and the subtask $Navigate(t)$, which consists of navigating to one of the locations indicated by the parameter t. (A parameterized subtask like this is shorthand for multiple copies of the subtask, one for each value of the parameter.) This subtask $Navigate(t)$ is decomposed into the primitive actions that are moves *North*, *South*, *East*, or *West*. The subtasks and primitive actions into which a subtask m_i is decomposed are called the "children" of M_i. An important aspect of a task graph is that the order in which a subtask's children are shown is arbitrary and the choice that the higher level controller makes depends on its policy. The graph just restricts the action choices that can be made at each level.

Each subtask, M_i, consists of three components. First, it has a subtask policy, p_i, that can select other subtasks from the set of M_i 's children. Here, as with options, primitive actions are special cases of subtasks. In addition, the subtask policies are assumed to be deterministic. Second, each subtask has a termination predicate that partitions the state set, s, of the core MDP into s_i, the set of active states, in which M_i's policy can execute, and t_i, the set of termination states, which, when entered causes the policy to terminate. Third, each subtask m_i has a pseudo-reward function that assigns reward values to the states in t_i. The pseudo-reward function is only used during learning, which we discuss after first describing how the task graph hierarchy allows value functions to be decomposed.

A subtask is very much like a hierarchical option, I_i, π_i, β_i, with the addition of a pseudo-reward function. The policy over options, M_i, corresponds to the subtask's π_i; the termination condition, β_i, in this case assigns to states termination probabilities of only 1 or 0; and the option's input set I_i corresponds to s_i. Unlike the option formalism, however, which treats semi-Markov options, MAXQ explicitly adds a component to each state that gives the current contents, K, of a pushdown stack containing the names and parameter values of the hierarchy of calling subtasks, as in subroutine handling of ordinary programming languages. At any time step, the top of the stack contains the name of the subtask currently being executed. Thus, while a subtask's policy is non-Markov with respect to the state set of the core MDP, it is Markov with respect to this augmented state set. As a consequence, each subtask policy has to assign actions to every combination, $[s, K]$, of core state s and stack contents K.

Given a hierarchical decomposition of M into n subtasks as given by a task graph, a hierarchical policy is defined to be $\pi = \{\pi_0, \ldots, \pi_n\}$, where π_i is the policy of M_i.

Given this, one can write a Bellman equation for the SMDP corresponding to subtask M_i:

$$V^\pi(i, s) = V^\pi(\pi_i(s), s) + \sum_{s', t} P_i^\pi(s', t | s, \pi_i(s)) \gamma^t V^\pi(i, s')$$

where $V^\pi(i, s')$ is the expected return for completing subtask M_i starting in state s' and $V^\pi(a, s)$ is the selected immediate reward of executing a in state s.

The action-value function, Q, can be extended to apply to subtasks: for hierarchical policy π, $Q^\pi(i, s, a)$ is the expected return for action a (a primitive action or a child subtask) being executed in subtask M_i and then π being followed until M_i terminates. In terms of this subtask action-value function, this observation takes the form:

$$Q^\pi(i, s, a) = V^\pi(a, s) + \sum_{s',t} P_i^\pi(s', t|s, a)\gamma^t Q^\pi(i, s', \pi(s'))$$

3.4.2 Hierarchical Abstract Machines

Parr [24] developed an approach to hierarchically structuring MDP policies called Hierarchies of Abstract Machines or HAMs. Like the options formalism, HAMs exploit the theory of SMDPs, but the emphasis is on simplifying complex MDPs by restricting the class of realizable policies rather than expanding the action choices. In this respect, as pointed out in [24], it has much in common with the multilayer approach for controlling large Markov chains which considered a two-layer structure in which the lower level controls the plant via one of a set of pre-defined regulators. The higher level, the supervisor, monitors the behavior of the plant and intervenes when its state enters a set of boundary states [4].

Intervention takes the form of switching to a new low-level regulator. This is not unlike many hybrid control methods except that the low-level process is formalized as a finite MDP and the supervisor's task as a finite SMDP. The supervisor's decisions occur whenever the plant reaches a boundary state, which effectively "erases" the intervening states from the supervisor's decision problem, thereby reducing its complexity. In the options framework, each option corresponds to a low-level regulator, and when the option set does not contain the one-step options corresponding to all primitive actions, the same simplification results. HAMs extend this idea by allowing policies to be specified as hierarchies of stochastic finite-state machines.

The idea of the HAM approach is that policies of a core MDP are defined as programs which execute based on their own states as well as the current states of the core MDP. Departing somewhat from Parr's [24] notation, let M be a finite MDP with state set S and action sets A_s,$s \in S$. A HAM policy is defined by a collection of stochastic finite-state machines, $\{h_i\}$, with state sets s_i, stochastic transition functions δ_i, and input sets all equal to M's state set, s. Each machine h_i also has a stochastic function $I_i : S \rightarrow S_i$ that sets the initial state of M in the manner described below. Each h_i has four types of states: *action*, *call*, *choice*, and *stop*. An action state generates an action of the core MDP, M, based on the current state of M and the current state of the currently executing machine, say h_i. That is, at time step t, the action is $a_t = \pi(m_t^i, s_t) \in A_s$, where m_i^t is the current state of h_i and s_t is the current state of M. A *call* state suspends execution of the currently executing h_i and initiates execution of another machine, say h_j, where j is a function of h_i's state m_i^t. Upon being called, the state of h_j is set to $I_j(s_t)$. A *choice* state nondeterministically selects a next state of h_i. Finally, a *stop* state terminates execution of h_i and returns control to the machine that called it (whose execution commences where it was suspended). Meanwhile, the core MDP, upon receiving an action, makes a transition to a next state according to its transition probabilities and generates an immediate reward [4].

If no action is generated at step t, then M remains in its current state. Parr defines a HAM H to be the initial machine together with the closure of all machine states in all machines reachable from the possible initial states of the initial machine. Let us call this state set S_H. For convenience, he also assumes that the initial machine does not have a *stop* state and that there are no infinite, i.e. probability 1, loops that do not contain *action* states. This ensures that the core MDP continues to receive primitive actions.

Figure 3.8 [24] shows a simple HAM state-transition diagram similar to an example given in the previous section for controlling a simple simulated mobile robot. This HAM runs until the robot reaches an intersection. Whenever an obstacle is encountered, a *choice* state is entered that allows the robot to decide to back away from the obstacle by calling the machine back-off or to try to get around the obstacle by calling the machine follow-wall. Each of these machines has its own state-transition structure, possibly containing additional *choice* and *call* states. When this HAM is selected, it deterministically starts by calling the follow-wall machine.

The composition of a HAM H and an MDP M, as described above, yields a discrete-time

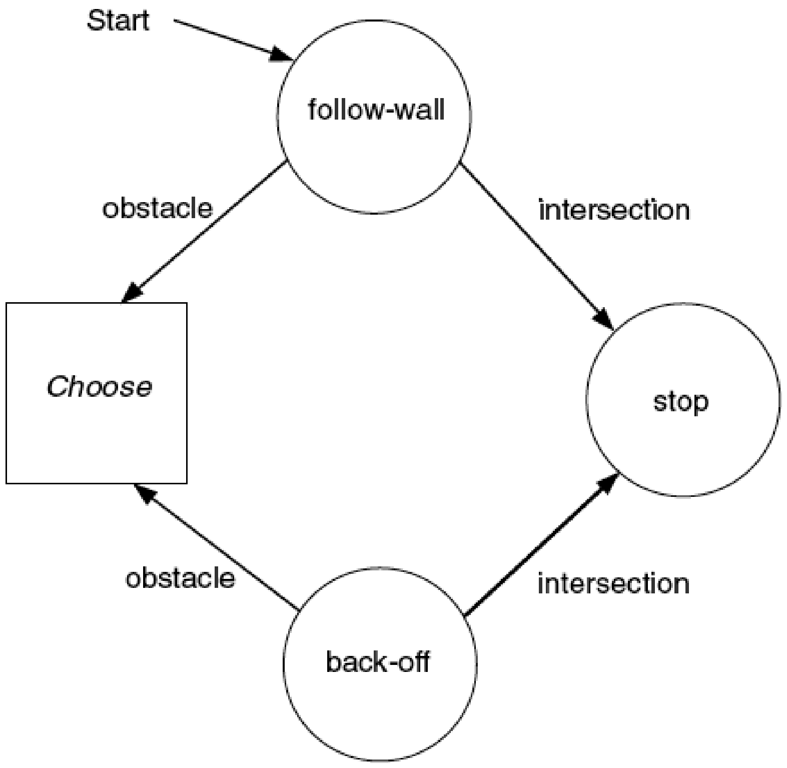


Figure 3.8. State-transition structure of a simple HAM Parr [24].

SMDP denoted by $H \circ M$. The state set of $H \circ M$ is $S \times S_H$, and its transitions are determined by the parallel actions of the transition functions of H and M. The only actions of $H \circ M$ are the choices allowed in the choice points of $H \circ M$, which are the states whose H components are *choice* states. These actions change only the HAM component of each state. This is an SMDP because after a choice is made, the composition of H and M runs autonomously until another choice point is reached [4].

All the primitive actions to M during this period are fully determined by the *action* states of H. The expected immediate rewards of $H \circ M$ are the expected returns accumulated during these periods between choice points, and they are determined by the immediate rewards of M together with rewards of zero for the time steps in which M's state does not change. Thus, one can think of a HAM as a method for delineating a possibly drastically restricted set of policies for M. This restriction is determined by the prior knowledge that the HAM's designer, or programmer, has about what might be good ways to control M.

CHAPTER 4

HIERARCHICAL ACTIONS AND STATE SPACE CONSTRUCTION

This section describes the main contribution of this dissertation. In this approach, the agent uses information acquired while learning one task to discover subgoals for similar tasks. The agent is able to transfer knowledge to subsequent tasks and to accelerate learning by creating useful new subgoals and by off-line learning of corresponding subtask policies as abstract actions (options). At the same time, the subgoal actions are used to construct a more abstract state representation using action-dependent state space partitioning. This representation forms a new level in the state space hierarchy and serves as the initial representation for new learning tasks. In order to ensure that tasks are learnable, value functions are built simultaneously at different levels of the hierarchy and inconsistencies are used to identify actions to be used to refine relevant portions of the abstract state space. Together, these techniques permit the agent to form more abstract action and state representations over time. Figure 4.1 illustrates these procedures.

4.1 Autonomous Subgoal Discovery

In this section we introduce our subgoal discovery algorithm and we show how it can be improved using Monte Carlo sampling. The discovered subgoals will be used in the next section in order to build a more abstract state space which is action dependent. An example that shows the importance of a subgoal is a room to room navigation task where the agent should discover the utility of doorways. If the agent discovers that a doorway is a subgoal it can learn an option to reach the doorway which, in turn, can accelerate learning of new navigation tasks. The idea of using subgoals is not, however,

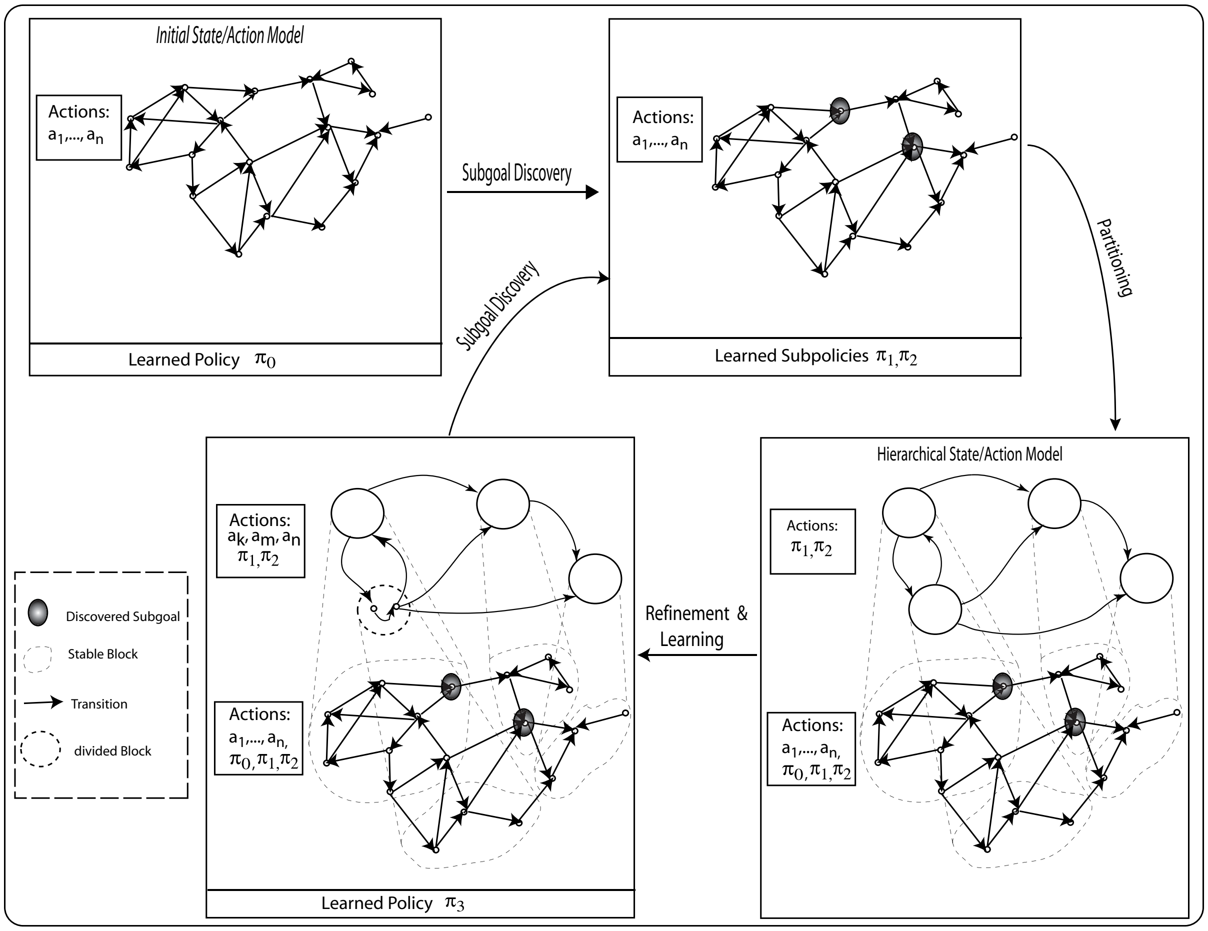

Initial State/Action Model
Actions: $a_1,\ldots,a_n$
Learned Policy π_0
Subgoal Discovery
Actions: $a_1,\ldots,a_n$
Learned Subpolicies π_1,π_2
Partitioning
Subgoal Discovery
Hierarchical State/Action Model
Actions: π_1,π_2
Actions: $a_1,\ldots,a_n,$ π_0,π_1,π_2
Refinement & Learning
Actions: a_k,a_m,a_n π_1,π_2
Actions: $a_1,\ldots,a_n,$ π_0,π_1,π_2
Learned Policy π_3
Discovered Subgoal
Stable Block
Transition
divided Block

Figure 4.1. Overview of the approach for hierarchical state and action abstraction. Here, boxes indicate state/action models where the transition diagram indicates the action dependent state space model and the action set to its left represents the action set used for its construction. Policies listed at the bottom of a box indicate the policies learned in the corresponding step. Starting from an initial MDP and action set, the agent first learns a task policy π_0. Using this policy, it extracts subgoals and corresponding subgoal policies, π_1 and π_2, which are used to build a new, more efficient representation. To ensure that the new task is learnable, the system maintains a separate value function for the original state space and determines if there are significant inconsistencies between this value function and the one derived on the abstract partition space. When an inconsistency between the values for the best actions (in this case a_k, a_m, a_n) in both representations is discovered, these actions are used to refine the block in the abstract state partition in which the inconsistency was discovered, resolving the inconsistency.

limited to grid worlds. For example, for a robot arm to pick up an object, an important subtask is the recognition of the object and thus being aware of its presence would be a subgoal.

4.1.1 Subgoal Discovery Using a Count Metric

The main goal of automatic subgoal discovery is to find useful subgoals in the agent's state space. Once they are found, options to those subgoals can be learned and added to the behavioral repertoire of the agent. In the approach presented here, subgoals are identified as states with particular structural properties in the context of a given policy. In particular, we define subgoals as states that, under a given policy, lie on a substantially larger number of paths than would be expected by looking at its successor states. In other words, we are looking for states that form a "funnel" for state space trajectories under the learned policy. This method is based on the subgoal discovery method introduced by Goel and Huber [14] which was presented in Sections 3.4.1 and 3.4.2. However, while their subgoal discovery method discovers useful subgoals, it needs to compute the count metric exhaustively. This section describes an extension to the subgoal discovery method introduced by Goel and Huber [14] which uses Monte Carlo sampling in order to accelerate the task of finding subgoals.

Definition 4.1: *A state s' is a direct predecessor of state s, if under a learned policy the action a in state s' can lead to state s i.e. $P(s|s', a) > 0$.*

Definition 4.2: *The count metric for state s under a learned policy, π, is the sum over all possible state space trajectories weighed by their accumulated likelihood to pass through state s.*

Definition 4.3: The normalized count metric $\hat{C}^*_\pi(s)$ for state s under a learned policy π is the average number of times that a state s is encountered on a state space trajectory.

Let $\hat{C}^*_\pi(s)$ be the normalized count for state s and $\hat{C}^t_\pi(s)$ be the likelihood of a trajectory passing through state s at time t, then:

$$\hat{C}^1_\pi(s) = \frac{1}{|S|}\sum_{s' \neq s} P(s|s', \pi(s'))$$

where S is a number of states and

$$\hat{C}^t_\pi(s) = \sum_{s' \neq s} P(s|s', \pi(s'))\hat{C}^{t-1}_\pi(s')$$

$$\hat{C}^*_\pi(s) = \sum_{i=1}^{n} \hat{C}^i_\pi(s) \tag{4.1}$$

where n is such that $\hat{C}^n_\pi(s) = \hat{C}^{n+1}_\pi(s)$ or $n = |S|$. The condition $s' \neq s$ prevents the counting of self loops and $P(s|s', \pi(s'))$ is the probability of reaching state s from state s' by executing action $\pi(s')$. The standard count metric can be obtained from the normalized count metric, since $C^*_\pi(s) = |S|\hat{C}^*_\pi(s)$. The slope of $C^*_\pi(s_t)$ along a path, ρ, under policy π is:

$$\Delta_\pi(s_t) = C^*_\pi(s_t) - C^*_\pi(s_{t-1})$$

where s_t is the t^{th} state along the path. In order to identify subgoals, the gradient ratio $\Delta_\pi(s_t)/\max(1, \Delta_\pi(s_{t+1}))$ is computed for states where $\Delta_\pi(s_t) > 0$ [1]. A state s_t is considered a potential subgoal candidate if the gradient ratio is greater than a specified threshold $\mu > 1$. Appropriate values for this user-defined threshold depend largely on the characteristics of the state space and result in a number of subgoal candidates that is inversely related to the value of μ. This approach is an extension of the criterion in [14]

[1]Alternatively, the gradient ratio could also be computed using the normalized count metric as $\hat{\Delta}_\pi(s_t)/\max(1/|S|, \hat{\Delta}_\pi(s_{t+1}))$ where $\hat{\Delta}_\pi(s_t) = \hat{C_\pi}^*(s_t) - \hat{C_\pi}^*(s_{t-1})$.

with $\max(1, \Delta_\pi(s_{t+1}))$ addressing the effects of potentially obtaining negative gradients due to nondeterministic transitions. In this approach we focus our attention on state spaces that are regular, i.e., every state has approximately the same expected number of direct predecessors according to a learned policy, but it could easily be extended to irregular state spaces.

4.1.1.1 Detecting Significant States

To make this subgoal discovery completely autonomous it is important to be able to automatically detect states that have gradient ratios that are significantly larger than expected, and thus to automatically determine the detection threshold μ. Assuming that there are a small number of subgoals as compared to the size of the space, the theory of the t-test is applied here to compute a threshold. Based on this assumption it can be stated that the distribution of gradient ratios over the entire space represents approximately the distribution in a space free of subgoals. Another assumption made here is that the gradient ratios at any state are randomly drawn from the cumulative distribution of the gradient ratios which has a mean of ν and a standard deviation of σ. Given this assumption, it can be tested using a t-test whether the gradient ratio distribution at any state belongs to the cumulative distribution and its affirmation means that the state is not a subgoal. To run a t-test for a given state s, the mean, ρ, of a sample of gradient ratios of size N is computed by randomly choosing N trajectories going through state s under the learned policy. Then Equation 4.2 is used to compute the t value.

$$t = \frac{\rho - \nu}{\sigma / \sqrt{N}} \tag{4.2}$$

To avoid performing a t-test at every state, a slightly different approach is taken here. Based on the desired p value (the probability of obtaining a particular sample result given the null hypothesis), the sample mean ρ required to pass the test is computed using the t-test formula (Equation 4.2).

The value for ρ determined in this fashion plays the role of threshold, μ, required in this approach because in order for the t-test to provide a positive result for the sample distribution at a given state s, there should be at least one path through s along which the gradient ratio at state s is equal or greater than ρ.

4.1.1.2 Detecting Subgoals Using Monte Carlo Sampling

While the subgoal discovery technique provides a way to automatically detect states that have significant structural property and form local "funnels" in the state space, the complexity of computing and evaluating the count metric can be prohibitive in large state spaces. To address this, the method has been augmented with an approach that permits the calculation of the count metric using Monte Carlo sampling.

Definition 4.4 *Let $H = \{h_1, ..., h_N\}$ be N sample trajectories induced by policy π, then the sampled normalized count metric, $\hat{C}^*_H(s)$, for each state s that is on the path of at least one path h_i can be calculated as the average of the accumulated likelihoods of each path, h_i, $1 \leq i \leq N$.*

If we need to estimate the value of $\hat{C}^*_\pi(s)$ from independent, identically distributed (i.i.d) samples induced by policy π, then after taking N samples h_i, $i \in \{1, \ldots, N\}$ we have:

$$\hat{C}^*_H(s) = \frac{1}{N} \sum_i C^*_{h_i}(s)$$

The expected value of this estimator is $\hat{C}^*_\pi(s)$. As in the case of the exhaustive count metric, the standard sampled count metric $C^*_H(s_t)$ can be computed from the normalized sampled count metric as $C^*_H(s_t) = |S|\hat{C}^*_H(s_t)$. Theorem 4.2 shows that for a sufficient number of samples the difference between count metrics computed exhaustively and by using Monte Carlo sampling can be arbitrarily small and as a result Monte Carlo sampling gives us a good estimate of count metric. Note, that this theorem applies equally to the count metrics and the normalized count metrics.

Theorem 4.1 (Bernstein [31]) *Let $\xi_1, \xi_2, \ldots$ be independent random variables with means $E\xi_i$, bounded by some constant $E\xi_i \leq a$, $a > 0$. Also let $Var(M_N) = E\xi_1^2 + \ldots + E\xi_N^2 \leq L$ and $Z_N = E\xi_1 + \ldots + E\xi_N$. Then the partial sum $M_N = \xi_1 + \ldots + \xi_N$ obeys the following inequality for all $\epsilon_N > 0$:*

$$Pr\left(\left|\frac{1}{N}M_N - \frac{1}{N}Z_N\right| > \epsilon_N\right) \leq 2exp\left(-\frac{1}{2}\frac{\epsilon_N^2 N}{L + a\epsilon_N}\right)$$

The following theorem shows that for a large number of samples the difference between C_π^* and C_H^* would be smaller than a real value ϵ_N and thus subgoal discovery by using Monte Carlo sampling can find the same subgoals that are discovered by exhaustive computation of the count metric, when the number of samples is sufficiently large.

Theorem 4.2 *In a regular space,for sample size*

$$N \geq \frac{\max_t C_H^*(s_t)}{\epsilon_N^2} 2(1 + \epsilon_N) ln(\frac{2}{1-p}) \tag{4.3}$$

it is true that

$$|C_H^*(s_t) - C_\pi^*(s_t)| \leq \epsilon_N$$

with probability p.

Proof: Let $\xi_i = C^*_{h_i}(s_t)$ then $E\xi_i = E[C^*_{h_i}(s_t)]$. Let

$$\begin{aligned} Z_N &= \\ E\xi_1 + \ldots + E\xi_N &= \\ E[C^*_{h_1}(s_t)] + \ldots + E[C^*_{h_N}(s_t)] &= \\ E[C^*_{h_1}(s_t) + \ldots + C^*_{h_N}(s_t)] \end{aligned}$$

since ξ_i are independent.

Thus

$$\frac{1}{N} Z_N = \frac{1}{N} E[C^*_{h_1}(s_t) + \ldots + C^*_{h_N}(s_t)] = E[C^*_H(s_t)] = C^*_\pi(s_t).$$

Let

$$a = \max_t C^*_H(s_t)$$

then $|E\xi_i| \leq a$, $a > 0$. Also for

$$M_N = E\xi_1 + \ldots + E\xi_N$$

we have

$$Var(M_N) = E\xi_1^2 + \ldots + E\xi_N^2 \leq N(\max_t C^*_H(s_t))^2.$$

Let

$$L = N(\max_t C^*_H(s_t))^2$$

and according to Bernstein's inequality:

$$Pr\left(|C^*_H(s_t) - C^*_\pi(s_t)| > \epsilon_N\right) \leq$$

$$2exp\left(-\frac{1}{2}\frac{\epsilon_N^2 N}{\max_t C^*_H(s_t)^2 + \max_t C^*_H(s_t)\epsilon_N}\right)$$

then

$$Pr\left(|C^*_H(s_t) - C^*_\pi(s_t)| > \epsilon_N\right) \leq 2exp\left(-\frac{1}{2}\frac{\epsilon_N^2 N}{\max_t C^*_H(s_T)\epsilon_N}\right)$$

by letting

$$1 - p = Pr\left(|C^*_H(s_t) - C^*_\pi(s_t)| > \epsilon_N\right)$$

we have

$$1 - p \leq 2exp\left(-\frac{1}{2}\frac{\epsilon_N^2 N}{\max_t C^*_H(s_t)\epsilon_N}\right)$$

After solving for N, we get the statement of theorem 4.2.□

Given the result of Theorem 4.2, it can be shown that for a sufficient large number of samples, subgoals discovered by Monte Carlo sampling will likely find the same subgoals as the exhaustive subgoal discovery method.

Theorem 4.3 *Let $H = \{h_1, ..., h_N\}$ be N sample trajectories induced by policy π with N selected according to Equation 4.3. If*

$$\frac{\Delta_H(s_t)}{\max(1, \Delta_H(s_{t+1}))} > \mu + \frac{2\epsilon_N}{\max(1, \Delta_H(s_{t+1}))}$$

then

$$\frac{\Delta_\pi(s_t)}{\max(1, \Delta_\pi(s_{t+1}))} > \mu$$

with probability $\geq p$.

Proof: Let

$$N \geq \frac{\max_t C^*_H(s_t)}{\epsilon_N^2} 2(1 + \epsilon_N) ln(\frac{2}{1-p})$$

then $|C^*_H(s_t) - C^*_\pi(s_t)| \leq \epsilon_N$ and Similarly $|C^*_H(s_{t+1}) - C^*_\pi(s_{t+1})| \leq \epsilon_N$, thus

$$\begin{aligned} |\Delta_\pi(s_t) - \Delta_H(s_t)| &= \\ |C^*_\pi(s_t) - C^*_\pi(s_{t+1}) - C^*_H(s_t) + C^*_H(s_{t+1})| &\leq \\ |C^*_\pi(s_t) - C^*_H(s_t)| + |C^*_\pi(s_{t+1}) - C^*_H(s_{t+1})| &\leq \\ 2\epsilon_N & \end{aligned}$$

and

$$-2\epsilon_N \leq \Delta_\pi(s_t) - \Delta_H(s_t) \leq 2\epsilon_N$$

and

$$\Delta_\pi(s_t) - 2\epsilon_N \leq \Delta_H(s_t) \leq \Delta_\pi(s_t) + 2\epsilon_N$$

Similarly we can show that

$$\Delta_\pi(s_{t+1}) - 2\epsilon_N \leq \Delta_H(s_{t+1}) \leq \Delta_\pi(s_{t+1}) + 2\epsilon_N \tag{4.4}$$

by dividing inequality Equation 4.4 by $\max(1, \Delta_H(s_{t+1}))$ we have:

$$\frac{\Delta_H(s_t)}{\max(1, \Delta_H(s_{t+1}))} \leq \frac{\Delta_\pi(s_t) + 2\epsilon_N}{\max(1, \Delta_H(s_{t+1}))} \tag{4.5}$$

if

$$\frac{\Delta_H(s_t)}{\max(1, \Delta_H(s_{t+1}))} > \mu + \frac{2\epsilon_N}{\max(1, \Delta_H(s_{t+1}))}$$

then according to Equation 4.5

$$\frac{\Delta_\pi(s_t) + 2\epsilon_N}{\max(1, \Delta_H(s_{t+1}))} \geq \mu + \frac{2\epsilon_N}{\max(1, \Delta_H(s_{t+1}))}$$

and as a result $\frac{\Delta_\pi(s_t)}{\max(1, \Delta_\pi(s_{t+1}))} > \mu$.$\square$

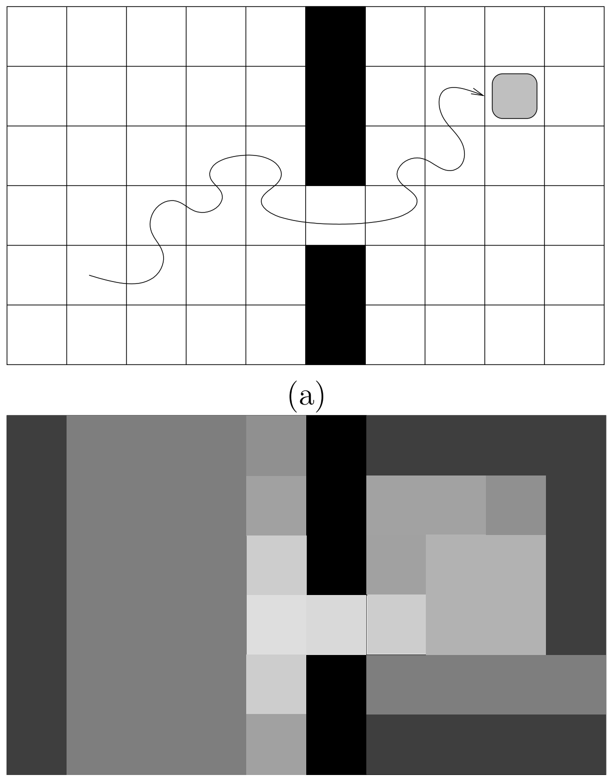
Theorem 4.3 implies that for a sufficiently large sample size the exhaustive and the sampling method predict the same subgoals with high probability. The main advantage of the Monte Carlo sampling method can be seen in the lower computational complexity as the size of the state space increases and its ability to trade off computation time against precision in finding the desired subgoals.

4.1.2 Example

Figure 4.2(a) shows a two-room example environment on a 10×6 grid. For this experiment, the goal state is placed in the upper right hand portion (gray cell) and each trial is started from the same state in the lower left corner. The action space consists of eight primitive actions (North, East, South, West, Northwest, Northeast, Southwest and Southeast). The world is deterministic and each action succeeds in moving the agent in the chosen direction. With every action the agent receives a negative reward of -1 for a straight action and -1.2 for a diagonal action. In addition, the agent receives a reward of $+10$ when it reaches the goal state. Policy π is learned using Q-learning and the count metric for every state is computed. The agent then evaluates the gradient ratio along the count curve by choosing 10 random trajectories according to π, and picks the states in which the ratio is higher than the specified threshold as subgoal states. Figure 4.2(b) shows the values of the gradient ratio for each state. In this example, the gradient ratio is less than 3 in all states except the doorway where it is 3. The mean of the distribution of gradient ratios over the state space is 0.43 and the standard deviation is 0.57. Using t-test and probability 0.025 the threshold μ is chosen to be 1.8 resulting in the discovery of one subgoal in the location of the doorway. This subgoal could now be used to learn similar tasks in this environment.

4.2 Action Dependent State Abstraction

Once potential subgoals are discovered, options that terminate in the subgoal states can be learned and added as abstract actions to the action hierarchy available to the agent.



(b)

Figure 4.2. (a) A two room environment with a connecting doorway. The goal is illustrated in gray and the line shows a sample trajectory. (b) Cells are shaded according to the gradient ratio over 10 trajectories with higher ratios indicated by lighter shading.

In addition, these options can be used to build a more compact representation of the state space. Abstraction is achieved here by partitioning the state space of the original MDP into blocks of states that have similar properties (i.e. transition probabilities and reward values).

ϵ-reductions were introduced by Dean et al. [9] as a mechanism to derive state space partitions of a MDP that ensure approximately optimal policies to be learned. These partitions depend on the action space and the particular reward function of the task. Kim and Dean, [18] introduced an algorithm to derive a set of such partitions and used it to learn a policy for the task indicated by the reward function. The resulting policy is ensured to be within an ϵ-dependent quality bound.

Two shortcomings of this algorithm are that it does not address temporally abstract actions and that it requires a complete re-computation of the partitions when a new action is introduced. In addition it requires knowledge of the reward function prior to partitioning, and thus no part of the partitioning transfers across tasks.

4.2.1 Multi-Phase State Space Partitioning for SMDPs

To address the issues of ϵ-reduction, the algorithm introduced in this section derives partitions with the same approximate optimality properties for the SMDP framework. In addition, it derives them in multiple phases, the first of which derives separate partitions for all actions in a reward-independent fashion using the action's subgoal or termination states as the initial partition for subsequent probability-based refinement. These partitions can be completely precomputed at the time the actions are acquired. In the second phase, the action-specific partitions for a relevant subset of the actions are selected, intersected, and used for a second reward-independent, transition probability-based refinement step. Again, this phase is task-independent and can thus be performed off-line and its resul can be transferred to new tasks. The third stage finally refines the reuslt of the previous stage using the task-specific reward, once it becones available. Overall, this construction process allows the system to effectively transfer state abstractions even prior to having knowledge of the specific task to be learned, and permits fast adaptation to changing action sets. Furthermore, itmaintains the bounded optimality guarantees of the ϵ-reduction method [18].

The method introduced here also provides for extensions to ϵ-reduction that permit its use with options in an SMDP framework. Overall, this approach first constructs the options $o_i = (I_i, \pi_i, \beta_i)$ according to the discovered subgoals. It then computes the transition probability function $F(s|s', o_i)$ and the reward function $R(s, o_i)$ using Equations 3.1

and 3.2. The transition function can here be completely pre-computed at the time when the policy itself is learned and as a result, only the discounted reward estimate has to be re-computed for each new learning task.

4.2.1.1 ϵ, δ-Reduction for SMDPs

To address temporal abstraction and the previously learned subgoal options within the state space reduction framework, two extensions have been added to the ϵ-reduction framework. The first provides more flexibility in terms of the system reward functions by decoupling the BPMDP bounds on transition probabilities and rewards. The second extends the definitions to accomodate SMDP options, providing a framework for Bounded Parameter Semi-Markov Decision Processes (BPSMDP).

Definition 4.4: *A partition $P = \{B_1, \ldots, B_n\}$ of the state space of an SMDP has approximate stochastic bisimulation homogeneity if and only if for each $B_i, B_j \in P$, for each $s, s' \in B_i$, and for all actions $o_i \in O$:*

$$\left| \sum_{s'' \in B_j} F(s''|s, o_i(s)) - \sum_{s'' \in B_j} F(s''|s', o_i(s')) \right| \leq \delta \tag{4.6}$$

and

$$|R(s, o_i) - R(s', o_i)| \leq \epsilon \tag{4.7}$$

where $0 \leq \delta \leq 1$ and $\epsilon \geq 0$.

Definition 4.5: *A block B_i is δ-stable with respect to block B_j if and only if Inequality 4.6 holds. B_i is δ-stable if B_i is δ-stable with respect to all blocks of P.*

To form partitions, an initial partition can be formed here either according to the reward criterion in Inequality 4.7 or according to an alternate termination criterion. Each block is constructed according to its available options and it is checked for δ-stability and unstable blocks are split until no unstable blocks remain. When a block B_k is found to be unstable with respect to block B_l, we replace B_k by a set of sub-blocks $B_{k_1}, \ldots, B_{k_m}$ such that B_{k_i} is a maximal sub-block of B_k that is δ-stable with respect to B_l. This process, when started from the initial reward partition, will result in a state space model that has bounded optimality properties similar to the ones of standard ϵ-reduction [18] and can be used to learn new tasks more efficiently.

However, this reduction algorithm requires that the reward function of the task that should be learned is known prior to partitioning and that the action set is constant. If either the reward or the set of actions changes, this basic algorithm would have to be re-run from scratch.

In many real world situations, however, the reward might not be known a priori. Moreover, in the hierarchical learning approach presented here, new actions are frequently added to the system in the form of subgoal options. As a consequence, a more efficient partitioning algorithm is required which can transfer knowledge more efficiently when actions change, and which can deal with situations where the reward is unknown or only partially known.

4.2.1.2 Multi-Phase, Action-Dependent Partitioning of the State Space

The intuition behind learning subgoal options and using them in an SMDP action hierarchy is that complex policies will involve fewer decision points and as a result can be learned substantially faster. In hierarchical learning systems, it is thus useful to remove primitive actions from consideration as more complex actions become available and capable of addressing new problems. To take full advantage of newly learned action capabilities, such a limitation of the action space should also reflect in the state representation. In particular, once low-level actions can be ignored, much more abstract state representations should be sufficient to address the same tasks. However, it is generally not known beforehand at which point lower level actions can be safely ignored without compromising the set of tasks that can be addressed. The state reduction technique has therefore to be flexible and able to adjust to changes in the action space efficiently without incurring the overhead of completely re-computing a partition.

To permit such flexibility, and to facilitate modifications in the action space, the partitioning approach introduced here first derives partitions on a per-action basis independent of the task reward. The blocks of the final partition are then formed by intersecting all blocks for each o_i that are used in the new learning task, followed by a refining stage that achieves δ-stability for the intersections [1, 2]. This reduces the overhead required when the action set changes to the intersection and the final refinement step. Furthermore, it is still independent of the task reward and thus transfers accros tasks. If the reward structure becomes available, the third phase of the partitioning technique further refines the partition using the reward criterion in Inequality 4.7.

Given a particular subset of options, an appropriate abstract state space representation for the learning task can here be derived which is stable according to the criteria in Inequalities 4.6 and 4.7. Furthermore, representation changes due to changes in the action set can be performed efficiently and a simple mechanism can be provided to use the previously learned value function as a starting point when such representation changes occur. This is particularly important if actions are added over time to permit refinement of the initially learned policy by permitting finer-grained decisions.

In the first phase, the multi-phase partitioning approach derives a partition P_i for each action o_i. To do this, it first generates a partition consisting of a termination and a non-termination block. In particular, given $\{s_1, ..., s_n\}$ as the discovered subgoals and $\{o_1, ..., o_n\}$ as the corresponding subgoal options, initial partitions for the options o_i are derived as $P_i = \{B_i^1, B_i^2\}$ where $B_i^1 = \{s_i\}$ and $B_i^2 = \{s | s \in S, s \neq s_i\}$. This termination partition for action o_i is then refined until all blocks are δ-stqable for action o_i.

The second phase then selects all partitions for a given set of actions, intersects them and refines them further until the result is δ-stable for all actions in the selected action set. In particular, let M be a $SMDP$ with a set of n different actions $o_1, \ldots, o_n$ and let $P_1, ..., P_n$ be the action dependent partitions corresponding to each action, where $P_i = \{B_i^1, ..., B_i^{m_i}\}$ for $i \in W = \{i | o_i \in O\}$. Define $\Phi = P_1 \times P_2 \times \ldots \times P_n$, as the cross product of all partitions. Each element of Φ has the form $\phi_j = (B_1^{\sigma_1(j)}, \ldots, B_n^{\sigma_n(j)})$ where σ_j is a function with domain Φ and range $1, \ldots, m_i$. Each element $\phi_j \in \Phi$ corresponds to a $\bar{B} = \cap_{i \in A} B_1^{\sigma_i(j)}$. Since $B_i^k \cap B_i^l = \emptyset$ for all $1 \leq k, l \leq m_i$, Φ is a partition over all actions.

Given a particular subset of actions, a partition for a learning task can thus be derived as the set of all nonempty blocks resulting form the intersection of the subsets for the participating actions. A block in the resulting partition can here be represented by a vector over the actions involved, where each entry indicates the index of the block within the corresponding single action partition. Once the initial blocks are constructed by the above algorithm, these blocks will be refined until they are δ-stable according to Definition 4.6. Changes in the action set therefore do not require a recalculation of the individual partitions but only changes in the length of the vectors representing new states and a recalculation of the final refinement step. This means that changes in the action set can be performed efficiently and a simple mechanism can be provided to use the previously learned value function even beyond the change of actions and to use it as a starting point for subsequent additional learning.

Once this action-dependent but reward-independent partition has been derived, the third phase of the partitioning process uses reward information for the new learning task as it becomes available to further refine this partition using the criterion in Inequalities 4.6

and 4.7.

This last phase is the only task-specific partitioning phase and can thus only be performed once task information is available. All other phases can be performed off-line and thus facilitate the efficient transfer not ony of subgoal and action information but also of abstraction knowledge to new tasks.

4.2.1.3 Task-Solvability on Reward Independent Partitions

Real environments usually do not provide all the necessary information for an agent, and the agent needs to find out these details by itself. For example, it is common that an agent does not have full information of the reward structure. In these situations, constructing the immediate reward partition is not possible and no reward refinement can be performed prior to learning. In these situations the multi-phase partitioning approach presented in the previous section can still construct a reward-independent partition by distinguishing terminal states for available actions from non-terminal states and refining them using the transition probabilities by executing the first two partitioning phases. If the reward structure becomes available later on, it can further refine this partition using the reward and transition criterion.

The advantage of this construction is that the learning process can already utilize and benefit from the reward-independent partition while learning the task and potentially performing the reward abstraction. Furthermore, whenever a change in the reward criteria (and thus in the task) occurs, only the final refinement part has to be recomputed.

To fully appreciate the benefit and potential of this initial, task-independent partition, Theorem 4.4 (below) shows that under certain conditions this task-independent, action-specific state representation is sufficient to ensure that the new task can be learned. In particular, this theorem shows that if the goal is reachable by subgoal options and a task is learnable on the original state space, then this task can also be learned in the abstract state space.

Theorem 4.4: *For any policy π for which the goal G can be represented as a conjunction of terminal sets (subgoals) of the available actions in the original MDP M, there is a policy π_P in the reduced MDP, M_P , that achieves G as long as for each state s_t in M for which there exists a path to G , there exists a path such that $F(G|s_t, \pi_P(s_t)) > \delta$.*

Proof: The blocks of partition $\Phi = \{B_1, ..., B_n\}$ have the following property

$$\left| \sum_{s \in B_j} F(s|s_1, o_i(s_1)) - \sum_{s \in B_j} F(s|s_2, o_i(s_2)) \right| \leq \delta \tag{4.8}$$

For every policy π that fulfills the requirements of the proposition, there exists a policy π_Φ in partition space such that for each $n \in \aleph$, if there is a path of length n from state s_0 to a goal state G, under policy π, then there is a path for block B_{s_0} containing s_0 to block B_G containing G, under policy π_Φ.

Case $k = 1$: if $F(G|s_0, \pi(s_0)) > \delta$ then by inequality 4.8 for all $s \in B_{s_0}$,

$$\left| \sum_{s' \in B_G} F(s'|s_0, \pi(s_0)) - \sum_{s' \in B_G} F(s'|s, \pi(s_0)) \right| \leq \delta$$

thus $\forall s \in B_{s_0}$ we have

$$F(G|s, \pi(s_0)) > F(G|s_0, \pi(s_0)) - \delta > 0.$$

where policy π_Φ is such that $\pi_\Phi(B_{s_0}) = \pi(s0)$, then $F(B_G|B_{s_0}, \pi_\Phi(B_{s_0})) > 0$.

Case $k = n - 1$: Assume for each path of length less than or equal to $n - 1$ that reaches state G from s_0 under policy π, there is a path under policy π_Φ in the partition space.

Case $k = n$: Each path that reaches to G from s_0 under policy π in n steps contains a path with $n-1$ steps, that reaches G from s_1 under policy π. By induction hypothesis, there is a policy π_Φ that leads to B_G from B_{s_1}. Now if s_0 is an element of $B_{s_n} \cup B_{s_{n-1}} \cup, \ldots, \cup B_{s_1}$, the blocks already chosen by paths with length less than or equal $n-1$, then there is a policy π_Φ that leads to B_G from B_{s_0} under policy π_Φ and the policy $\pi_\Phi(B_{s_0})$ is already defined. But if $s_0 \notin B_{s_n} \cup B_{s_{n-1}} \cup, ..., \cup B_{s_1}$ then by induction hypothesis it has only to be shown that there is a policy π_Φ that fulfills the induction hypothesis and which leads from B_{s_0} to B_{s_1} such that $F(B_{s_1}|B_{s_0}, \pi(s_0)) > 0$. By inequality 4.8 $\forall s_1, s_2 \in B_{s_0}$ we have

$$|\sum_{s' \in B_G} F(s|s_0, \pi_i(s_0)) - \sum_{s' \in B_G} F(s'|s, \pi_i(s_0))| \leq \delta$$

thus

$$\begin{aligned} F(B_{s_1}|B_{s_0}, \pi_\Phi(B_{s_0})) &= \sum_{s' \in B_{s_1}} F(s'|s, \pi(s_0)) \\ &\geq \sum_{s_0 \in B_{s_1}} F(s'|s_0, \pi(s_0)) - \delta > 0. \square \end{aligned}$$

This suggests that, if competent actions are available and the correct action set can be selected, a policy can often be learned even before the complete reward structure has been experienced. As a consequence the multi-phase partitioning technique promises to provide significant improvement in learning performance as compared to ϵ-reduction or flat state space techniques by providing a compact state space which encodes important functional aspects of the action set.

4.2.2 Example

In this example we assume a grid world with a mobile robot which can perform four primitive deterministic actions: LEFT, RIGHT, UP and DOWN. Rewards for actions that lead the agent to another cell are assumed to be -1 and the state marked with * has a reward of 20. The goal of the agent is to find the state marked with *. The bold bars in the grid world represent obstacles for the robot. In order to construct an option we define a policy with each action. The termination condition is hitting the wall and the option repeats each action until it terminates. Figure 4.3 shows this scenario. Furthermore, let $\delta = 1$ and $\epsilon = 20$ for the multi-phase partitioning using the four options.

Figure 4.4 shows the possible partitions for the four options. Each partition in these

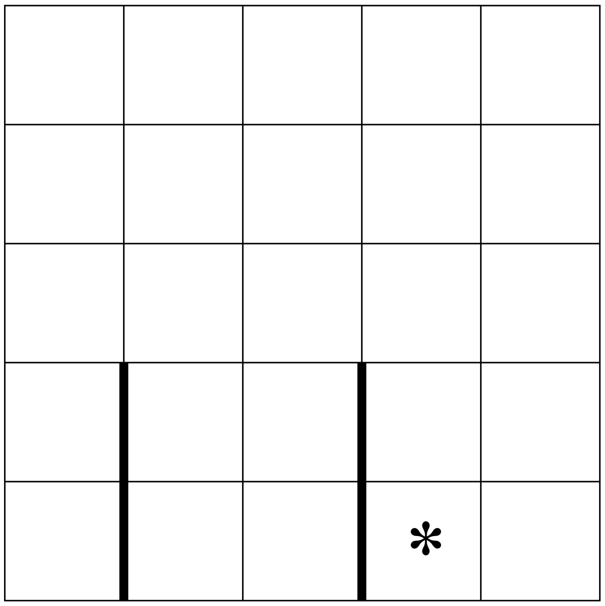


Figure 4.3. Grid world with a mobile robot which can perform four primitive deterministic actions: LEFT, RIGHT, UP and DOWN. Rewards for actions that lead the agent to another cell are assumed to be -1 and the state marked with * has a reward of 20. The goal of the agent is to find the state marked with *. The bold bars in the grid world represent obstacles for the robot. In order to construct an option we define a policy with each action. The termination condition is hitting the wall and the option repeats each action until it terminates.

figures is divided only in two blocks as all the states satisfy the probability criterion but state * is different from the other states in terms of the reward criterion. Let B_i^j be block j for partition i derived by option o_i.

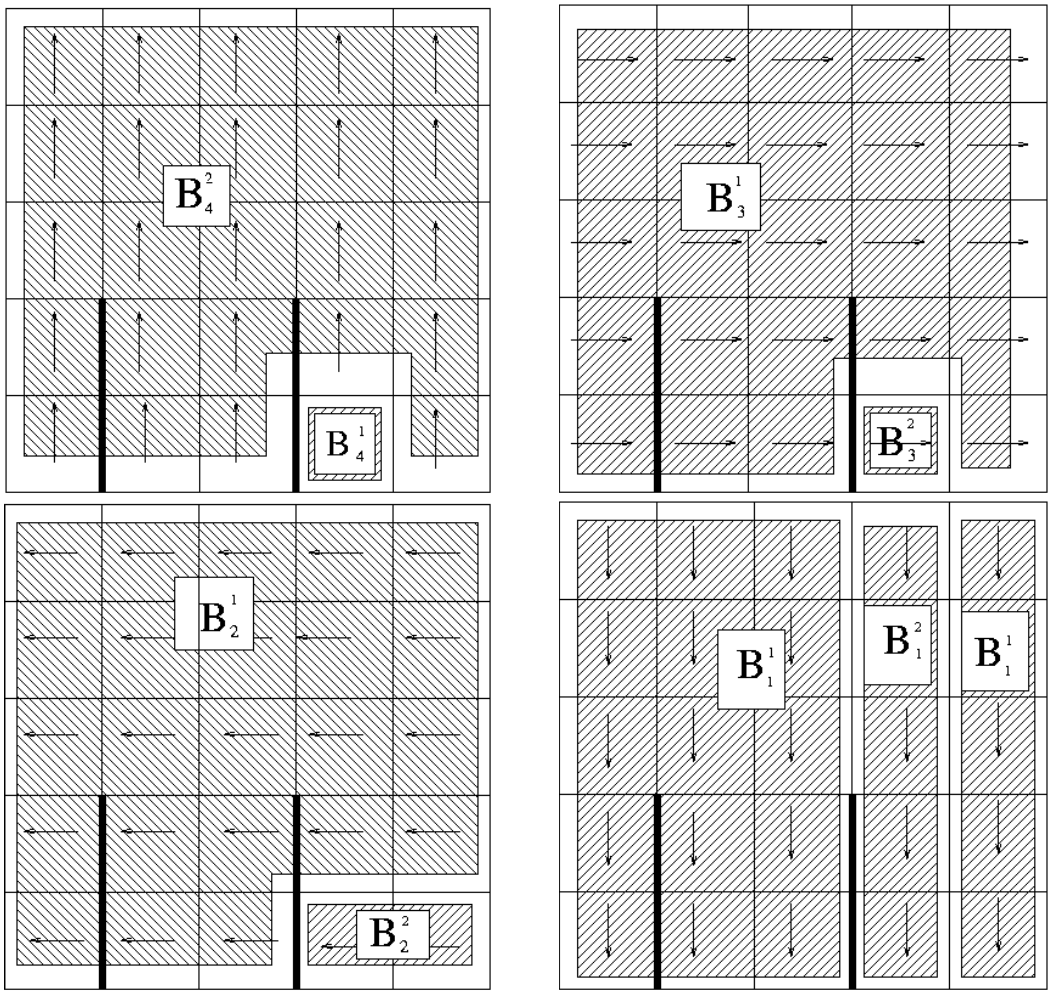


Figure 4.4. Blocks for options *UP, RIGHT, LEFT* and *DOWN*. Each partition is divided only in two blocks as all states satisfy the probability criterion, but state * is different from the other states in terms of the reward criterion.

These blocks can be combined by intersecting them in order to derive a partition that consists of new blocks that are defined for each option. For example:

$$\begin{aligned} B_1 &= B_1^1 \cap B_4^1 \cap B_2^1 \cap B_3^1 \\ B_2 &= B_1^2 \cap B_4^1 \cap B_2^1 \cap B_3^1 \end{aligned} \tag{4.9}$$

Figure 4.5(a) illustrates the intersection of the partitions. These blocks form the initial blocks for the reduction technique. The result of refinement is illustrated in Figure 4.5(b). While performing an action on each state the result would be another block instead of a state so each block of Figure 4.5(b) can be considered a single state in the resulting state space. Now if the agent is in block B_1 and executes the action RIGHT the agent will arrive in a state in block B_6.

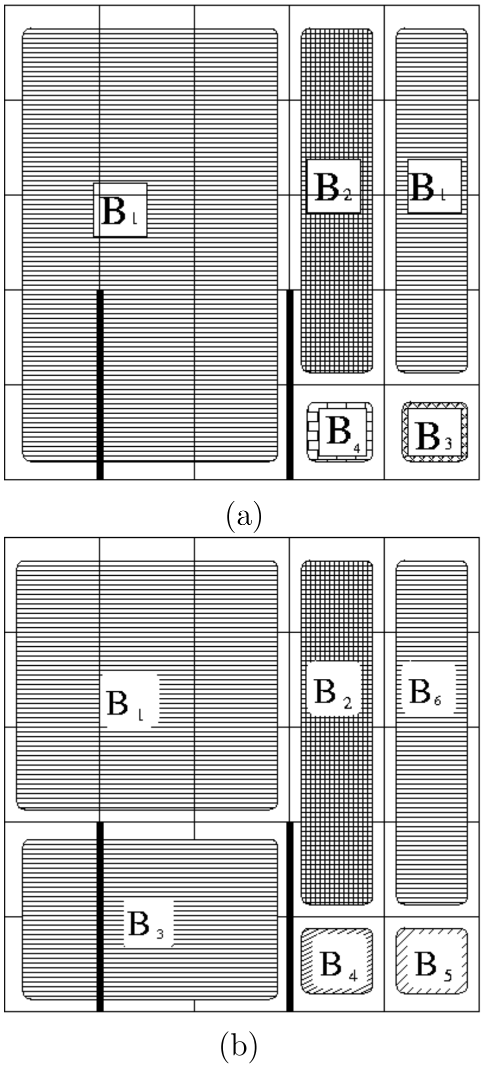


Figure 4.5. (a) Intersection of blocks (b) Blocks of partition after final refinement.

4.3 Learning by Computing Reward Variation

While the theorem in Section 4.2.1.3 shows that under certain conditions a new task is learnable on an abstract partition space with a reduced, higher-level action set, its conditions require that the goal of the new task is addressable by the selected actions and that the actions are sufficient to construct a policy. Although this would be easy to achieve by simply including all primitive actions, this would dramatically reduce the abstraction capabilities of the partitioning method and likely result in a partition where blocks correspond largely to single states or very small sets of states. As a result, the state space partitioning would likely yield only a very limited preformance gain.
To address this, this section proposes two methods aimed at making the goal achievable in the partition space generated by the multi-phase method even without all primitive actions, and to speed up the learning task on the partition space. The first method is proposing to do so by adding additional subgoals before performing the partitioning and the second method is to refine the action space after performing the two phase partitioning.

4.3.1 Subgoals and Rewards

The potential candidates for subgoals in the presented method are those states whose ratio of predecessor counts are greater than a user defined threshold. However, it is possible that this method does not find all necessary subgoals. In particular, in order to learn the task after partitioning, the goal states must belong to the set of all subgoals. This requirement may not be met by the autonomous subgoal discovery and hence additional observations need to be taken. First we consider that the task is known before learning it. An example of this kind of task is when a robot arm should pick up an object. In this situation the goal state is known and can be added to the set of subgoals. The second case is when the task is not known a priori and the task of the agent is to maximize its expected reward over time. In this situation the reward structure plays an important role and we need to look at the reward variations in order to add states with potentially high reward as subgoals. The reason for this consideration is the fact that the agent needs to have a defined option for capturing all states with high rewards in order to maximize its overall expected reward.

Let $\check{R} = \{R(s, a) | s \in S,\ a \in A\}$, and $\check{r} = \min \check{R}$, then for each state s and for all actions $a \in A(s)$, if $R(s, a) > \check{r}$ then state s needs to be considered as a subgoal. In this way, there exists a path from every state to those states that have a reward larger than the minimum reward. Note that, even though this method adds more subgoals to the subgoals found be the autonomous subgoal discovery method, it assures the reachability of the goal (i.e. maximizing the expected reward). The blocks of this partition are such that all subgoals including the goal state are reachable as stated in Theorem 4.4.

4.3.2 Refining the Action Space After Partitioning

Let $P = \{B_1, \ldots, B_n\}$ be the partition space constructed with the multi-phase method, then the action space available for P consists of A, all original actions available on the original MDP M, and the multiple step actions $o_1, \ldots, o_k$, the options constructed according to the extracted subgoals found by the autonomous subgoal discovery method. In order to speed up the learning process we need to ignore the unnecessary actions (if possible). Consider a block B_i which neither consists of a goal state nor any state with a noticeable reward. If we ignore the underlying actions in this block then the agent does not need to search B_i and the agent can jump from B_i to a block B_j with available options. This can be done using the minimum reward value described in the previous section.

Let $\check{R} = \{R(s, a) | s \in S,\ a \in A\}$, and $\check{r} = \min \check{R}$. If for each block B_i, state $s \in B_i$, and action $a \in A$, the reward $R(s, a) \leq \check{r}$ then all the underlying actions in B_i can be removed and only multiple step actions need to be considered in B_i. The reason for this is that executing underlying actions in B_i neither helps finding a goal state nor makes a significant change in maximizing the reward.

The main advantage of this method is that, given that the reward information is available, construction of the refinement is simple. On the other hand, use of this reward criterion can lead to a significant increase in the state space even when not required because refinement will happen even in situations where the reward state causing a split does not actually contribute to the final policy that is learned.

4.3.3 Learning by Solving Inconsistencies

Action dependent partitioning builds a representation that forms a new level in the state space hierarchy which serves as the initial representation for new learning tasks. In this section we introduce a method to ensure that tasks are learnable in this initial representation. The method described in this section simultaneously builds value functions on the flat MDP and the abstract representation derived using multi-phase partitioning (and thus at different levels of the hierarchy) which are then monitored for inconsistencies that identify actions that should contribute to the final policy and thus be used to refine relevant portions of the abstract state space.

Let $P = \{B_1, \ldots, B_n\}$ be a partition for state space S derived by the action-dependent partitioning method using subgoals $\{s_1, \ldots, s_k\}$ and let $\{o_1, \ldots, o_k\}$ be the options to these subgoals. If the goal state G of the task belongs to the set of subgoals $\{s_1, \ldots, s_k\}$ and if task G is achievable on the flat MDP using only the options, then G is achievable by options $\{o_1, \ldots, o_k\}$ on the abstract state space partition, and the task is learnable according to Theorem 4.4. However, if $G \notin \{s_1, \ldots, s_k\}$ then the task may not be solvable using only the options that terminate at subgoals. The proposed approach solves this problem by maintaining a separate value function for the core MDP while learning a new task on the partition space derived from only the subgoal options. During learning the agent has access to the original actions as well as to all options, but makes decisions only based on the abstract partition space information.

While the agent tries to solve the task on the abstract partition space, it computes the difference in Q-values between the best actions in the current state in the abstract state space and in the original state space. If the difference is larger than a constant value (given by Theorem 4.5, described below), then there is a significant difference between different states underlying the particular block that was not captured by the subgoal options. This approach is similar to McCallum's utile distinction memory [21] which combines Hidden Markov Models (HMMs) and Q-learning to solve tasks with only a few fields by splitting "inconsistent" HMM states whenever the agent fails to predict their utilities. In the presented learning method, a similar idea is used which determines inconsistent model states based on Q-value inconsistencies.

Theorem 4.5 [18] shows that if blocks are stable with respect to all actions, the difference between the Q-values in the partition space and in the original state space is bounded by a constant value.

Theorem 4.5 [18]: *Given an MDP $M = (S, A, T, R)$ and a partition P of the state space M_P , the optimal value function of M given as V^* and the optimal value function of M_P given as V_P^* satisfy the bound on the distance*

$$\| V^* - V_P^* \|_\infty \leq 2 \left(1 + \frac{\gamma}{1-\gamma} \max\{\epsilon, \delta\} \right)$$

This theorem ensures bounded optimality of the policy learned on the partition space if all required actions are available during refinement, but also provides a method to detect situations in which this condition is not fulfilled.

Let $M = (S, A, P, R)$ be a MDP and M_P be the corresponding reduced MDP with $P = \{B_1, \ldots, B_n\}$, where B_i, $1 \leq i \leq n$, are δ-stable (and they are possibly stable according to Inequality 4.7). This method maintains two separate tables for computing V and V_P, where V_P is the value function for M_P. If $V(s) - V_P(s) > 2(1 + \frac{\gamma}{1-\gamma} \max\{\epsilon, \delta\})$ for $s \in B_i$ then the primitive action that achieves the highest value on the original state in the MDP will be added to the action space of the states in block B_i, i.e.

$$A_{B_i} \leftarrow A_{B_i} \cup \{a | \max_a V(s)\} \quad \forall s \in B_i$$

and block B_i is refined according to Inequalities 4.6 and 4.7 until it is stable for the new action set. Once no such significant difference exists, the goal will be achievable in the resulting state space according to Theorem 4.4 and the learned policy will be within the abovelisted bound of optimality. This procedure is illustrated in Figure 4.6.

While this method, as opposed to the one presented in Section 4.3.2, has to maintain a

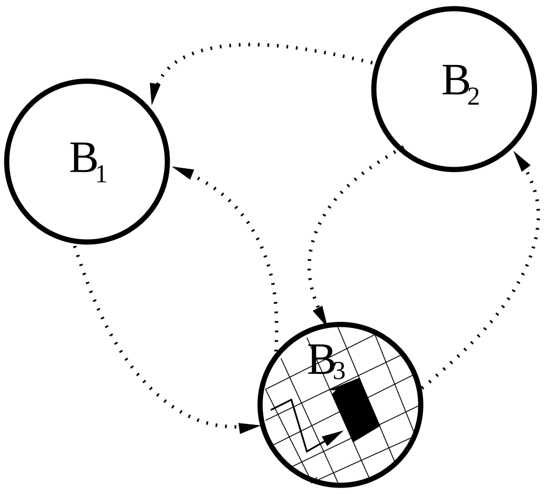


Figure 4.6. An abstract state space with 3 blocks (B_1, B_2, B_3). Options are shown using dotted curves and original actions are illustrated with solid lines. The black cell in block B_3 is the goal state. Since the Q-value in block B_3 is significantly smaller than the one of the underlying goal state this block is refined using the options and the primitive actions.

second, substantially larger value function, it will perform state space splits only if the existing actions are not sufficient to represent a bounded optimal policy. As a result, this method frequently results in a significantly smaller final state space partition. In addition, since learning is driven by the abstract state space partition (the decision layer), learning time is largely determined by the complexity of this representation, promising larger performance gains in terms of learning time.

4.4 Autonomous Hierarchy Construction

In the multi-phase partitioning and hierarchical learning method discussed in the previous section, it has so far been assumed that either the correct set of actions for constructing an abstract state space is available or that, as a simple heuristic, all subgoal options are selected as the relevant action set. While the latter can lead to good results when used in conjunction with the learning method in Section 4.4, it might lead to an ever increasing action set if a large sequence of tasks is to be learned. In particular, this heuristic has the limitation that it can never remove an option from the action set used for multi-phase partitioning, even if it is not used for any of the tasks. To address this limitation, this section presents a method aimed at automatically constructing the abstract representation based on the information contained in the previously learned task policies.

In order to estimate the structure of the state space for learning future tasks, we construct the decision layer here based on an estimate of the expected time to learn a new task according to previously learned tasks. Let $\Pi = \{\pi_1, \ldots, \pi_n\}$ be the set of previously learned polices and $P_i = \{B_{i,1}, \ldots, B_{i,n}\}$ be the corresponding partitions. Also let the triple $T_i = (\pi_i, P_i, Q_i)$ be a task on partition $P_i = \{B_{i,1}, \ldots, B_{i,n}\}$ with policy π_i and the Q-function Q_i. The expected number of experiences required to learn a task, with high probability, on partition P with action set O using a DP-based version of Q-learning is [33]:

$$T_{conv}(P,O) = c|P|^2|O|$$

where c is a constant and it is assumed that the task is learnable on P with action set O.

The expected time required to learn task T_i on state representation P (including the refinement process) can be obtained by calculating the number of experiences that are needed for learning T_i on partition P plus the amount of time that is need to refine a block P, that is:

$$E[t_{T_i}|P] = t_{conv}(P,O) + \sum_{B_j \in P} P_{refine}(B_j|T_i) t_{conv}(\{B_{i,k}|B_{i,k} \cap B_j \neq \emptyset\}, O)$$

We compute the likelihood that a block B_j has to be refined during the exploration and learning of task T_i with the following equation:

$$P_{refine}(B_j|T_i) = \sum_{B_{i,k}:B_{i,k} \cap B_j \neq \emptyset} P_{refine}(B_j|B_{i,k}, T_i) P(B_{i,k}|T_i)$$

where

$$P_{refine}(B_j|B_{i,k}, T_i) = \begin{cases} 1 & \text{if } max_a(Q_i(B_{i,k}, a) > max_{a \in O_{B_{i,k}}}(Q_i(B_{i,k}, a))) + L \\ 0 & \text{otherwise} \end{cases}$$

where $L = 2(1 + (\frac{\gamma}{1-\gamma})) \max\{\epsilon, \delta\}$ and $P_{refine}(B_j|B_k^i, T_i)$ is the probability that block B_j has to be refined during the exploration and learning of T_i due to encountering block $B_{i,k}$ which is at least partially contained in B_j and for which an action a which is not contained in the currently considered action set $O_{B_{i,k}}$, with significantly higher value should then be included using the hierarchical learning scheme described in Section 4.4.

We compute the expected time required to learn a task randomly chosen from the distribution of previously learned tasks according to an importance distribution $U(T_i)$ which indicates the weight that should be put on each tasks by:

$$E[t_{learn}|P] = \sum_i \frac{U(T_i)}{\sum_i U(T_i)} E[t_{T_i}|P]$$

Algorithm 3 illustrates the process of autonomous hierarchy construction, in particular this is a greedy algorithm that finds action-dependent partitions that have the smallest expected learning time given previously learned tasks. The reason for the greedy approach is to reduce the complexity sufficiently to make it tractable. This approach is very similar to McCullum's U-tree algorithm [22] except that splits are driven not by reward but by the expected learning time metric derived before. This procedure can be done either by splitting the blocks separately or by limiting the inclusion of actions across the state space. While the latter saves us more computational time, the former will give us more nuanced splits.

Algorithm 3 Autonomous Hierarchy Construction

```
Require: $O_0 = \emptyset$, $P_0 = \{s\}$
  $n = 0$
  repeat
    for all $B_j$ in $P_n$ and $o_i \in O - O_{n,B_j}$ do
      $P_{n+1,(i,j)} = P_n$ where $B_j$ is refined with $o_i$
    end for
    $(k,l) = argmin_{(b,c)} E[t_{learn} | P_{n+1,(b,c)}]$
    $P_{n+1} = P_{n+1,(k,l)}$
    $B = B_l$
    for all $B_i \in P_n$ do
      for all $B_j \in P_{n+1}, B_j \subseteq B_i$ do
        if $B_i = B$ then
          $O_{n+1,B_j} = O_{n,B} \cup \{o_k\}$
        else
          $O_{n+1,B_j} = O_{n,B_i}$
        end if
      end for
    end for
    $n = n + 1$
  until $E[t_{learn} | P_n] \geq E[t_{learn} | P_{n-1}]$
  return $P_{n-1}$
  END
```

CHAPTER 5

EMPIRICAL RESULTS

This section describes experimental results in stochastic and deterministic domains. The first experiment has been performed on a grid world in order to investigate the result of subgoal discovery and action-dependent partitioning in deterministic domains.
The second experiment shows the evaluation of action-dependent partitioning on a stochastic domain and shows how an agent can use information acquired while learning one task to discover subgoals for similar tasks. The agent is able to transfer knowledge to subsequent tasks and to accelerate learning by creating useful new subgoals and by off-line learning of corresponding subtask policies as abstract actions (options). At the same time, the subgoal actions are used to construct a more abstract state representation using action-dependent state space partitioning.
The third experiment shows the result of the presented approach in a game domain that is more complex and more similar to real environments. While all these experiments use the heuristic of using all subgoal actions to construct the abstract decision layer, the fourth experiment in the same game domain investigates the autonomous hierarchy construction approach in order to illustrate the construction of an approximate partition using the information of the previously learned polices.

5.1 Autonomous Subgoal Discovery and Hierarchical Learning

This section describes experiments for subgoal discovery and hierarchical learning in three different setups. The results of the experiments in this section show that the action-dependent partitioning method and autonomous hierarchical learning can significantly outperform the learning time on the flat state space.

5.1.1 Experiments in a Deterministic Domain

To illustrate the result of the multi-phase partitioning method using subgoal discovery, a number of experiments have been performed with randomly changing goal locations in a chosen environment. This environment consists of three connected grid worlds in order to illustrate a 3-D environment. Each of these grid worlds consists of different randomly chosen rooms. The connection between these rooms are stairways. Discovered subgoals are compared to hand-design subgoals (doorways and entrances to the hallways), in order to evaluate the efficiency of the subgoal discovery method. The hand-designed subgoals which serve also as the terminal states of the available options for the corresponding experiments are illustrated in black in Figure 5.1.
The actions for each of the subgoal states are multi-step actions which terminate when they reach the subgoal in the same room. The actions for the stairways are multi-step actions consisting of a sequence of GoUp and GoDown actions. The underlying deterministic actions are GoUp, GoDown, GoLeft and GoRight. The reward of the goal state is +100. In the following experiment, a sequence of two tasks is learned. The first task is a navigation task where the system is rewarded when it reaches a fixed location, while the second is an object retrieval and delivery task where the system is rewarded for picking up and delivering an object.

5.1.2 Subgoal Discovery and Action-Dependent Partitioning

In order to extract the subgoals in the environment illustrated in Figure 5.1, Q-learning has been used to learn a policy for a defined goal in order to compute the counts

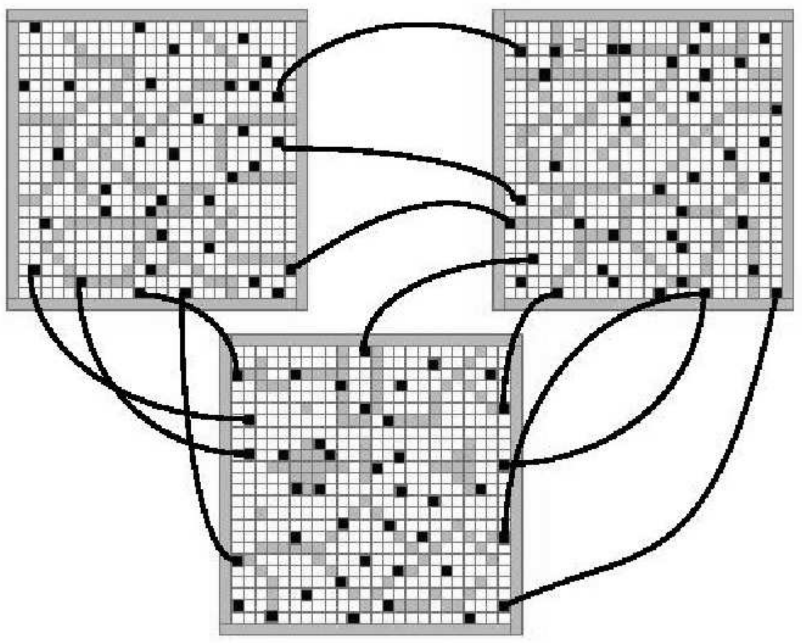


Figure 5.1. A three room environment connected by stairways (dark curves), simulating a deterministic grid world. The black cells indicate the hand-designed subgoals. The main task for the agent is to navigate this environment and find an object.

of predecessors for every state using Equation 4.1. The mean of the distribution of gradient ratios over the space is 5.427, the standard deviation is 18.09, and using the t-test and a p value of 0.025, the computed threshold is 35. Once the subgoals have been extracted using this threshold and subgoal options have been learned, this state space has been partitioned according to the multi-phase method with $\epsilon = 10$ and $\delta = 0.3$.

The number of extracted subgoals is less than the total number of hand designed subgoals and thus the number of options learned with extracted subgoals is smaller than the number of options learned using predefined subgoals.

In order to show the efficiency of the subgoal discovery method, the learning performance using the hierarchical learning approach with automatically discovered subgoals is compared to the one with the one with hand-designed subgoals.

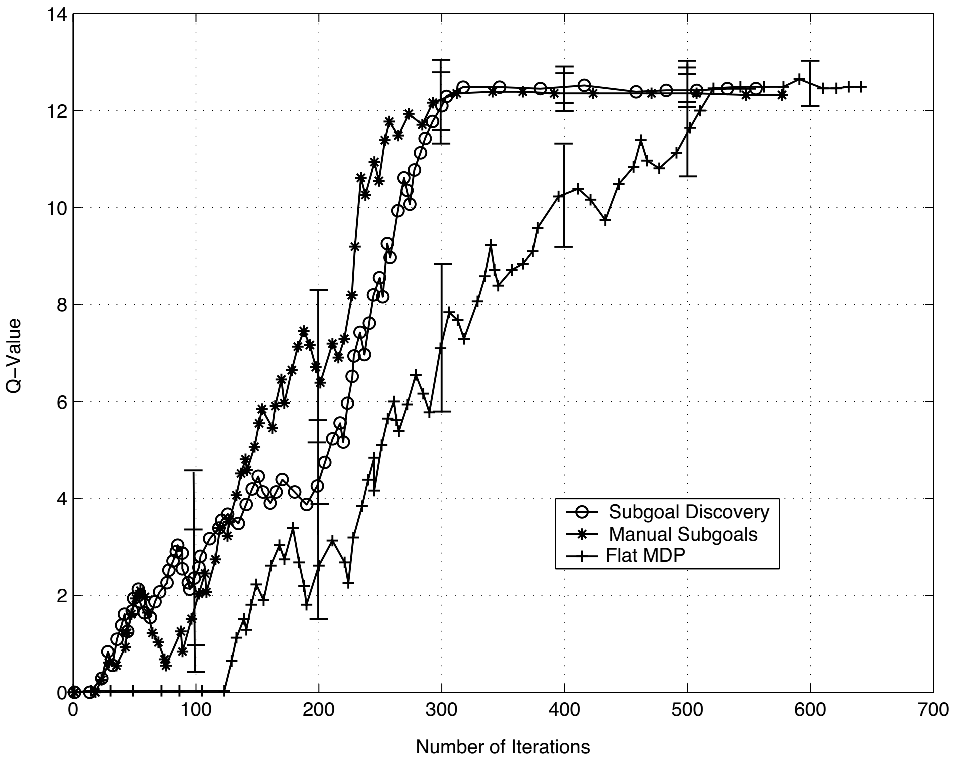


Figure 5.2. Comparison of optimal policy (policy derived from original state space and original actions) and policies derived with options to manual subgoals and discovered subgoals before partitioning the state space.

Figure 5.2 shows the difference between the average of 50 different policies computed on the original MDP and on the SMDPs constructed based on the hand designed subgoals and the subgoals extracted with the subgoal discovery method. These results show that the difference in values of policies is small and thus executing the tasks on the constructed SMDPs is almost optimal. However, the average Q-value converges in 300 iterations in the constructed SMDPs while the Q-values in the MDP converge only after 500 iterations. In addition, it can be seen that the system with automatically discovered subgoals performs almost as well as the one with hand-designed subgoals, illustrating the utility of the used subgoal criterion.

As illustrated in Figure 5.3 the performance difference between the MDP and the hierarchical systems is even more significant in the partition space where partitions are built using the multi-phase partitioning approach with hand-design subgoals or with subgoals derived with the subgoal discovery method. Much of this additional performance gain is probably due to the substantially lower number of blocks in partitions (approximately 63) as compared to the number of states in the underlying MDP (2900). As a consequence, the Q-values in partition spaces converge already after 30 iterations.

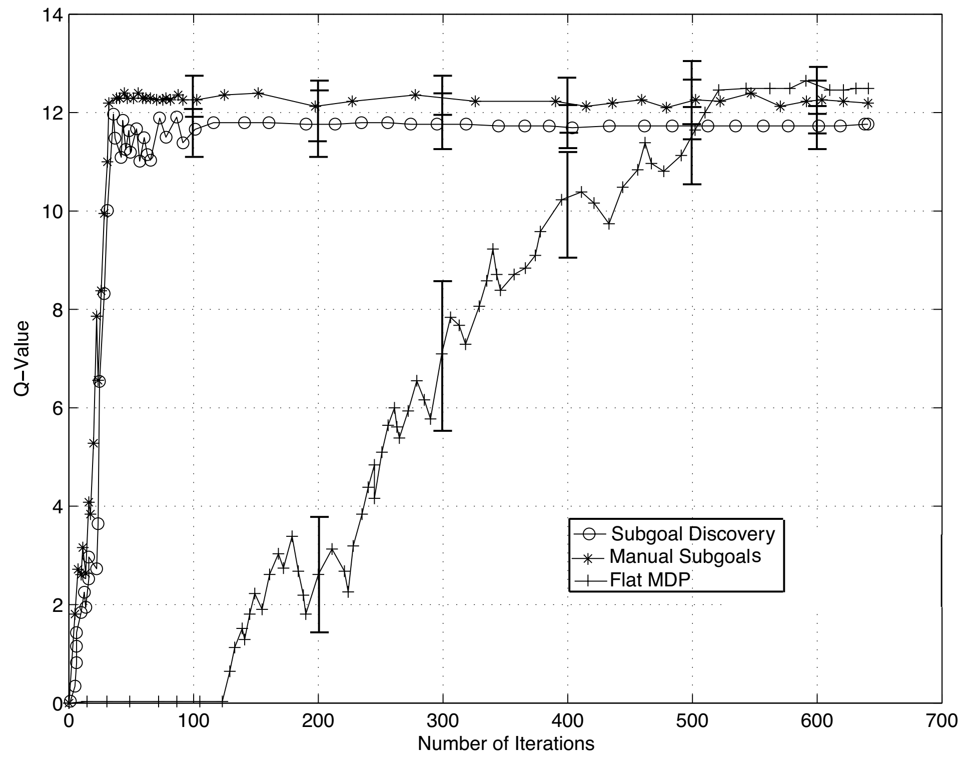


Figure 5.3. Comparison of optimal policy (policy derived from original state space and original actions) and policies derived with options to manual subgoals and discovered subgoals after partitioning the state space.

5.1.3 Subgoal Discovery Using Reward Variation and Sampling

Autonomous subgoal discovery determines subgoal states as states with local structural properties in the state space. However, as discussed previously, it is possible that this does not find all subgoals necessary to solve any new task using only the subgoal options. In particular, it does not guarantee that the goal states of new tasks belong to the set of detected subgoals and hence additional observations need to be taken. In the experiments presented in the previous section, hierarchical learning with refinement was used to address this by refining individual blocks as inconsistencies between the low-level and the high-level value function are detected. In this section the same experiments are repeated using the reward variation approach presented in Section 4.3.1 instead of the hierarchical learning and refinement technique to investigate the overhead of on-line detection of inconsistencies.

When using reward variation to ensure learnability using subgoal options on the partition space derived using multi-phase partitioning, we first consider that the task is known before learning it. Let the minimum reward be $\check{r} = 0$, and thus for each state s and for all actions $a \in A_s$, if $R(s, a) > 0$ then s will be added to the set of subgoals. In this way, there exists a path from every state to those states that have a reward greater than 0. Note that while this method adds more subgoals to the subgoals found be the autonomous subgoal discovery method, it also assures the reachability of the goal (i.e. maximizing the expected reward). The blocks of the partition are such that all subgoals, including the goal state are, reachable as stated in the Theorem 4.4.

Using these subgoals the state space is partitioned according to the procedure described in the previous section. Figure 5.4 shows the differences between the average of 50 different policies learned in the core MDP and on the SMDPs with subgoal action constructed based on the hand-designed subgoals and based on the subgoals extracted with the subgoal discovery method and the reward variation method.

These results show, similar to the experiments in the previous section, that the difference

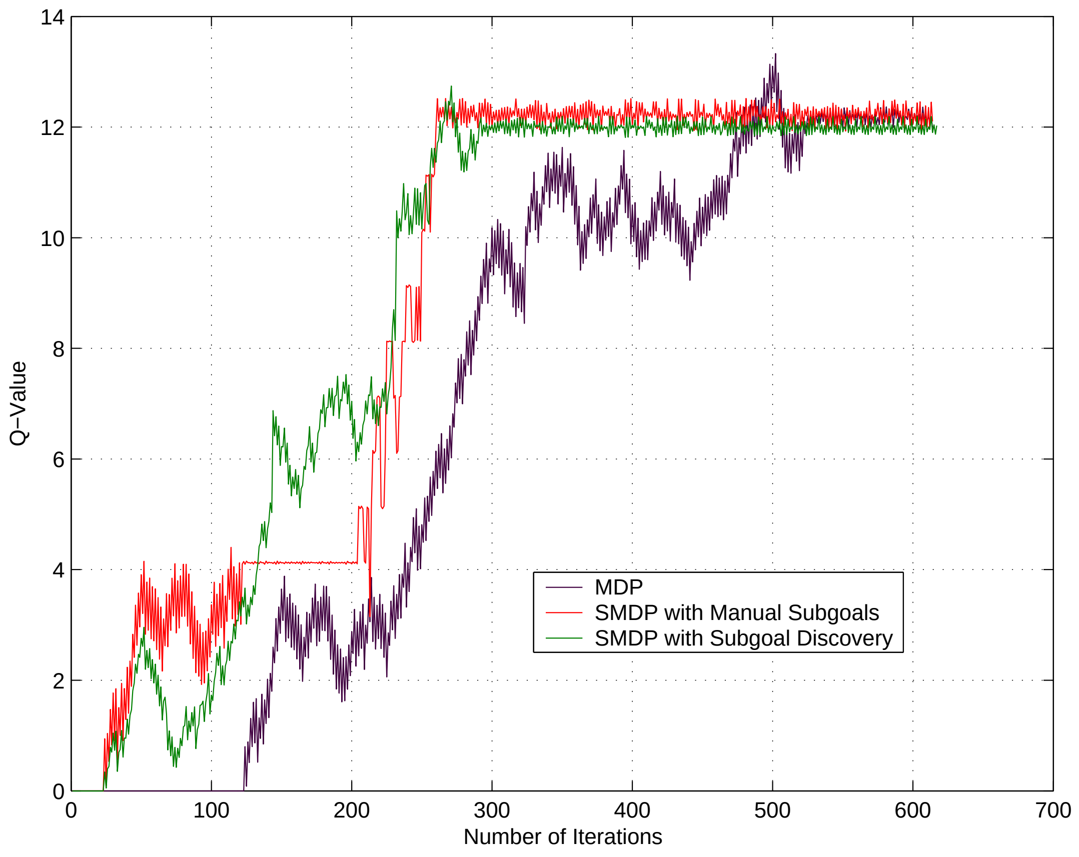


Figure 5.4. Comparison of the policies and learning time before partitioning.

in the values of the policies are small, and thus executing the tasks on the constructed SMDPs is almost optimal. However the average Q-values converge in 300 iterations on the constructed SMDPs while they require approximately 500 iterations in the original MDP. As illustrated in Figure 5.5 this difference is more significant if the partition space is used, where partitions are build from the subgoal options derived from hand-design subgoals or using the subgoal discovery and reward variation methods. As the number of blocks in each partition is substantially smaller than the number of states in the original state space, the Q-values in the partition spaces converge significantly faster.

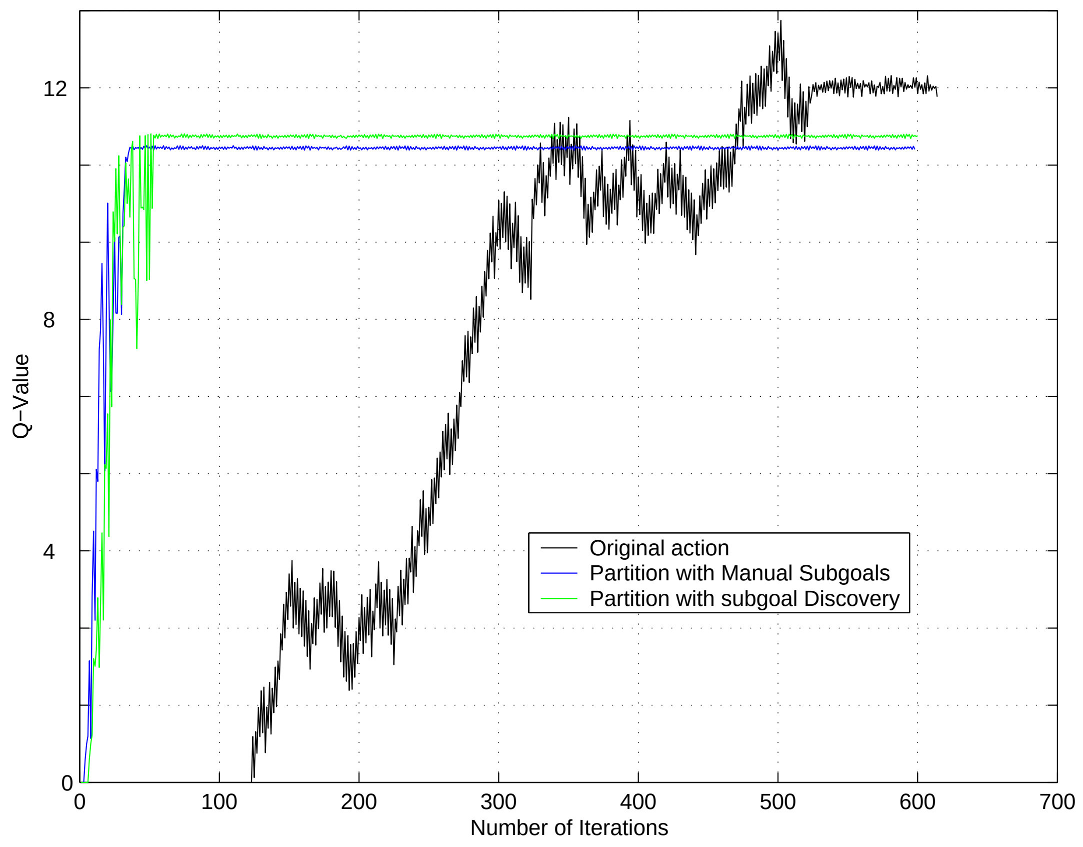


Figure 5.5. Comparison of the policies after partitioning.

Comparing the results with the ones in the previous section also shows that in this environment the hierarchical learning method with refinement and the reward variation technique perform approximately equally well. This is probably due to the fact that there are only a very limited number of reward states in this environment, allowing reward variation to achieve a very compact subgoal set.

5.1.4 Experiment in a Stochastic Domain

The main goal of this experiment is to show the potential of hierarchical learning with autonomous subgoal discovery using Monte Carlo sampling and action-dependent partitioning for accelerating learning in a stochastic environment. The main learning task of the agent is to find an object in a randomly chosen cell, to pick it up, and drop it in another randomly chosen state. Figure 5.6 illustrates the environment for this experiment. The state space, similar to the deterministic environments in the previous experiments, consists of three grid worlds that are connected through stairways (arrows). The dark cells represent obstacles and the actions are GoNorth, GoEast, GoSouth, GoWest, GoUp, GoDown, OpenArm, CloseArm, Pickup and Drop. The available actions for stairways are macro actions defined by sequences of GoUp or GoDown where the size of each sequence is 10.

The cost for each single step action is -1, and each action for navigation succeeds with

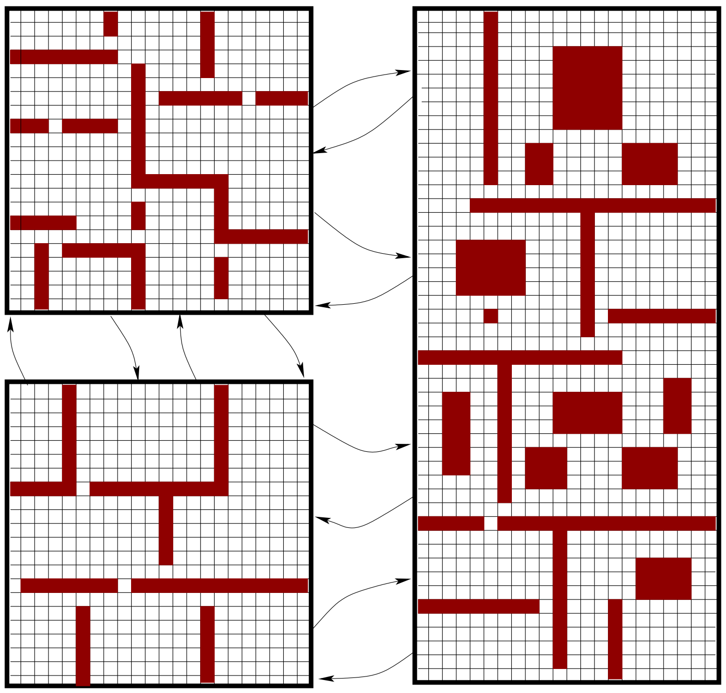


Figure 5.6. A three room environment connected by stairways (arrows), simulating a 3D grid world. The black cells indicate the obstacles. The main task for the agent is to navigate this environment and find an object. Once the agent finds the object it must pick it up, move it to another randomly chosen locations and drop it off.

probability 0.5 and with probability 0.25 causes the agent to move to the sides. Actions

OpenArm, CloseArm, Pickup and Drop always succeed with probability 1. The reward in the goal state is 100. In this experiment, the agent first learns a policy to move from a fixed starting location to a particular goal point. It then uses this policy to extract subgoals by generating random samples according to the policy. These subgoals are subsequently used to learn subgoal options which, in turn are used as abstract actions and to partition the state space using the multi-phase partitioning approach.

To evaluate the performance of the sampling-based subgoal discovery method, the count metric is first computed exhaustively, resulting in a mean of the distribution of gradient ratios over the space of 8.839 with a standard deviation of 20.12. Using the t-test criterion and a probability value of 0.025, this results in a threshold μ of 30 which is then used to extract 42 subgoal states.

In order to determine subgoal states by Monte Carlo sampling, it is important to draw the correct number of samples to assure that the resulting subgoal states will be almost the same as the ones derived using the exhaustive calculation method while minimizing the computational overhead of deriving the trajectories. One way to determine the number of sample trajectories needed would be to calculate an a priori upper bound using the result in Equation 4.3. In order to do this, a priori conservative estimates for $\max_t C^*_H(s_t)$ and for the threshold ϵ_N have to be determined. Assuming that the normal count metric is used, the a priori, worst case estimate for $\max_t C^*_H(s_t)$ would be $n|S|$, where n is the maximum length of a until it either terminates or the likelihood of its final state becomes negligible. If the assumption is made that the policy π does not create loops, then the estimate for $\max_t C^*_H(s_t)$ can be reduced to $|S|$ since each state can lie at most once on each trajectory. Using this term (in this experiment this implies $\max_t C^*_H(s_t) = |S| = 3600$) in Equation 4.3 and using $\epsilon_N = 5 \geq |C^*_H(s_t) - C^*_\pi(s_t)|$ and a probability value of 0.9, this would lead to a worst case estimate for the number of samples of

$$N \geq \frac{3600}{25} \times 2(1+5)ln\frac{2}{1-0.9} = 5176$$

However, since this a priori value does not take into account any of the characteristics of the state space and requires a guarantee of being correct within the given probability limit, this estimate generally significantly overestimates the actual number of samples required while forcing the selection of a frequently unreasonably low threshold. As a result, this a priori estimate is often not an efficient criterion to determine the number of samples.

Rather than using this a priori estimate, the same criterion from Equation 4.3 cold be used in an empirical fashion to monitor if sufficient samples have been taken. In this case, the actual value for $\max_t C_H^*(s_t)$ computed from the current samples can be used instead of the a priori estimate and sampling can continue until the inequality holds for the current number of samples and values for $\max_t C_H^*(s_t)$. Similarly, other on-line monitoring techniques can be used to observe the convergence of the count metric or of the gradient ratios to stop the sampling process.

In this experiments, the sampling process converges after collecting approximately 140 samples. At this point we compute the count metric and gradient ratio according to the collected samples and choose the threshold $\mu \geq 30 + \frac{2\times 2}{2.1} = 31.9$. Figure 5.7 illustrates the number of subgoals that are discovered using Monte Carlo sampling. As illustrated in this figure, the total number of samples that are needed to learn almost all of the subgoals is 140. Figure 5.8 shows that the total time spent on subgoal discovery for these 140 samples is less than 30 seconds, which is 5 times faster than using the entire state space for extracting the subgoals. The extracted subgoals in this experiment consist of a set of doorways, opening and closing points of stairways, and the states indicating

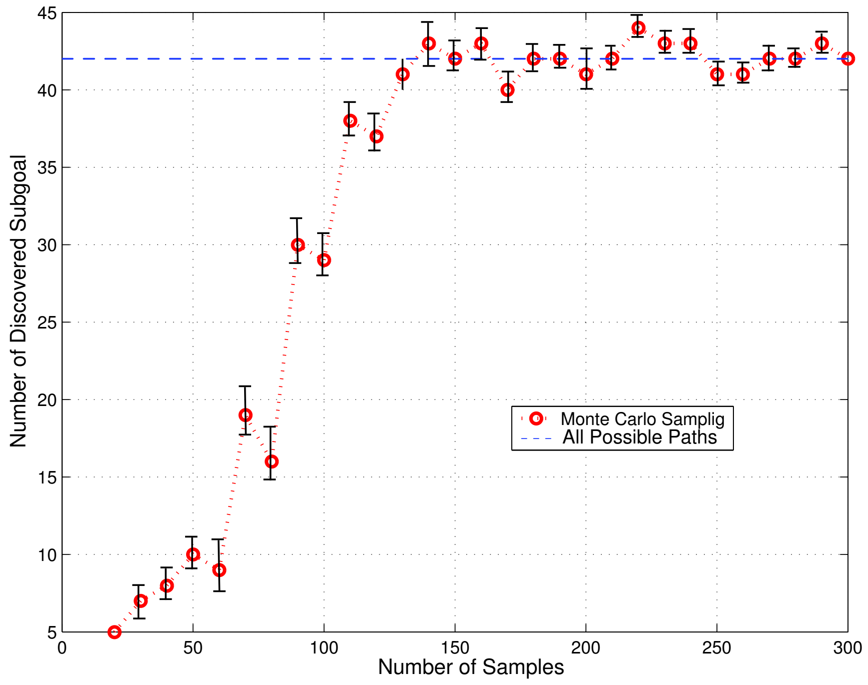


Figure 5.7. Number of samples needed to discover all useful subgoals.

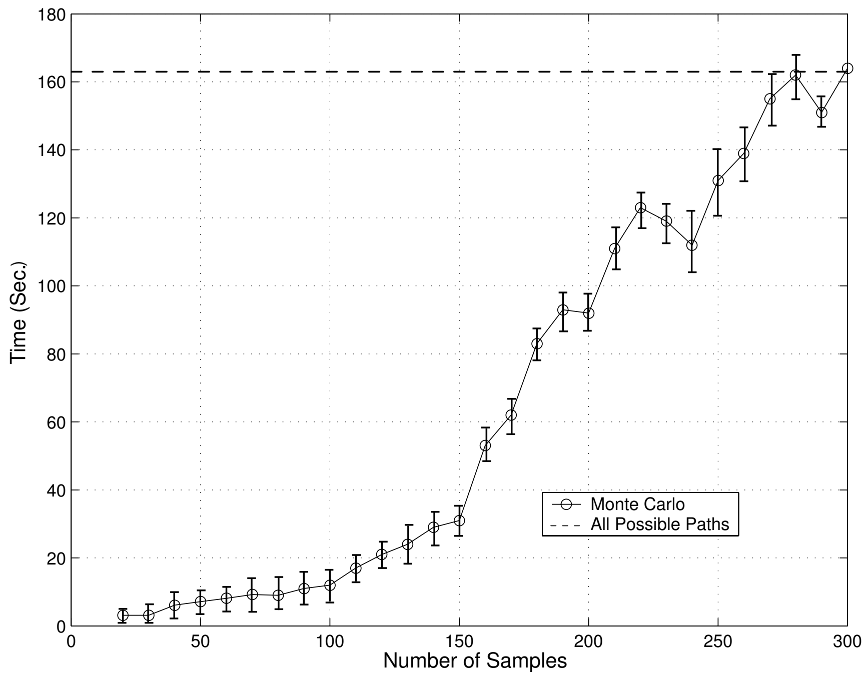


Figure 5.8. Comparison between the run times for subgoal discovery using the entire state space and Monte Carlo sampling.

that the agent is holding the object.

Figure 5.9 shows the quality of the learned policies and the acceleration of learning with and without refinement of abstract states based on Q-value inconsistencies. This

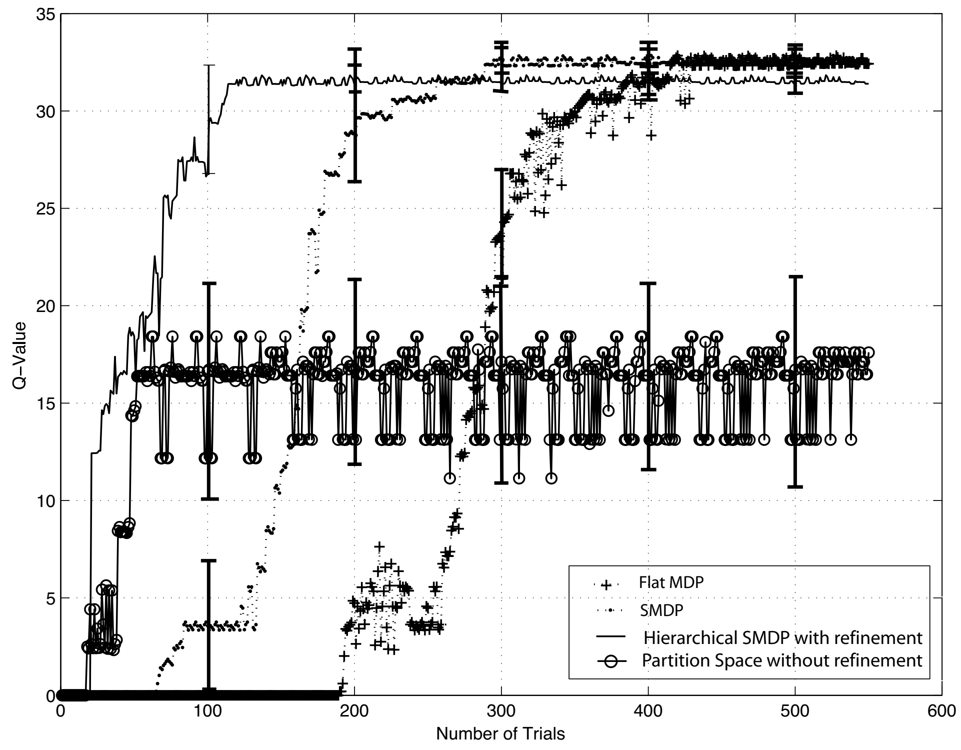


Figure 5.9. Comparison of policies derived in partition space and the original state space. The policies using the options to subgoals in the partition space converge significantly faster than the optimal policy in the original state space with only the original actions. Note that the policy in partition space is not optimal but within a fixed bound from the optimal policy.

experiment shows that when the goal state is not reachable with a subgoal option, learning is impossible without further refinement. However, according to Theorems 4.2 and 4.3, further refinement of the blocks that have inconsistencies between their flat and partition space value functions ensures the achieveablity of the goal state. The comparison of the number of states in the original state space and partition spaces is illustrated in Figure 5.10.

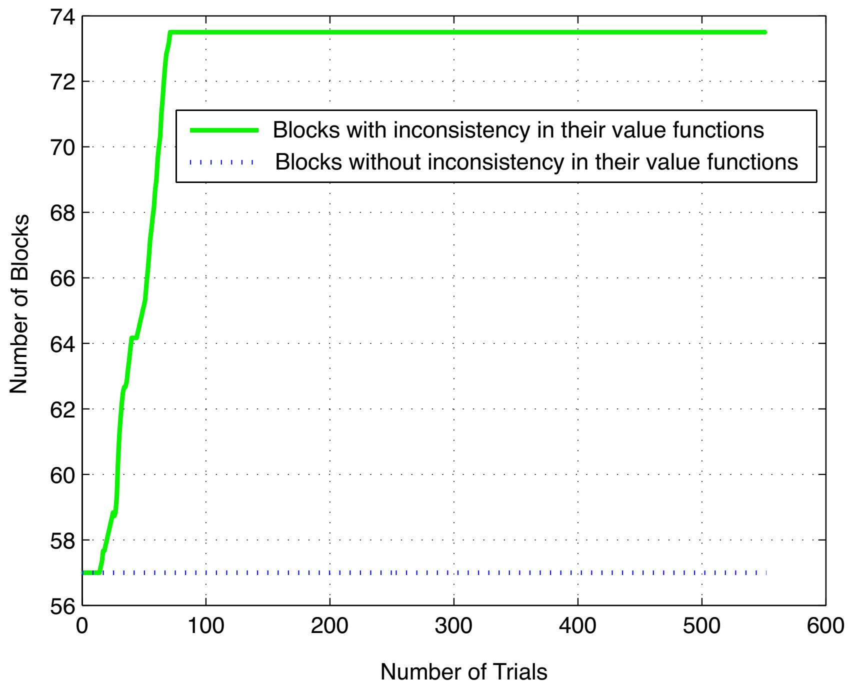


Figure 5.10. Number of states in the partition spaces constructed before and after resolving inconsistencies. When an inconsistency between the value functions of each state space is found, the blocks that cause the inconsistencies will be split according to the action that achieves the highest value in the original state and will be refined according to Criterions 4.6 and 4.7.

5.1.5 Experiment in the Game Domain

To further evaluate the approach, it has been implemented on the Urban Combat Testbed (UCT), a computer game. For the experiments presented here, the agent is given the abilities to move through the environment, shown in Figure 5.11, and to retrieve and deposit objects.

The actions are GoUp, GoDown, TurnLeft, TurnRight, PickUp and DropOff. The cost

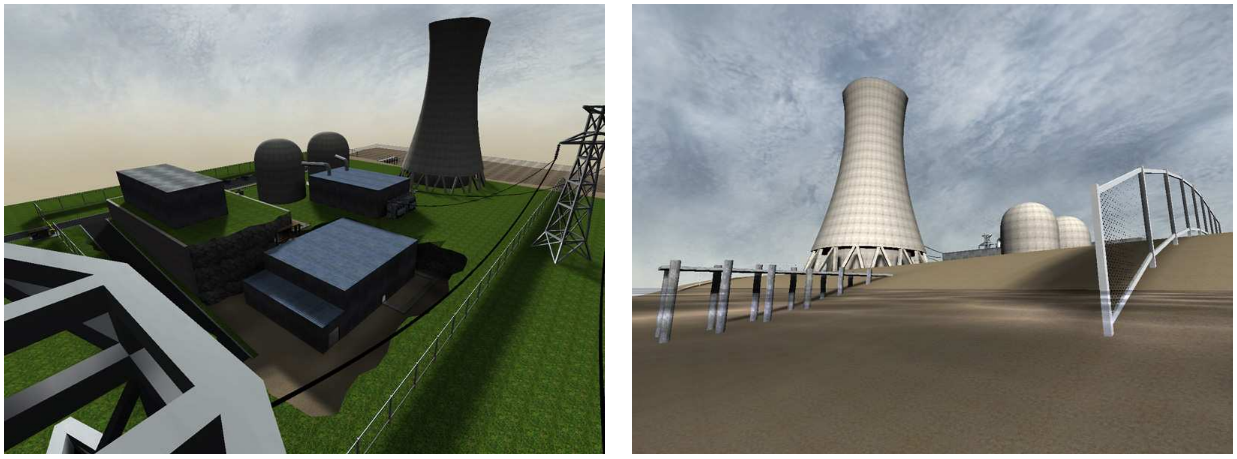


Figure 5.11. The game domain where the agent is able to navigate the environment in order to construct the action dependent partition. The agent is able to transfer the knowledge acquired from leaning a task to learn subsequent tasks.

for each single step action is -1 and each action for navigation succeeds with probability 1. The reward in the goal state where the agent can pick up and drop off the object is 100. The state is here characterized by the agent's pose as well as by a set of local object percept, resulting in an effective state space with $20,000$ states. The agent is first presented with a reward function to learn to move to a specific location. Once this task is learned, subgoals are extracted by generating random sample trajectories as shown in Figure 5.12.

As in the previous experiment, the count metric is first calculated exhaustively to provide an evaluation for the sampled subgoal discovery technique. In this case the mean of the distribution of gradient ratios over the space can be determined to be 23.934 with a standard deviation of 42.26. Using these values for the t-test and a probability value of 0.025 the threshold μ is calculated to be 40 and 29 subgoals can be found.

As described in the previous section, a conservative a priori estimate for the number of samples required for Monte Carlo-based subgoal discovery could be derived using

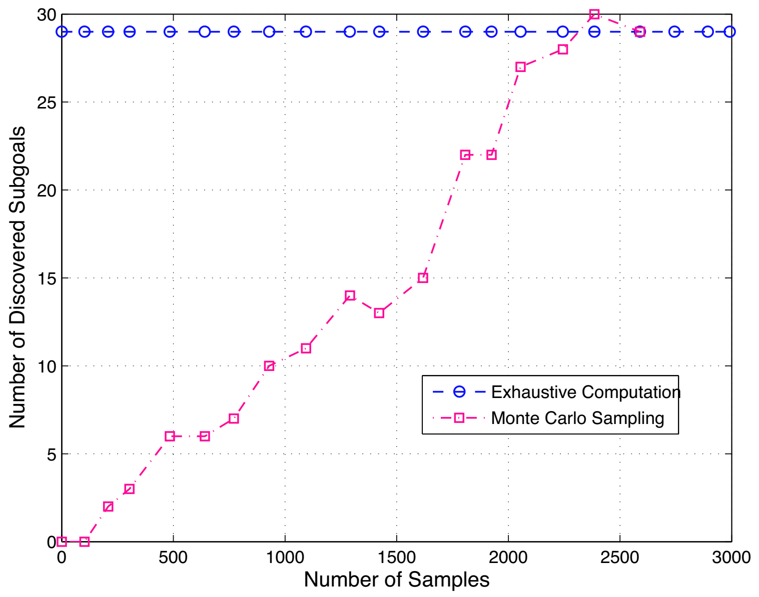


Figure 5.12. Number of samples needed to discover all useful subgoals.

Equation 4.3. In this case, let $\epsilon_N = 5$, $\max_t C^*_H(s_t) = |S| = 20000$, and the probability be 0.9. Then the worst case estimate for the number of samples could be derived as

$$N \geq \frac{20000}{25} \times 2(1+5)ln\frac{2}{1-0.9} = 28759$$

However, as previously discussed this is an overly conservative estimate since it does not make any assumptions about the environment and the sampling process. Instead, a monitoring process is used again which monitors the convergence of the sampling process. As the number of samples increases, the system identifies an increasing number of subgoals until, after approximately 2600 samples the sampling process converges (see Figure 5.12. After collecting the samples, we compute the count metric and gradient ratio according to the collected samples and we choose a threshold of $\mu \geq 40 + \frac{5\times 2}{4.6} = 42.17$ in order to select the same subgoals that are discovered by calculating the count metric exhaustively. Using these subgoal states, subgoal options are learned and the state space is partitioned using the multi-phase method with $\epsilon = 20$ and $\delta = 1$. On the resulting partition space, the main task of moving to a location, picking up the object there, and delivering it to a different location is learned using the hierarchical learning method with on-line refinement due to value function inconsistencies. Figure 5.13 shows the number of states in the partition space (the decision layer) during the learning process.

Figure 5.14 shows the quality of the learned policy and the acceleration of learning with

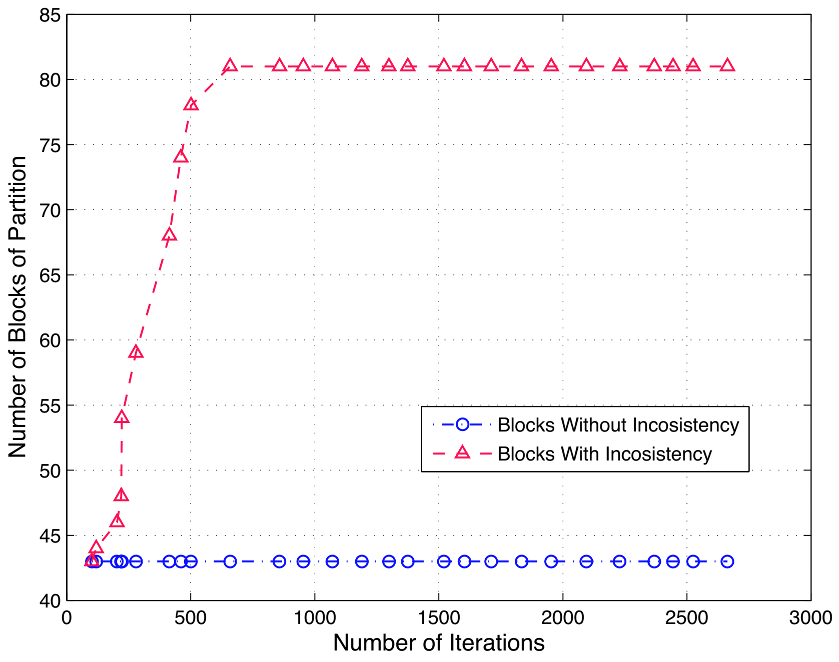


Figure 5.13. Number of Blocks during learning with refinement according to inconsistencies.

and without refinement of abstract states based on Q-value inconsistencies. These graphs show that while the use of discovered subgoal options alone within the SMDP framework can yield significant improvement in learning performance, the additional use of the multi-phase partitioning approach with its resulting reduction in the state space size used for learning leads to another significant acceleration of the learning process.

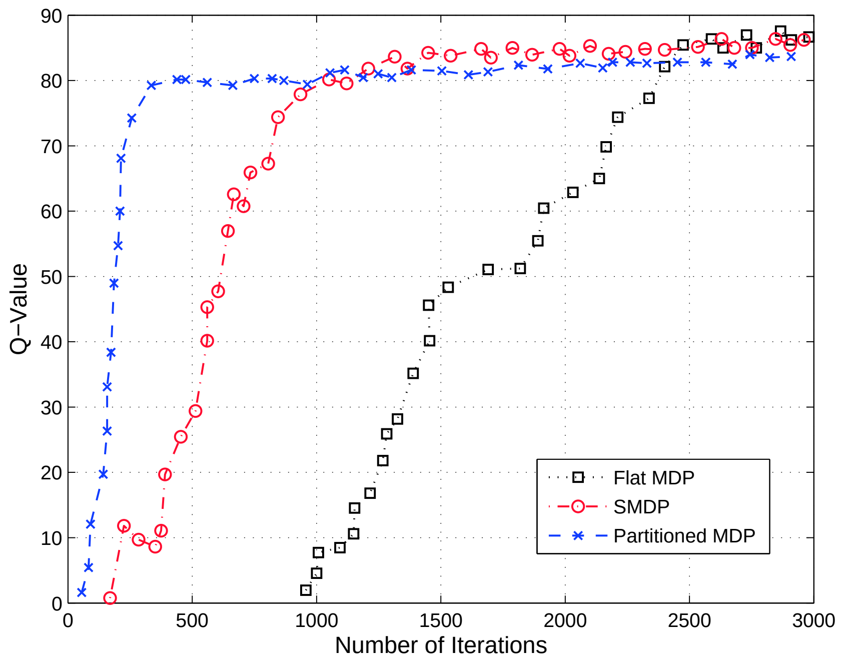


Figure 5.14. Comparison of policies for core MDP, subgoal option SMDP, and with action-dependent partitioning.

5.2 Autonomous Hierarchy Construction

In order to show the automatic construction of the decision layer, another experiment consisting of a sequence of five different tasks is learned in the same game environment used for the previous experiment. The first task is to navigate the environment, i.e. the agent learns how to move from one location to another location. The second task is to navigate the environment and pick up an object. The goal of third task is to navigate a different region of the environment, and in the fourth task the agent learns how to navigate and drop off an object in another location. The fifth task is a combination of the capabilities required for the first four tasks in that the agent has to learn to navigate the environment, to pick up an object, and to drop it off in another location.

At each step in this task sequence, all subgoal options discovered in the previously learned tasks are available to the agent and it uses Algorithm 3 described in the Section 4.3.3 to derive a partition space which attempts to minimize the expected learning and refinement time based on the previously learned policies.

Figures 5.15, 5.16, 5.17 and 5.18 show the learning performance for the second, third, fourth, and fifth task on the automatically constructed state space hierarchy compared against the learning performance on the core MDP with no knowledge transfer. Task one is not displayed here since, as the first task, it has inherently to be learned on the core MDP in the presented approach as no prior subgoals or reward information is available.

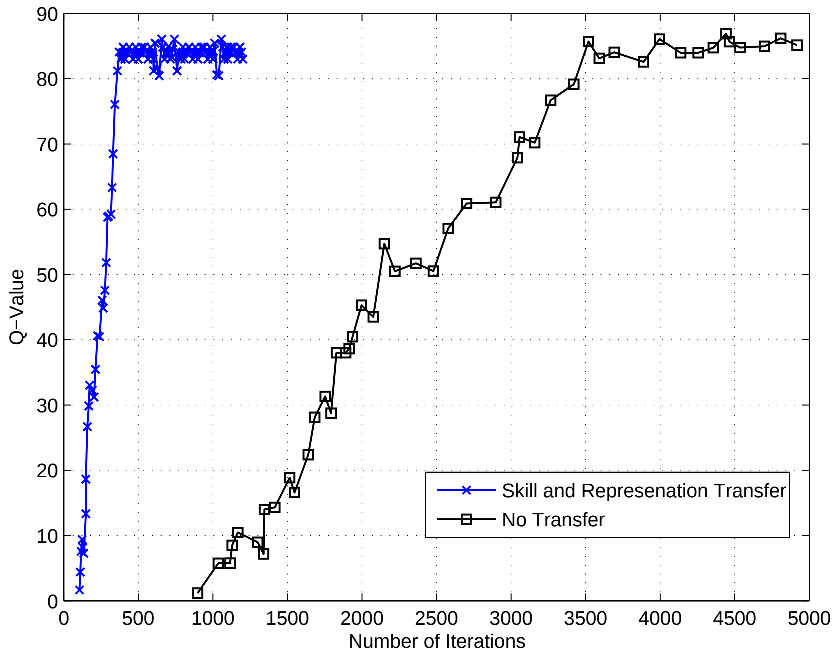


Figure 5.15. Learning curves for the first task. The agent learns to navigate the environment and the information acquired by learning this task will be used for constructing a partition for the next task ,i.e, the navigation and pickup tasks.

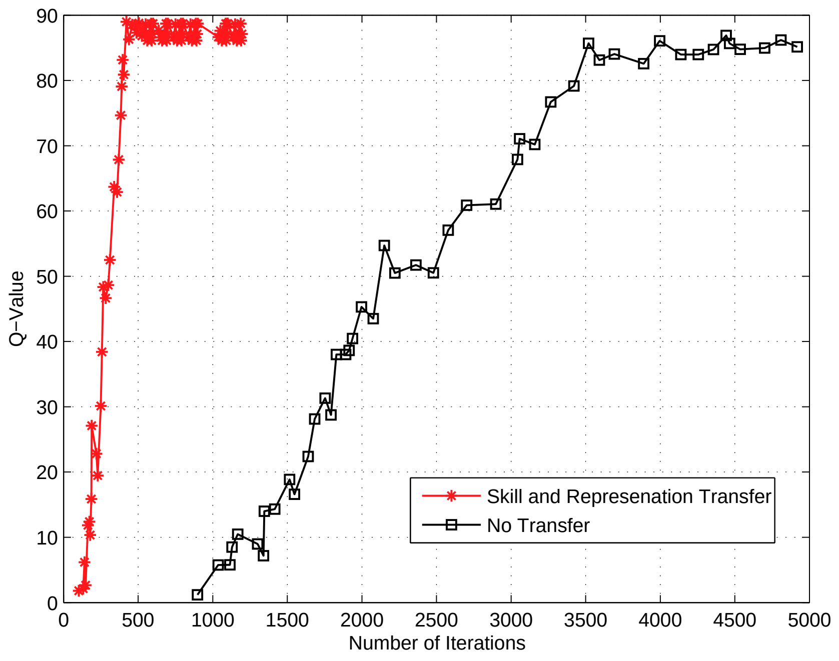


Figure 5.16. Learning curves for the Third task, i.e., the navigation and pickup tasks. The information acquired by learning this task and the first task will be used for constructing a partition for the fourth task.

Figure 5.19 shows the number of blocks in the final partition at the decision layer for tasks two, three, four, and five. As in the case of the learning curves, task one is not included here as it is learned on the core MDP and thus on a state space consisting of 20000 states. This graph shows that the number of blocks constructed to learn a new task seems initially to increase but then, as more learned tasks are available to gauge the importance of different actions (and thus state space splits), it seems to decrease although the complexity of the tasks to be learned actually increases. One possible rationale for this could be seen in the fact that the metric used to evaluate the estimated learning time is initially prone to over fitting the small number of problems available to evaluate it. As more learned tasks become available for evaluation, splits due to over fitting (i.e. ones that only match a very small subset of the tasks) will be increasingly penalized by the

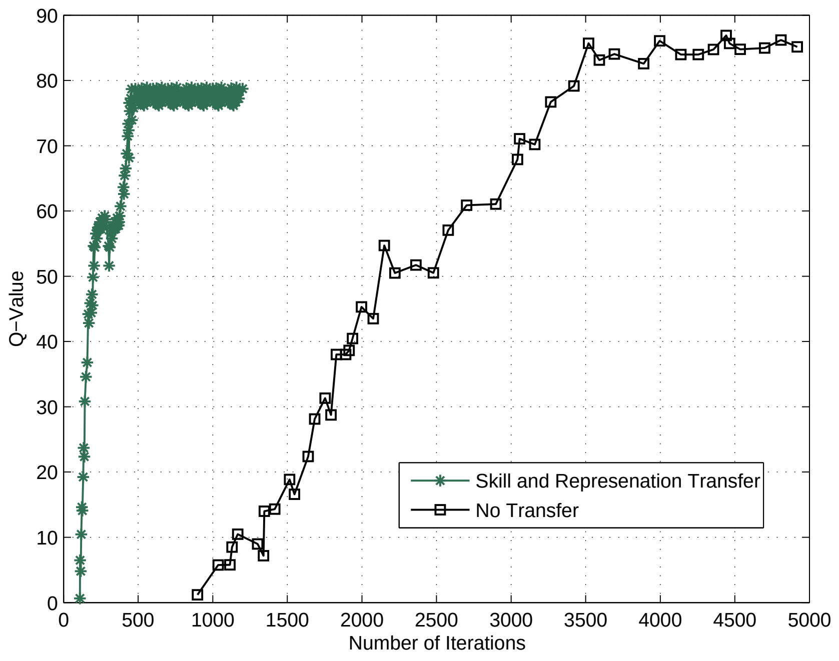


Figure 5.17. Learning curves for the fourth task, i.e., the second navigation task. The information acquired by learning this task and the previous two tasks will be used for constructing a partition for the fifth task.

metric, leading to a focusing on partitions that represent largely the common properties of larger subsets of the already learned tasks.

To analyze the efficiency of automatic hierarchy construction using estimated learning time, the partition space constructed for task five has been compared to the one resulting from multi-phase partitioning using only the available subgoal options. Figure 5.20 shows the size of the initial partition spaces for both cases as well as their development during learning and refinement. This figure shows that while the automatically constructed decision layer had significantly more blocks initially (72 versus 43 for the case of partitioning using only subgoals), the final partition size after learning is only slightly larger. This suggests that a number of the initial splits made by the automatic hierarchy construction were actually beneficial for the new learning task, indicating that the auto-

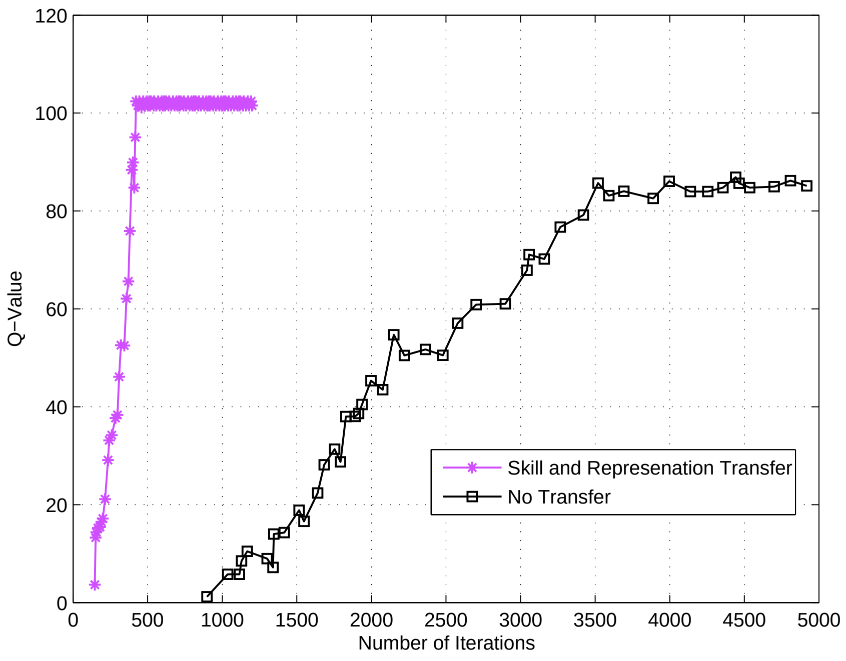


Figure 5.18. Learning Curves for the fifth task, i.e., the navigation and dropoff tasks. A new partition will be constructed by using a history of previously learned tasks for future subsequent tasks.

matic construction method can detect important aspects of the state and action space and translate them into an appropriate representation.

Figure 5.21 illustrates the learning curves comparing the performance for task five using the automatically constructed state space hierarchy and the system where the decision layer is constructed only from the available subgoal actions. These graphs shows that for this problem both methods to derive the abstract partition space lead to the same learning performance. In both cases, a policy that is within the pre-set bound of optimal is learned within approximately 450 iterations (as opposed to more than 5000 iterations when using the core MDP). This shows that the initially larger partition space of the automatic hierarchy construction approach using the estimated learning time metric does not lead to a decrease in performance. On the other hand, this problem does also not

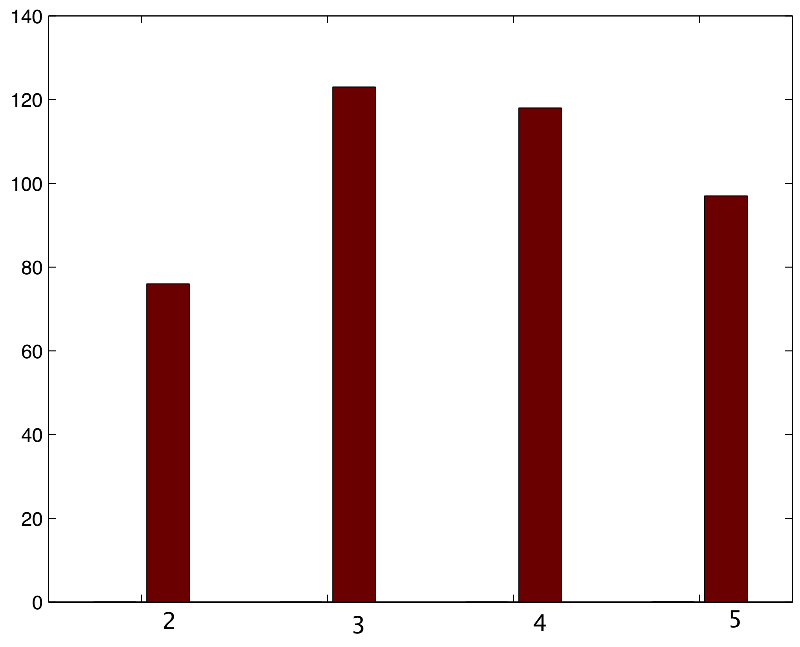


Figure 5.19. Number of blocks constructed for learning task two through five.

seem to lead to an undue burden for the method that selects all subgoal options to construct the decision layer, suggesting that in this domain the set of irrelevant subgoal options does not increase rapidly.

5.3 Summary of the Experimental Results in the Game Domain

The experiments in the game domain show the complete procedure of the research presented in this dissertation. In these experiments, the agent starts by learning an arbitrary task and uses the information acquired while learning this task to extract information for discovering useful subgoals. The agent uses the discovered subgoals in order to learn options to the discovered subgoals and to construct an initial abstract representation according to action-dependent partitioning method. In order to ensure that tasks can be learned in this abstract representation, relevant portions of the abstract

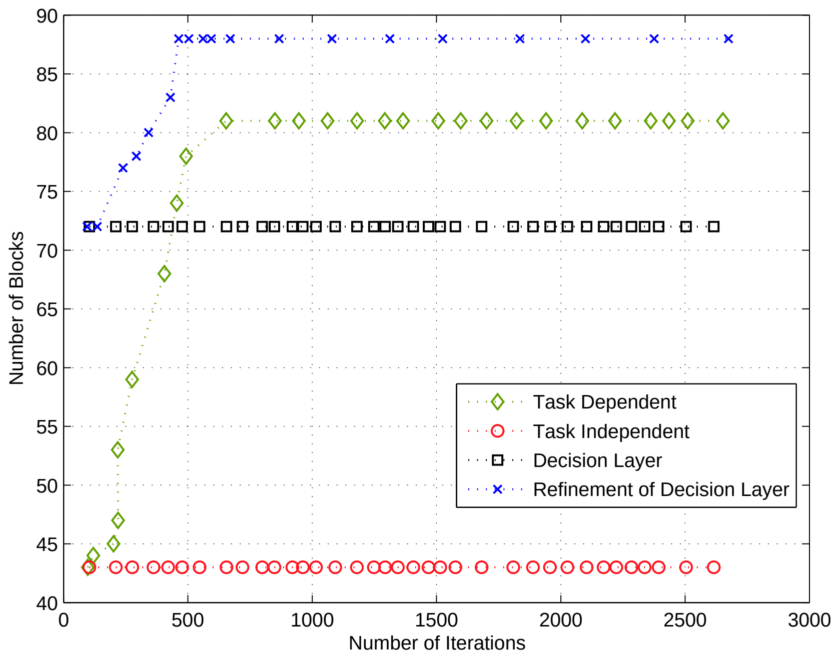


Figure 5.20. Number of blocks in the decision layer for task five before and during learning and refinement for the automatically constructed state hierarchy and for the case where only subgoal options are used to derive the initial partition.

state space are refined whenever there is an inconsistency between the value functions in flat and abstract representations.

The constructed initial abstract representation serves as a first layer of the hierarchy. In order to estimate the structure of the state space for learning future tasks, the decision layer is constructed based on an estimate of the expected time to learn a new task and the system's experience with previously learned tasks. To do so, five tasks have been learned in this domain and the policy derived by solving each task has been added to the set of policies. The previously learned polices (previous experiences) have been used to estimate the structure of the state space for learning the fifth task, resulting in a decision layer that is more relevant for the solution of subsequent tasks.

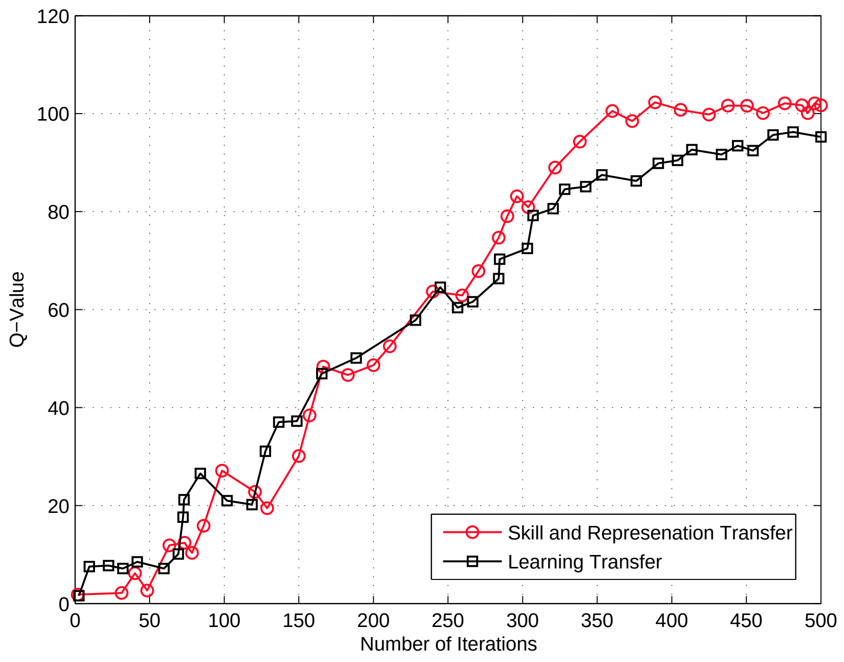


Figure 5.21. Learning on a partition space obtained by autonomous hierarchy construction method by using the first four tasks. This experiment shows how a new partition can be constructed by using a history of previously learned tasks while it ensures that the new policy is within a fixed bound from the optimal policy.

CHAPTER 6

Comparison with MAXQ Decomposition

Dietterich [11] developed an approach to hierarchical RL called the MAXQ Value Function Decomposition, which is also called the MAXQ method. Like options and HAMs, this approach relies on the theory of SMDPs. Unlike options and HAMs, however, the MAXQ approach does not rely directly on reducing the entire problem to a single SMDP. Instead, a hierarchy of SMDPs is created whose solution can be learned simultaneously. The MAXQ approach starts with a decomposition of a core MDP M into a set of subtasks $\{M_0, \ldots, M_n\}$. The subtasks form a hierarchy with M_0 being the root subtask, which means that solving M_0 solves M. Actions taken in solving M_0 consist of either executing primitive actions or policies that solve other subtasks, which can in turn invoke primitive actions or policies of other subtasks, etc.

The structure of the hierarchy is summarized in a task graph, an example of which is given in Figure 3.7 for a Taxi problem that Dietterich used as an illustration. Each episode of the overall task consists of picking up, transporting, and dropping off a passenger. The overall problem, corresponding to the root node of the graph, is decomposed into the subtask *Get*, which is the subtask of going to the passenger's location and picking them up, and the subtask *Put*, which is the subtask of going to the passenger's destination and dropping them off. These subtasks, in turn, are respectively decomposed into the primitive actions *Pickup* or *Dropoff*, which respectively pick up and drop off a passenger, and the subtask $Navigate(t)$, which consists of navigating to one of the locations indicated by the parameter t. (A subtask parameterized like this is shorthand for multiple copies of the subtask, one for each value of the parameter.) This subtask $Navigate(t)$ is decomposed into the primitive actions that are *GoNorth*, *GoSouth*, *GoEast*, or *GoWest*. The subtasks and primitive actions into which a subtask m_i is decomposed are called the "children" of M_i. An important aspect of a task graph is that the order in which a subtask's children are shown is arbitrary and the choice the higher level controller makes depends on its policy. The graph just restricts the action choices that can be made at each level.

Each subtask, M_i, consists of three components. First, it has a subtask policy, p_i, that can select other subtasks from the set of M_i 's children. Here, as with options, primitive actions are special cases of subtasks. We also assume the subtask policies are deterministic. Second, each subtask has a termination predicate that partitions the state set, s, of the core MDP into s_i, the set of active states in which M_i's policy can execute, and t_i, the set of termination states which, when entered, causes the policy to terminate. Third, each subtask m_i has a pseudo-reward function that assigns reward values to the states in t_i. The pseudo-reward function is only used during learning.

In order to compare our results with the MAXQ decomposition method we use Parr's domain [24] illustrated in Figure 6.1. This maze has a high-level structure (i.e. a series of hallways and intersections) and a low-level structure (a series of obstacles that must be avoided in order to move through the hallways and intersections). In each trial the agent starts in the top left corner, and it must move to any state in the bottom right corner room. The agent has the usual four primitive actions, *GoNorth*, *GoSouth*, *GoEast*, and *GoWest*. The actions are stochastic; with probability 0.8, they succeed, but with probability 0.1 the action will move to the left and with probability 0.1 the action will move to the right instead (e.g. a *GoNorth* action will move east with probability 0.1 and west with probability 0.1). If an action would collide with a wall or an obstacle, it has no effect.

The maze is structured as a series of rooms, each containing a 12×12 block of states (and various obstacles). Some rooms are parts of hallways because they contain walls on two opposite sides, and they are open on the other two sides. Other rooms are intersections where two or more hallways meet.

To test the representational power of the action dependent method, we want to see how

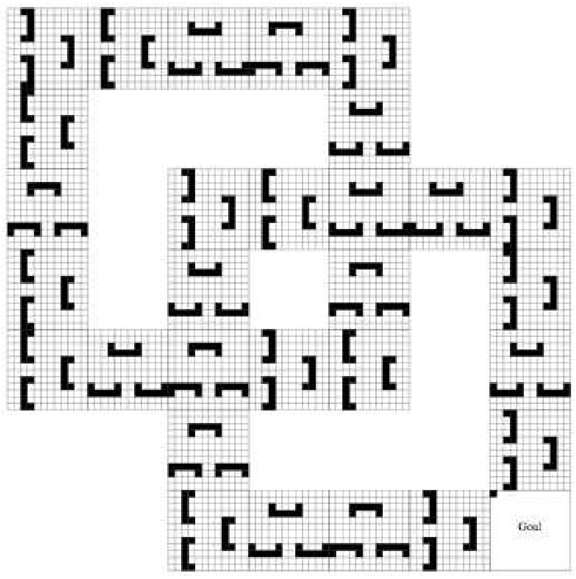


Figure 6.1. Parr's domain for comparison with MAXQ [24].

well it can represent the prior knowledge that MAXQ is able to represent. We begin by describing MAXQ for this maze task, and then we will compare it to the hierarchical action-dependent method.

The subtasks in MAXQ are hand-designed and are defined as follows:

- *Root*. It must choose a direction d and invoke *Go*. It terminates when the agent enters a terminal state. This is also its goal condition (of course).
- *Go*(d, r). The parameter r is bound to the current 12×12 room in which the agent is located. *Go* terminates when the agent enters the room at the end of the hallway in

direction d or when it leaves the desired hallway (e.g. in the wrong direction). The goal condition for Go is satisfied only if the agent reaches the desired intersection.

- $ExitInter(d, r)$. This terminates when the agent has exited room r. The goal condition is that the agent exit room r in direction d.
- $ExitHall(d, r)$. This terminates when the agent has exited the current hall (into some intersection). The goal condition is that the agent has entered the desired intersection in direction d.
- $Sniff(d, r)$. $Sniff$ has two child subtasks, $ToWall$ and $FollowWall$.
- $ToWall(d)$. This is equivalent to part of $Sniff$, and it terminates when there is a wall in front of the agent in direction d. The goal condition is the same as the termination condition.
- $FollowWall(d, p)$. It moves in direction p until the wall in direction d ends (or until it is stuck in a corner with walls in both directions d and p). The goal condition is the same as the termination condition.
- $BackOne(d, x, y)$. This moves the agent one step backwards (in the direction opposite to d. It needs the starting x and y position in order to tell when it has succeeded. It terminates if it has moved at least one unit in direction d or if there is a wall in this direction. Its goal condition is the same as its termination condition.
- $PerpThree(p, x, y)$. This moves the agent three steps in the direction p. It needs the starting x and y positions in order to tell when it has succeeded. It terminates when it has moved at least three units in the direction p or if there is a wall in that direction. The goal condition is the same as the termination condition.
- $Move(d)$. This is a parameterized primitive action. It executes one primitive move in direction d and terminates immediately.

MAXQ uses this information to learn a task as described in Section 3.6.1.

To address this experiment with the hierarchical learning approach introduced in this dissertation, the agent first learns a policy to move from a fixed starting location to a particular goal point. It then uses this policy to extract subgoals according to the autonomous, sampling-based subgoal discovery method described in Section 4.1.1. After subgoals are extracted, the approach is presented with the same learning problem as MAXQ which it addresses using the subgoal options and multi-phase partitioning with the hierarchical learning method.

After learning the first task using Q-learning, the subgoal discovery approach forst generates a set of random trajectories according to the learned policy in order to compute the count metric. Then it uses the t-test criterion to determine the significance threshold μ and uses it, together with the sampled trajectories to determine the set of subgoals. In this case, the method converges after 140 sample trajectories, leading to a distribution of gradient ratios with a mean of approximately 7.3 and a standard deviation of approximately 18.2. Using these values and a p value of 0.025, the threshold μ for the sampling-based approach is determined as 32 and the set of subgoals is extracted accordingly followed by the learning of corresponding subgoal options.

After the subgoals are extracted, the hierarchical learning technique is presented with the same learning problem as MAXQ and two experiments are performed. In the first experiment only the subgoal options are transferred to the new task and a standard SMDP policy is learned using the original state space representation. For the second experiment, the multi-phase partitioning technique is used with the set of learned subgoal options to first construct a reward-independent state space partition for the new learning task. Starting with this initial partition, the hierarchical learning approach is then used to learn a bounded optimal policy for the new task.

Figure 6.2 shows the comparison between the MAXQ decomposition and the learning of an SMDP with the sampling-base subgoal discovery but without action-dependent partitioning. This experiment illustrates that MAXQ will outperform an SMDP with options

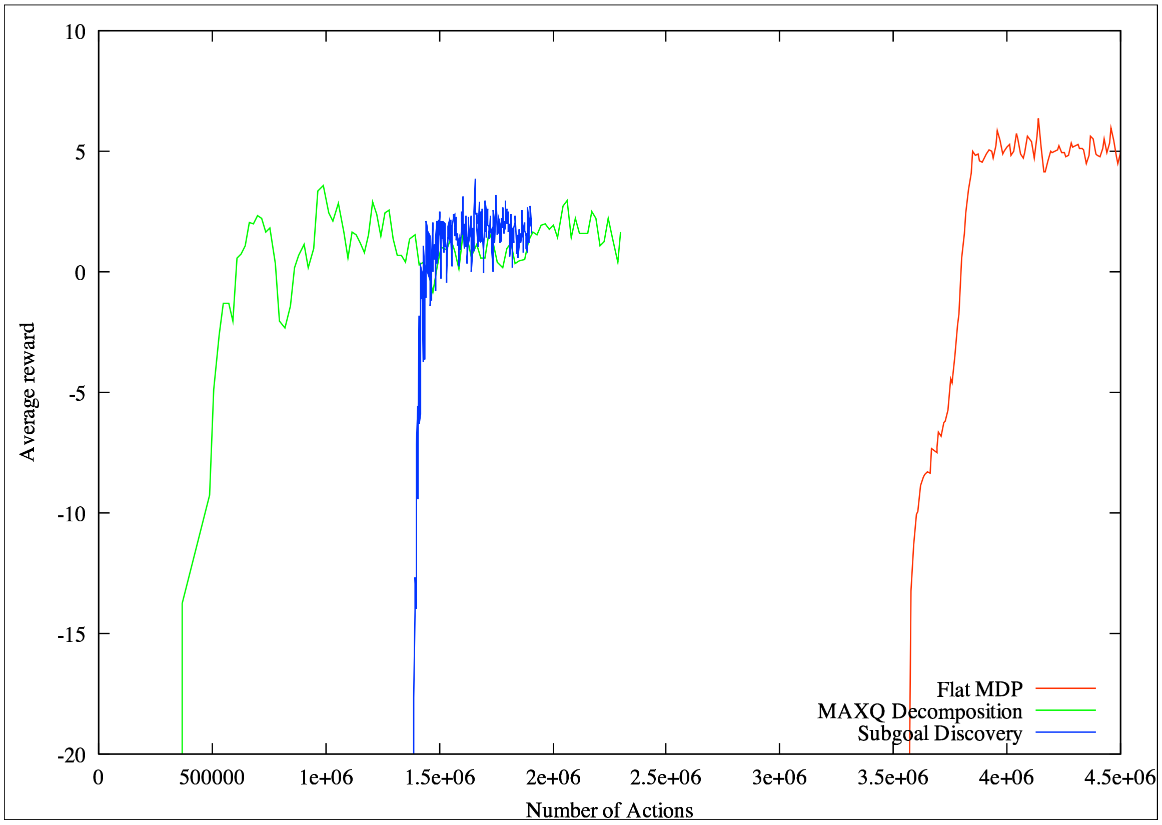


Figure 6.2. Comparison of policies derived with the MAXQ method and with a SMDP with sampling-based subgoal discovery.

to the subgoals that are discovered by sampling-based subgoal discovery. The reason for this is that while subgoals are hand designed in the MAXQ decomposition, the sampling-based method is fully autonomous and does not rely on human decision. As a result, subgoal discovery generates additional subgoal policies that are not required for the task at hand and might not find the optimal option set. Figure 6.3 illustrates the comparison between learning time in MAXQ and the BPSMDP constructed by the action-dependent partitioning method. This experiment shows that action-dependent partitioning can significantly outperform the MAXQ decomposition since it constructs state and temporal abstractions resulting in a more abstract state space. In this form, it can transfer the information contained in previously learned policies for solving subsequent tasks.

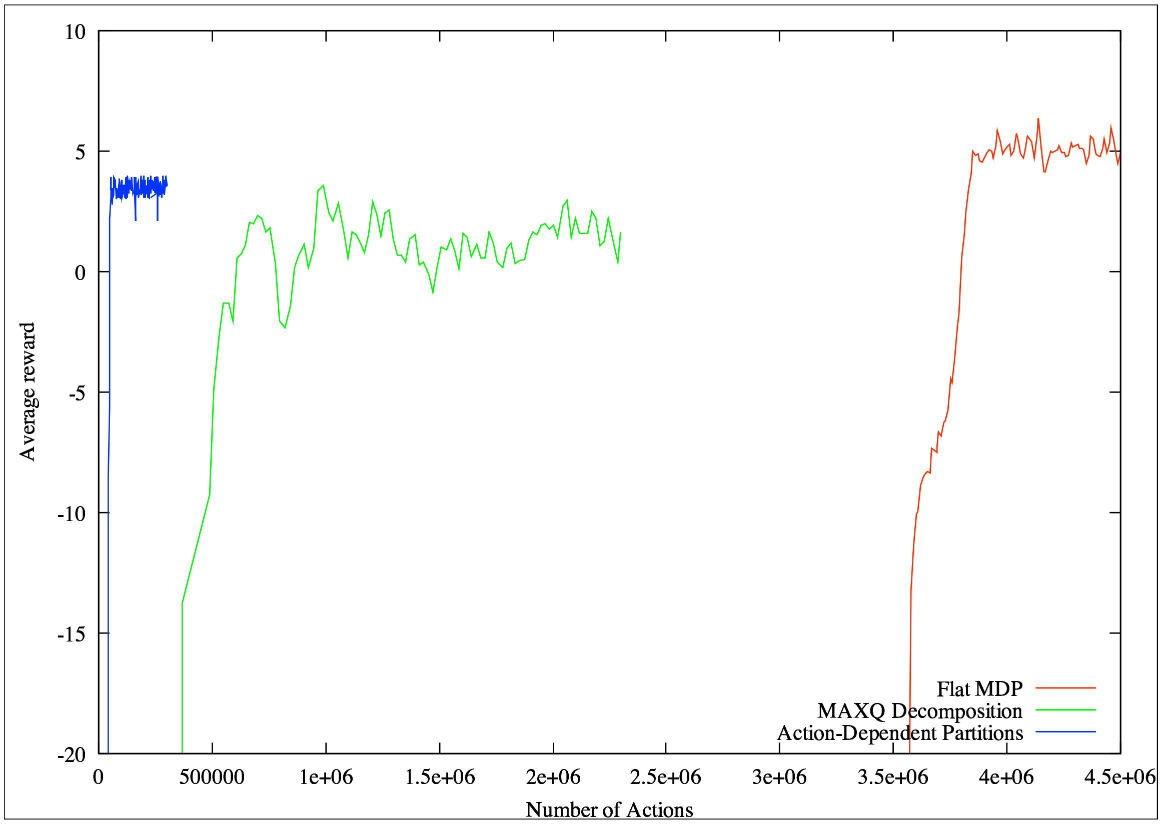


Figure 6.3. Comparison of policies derived with the MAXQ method and action-dependent partitioning with autonomous subgoal discovery.

CHAPTER 7

CONCLUSION

This dissertation presents an efficient method for constructing a hierarchical state and action space for SMDPs. To do this, it first discovers subgoals by analyzing previously learned policies for states with particular structural properties. Once subgoals are derived, it learns options to achieve these subgoals off-line and includes these into the action space available to the agent. Using the subgoal options, it then uses action-dependent state space partitioning to derive an abstract state space. Learning of subsequent tasks is addressed on the abstract state space. To ensure that the new task is learnable, the system presented here maintains a separate value function for the original state space and determines if there are significant inconsistencies between this value function and the one derived on the abstract partition space. When an inconsistency between the values for the best actions in both representations is discovered, the original actions are used to refine the block in the abstract state partition in which the inconsistency was discovered, resolving the inconsistency. This representation serves as a first layer of the hierarchy. In order to estimate the structure of the state space for learning future tasks, the decision layer will be constructed based on the expected time to learn a new task according to previously learned tasks. Together, these techniques permit the agent to form more abstract action and state representations over time.

The experimental results presented in this dissertation illustrate how an autonomous hierarchy can be constructed by using the action-dependent partitioning method. These experiments show a significant reduction in the number of states in the abstract state space, resulting in faster convergence of the value function. Furthermore, these experiments show a procedure to estimate the structure of the state space for learning future tasks and to construct the decision layer based on the expected time to learn a new task according to previously learned tasks.

One of the future goals is to find even more efficient machine learning methods for control tasks. Algorithms can be developed for statistical generalization and reasoning about the algorithms that learn to incrementally scale up to analyze even more complex tasks. Discovering hierarchy in task structure and world structure is an important means in achieving this end. Algorithms need to be developed that learn to reason about their environment in a combinatorial way and learn to develop more cognitive internal representations that mimic relational structures. Integration of more powerful representations such as factorial HMMs and POMDPs are a potential follow-up to this work. Smarter hierarchical algorithms must be found to deal with larger tasks, and research must be directed at more intelligent representational design not only for incorporating hierarchy but also for sharing substructures.

REFERENCES

[1] M. Asadi and M. Huber. Accelerating Action Dependent Hierarchical Reinforcement Learning Through Autonomous Subgoal Discovery. In *Proceedings of the ICML 2005 Workshop on Rich Representations for Reinforcement Learning*, 2005.

[2] M. Asadi and M. Huber. Autonomous Subgoal Discovery and Hierarchical Abstraction for Reinforcement Learning using Monte Carlo Method. In *Proceedings of the 20th Conference of American Association of Artificial Intelligence*, 2005.

[3] A. Barto and S. Mahadevan. Recent Advances in Hierarchical Reinforcement Learning. *Discrete Event Dynamic Systems*, 13:341–379, 2003.

[4] Andrew G. Barto and Sridar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341–379, 2003.

[5] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA, 1996.

[6] Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[7] R. H. Crites and A. G. Barto. Elevator group control using multiple reinforcement learning agents. *Machine Learning*, 33:235–262, 1998.

[8] T. Dean, R. Givan, and M. Greig. Equivalence Notions and Model Minimization in Markov Decision Processes. In *Special issue on planning with uncertainty and incomplete information*, pages 163–223, 2003.

[9] T. Dean, R. Givan, and S. Leach. Model Reduction Techniques for Computing Approximately Optimal Solutions for Markov Decision Processes. In *Proceedings*

of the 13th Annual Conference on Uncertainty in Artificial Intelligence (UAI-97), pages 124–131, San Francisco, CA, 1997. Morgan Kaufmann Publishers.

[10] T. G. Dietterich. An Overview of MAXQ Hierarchical Reinforcement Learning. *Lecture Notes in Computer Science*, 1864, 2000.

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

[12] B. Digney. Emergent hierarchical control structures: Learning reactive / hierarchical relationships in reinforcement environments. In *Proceedings of the Fourth Conference on the Simulation of Adaptive Behavior*, 1996.

[13] C. Drummond. Using a Case Base of Surfaces to Speed-Up Reinforcement Learning. In *Proceedings of the Second International Conference on International Conference on Case-Based Reasoning*, pages 435–444, 1997.

[14] S. Goel and M. Huber. Subgoal Discovery for Hierarchical Reinforcement Learning Using Learned Policies. In *Proceedings of the 16th International FLAIRS Conference*, pages 346–350. AAAI, 2003.

[15] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[16] Manfred Huber and Roderic A. Grupen. Learning to coordinate controllers - reinforcement learning on a control basis. In *IJCAI*, pages 1366–1371, 1997.

[17] G. A. Iba. A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3(4):285–317, June 1989.

[18] K. Kim and T. Dean. Solving Factored MDPs using Non-Homogeneous Partitions. *Artificial Intelligence*, 147:225–251, 2003.

[19] R. E. Korf. *Learning to Solve Problems by Searching for Macro-Operators*. Pitman, Boston, MA, 1985.

[20] S. Mahadevan, N. Marchalleck, T. Das, and A. Gosavi. Self-improving factory simulation using continuous-time average-reward reinforcement learning. In *Proceedings of the 14th International Conference on Machine Learning*, Nashville, TN, 1997.

[21] A. McCallum. Overcoming Incomplete Perception with Utile Distinction Memory. In *Proceedings of the Tenth International Machine Learning Conference*, 1993.

[22] A. McCallum. Learning to use selective attention and short-term memory in sequential tasks. In *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, pages 315–324. The MIT Press, 1996.

[23] A. McGovern and A. Barto. Automatic Discovery of Subgoals in Reinforcement Learning using Diverse Density. In *Proceedings of the 18th International Conference on Machine Learning*, pages 361–368, 2001.

[24] R. Parr. *Hierarchical Control and Learning for Markov Decision Processes*. PhD thesis, University of California, Berkeley, CA, 1998.

[25] R. Parr and S. Russell. Reinforcement learning with hierarchies of machines. in. In *Proceedings of the 1997 Conference on Advances in Neural Information Processing Systems*, Cambridge, MA, 1998. MIT Press.

[26] Theodore J. Perkins and Andrew G. Barto. Heuristic search in infinite state spaces guided by lyapunov analysis. In *IJCAI*, pages 242–247, 2001.

[27] M. L. Puterman. *Markov Decision Problems*. Wiley, New York, 1994.

[28] Fikes R. E., P. E. Hart, and N. J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[29] S. Ross. *Introduction to Stochastic Dynamic Programming*. Academic Press, New York, 1983.

[30] G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Department, Cambridge, U.K., 1994.

[31] A. N. Shiryaev. *Probability*. Springer, New York, 1996.

[32] S. Singh and D. Bertsekas. Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *Proceedings of the 1996 Conference on Advances in Neural Information Processing Systems*, Cambridge, MA, 1997. MIT Press.

[33] Satinder Singh, Tommi Jaakkola, Michael Littman, and Csaba Szpesvari. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning Journal*, 38(3):287–308, 2000.

[34] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[35] R.S. Sutton, D. Precup, and S. Singh. Between MDPs and Semi-MDPs: Learning, Planning, and Representing Knowledge at Multiple Temporal Scales. *Artificial Intelligence*, 112:181–211, 1999.

[36] G. J. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8:257–277, 1992.

[37] Sebastian Thrun and Anton Schwartz. Finding structure in reinforcement learning. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 385–392. The MIT Press, 1995.

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

[39] C. J. C. H. Watkins. *Learning from Delayed Rewards.* PhD thesis, Cambridge University, 1989.

[40] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

BIOGRAPHICAL STATEMENT

Mehran Asadi received his B.S. degree from the Tehran Polytechnic University, Tehran, Iran, in 1994, in Applied Mathematics and Computer Science and his M.S. and PhD degrees from the University of Texas at Arlington in 2003 and 2006, respectively, both in Computer Science and Engineering. From 1994 to 2001, he was with the Semi Conductor Lab, in the Iran Electronic Industries and SAPCO Co., Tehran, Iran. His current research interest is in the area of Machine Learning, in particular, Reinforcement Learning, Statistical and Probabilistic Inference.