ABSTRACTION IN REINFORCEMENT LEARNING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

SERTAN GİRGİN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

MARCH 2007

Approval of the Graduate School of Natural and Applied Sciences.

Prof. Dr. Canan Özgen
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Doctor of Philosophy.

Prof. Dr. Volkan Atalay
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Doctor of Philosophy.

Prof. Dr. Faruk Polat
Supervisor

Examining Committee Members

Prof. Dr. H. Altay Güvenir	(Bilkent Univ., CS)
Prof. Dr. Faruk Polat	(METU, CENG)
Prof. Dr. Kemal Leblebicioğlu	(METU, EEE)
Assoc. Prof. Dr. İ. Hakkı Toroslu	(METU, CENG)
Assoc. Prof. Dr. Halit Oğuztüzün	(METU, CENG)

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: Sertan Girgin

Signature :

ABSTRACT

ABSTRACTION IN REINFORCEMENT LEARNING

Girgin, Sertan

M.S., Department of Computer Engineering

Supervisor : Prof. Dr. Faruk Polat

March 2007, 111 pages

Reinforcement learning is the problem faced by an agent that must learn behavior through trial-and-error interactions with a dynamic environment. Generally, the problem to be solved contains subtasks that repeat at different regions of the state space. Without any guidance an agent has to learn the solutions of all subtask instances independently, which degrades the learning performance.

In this thesis, we propose two approaches to build connections between different regions of the search space leading to better utilization of gained experience and accelerate learning is proposed. In the first approach, we first extend existing work of McGovern and propose the formalization of stochastic conditionally terminating sequences with higher representational power. Then, we describe how to efficiently discover and employ useful abstractions during learning based on such sequences. The method constructs a tree structure to keep track of frequently used action sequences together with visited states. This tree is then used to select actions to be executed at each step.

In the second approach, we propose a novel method to identify states with similar sub-policies, and show how they can be integrated into reinforcement learning framework to improve the learning performance. The method uses an efficient data structure to find common action sequences started from observed states and defines a similarity function between states based on the number of such sequences. Using this similarity function, updates on the action-value function of a state are reflected to all similar states. This, consequently, allows experience acquired during learning be applied to a broader context.

Effectiveness of both approaches is demonstrated empirically by conducting extensive experiments on various domains.

Keywords: Reinforcement Learning, Abstraction, Similarity, Options, Conditionally Terminating Sequences

ÖZ

PEKİŞTIRMELI ÖĞRENMEDE SOYUTLAMA

Girgin, Sertan

Doktora, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Faruk Polat

Mart 2007, 111 sayfa

Pekiştirmeli öğrenme dinamik bir ortam ile deneme-yanılma etkileşimleri aracılığyla davranış öğrenmeye çalışan bir etmenin karşılaştığı problemdir. Genellikle, çözülmesi gereken problem durum uzayının farklı bölgelerinde tekrar eden altgörevler barındırır. Herhangi bir yönlendirme olmadan etmen tüm bu tekrarlamaları birbirinden bağımsız olarak öğrenmek zorundadır ve bu durum da öğrenme performansının düşmesine yol açmaktadır.

Bu tezde, arama uzayının farklı bölgeleri arasında bağlantı kurarak edinilen deneyimin daha verimli kullanımını ve öğrenmenin hızlanmasını sağlayan iki yaklaşım önerilmektedir. Birinci yaklaşımda, McGovern'in mevcut çalışması geliştirilerek daha yüksek temsil gücüne sahip stokastik koşullu sonlanan diziler tanımlanmıştır. Daha sonra, bu dizilere dayalı olarak öğrenme esnasında yararlı soyutlamaların nasıl keşfedilebileceği ve kullanılabileceği anlatılmıştır. Yöntem sıkça kullanılan hareket dizilerini ziyaret edilen durumlar ile birlikte takip edebilmek için bir ağaç yapısı kurmaktadır. Bu ağaç ile her adımda seçilecek hareketlere karar verilmektedir.

İkinci yaklaşımda, benzer alt-davranış biçimlerine sahip durumları belirlemek için özgün bir yöntem önerilmiş ve mevcut algoritmalar ile nasıl entegre edilebileceği gösterilmiştir. Yöntem gözlemlenen durumlardan başlayan ortak hareket dizilerini bulmak için verimli bir veriyapısı kullanmakta ve bu dizilerin sayısına bağlı olarak durumlar arasında bir benzerlik fonksiyonu tanımlanmaktadır. Bu fonksiyon ile bir durumun hareket-değer fonksiyonu üzerindeki güncellemeler tüm benzer durumlara yansıtılmakta ve dolayısıyla öğrenme esnasında edinilen deneyimin daha geniş bir alana uygulanmasına olanak sağlamaktadır.

İki yaklaşımın da başarısı çeşitli problemler üzerinde kapsamlı deneyler ile gösterilmiştir.

Anahtar Kelimeler: Pekiştirmeli Öğrenme, Soyutlama, Benzerlik, Opsiyonlar, Koşullu Sonlanan Diziler

ACKNOWLEDGMENTS

I, foremost, would like to thank Prof. Dr. Faruk Polat for his excellent advice, help and mentoring during the development of this work. I also would like to thank and express gratitude to Prof. Dr. Reda Alhajj for his guidance and support during the period spent as a visiting researcher at the Department of Computer Science, University of Calgary, Alberta, Canada. Finally, thanks to my family for their patience and continuous support.

The work presented in this manuscript is partially supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK) under Grant No. 105E181 (HD-7).

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

List of Algorithms

CHAPTER 1

INTRODUCTION

"What we have to learn to do, we learn by doing."
Aristotle

Single and multi-agent systems are computational systems in which one or more *agents* situated in an environment perform some set of tasks or try to satisfy some set of goals. There are many existing definitions of agents. Russell and Norvig [44] define an agent as *"anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors"* (Figure 1.1). Franklin and Graesser [10] extend this definition as *"An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future"*. According to Maes acting is goal-oriented: *"Autonomous agents are computational systems that inhabit some complex dynamic environment, sense and act autonomously in this environment, and by doing so realize a set of goals or tasks for which they are designed"*. [28]. Hayes-Roth insists that agents reason during the process of action selection in her definition - *"Intelligent agents continuously perform three functions:*

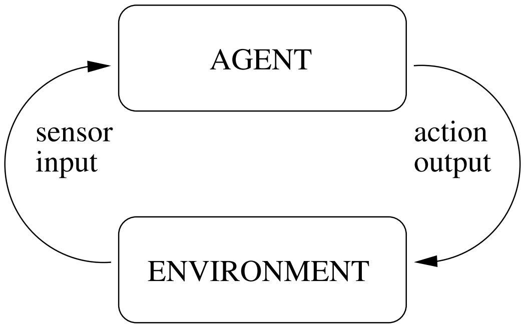


Figure 1.1: An agent in its environment.

perception of dynamic conditions in the environment; action to affect conditions in the environment; and reasoning to interpret perceptions, solve problems, draw inferences, and determine actions" [43]. Wooldridge and Jennings [56] distinguish two different notions of agency. In weak notion of agency, an agent is an entity that has / is capable of

autonomy operating without direct external intervention

social ability capability of interacting with other agents

reactivity being able to perceive the environment and respond in timely fashion to changes that occur in it

pro-activeness exhibiting goal-directed behavior by taking the initiative in order to satisfy design objectives.

For a stronger notion of agency they define the need for mental notions (i.e. knowledge, belief, intention, and obligation) and emotions in addition to the characteristics presented above.

Regardless of its definition, when designing an agent (or a system of agents), unless the environment and the problem to be solved is very small and restricted, it is almost impossible to foresee all situations that the agent may encounter and specify an (optimal) agent behavior in advance [2]. For example, the state space can be too large for explicit encoding and/or the environment, including the (behaviors of) agents that it accommodates, can be non-stationary and may change with time. In order to handle situations that are yet unknown or with different consequences than what is observed before, an agent must be able process the the information that it receives to update or increase its knowledge and abilities, i.e. modify its behavior through experience or conditioning; it must possess the ability to *learn.* In computer science, the development of algorithms and techniques that allow computerized entities to learn is studied under the *machine learning* subfield of artificial intelligence [36].

Reinforcement learning (RL) is the problem faced by an agent that must learn behavior through trial-and-error interactions with a dynamic environment by gaining percepts and rewards from the world and taking actions to affect it [23, 52] (Figure 1.2). This is very similar to the kind of learning and decision making problems that people and animals face in their daily lives (for example, those related to physical activities,

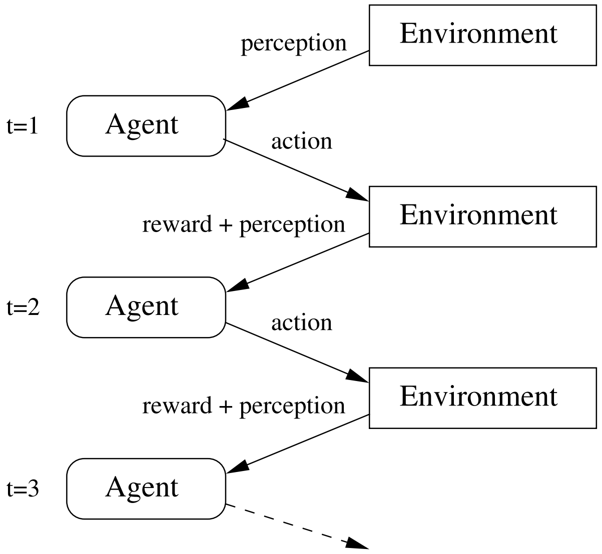


Figure 1.2: Agent–Environment interactions in reinforcement learning.

such as riding a bicycle – everybody falls a couple of times before successfully riding a bicycle.). A particular class of machine learning algorithms that fall into the scope of reinforcement learning, called *reinforcement learning algorithms*, try to find a policy that maximizes an objective function which is based on the rewards received by the agent from the environment. Some of the possible objective functions are

- total reward over a fixed horizon,
- discounted cumulative reward, and
- average reward in the limit.

Although the methods presented in this thesis are applicable to different such functions, we will be using discounted cumulative reward which is the most analytically tractable and most widely studied objective function. Reinforcement learning problems are generally modeled using *Markov Decision Processes* and policies are defined as mappings from states to actions that the agent can take. It is assumed that the states possess the *Markov property*, i.e. if the current state of the decision process at time t is known, transitions to a new state at time $t+1$ are independent of all previous states. By storing the experience of the agent in terms of observed (internal) state and taken action together with the outcome, i.e. received reward and next state, it is possible to find an optimal policy using dynamic programming and temporal differencing techniques [19, 23, 36, 52].

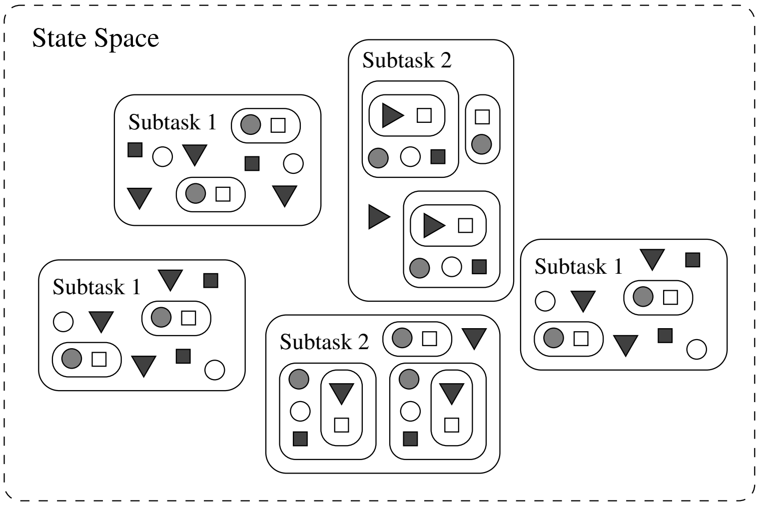


Figure 1.3: Hierarchical task decomposition in complex problems.

In most of the realistic and complex domains, the task that the agent is trying to solve is composed of various subtasks and has a hierarchical structure formed by the relations between them [3]. Each of these subtasks repeats many times at different regions of the state space (Figure 1.3). Although, all instances of the same subtask, or similar subtasks, have almost identical solutions (sub-behaviors), without any (self) guidance, an agent has to learn the solutions of all instances independently by going through similar learning stages again and again. This situation affects the learning process negatively, making it difficult to converge to optimal behavior in a reasonable time.

The main reason of the problem is the lack of connections, that would allow to share solutions, between similar subtasks scattered throughout the state space. One possible way to build connections is to use *temporally abstract actions*, or options, which are macro actions that generalize primitive actions and last for a period of time [38, 40, 20, 53, 8, 34, 3]. By providing meta-knowledge about the problem to be solved, they allow the agent to focus the search at a higher level, instead of learning to combine primitive actions each time. Although, in terms of performance, it is quite effective to define temporally abstract actions manually, doing so requires extensive domain knowledge and gets difficult as the complexity of the problem or the components that are involved increases. Therefore, several methods that try to discovery useful abstractions without user intervention are proposed in the literature. These methods either find states that are possible subgoals of the problem using statistical or graph theoretic approaches and generate sub-policies leading to them [9, 48, 33, 35, 45, 46, 30], or identify frequently occurring action patterns and convert them into macro-actions [31, 32]. In the first case, different abstractions are generated separately for each instance of the repeated subtask or similar subtasks since the subgoals states in each instance may differ. In the second case, the number of abstractions can be very large, since solutions of subtasks are not unique and action sequences that differ slightly may have the same consequences, unless frequent action sequences are restricted in an ad hoc manner.

Motivated by these shortcomings of existing approaches, as the first contribution of this thesis, we propose a method which efficiently discovers useful options in the form of a single meta-abstraction without storing all observations. We first extend conditionally terminating sequence formalization of McGovern [32] and propose stochastic conditionally terminating sequences which cover a broader range of temporal abstractions and show how a single stochastic conditionally terminating sequence can be used to simulate the behavior of a set of conditionally terminating sequences. Stochastic conditionally terminating sequences are represented using a tree structure, called sequence tree. Then, we investigate the case where the set of conditionally terminating sequences is not known in advance but has to be generated during the learning process. From the histories of states, actions and received rewards, we first generate trajectories of possible optimal policies, and then convert them into a modified sequence tree. This helps to identify and compactly represent frequently used sub-sequences of actions together with states that are visited during their execution. As learning progresses, this tree is constantly updated and used to implicitly run represented options. The proposed method can be treated as a meta-heuristic to guide any underlying reinforcement learning algorithm. We demonstrate the effectiveness of this approach by reporting extensive experimental results on various test domains. Also, we compared our work with acQuire-macros, the option framework proposed by [31, 32]. The results show that the proposed method attains substantial level of improvement over widely used reinforcement learning algorithms.

As the second contribution of this thesis, following homomorphism notion, we propose a method to identify states with similar sub-policies without requiring a model of the MDP or equivalence relations, and show how they can be integrated into reinforcement learning framework to improve the learning performance [12, 15]. Using the collected history of states, actions and rewards, traces of policy fragments are generated and then translated into a tree form to efficiently identify states with similar sub-policy behavior based on the number of common action-reward sequences. Updates on the action-value function of a state are then reflected to all similar states, expanding the influence of new experiences. We demonstrate the effectiveness of this approach by reporting test results on various test domains. Further, the proposed method is compared with other reinforcement learning algorithms, and a substantial level of improvement is observed on different test cases. Also, although the approaches are different, we present how the performance of our work compares with option discovery algorithms.

Before describing the our contributions in more detail, we give an overview of the standard reinforcement learning framework of discrete time finite Markov decision processes in Chapter 2. Semi-Markov Decision Processes and in particular the *options* framework of Sutton et al. [53] is also described in a separate section followed by related work on option discovery and equivalence in reinforcement learning. In Chapter 3, we present the test domains which are used to evaluate the performance of proposed methods. The main contributions of this thesis, namely automatically discovering and creating useful temporal abstractions in the form of stochastic conditionally terminating sequences and employing state similarity in reinforcement learning are presented in Chapters 4 and 5, respectively. In Section 5.4, we also combined these two methods together and analyzed their overall behavior. The final chapter, Chapter 6, concludes and discusses our plans for future research in this area.

CHAPTER 2

BACKGROUND

In this chapter, we introduce the background necessary to understand the material introduced in this thesis. We start by defining Markov decision processes and reinforcement learning problem. Then, we describe the generic temporal differencing method to solve an RL problem. We briefly present learning/update rules for Q-learning and $Sarsa(\lambda)$ algorithms, as they are used to evaluate our proposals.

2.1 Markov Decision Processes

Definition 2.1.1 (Markov Decision Process) *A* Markov decision process*, MDP in short, is a tuple* $\langle S, A, T, R \rangle$*, where*

- S *is a finite set of states,*
- A *is a finite set of actions,*
- $T : S \times A \times S \rightarrow [0,1]$ *is a state transition function such that* $\forall s \in S, \forall a \in A, \sum_{s' \in S} T(s,a,s') = 1$*, and*
- $R : S \times A \rightarrow \Re$ *is a reward function.*

$T(s,a,s')$ *denotes the probability of making a transition from state* s *to state* s' *by taking action* a*.* $R(s,a)$ *is the* immediate *expected reward received when action* a *is executed in state* s*.* ■

Given an MDP, a stationary *policy*, $\pi : S \times A \rightarrow [0,1]$, is a mapping that defines the probability of selecting an action from a particular state. In a non-stationary policy, the probability distribution may change with time. If $\forall s \in S, \pi(s, a_s) = 1$ and $\forall a \in A, a \neq a_s, \pi(s,a) = 0$ then π is called *deterministic policy.*

Definition 2.1.2 (State value function) *The* value *of a state* s *under policy* π*, denoted by* $V^{\pi}(s)$*, is the expected infinite discounted sum of reward that the agent will gain if it starts in state* s *and follows* π *[23]. It is computed as*

$$V^{\pi}(s) = \sum_{t=0}^{\infty} \gamma^t E(r_t | \pi, s_0 = s)$$

where r_t *is the reward received at time* t*, and* $0 \leq \gamma < 1$ *is the* discount factor*.* $V^{\pi}(s)$ *is called the policy's* state value function*.*

Let $Q^{\pi}(s, a)$ denote the expected infinite discounted sum of reward that the agent will gain if it selects action a at s, and then follows π:

$$Q^{\pi}(s, a) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V^{\pi}(s')$$

Then, we have

$$V^{\pi}(s) = \sum_{a \in A} \pi(s, a) Q^{\pi}(s, a)$$

Similar to $V^{\pi}(s)$, $Q^{\pi}(s, a)$ is called the policy's *state-action value function.*

In a Markov decision process, the objective of an agent is to find an *optimal policy*, π^*, which maximizes the state value function for all states (i.e., $\forall \pi, \forall s \in S, V^{\pi^*}(s) \geq V^{\pi}(s)$). Every MDP has a deterministic stationary optimal policy; and the following Bellman equation holds [4] $\forall s \in S$:

$$\begin{aligned} V^*(s) &= \max_{a \in A} \left(R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V^*(s') \right) \\ &= \max_{a \in A} Q^*(s, a) \end{aligned}$$

Here, V^* and Q^* are called the optimal value functions. Using Q^*, π^* can be specified as:

$$\pi^*(s) = \begin{cases} 1 & \text{if } a = \arg\max_{a' \in A} Q^*(s, a') \\ 0 & \text{otherwise} \end{cases}$$

π^* is a greedy policy; at state s it selects the action having the maximum $Q^*(s, \cdot)$ value, i.e. the one which is expected to be most profitable.

When the reward function, R, and the state transition function, T, are known, π^* can be found by using *dynamic programming* techniques [23, 52]. When such information is not readily available, a model of the environement (i.e. R and T functions) can be generated online, or Monte Carlo and temporal-difference (TD) learning methods can be used to find the optimal policy directly without building such a model. Instead of requiring complete knowledge of the underlying model, these approaches rely on experience in the form of sample sequences of states, actions, and rewards collected from on-line or simulated trial-and-error interactions with the environment.

TD learning methods are built on the bootstrapping and sampling principles. Estimate of the optimal state(-action) value function is kept and updated in part on the basis of other estimates.

Let $Q(s, a)$ denote the estimated value of $Q^*(s, a)$. In an episodic setting, a TD learning algorithm which uses the state-action value function has the following form:

```
 1: Initialize $Q$ arbitrarily (e.g., $Q(\cdot, \cdot) = 0$)
 2: repeat
 3:     Let $s$ be the current state
 4:     repeat                                              ▷ for each step
 5:         Choose $a$ from $s$ using policy derived from $Q$ with sufficient exploration
 6:         Take action $a$, observe $r$ and the next state $s'$
 7:         Update Q based on $s, r, a, s'$
 8:         $s = s'$                         ▷ Next state becomes the current state.
 9:     until $s$ is a6 terminal state
10: until a termination condition holds
```

The critical point is to update the estimate in such a way that it converges to the optimal function. Various algorithms basically differ from each other on how this update is realized. In well-known *Q-learning* algorithm [54], at step 7, $Q(s, a)$ is updated according to the following learning rule:

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha[r + \gamma \max_{a' \in A} Q(s', a')] \tag{2.1}$$

where $\alpha \in [0, 1)$ is the learning rate. If

1. every state-action pair is updated an infinite number of times, and

2. α is appropriately decayed over time (i.e. is square summable, but not summable)

then Q values are guaranteed to converge to optimal Q^* values when Equation (2.1) is used as the update formula [54, 23, 27]. In Sarsa algorithm [52], instead of the current state value of the next state, only the current value of $Q(s', a')$, where a' is the action selected at state s', is used as an estimate of future discounted rewards:

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha[r + \gamma Q(s', a')] \tag{2.2}$$

Although update rules are similar to each other, the convergence behavior of Q-learning and Sarsa algorithms are quite different. Q-learning is an *off-policy algorithm* which means that Q values converge to Q^* independent of how the actions are chosen at step 5; the agent can be acting randomly and still Q values will converge to optimal values. However, in Sarsa algorithm, the convergence depends on the policy being followed; if the agent acts randomly then it is very likely that convergence will not be achieved, and consequently an optimal policy could not be found. Algorithms having this property are called *on-policy algorithms*.

In simple TD learning algorithms, such as Q-learning or Sarsa, the update of estimation is based on just the immediate reward received by the agent and a single step approximation of the expected future reward – current value of the next state (or next state-action tuple) is assumed to represent the remaining rewards. n-step TD and TD(λ) algorithms extend this to include a sequence of observed rewards and discounted average of all such sequences, respectively. By keeping a temporary record of visited states and selected actions, at each step, change in the value of $Q(s, a)$ is gradually reflected backwards using mechanisms such as *eligibility traces* [52]. This improves the approximation and also increases the convergence rate.

2.2 Semi-Markov Decision Processes and Options

As a discrete time model, Markov Decision Processes introduced in the previous section and consequently algorithms based on MDP framework, are restricted in the sense that all actions are presumed to take unit time duration; it is not possible to model situations in which actions take variable amount of time, i.e., they are temporally extended. *Semi-Markov Decision Processes* extend MDPs to incorporate transitions with stochastic time duration. Generalizing Definition 2.1.1, a Semi-Markov Decision Process is formally defined as follows.

Definition 2.2.1 (Semi-Markov Decision Process) *A Semi-Markov Decision Process, SMDP in short, is a tuple* $\langle S, A, T, R, F \rangle$*, where*

- *S is a finite set of states,*
- *A is a finite set of actions,*
- *$T : S \times A \times S \rightarrow [0,1]$ is a state transition function such that $\forall s \in S, \forall a \in A, \sum_{s' \in S} T(s,a,s') = 1$,*
- *$R : S \times A \rightarrow \Re$ is a reward function, and*
- *F is a function giving probability of transition times for each state-action pair.*

$T(s,a,s')$ denotes the probability of making a transition from state s to state s' by taking action a. $F(t|s,a)$ denotes the probability that starting at s, action a completes within time t. $R(s,a)$ is the expected *reward that will be received until next transition when action a is executed in state s; it allows rewards be received during a transition from one state to another, and computed as*

$$R(s,a) = k(s,a) + \int_0^\infty \int_0^t \rho(s,a,t) dt dF(t|s,a)$$

where $k(s,a)$ is a fixed reward received upon executing action a at state s, and $\rho(s,a,t)$ is a reward rate given that the transition takes t time units. ■

It is worth noting that when F has the form

$$F(t|\cdot) = \begin{cases} 0 & ,t < 1 \\ 1 & ,t \geq 1 \end{cases}$$

i.e. a step function with a jump at 1, an SMDP turns into a MDP. During a transition from one state to another upon executing an action, the state of the environment may change continually, i.e., the agent may pass through some intermediate states. However, it has no direct effect on the course of events until the current action terminates.

Similar to MDPs, a (stationary) policy for an SMDP is a mapping from states to actions, and the following Bellman equations hold for an optimal policy [39]:

$$\begin{aligned} V^*(s) &= \max_{a \in A} \left(R(s,a) + \sum_{s' \in S} T(s,a,s') \int_0^\infty \gamma^t V^*(s') F(t|s,a) dt \right) \\ &= \max_{a \in A} \left(R(s,a) + \gamma(s,a) \sum_{s' \in S} T(s,a,s') V^*(s') \right) \\ &= \max_{a \in A} Q^*(s,a) \end{aligned}$$

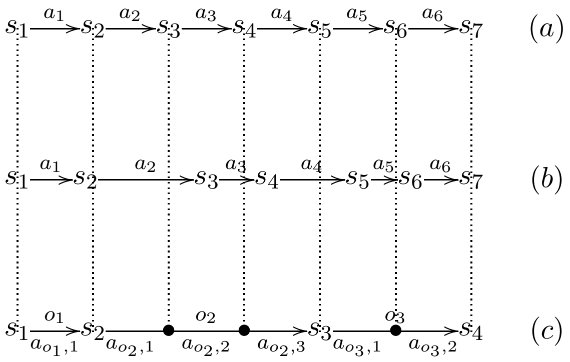


Figure 2.1: Time flow of (a) MDP, (b) SMDP, and (c) discrete-time SMDP embedded over MDP (options).

where

$$\gamma(s,a) = \int_0^\infty \gamma^t F(t|s,a)dt$$

is the variable discount rate based on state s and action a. As in the case of MDPs, when the reward function R and the state transition function T are known, an optimal policy for an SMDP can be found by using dynamic programming techniques. However, when the model is not known, the reinforcement learning algorithms for MDPs given in Section 2.1 can be generalized or adapted to SMDPs by taking into account the length of transitions [6, 29, 39]. For example, the update rule of Q-learning becomes:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha[r + \gamma^t \max_{a' \in A} Q(s',a')]$$

where a is the action selected at state s, s' is the observed state after a terminates, r is the appropriately weighted sum of rewards received during the transition from state s to s', and t is the time passed in between.

2.2.1 Options

In SMDP formalism, the actions are treated as "black boxes", indivisible flow of execution, which are used as they are irrespective of their underlying internals. Nevertheless, if the behavior of temporally extended actions are not determined for certain, such as when they need to adapt to changes in the environment or be learned from simpler actions, this assumption makes it hard to analyze and modify them. As demonstrated in Figure 2.1, by embedding a discrete-time SMDP over a MDP, the *options* framework of [53] extends the theory of reinforcement learning to include temporally extended actions with an explicit interpretation in terms of the underlying MDP. While keeping the unit time transition dynamics of MDPs, actions are generalized in the sense that they may last for a number of discrete time steps and referred to as *options*.

Definition 2.2.2 (Option) *An option is a tuple* $\langle I, \pi, \beta \rangle$, *where*

- $I \subseteq S$ *is the set of states that the option can be initiated at, called* initiation set,
- π *is the option's local policy, and*
- β *is the termination condition.*

Once an option is initiated by an agent at a state $s \in I$, π *is followed and actions are selected according to* π *until the option terminates (stochastically) at a specific condition determined by* β. ■

By changing I, π, which is a restricted policy over actions by itself, or β, one can now alter the behavior of an option. In a *Markov option*, action selection and option termination decisions are made solely on the basis of the current state, i.e., $\pi : S \times A \to [0, 1]$, and $\beta : S \to [0, 1]$. During option execution, if the environment makes a transition to state s, then the Markov option terminates with probability $\beta(s)$ or else continues, determining the next action a with probability $\pi(s, a)$. It is generally assumed that an option can also be initiated at a state where it can continue, which means that the set of states with termination probability less than one is a subset of I. In this case, the domain of π is restricted to I only instead of S.

Markov options are limited in their ability to represent some useful abstractions, such as terminating after a given number of steps, or carrying out a sequence of actions irrespective of the consequences (as in open-loop control). For more flexibility, *Semi-Markov options* extend Markov policies and allow π and/or β to depend on all prior events since the option was initiated.

Definition 2.2.3 (History) *Let* $s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, \ldots, r_\tau, s_\tau$ *be the sequence of states, actions and rewards observed by the agent starting from time* t *until* τ. *This sequence is called a* history *from* t *to* τ, *denoted* $h_{t\tau}$ *[53]. The length of* $h_{t\tau}$ *is* $\tau - t$.■

If the set of all possible histories is denoted by Ω, then in a Semi-Markov option, β and π are defined over Ω instead of S, i.e., $\beta : \Omega \to [0, 1]$, $\pi : \Omega \times A \to [0, 1]$.

Let O be the set of available options, which also include primitive actions as a special case, then a (stationary) policy over options, $\mu : S \times O \rightarrow [0,1]$, is a mapping that defines the probability of selecting an option from a particular state. If state s is not in the initiation set of an option o, then $\mu(s,o)$ is zero. In [53], it has been proved that for any MDP and any set of options defined on that MDP, a policy over options, executing each to termination, is an SMDP. Hence, results given above for SMDPs also hold for options, and optimal value functions and Bellman equations can be generalized to options and to policies over options. We have:

$$\begin{aligned} V^*(s) &= \max_{o \in O_s} E\{r + \gamma^k V^*(s')\} \\ &= \max_{o \in O_s} Q^*(s,o) \end{aligned}$$

where O_s denotes the set of options that can be initiated at s, s' is the state in which o terminates, k is the number of steps elapsed during the execution of o, and r is the cumulative discounted reward received between s and s' (i.e., $r = r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{k-1} r_{t+k}$ if o is initiated at time t); all conditional on the event that option o is initiated in state s. Consequently, SMDP learning methods are adapted to use option set instead of action set. In particular, the update rule of Q-learning is modified as follows:

$$Q(s,o) = (1-\alpha)Q(s,o) + \alpha[r + \gamma^t \max_{o' \in O_s} Q(s',o')]$$

which converges to optimal Q-values for all $s \in S$ and $o \in O$ under conditions similar to those for basic Q-learning.

2.3 Option Discovery

In most applications, options are part of the problem specification and provided by the system developer prior to learning. In order to define the options, the system developer needs to possess extensive knowledge about the problem domain and have a rough idea of possible solutions. Otherwise, faulty abstractions may be constructed that will have a negative effect on learning and move the agent away from the solution. However, this process of creating abstractions manually becomes more difficult as the complexity of the problem increases; as the number of variables increases it gets harder to handle the relations between them and their effect on the environment. On the positive side, compact and multi-leveled hierarchies of abstractions can be defined, leading to effective and efficient solutions. An alternative way is to construct macro actions automatically using domain information acquired during learning.

Most of the existing research on automatic discovery of temporally abstract actions are focused on two main approaches. In the first approach, possible subgoals of the problem are identified first, then abstractions solving them are generated and utilized. Digney [9], and Stolle and Precup [48] use a statistical approach and define subgoals as states that are visited frequently or have a high reward gradient. In [33], McGovern and Barto select most diversely dense regions of the state space (i.e., the set of states that are visited frequently on successful experiences, where the notion of success is problem dependent) as subgoals. Menache et al. [35] follow a graph theoretical approach and their Q-cut algorithm uses a Min-Cut procedure to find subgoals which are defined as bottleneck states connecting the strongly connected components of the graph derived from state transition history. Şimşek and Barto also use a similar definition of subgoals, but propose two methods for searching them locally as in Q-cut. Şimşek and Barto also use a similar definition of subgoals, but propose two methods for searching them locally as in Q-cut. In the first one, subgoal discovery is formulated as a classification problem based on relative novelty of states [45]; and in the second one as partitioning of local state transition graphs that reflect only the most recent experiences of the agent [46]. In another graph based method due to Mannor et al., state space is partitioned into regions using a clustering algorithm (based on graph topology or state values), and policies for reaching different regions are learned as macro actions [30]. In all of these methods, sub-policies leading to the discovered subgoals are explicitly generated by using auxiliary reinforcement learning processes, such as action replay [25], with artificial rewards executed on the "neighborhood" of the subgoal states. Note that, since different instances of same or similar subtasks would probably have different subgoals in terms of state representation, with these methods they will be discovered and treated separately.

In relatively less explored second approach, temporal abstractions are generated directly, without identifying subgoals, by analyzing common parts of multiple policies. An example of this approach is proposed by McGovern, where sequences that occur frequently on successful action trajectories are detected, and options are created for those which pass a static filter that eliminates sequences leading to similar results [31, 32]. One drawback of this method is that common action sequences are identified at regular intervals, which is a costly operation and requires all state and action histories be stored starting from the beginning of learning. Also, since every prefix of a frequent sequence has at least the same frequency, the number of possible options increases rapidly, unless limited in a problem specific way. Recently, Girgin et al. [11] also proposed a method that utilizes a generalized suffix tree structure to identify common sub-sequences within histories, and select a subset of them to generate corresponding options without any prior knowledge. Furthermore, they use a classifier to map states to options in order to generalize the domain of discovered options.

2.4 Equivalence in Reinforcement Learning

Temporally abstract actions try to solve the solution sharing problem inductively. Based on the fact that states with similar patterns of behavior constitute the regions of state space corresponding to different instances of similar subtasks, the notion of state equivalence can be used as a low level and more direct means of solution sharing compared to methods that make use of temporally abstract actions described above. By reflecting experience acquired on one state to all similar states, connections between similar subtasks can be established implicitly which, in effect, reduce the repetitions in learning and consequently improve the performance.

State equivalence is closely related with model minimization in Markov Decision Processes (MDPs), and various definitions exist. In [17], Givan et. al. define equivalence of states based upon stochastic bisimilarity, a generalization of the notion of bisimulation from the literature on concurrent processes. Two states are said to be equivalent if they are both action sequence and optimal value equivalent. Based on the formalism of MDP homomorphism Ravindran and Barto extended equivalence over state-action pairs [41, 42], which allow reductions and relations not possible in case of bisimilarity. Furthermore, they applied state-(action) equivalence to the options framework to derive more compact options without redundancies, called relativized options. However, relativized options are not automatically discovered, but rather defined by the user. Zinkevich and Balch [57] also addressed how symmetries in MDPs can be used to accelerate learning employing equivalence relations on the state-action pairs, in particular for multi-agent case where permutations of features corresponding to various agents are prominent, but without explicitly formalizing them.

CHAPTER 3

PROBLEM SET

In this chapter, we describe the details and properties of sample problems that are used to evaluate the performance of proposed methods in this work. The three test domains are: a six-room maze, various versions of Dietterich's taxi problem [8], and the keepaway subtask of robotic soccer [51]. We selected these domains due to their distinctive characteristics; the first problem contains bottleneck states and has a relatively small state space, the second one has repeated subtasks applicable at different regions of the state space, and the third one has a continuous state space and actions take variable amount of time, i.e. formulated in the SMDP setting.

3.1 Six-room Maze Problem

In the six-room maze problem, there is a grid-world containing six rooms in a 2×3 layout (Figure 3.1). Neighboring rooms are connected to each other with doorways. The task of the agent is to navigate from a randomly chosen position in the top left room to the gray shaded goal location in the bottom right room. The primitive actions are movement to a neighboring cell in four directions, *north*, *east*, *south*, and *west*. Actions are non-deterministic and each action succeeds with probability $p_{success}$,

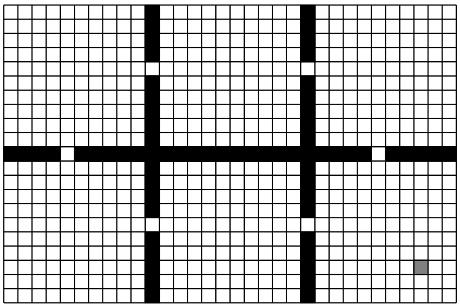
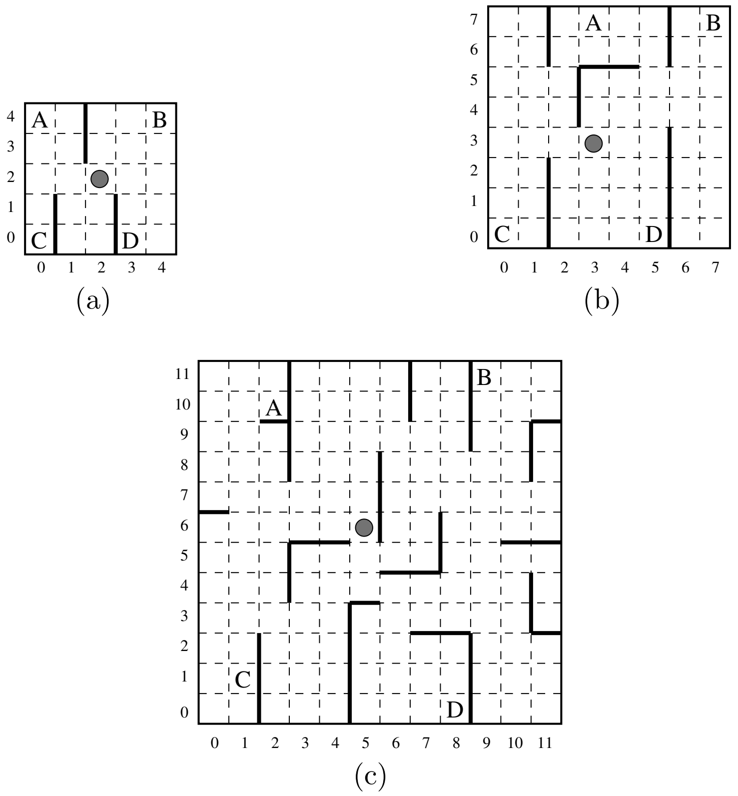


Figure 3.1: Six-room maze.



(a) (b) (c)

Figure 3.2: Taxi problem of different sizes. (a) 5 × 5 (Dietterich's original problem), (b) 8 × 8, and (c) 12 × 12. Predefined locations are labeled with letters from A to D.

or else moves the agent perpendicular, either clockwise or counter clockwise, to the desired direction with probability $1 - p_{success}$. Unless stated otherwise, $p_{success}$ is set to 0.9. If an action causes the agent to hit a wall (black squares), the action has no effect and the position of the agent does not change. The agent only receives a reward of 1 when it reaches the goal location. For all other cases, it receives a small negative reward of -0.01. The state space consists of 605 possible positions of the agent. The agent must learn to reach the goal state in shortest possible way to maximize total discounted reward.

In this task, for achieving a better performance, the agent needs to learn to make efficient use of the passages connecting the rooms. These passages can be regarded as the subgoals of the agent and are also bottleneck states in the solutions.

3.2 Taxi Problem

Our second test domain, the Taxi domain, is an episodic task in which a taxi agent moves around on an $n \times n$ grid world, containing obstacles that limit the movement

(Figure 3.2). The agent tries to transport one or more passengers located at predefined locations, to either the same location or to another one. In order to accomplish this task, the taxi agent must repeat the following sequence of actions for all passengers:

1. go to the location where a passenger is waiting,
2. pick up the passenger,
3. go to the destination location, and
4. drop off the passenger

At each time step, there are six different actions that can be executed by the agent: the agent can either move one square in one of four main directions, *north*, *east*, *south*, and *west*, attempt to *pickup* a passenger, or attempt to *drop-off* the passenger being carried. If a movement action causes the agent hit to wall/obstacle, the position of the agent does not change. The movement actions are non-deterministic; each movement action succeeds with probability $p_{success}$, or as in six-room maze problem with $1 - p_{success}$ probability agent may move perpendicular (either clockwise or counter-clockwise) to the desired direction. Unless stated otherwise, $p_{success}$ is set to 0.8. Passengers can not be co-located at the same position but their destinations can be the same.

An episode ends when all passengers are successfully transported to their destinations. There is an immediate reward of $+20$ for each successful transportation of a passenger, a high negative reward of -10 if pickup or drop-off actions are executed incorrectly (i.e., if pickup is executed but there is no passenger to pickup at current position, or drop-off is executed but there is no passenger being carried or current position is different than the destination of the passenger being carried) and -1 for any other action. Dietterich's original version of the taxi problem is defined on a 5×5 grid with a single passenger as presented in Figure 3.2 (a), and used as a testbed for the Max-Q hierarchical reinforcement learning algorithm [8].

In order to maximize the overall cumulative reward, the agent must transport the passengers as quickly as possible, i.e., using minimum number of actions. Note that if there are more than one passenger, the ordering of passengers to be transported is also important, and the agent needs to learn the optimal ordering for highest possible reward. Initial position of the taxi agent, locations and destinations of the passengers are selected randomly with uniform probability.

We represent each possible state using a tuple of the form $\langle r, c, l_1, d_1, \ldots, l_k, d_k \rangle$, where r and c denote the row and column of the taxi's position, respectively, k is the number of passengers, and for $1 \leq i \leq k$, l_i denotes the location of the i^{th} passenger (either (i) one of predefined locations, (ii) picked-up by the taxi, or (iii) transported), and d_i denotes the destination of the passenger (one of predefined locations). The size of the state space is $RC(L+1)^k L^k$ where $R \times C$ is the size of the grid, L is the number of predefined locations, and k is the number of passengers. For the single passenger case, since there is only one passenger to be carried, it reduces to $RC(L+1)L$. In particular, for the original version of the problem on a 5×5 grid with one passenger, there are 500 different tuples in the state space.

Compared to the six-room maze domain, the taxi domain has a larger state space and possesses a hierarchical structure with repeated subtasks, such as navigating from one location to another. These subtasks are difficult to describe as state based subgoals because state trajectories are different in each instance of a subtask. For example, if there is a passenger at location A, the agent must first learn to navigate there and pick up the passenger, which has the same sub-policy irrespective of the destination of the passenger or other state variables at the time of execution.

3.3 Keepaway Subtask of Robotic Soccer

Our last test domain is a subtask of robotic soccer. Robotic soccer is a fully *distributed*, *multiagent* domain with both *teammates* and *adversaries* in which two teams of autonomous agents play the game soccer against each other [1]. It is supported by the Robot World Cup Initiative, or *RoboCup* for short, an international joint project which uses game of soccer as a central topic of research to promote artificial intelligence, robotics, and related field. It aims to reach the ultimate goal of "*developing a team of fully autonomous humanoid robots that can win against the human world champion team in soccer.*" by 2050 [1]. RoboCup is divided into five leagues in which teams consisting of either robots or programs cooperate in order to defeat the opponent team, i.e. score more goals than the other team. In the simulation league, a realistic simulator, called the *soccerserver*, provides a virtual field and simulates all

catch *dir* Catch ball into direction *dir* if the ball is within the catchable area and the goalie is inside the penalty area. The goalie is the only player that can execute this action.

change_view *width quality*

dash *pow* Accelerate the agent with power *pow* in the direction of its body.

kick *pow dir* Kick the ball towards direction *dir* with power *pow* if the distance between ball and agent is less than kickable margin.

say *mesg* Broadcast message *mesg* to other agents.

turn *mnt* Change body direction with moment *mnt*.

turn_neck *ang* Change the neck angle of the player relative to its body. It can be executed during the same cycle as turn, dash and kick actions.

Table 3.1: List of primitive actions.

movements of a ball and players. Each agent is controlled by an independent single client. Environment is partially observable, which means there is hidden state, and agents receive noisy and inaccurate visual and auditory sensor information every 150 msec. indicating the relative distance and angle to visible objects in the world, such as the ball and other agents. Agent can communicate with each other only through the server, which is subject to communication bandwidth and range constraints. They may execute a primitive, parameterized action such as *turn*, *dash* and *kick* every 100 msec. (see Table 3.1 for a complete listing). The actions are non deterministic which means that agents may not be able to affect the world exactly as intended. Actions may sometimes fail or succeed partially (for example trying to turn 20 degrees may result in a rotation of less than 20 degrees depending on the current state of the agent and the environment). More detailed information about the soccerserver can be found in [37].

One important property of robotic soccer is that the perception and action cycles are *asynchronous* due to the difference in their frequencies. This situation necessitates a need for keeping an internal world model and predicting the current and future states of the world since, although possible, it is not feasible to directly map perceptual input to actions. All these domain characteristics make simulated robot soccer a complex, realistic and challenging domain for artificial intelligence studies [51].

Keepaway is an episodic subtask of RoboCup soccer played with two teams of

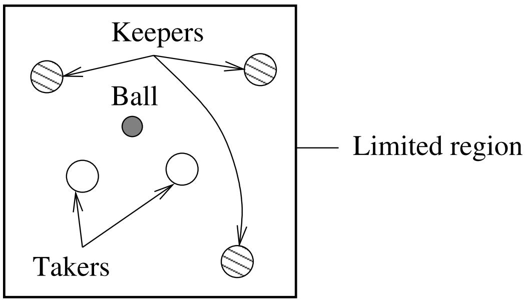


Figure 3.3: 3 vs. 2 keepaway.

reduced size, each consisting of two to five agents. At the beginning of each episode, members of one team, the *keepers*, are distributed evenly near the corners of a limited region (ranging from 25m x 25m to 35m x 35m and above) inside the soccer field. Members of the other team, the *takers*, are all placed at the bottom left corner of the region and the ball is placed next to the keeper at the top left corner. Typically, the number of takers is (one) less than the number of keepers.

The aim of the keepers is to maintain the possession of the ball and at the same time keep it within the limited region as long as possible by passing the ball to each other. The takers, on the other hand, try to intercept the ball and gain possession or send it outside of the region (Figure 3.3). Whenever the takers have the ball or the ball leaves the region, the episode ends and players are reset for another episode. Recent versions of RoboCup soccer simulator directly support keepaway subtask and all steps described above are handled automatically by invoking the server using appropriate parameters.

Although much simpler than the whole game of soccer, keepaway has *continuous state and action spaces* and involves both *cooperation* (within team members) and *competition* (between the members of two teams) making it a complex yet manageable benchmark problem for machine learning.

In a series of papers, Stone, Sutton and Kuhlmann treated the problem as a *semi-Markov* decision process by using action choices consisting of high level skills (see Table 3.2), which may persist for multiple time steps, instead of primitive actions provided by the simulator [49, 51, 50, 24]. They focus on learning policies for *active keepers*, keepers that are close enough to the ball to kick it, when playing against other players (keepers and takers) with predefined behaviors (random or hand-coded). Note that there may be more than one active keeper, since at a given time ball can

HoldBall() Remain stationary while keeping possession of the ball in a position that is as far away from the opponents as possible.

PassBall(k) Kick the ball directly towards keeper k.

GetOpen(p) Move to a position that is free from opponents and open for a pass from position p.

GoToBall() Intercept a moving ball or move directly towards a stationary ball.

BlockPass(k) Move to a position between the keeper with the ball and keeper k.

Table 3.2: High level skills used in the keepaway subtask of robotic soccer. All these skills except PassBall(k) are simple functions from state to a corresponding action; an invocation of one of these normally controls behavior for a single time step. PassBall(k), however, requires an extended sequence of actions, using a series of kicks to position the ball, and then accelerate it in the desired direction. Its execution influences behavior for several time steps. [50]

be kickable by multiple players depending on their position. A passive keeper sticks to the following policy:

Algorithm 1 Policy of a passive keeper

```
repeat
    if fastest player in the team to the ball then
        execute GoToBall() for one time step
    else
        Let $p$ be the predicted position of the ball when the fastest teammate reaches it
        execute GetOpen(p) for one time step
    end if
until ball is kickable or episode ends
```

When a taker has the ball, it tries to maintain possession by invoking HoldBall() for one time step. Otherwise, all takers either

1. choose with uniform probability one of $\{GoToBall(), BlockPass(1), \ldots, Block-Pass(k)\}$ where k is the number of keepers and execute for one time step (*Random*),
2. GoToBall() for one time step (*All-to-Ball*), or
3. use the *Hand-Coded* policy given in Algorithm 2.

Algorithm 2 Hand-coded policy of a taker

```
repeat
    if not closest or second closest taker to the ball then
        Mark the most open opponent (i.e. with the largest angle with vertex at the ball that is clear of takers) using BlockPass() option
    else if another taker doesn't have the ball then
        execute GoToBall() for one time step
    end if
until episode ends
```

An active keeper has multiple choices: it may hold the ball or pass to one of its teammates. Therefore, there are k options $\{HoldBall(), PassBall(2), \ldots, PassBall(k)\}$ to select. From the internal world model, the following state variables are available to the agent:

- $dist(K_i, C)$ $i = 1..k$
- $dist(T_i, C)$ $i = 1..t$
- $dist(K_1, K_i)$ $i = 2..k$
- $dist(K_1, T_i)$ $i = 1..t$
- $\min(dist(K_i, T_1), \ldots, dist(K_i, T_t))$ $i = 2..k$
- $\min(ang(K_i, K_1, T_1), \ldots, dist(K_i, K_1, T_t))$ $i = 2..k$

where k is the number of keepers, t is the number of takers, C denotes the center of the playing region, K_1 is the self position of the agent, $K_2 \ldots K_k$ and $T_1 \ldots T_t$ are positions of other keepers and takers ordered by increasing distance from the agent, respectively, $dist(a, b)$ is the distance between a and b, and $ang(a, b, c)$ is the angle between a and c with vertex at b. All variables are continuous values. The total number of the variables is $4 * k + 2 * t - 3$, which is linear with respect to number of players involved. The immediate reward received by each agent after selecting a high level skill is the number of primitive time steps that elapsed while following the higher-level action.

CHAPTER 4

IMPROVING REINFORCEMENT LEARNING BY AUTOMATIC OPTION DISCOVERY

In this chapter, we propose a method which efficiently discovers useful options in the form of a single meta-abstraction without storing all observations [14, 13]. We first extend conditionally terminating sequence formalization of McGovern and propose stochastic conditionally terminating sequences which cover a broader range of temporal abstractions and show how a single stochastic conditionally terminating sequence can be used to simulate the behavior of a set of conditionally terminating sequences. Stochastic conditionally terminating sequences are represented using a tree structure, called sequence tree. Then, we investigate the case where the set of conditionally terminating sequences is not known in advance but has to be generated during the learning process. From the histories of states, actions and reward, we first generate trajectories of possible optimal policies, and then convert them into a modified sequence tree. This helps to identify and compactly represent frequently used subsequences of actions together with states that are visited during their execution. As learning progresses, this tree is constantly updated and used to implicitly run represented options. The proposed method can be treated as a meta-heuristic to guide any underlying reinforcement learning algorithm. We demonstrate the effectiveness of this approach by reporting test results on three domains, namely room-doorway, taxi cab and keepaway in robotic soccer problems. The results show that the proposed method attains substantial level of improvement over widely used RL algorithms. Also, we compared our work with acQuire-macros, the option framework proposed by McGovern and Barto [31, 32].

The rest of the chapter is organized as follows. Section 4.1 covers conditionally terminating sequences and extends them into stochastic conditionally terminating sequences that have higher representational power. Based on stochastic conditionally terminating sequences, a novel method to discover useful abstractions during the learning process is covered in Section 4.1.2. Experimental results are reported in Section 4.2.

4.1 Options in the Form of Conditionally Terminating Sequences

In this section, we present the theoretical foundations and building blocks of our automatic option discovery method. We first describe a special case of Semi-Markov options in the form of *conditionally terminating sequences* as defined by McGovern [32] in Section 4.1.1. In Section 4.1.2, we highlight their limitations and propose the novel concept of *sequence trees* and the corresponding formalism of *stochastic conditionally terminating sequences* that extend conditionally terminating sequences and enable richer abstractions; they are capable of representing and generalizing the behavior of a given set of conditionally terminating sequences.

Finally, a method which utilizes a modified version of a sequences tree to find useful abstractions online, i.e. during the course of learning, is introduced; it is based on the idea of reinforcing the execution of action sequences that are experienced frequently by the agent and yield a high return.

4.1.1 Conditionally Terminating Sequences

Definition 4.1.1 (Conditionally Terminating Sequence) *A* conditionally terminating sequence *(CTS) is a sequence of* n *ordered pairs*

$$\sigma = \langle C_1, a_1\rangle\langle C_2, a_2\rangle \ldots \langle C_n, a_n\rangle$$

where n *is the length, denoted* $|\sigma|$, *and each ordered pair* $\langle C_i, a_i\rangle$ *consists of a continuation set* $C_i \subseteq S$ *and action* $a_i \in A$. *At step* i, a_i *is selected and the sequence advances to the next step (i.e.,* a_i *is performed) if current state* s *is in* C_i*; otherwise the sequence terminates.* ■

C_1 is the initiation set of σ and denoted by $init_\sigma$. The sequence $act\text{-}seq_\sigma = a_1 a_2 \ldots a_n$ is called the *action sequence* of σ. We use $C_{\sigma,i}$ and $a_{\sigma,i}$ to denote the i^{th} continuation set and action of σ.

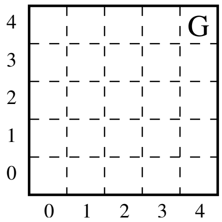


Figure 4.1: 5 × 5 grid world.

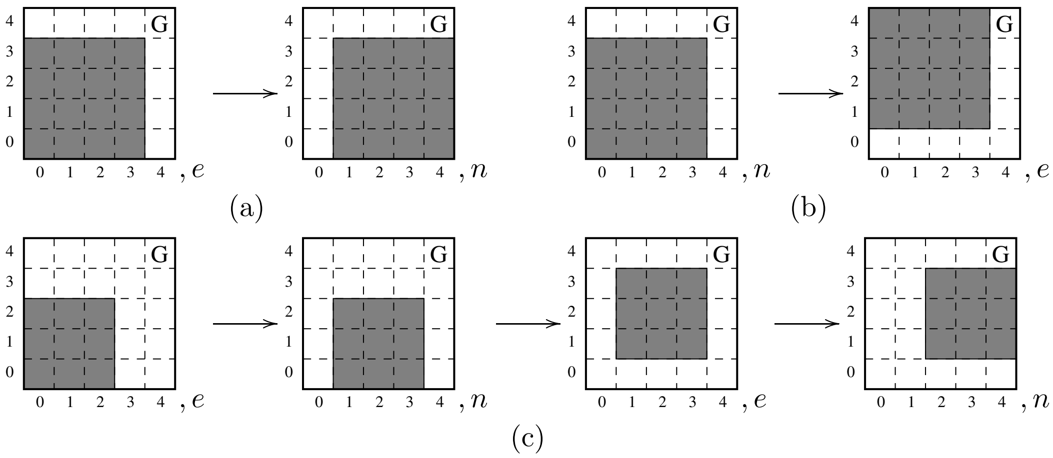
Lemma 4.1.2 *For every CTS σ, one can define a corresponding Semi-Markov option o_σ.*

Proof. Let $\sigma = \langle C_1, a_1\rangle \dots \langle C_n, a_n\rangle$ be a conditionally terminating sequence. A history $h_{t\tau} = s_t, a'_t, r_{t+1}, s_{t+1}, a'_{t+1}, \dots, r_\tau, s_\tau$ is said to be compatible with σ if and only if its length is less than the length of σ, and for $i = t, \dots, \tau - 1$, $s_i \in C_{i-t+1} \wedge a'_i = a_{i-t+1}$, and $s_\tau \in C_{\tau-t+1}$, i.e., observed states were consecutively in the continuation sets of σ starting from I_1 and at each step actions determined by σ were executed. Let H_σ denote the set of possible histories in Ω that are compatible with σ. We can construct a Semi-Markov option $o_\sigma = \langle I, \pi, \beta\rangle$ as follows:

$$
\begin{aligned}
I &= C_{\sigma,1} \\
\pi(h_{t\tau}, a) &= \begin{cases} 1, \text{if } h_{t\tau} \in H_\sigma \wedge a = a_{\sigma,\tau-t+1} \\ 0, \text{otherwise} \end{cases} \\
\beta(h_{t\tau}) &= \begin{cases} 0, \text{if } h_{t\tau} \in H_\sigma \\ 1, \text{otherwise} \end{cases}
\end{aligned}
$$

o_σ can only be initiated at states where σ can be initiated. When initiated at time t, the execution of o_σ continues if and only if the state observed at time $t+k$, $0 \leq k < n$, is in I_{k+1}. At time $t+k$, action a_{k+1} is selected, for every other possible action $a \neq a_{k+1}, \pi(\cdot, a) = 0$. Therefore, o_σ behaves exactly as σ. ■

The most important feature of CTSs is that they can be used to represent frequently occurring and useful patterns of actions in a reinforcement learning problem. For example, consider the 5 × 5 grid world shown in Figure 4.1. Starting from any location, the agent's goal is to reach the top rightmost cell marked with "G" as soon as possible (i.e., with minimum number of actions). At each time step, the agent can move in one of four directions; assume $A = \{n, s, e, w\}$ is the set of actions and $S = \{(i,j) | 0 \leq i, j \leq 4\}$ is the set of states where (i,j) denotes the coordinates of



(a) (b) (c)

Figure 4.2: (a) σ_{en}, (b) σ_{ne}, and (c) σ_{enen}. Shaded areas denote the continuation sets.

the agent. Note that the rectangular region on the grid with corners at (r_1, c_1) and (r_2, c_2), represented by $[(r_1, c_1), (r_2, c_2)]$, is a subset of S.

In order to reach the goal cell, one of the useful action patterns that can be used by the agent is to move diagonally in the north-east direction, i.e., e followed by n or alternatively n followed by e. These patterns can be represented by the following CTSs presented in Figure 4.2 (a) and (b):

$$\begin{aligned}\sigma_{en} &= \langle[(0,0),(3,3)],e\rangle\langle[(0,1),(3,4)],n\rangle \\ \sigma_{ne} &= \langle[(0,0),(3,3)],n\rangle\langle[(1,0),(4,3)],e\rangle\end{aligned}$$

Conditionally terminating sequences allow an agent to reach the goal more directly by shortening the path to a solution; in our grid world example, any primitive action can reduce the Manhattan distance to the goal position (i.e., $|4-i|+|4-j|$ where (i,j) is the current position of the agent) by 1 at best, whereas σ_{en} and σ_{ne} defined above reduce it by 2 when they are applicable. As the complexity of the problem increases, this shortening through the use of CTSs makes it possible to efficiently explore the search space to a larger extent. Consequently, this leads to faster convergence and improves the performance of learning. Although they have a simple structure, a set of CTSs is quite effective in exploiting temporal abstractions.

Now, consider a longer CTS σ_{enen} given in Figure 4.2 (c) that represents moving diagonally in the north-east direction two times; it is defined as:

$$\begin{aligned}\sigma_{enen} &= \langle[(0,0),(2,2)],e\rangle\langle[(0,1),(2,3)],n\rangle \\ &\quad\ \langle[(1,1),(3,3)],e\rangle\langle[(1,2),(3,4)],n\rangle\end{aligned}$$

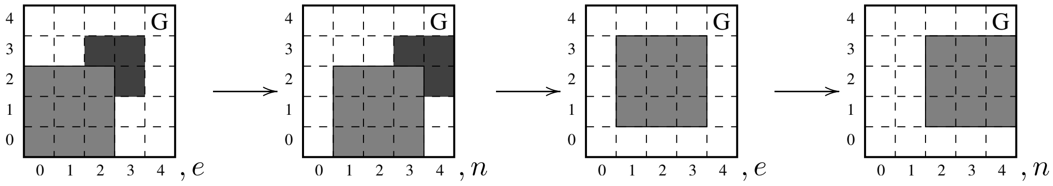


Figure 4.3: $\beta_{en} \cup \sigma_{enen}$. Dark shaded areas denote the continuation sets of β_{en}.

Note that the action sequence of σ_{enen} starts with the action sequence of σ_{en}; for the first two steps, action selection behaviors of σ_{en} and σ_{enen} are the same. Therefore, by taking the union of the continuation sets, it is possible to merge σ_{en} and σ_{enen} into a new CTS $\sigma_{en-enen}$ which exhibits the behavior of both sequences:

$$\begin{aligned}\sigma_{en-enen} &= \langle C_{\sigma_{en},1} \cup C_{\sigma_{enen},1}, e\rangle\langle C_{\sigma_{en},2} \cup C_{\sigma_{enen},2}, n\rangle\langle C_{\sigma_{enen},3}, e\rangle\langle C_{\sigma_{enen},4}, n\rangle \\ &= \langle C_{\sigma_{en},1}, e\rangle\langle C_{\sigma_{en},2}, n\rangle\langle C_{\sigma_{enen},3}, e\rangle\langle C_{\sigma_{enen},4}, n\rangle\end{aligned}$$

The action sequence of $\sigma_{en-enen}$ is *enen* and initially it behaves as if it is σ_{en} and σ_{enen}, i.e., selects action e and then n at viable states. At the third step, if the current state is a state where σ_{enen} can continue (i.e., in $C_{\sigma_{enen},3}$), then the sequence continues execution like σ_{enen}; otherwise it terminates. We call $\sigma_{en-enen}$ the *union* of σ_{en} and σ_{enen}.

Definition 4.1.3 (Union of two CTSs) *Let* $u = \langle C_{u,1}, a_{u,1}\rangle \ldots \langle C_{u,m}, a_{u,m}\rangle$ *and* $v = \langle C_{v,1}, a_{v,1}\rangle \ldots \langle C_{v,n}, a_{v,n}\rangle$ *be two CTSs such that the action sequence of* v *starts with the action sequence of* u*, i.e.* $m \leq n$ *and for* $1 \leq i \leq m, a_{u,i} = a_{v,i}$*. The CTS* $u \cup v$ *defined as*

$$\begin{aligned}u \cup v &= \langle C_{u,1} \cup C_{v,1}, a_{v,1}\rangle\langle C_{v,2} \cup C_{v,2}, a_{v,2}\rangle \ldots \langle C_{u,m} \cup C_{v,m}, a_{v,m}\rangle \\ &\quad \langle C_{v,m+1}, a_{v,m+1}\rangle \ldots \langle C_{v,n}, a_{v,n}\rangle\end{aligned}$$

is called the union of u *and* v*.* ■

Note that, given a sequence of observed states there may be cases in which both u and v would terminate within $|u| = m$ steps but $u \cup v$ continues to execute. For example, let $\beta_{en} = \langle[(2,2)(3,3)], e\rangle\langle[(2,3)(3,4), n\rangle$ be a restricted version of moving diagonally in the north-east direction. We have

$$\begin{aligned}\beta_{en} \cup \sigma_{enen} &= \langle[(0,0)(2,2)] \cup [(2,2)(3,3)], e\rangle\langle[(0,1)(2,3)] \cup [(2,3)(3,4), n\rangle \\ &\quad \langle[(1,1),(3,3)], e\rangle\langle[(1,2),(3,4)], n\rangle\end{aligned}$$

as presented in Figure 4.3. Initiated at state $(3,2)$, $\beta_{en} \cup \sigma_{enen}$ would start to behave like β_{en} and select action e. Suppose that, due to non-determinism in the environment, this action moves the agent to $(2,2)$ instead of $(3,3)$. By definition, β_{en} can not continue to execute from $(2,2)$ since $(2,2) \notin C_{\beta_{en},2}$; however $(2,2)$ is in the continuation set of the second tuple of σ_{enen}, and therefore $\beta_{en} \cup \sigma_{enen}$ resumes execution from $(2,2)$, switching to σ_{enen}. Thus, the union of two CTSs also generalizes their behavior in favor of longer execution patterns, whenever possible, resulting in a more effective abstraction.

4.1.2 Extending Conditionally Terminating Sequences

One prominent feature of conditionally terminating sequences is that they have a linear flow of execution; actions are selected sequentially provided that the continuation conditions hold. In this respect, they cannot be used to represent situations in which different courses of actions may be followed depending on the observed history of events. On the other hand, such situations are frequent in most real life problems due to the hierarchical decomposition inherent in their structure; abstractions contain common action sequences that solve similar subtasks involved. When conditionally terminating sequences are to be utilized in these problems, a seperate CTS is required for each trajectory of a hierarchical component corresponding to a particular subtask. This consequently leads to a drastic increase in the number of conditionally terminating sequences that need to be defined as the complexity of the problem increases, and constitutes one of the drawbacks of CTSs. By extending them to incorporate conditional branching of action selection, it is possible to make use of existing abstractions in a more compact and effective way, and overcome this shortcoming.

As a demonstrative example, consider σ_{enen} defined in the previous section and two new CTSs presented in Figure 4.4 which are defined as:

$$\begin{aligned} \sigma_{ee} &= \langle[(0,0),(4,2)],e\rangle\langle[(0,1),(4,3)],e\rangle \\ \sigma_{enn} &= \langle[(0,0),(2,3)],e\rangle\langle[(0,1),(2,4)],n\rangle\langle[(1,1),(3,4)],n\rangle \end{aligned}$$

σ_{ee} has an action pattern of moving east twice, and σ_{enn} has an action pattern of moving east followed by moving north twice. Note that, the action sequences of these CTSs have common prefixes. They all select action e at the first step, and furthermore both σ_{enn} and σ_{enen} select action n at the second step. Suppose that

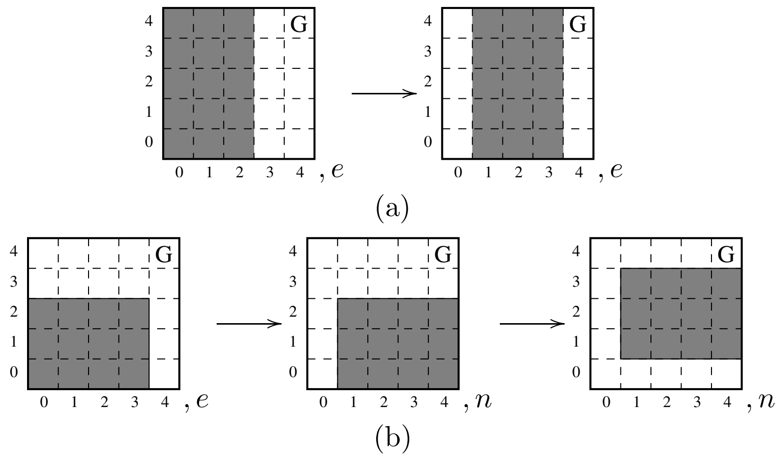


Figure 4.4: (a) σ_{ee}, and (b) σ_{enn}. Shaded areas denote the continuation sets.

the CTS to be initiated at state s is chosen based on a probability distribution $P : S \times \{\sigma_{ee}, \sigma_{enn}, \sigma_{enen}\} \rightarrow [0, 1]$; let $viable_i = \{\sigma \in \{\sigma_{ee}, \sigma_{enn}, \sigma_{enen}\} | \exists I_{\sigma,i}, s_i \in I_{\sigma,i}\}$ denote the set of CTSs which are compatible with the state s_i observed by the agent at step i, and σ_i be the CTS chosen by P over $viable_i$. Then, by taking the union of common parts and directing the flow of execution based on $viable_i$ it is possible combine the behavior of these CTSs as follows:

1. If $viable_1 = \emptyset$ then terminate. Otherwise, execute action e.
2. If $viable_2 = \emptyset$ then terminate. Otherwise,
    (a) If $\sigma_2 = \sigma_{ee}$ then execute action e.
    (b) Otherwise, i.e. if $\sigma_2 \in \{\sigma_{enn}, \sigma_{enen}\}$, execute action n.
        i. If $viable_3 = \emptyset$ then terminate. Otherwise,
            A. If $\sigma_3 = \sigma_{enn}$ then execute action n.
            B. Otherwise, i.e. if $\sigma_3 = \sigma_{enen}$, execute action e followed by $\sigma_{enen}^{[4:]} = \langle I_{\sigma_{enen},4}, n \rangle$.

Steps 2 and 2(b)i essentially introduce conditional branching to action selection; they are called *decision points*. The entire process can be encapsulated and represented in a tree form as depicted in Figure 4.5. $\emptyset$ represents the root of the tree and decision points are enclosed in a rectangle. At each node, shaded areas on the grid denote the states for which the corresponding action (label of the incoming edge) can be chosen. They are comprised of the union of continuation sets of compatible CTSs. We call such a tree a *sequence tree*.

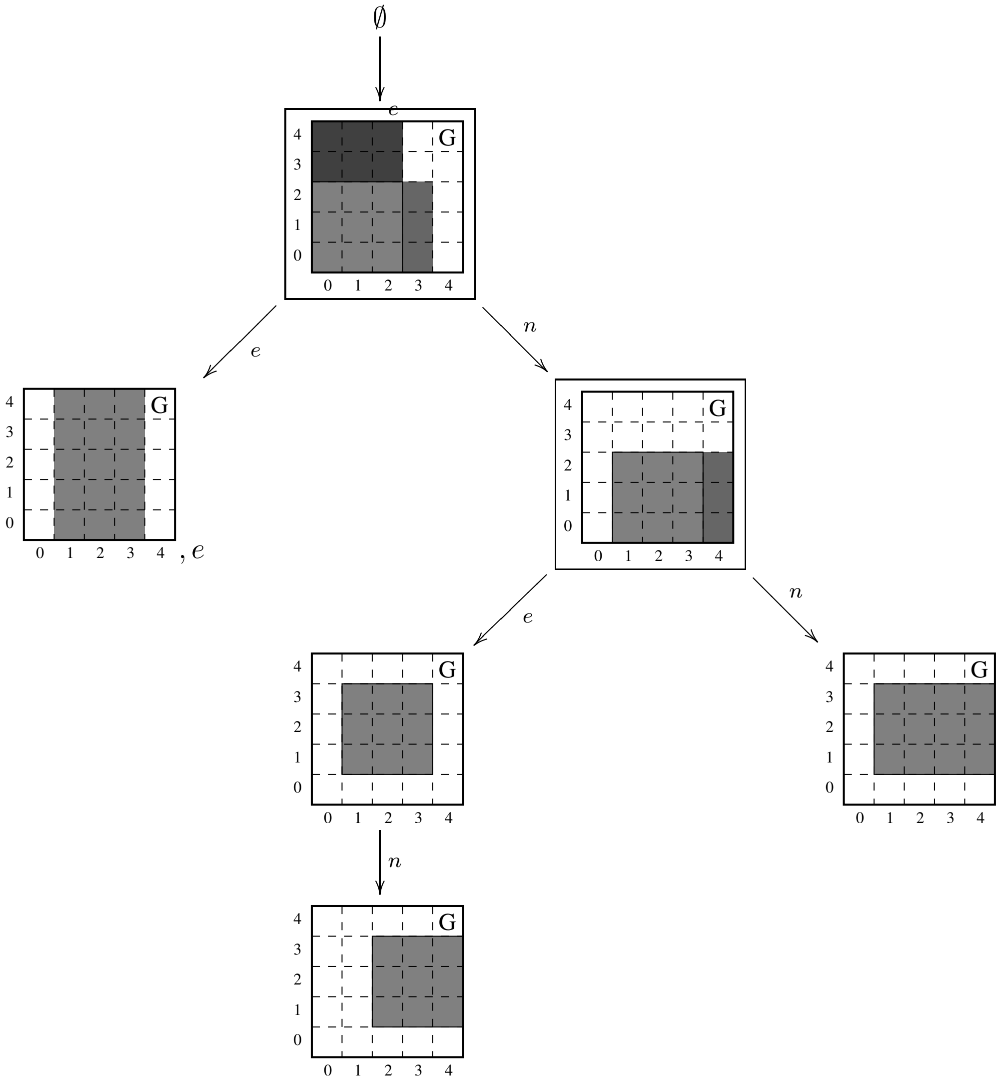


Figure 4.5: Combination of σ_{ee}, σ_{enn} and σ_{enen}. Shaded areas denote the set of states where the corresponding action (label of the incoming edge) can be chosen. Rectangles show the decision points.

Definition 4.1.4 (Sequence Tree) *A sequence tree is a tuple* $\langle N, E\rangle$ *where* N *is the set of nodes and* E *is the set of edges. Each node represents a unique action sequence; the root node, denoted by* $\emptyset$*, represents the empty action set. If the action sequence of node* q *can be obtained by appending action* a *to the action sequence represented by node* p*, then* p *is connected to* q *by an edge with label* a*; it is denoted by the tuple* $\langle p, q, a\rangle$*. Furthermore,* q *is associated with a continuation set* $cont_q$ *specifying the states where action* a *can be chosen after the execution of action sequence* p*. A node* p *with* $k > 1$ *out-going edges is called a* decision point *of order* k*.* ■

Algorithm 3 Algorithm to construct a sequence tree from a given set of conditionally terminating sequences.

```
function CONSTRUCT($\Sigma$)    ▷ $\Sigma$ is a set of conditionally terminating sequences.
    $N = \{\emptyset\}$    ▷ $N$ is the set of nodes. Initially it contains the root node.
    $E = \{\}$    ▷ $E$ is the set of edges.
    for all $\sigma \in \Sigma$ do    ▷ for each CTS in $\Sigma$
        $current = \emptyset$
        for $i = 1$ to $|\sigma|$ do    ▷ for each tuple of $\sigma$
            if $\exists\langle current, p, a_{\sigma,i}\rangle \in E$ then    ▷ Check whether $current$ is already connected to a node $p$ by an edge with label $a_{\sigma,i}$ or not.
                $cont_p = cont_p \cup I_{\sigma,i}$    ▷ Combine the continuation sets.
            else
                Create a new node $p$ with $cont_p = \{I_{\sigma,i}\}$
                $N = N \cup \{p\}$
                $E = E \cup \{\langle current, p, a_{\sigma,i}\rangle\}$    ▷ Connect $current$ to new node $p$ by an edge with label $a_{\sigma,i}$.
            end if
            $current = p$
        end for
    end for
    return $\langle N, E\rangle$
end function
```

Generalizing the example given above, given a set of conditionally terminating sequences $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ a corresponding sequence tree T_Σ that captures their behavior in a compact form can be constructed using CONTRUCT procedure presented in Algorithm 3. The algorithm initially creates a sequence tree comprised of the root node only. Then, CTSs in Σ are one by one added to the existing tree by starting from the root node and following edges according to their action sequence; new nodes are created as necessary and the continuation sets of the nodes are updated by uniting them with the continuation sets of the CTSs. The number of nodes in T_Σ is equal to the total number of unique action sequence prefixes of CTSs in Σ, and the space requirement is dominated by continuation sets of nodes which is bounded by the total space required for the continuation sets of CTSs.

Initiated at state s, a sequence tree T can be used to select actions thereafter by following a path starting from the root node and following edges according to the continuation sets of the outgoing nodes. Initially, the active node of T is the root node. At each time step, if there exist child nodes of the active node which contain the current state observed by the agent in their continuation sets, then:

(i) one of them is chosen using a probability distribution defined over the set of CTSs that T is contructed from,

(ii) the action specified by the label of the edge connecting the active node to the chosen node is executed,

(iii) active node is set to the chosen node.

Otherwise, the action selection procedure terminates.

A sequence tree is the representational form of a novel type of abstraction that we named *stochastic conditionally terminating sequence* (S-CTS) [14]. Stochastic conditionally terminating sequences (S-CTS) extend CTSs to allow alternative action sequences be followed depending on the history of events starting from its execution. They make it possible to define a broader class of abstractions in a compact form.

4.1.3 Stochastic Conditionally Terminating Sequences

Definition 4.1.5 (Stochastic Conditionally Terminating Sequence) *Let* $init_\varsigma$ *denote the set of states at which a stochastic conditionally terminating sequence* ς *can be initiated, and* $first\text{-}act_\varsigma$ *be the set of possible first actions that can be selected by* ς*. A stochastic conditionally terminating sequence (S-CTS) is defined inductively as:*

1. *A CTS σ is a S-CTS; its initiation set and first action set are $init_\sigma$ and $\{first\text{-}act_\sigma\}$, respectively.*

2. *Given a CTS u and a S-CTS v, their concatenation $u \circ v$, defined as executing u followed by v is a S-CTS. $init_{u \circ v}$ is equal to $init_u$ and $first\text{-}act_{u \circ v}$ is equal to $first\text{-}act_u$.*

3. *For a given set of S-CTSs $\Sigma = \{\varsigma_1, \varsigma_2, \ldots, \varsigma_n\}$ such that each ς_i conforms to either rule (1) or rule (2) and for any two ς_i and $\varsigma_j \in \Sigma, first\text{-}act_{\varsigma_i} \cap first\text{-}act_{\varsigma_j} = \emptyset$, i.e., their first action sets are disjoint, then $\odot_t \Sigma$ defined as defined as:*

$$\odot_t \Sigma \begin{cases} \varsigma_i & \text{, if } s \in init_{\varsigma_i} \setminus \bigcup_{j \neq i} init_{\varsigma_j} \\ \mu_{\Sigma,t} & \text{, otherwise} \end{cases}$$

   is a S-CTS. In this definition, s denotes the current state, and $\mu_{\Sigma,t} : \Omega \times \Sigma \rightarrow [0,1]$ is a branching function which selects and executes one of $\varsigma_1, \ldots, \varsigma_n$ according to a probability distribution based on the observed history of the last t steps. $\odot_t \Sigma$ behaves like ς_i if no other $\varsigma_j \in \Sigma$ is applicable at state s. $init_{\odot_t \Sigma} = init_{\varsigma_1} \cup \ldots \cup init_{\varsigma_n}$ and $first\text{-}act_{\odot_t \Sigma} = first\text{-}act_{\varsigma_1} \cup \ldots \cup first\text{-}act_{\varsigma_n}$. Note that, since they are of the form (1) or (2), the first action set of all S-CTSs in Σ has a single element. $\odot_t \Sigma$ in effect allows conditional branching of action selection and corresponds to a decision point of order $n = |\Sigma|$.

4. *Nothing generated by rules other than 1-3 is a S-CTS.* ■

Given a CTS $\sigma = \langle C_1, a_1 \rangle \ldots \langle C_n, a_n \rangle$, let $\sigma^{[i:j]} = \langle C_{\sigma,i}, a_{\sigma,i} \rangle \ldots \langle C_{\sigma,j}, a_{\sigma,j} \rangle$ be the CTS obtained from σ by taking continuation sets and action tuples starting from i up to and including j; let $\sigma^{[i:]}$ denote the suffix of σ which starts from the i^{th} position (i.e. $\sigma^{[i:|\sigma|]}$).

The action pattern that combines σ_{ee}, σ_{enn} and σ_{enen} can now be represented by the S-CTS:

$$\begin{aligned} \varsigma_{\sigma_{ee},\sigma_{enn},\sigma_{enen}} &= (\sigma_{ee}^{[1:1]} \cup \sigma_{enn}^{[1:1]} \cup \sigma_{enen}^{[1:1]}) \circ \odot_1 \{\sigma_{ee}^{[2:]}, (\sigma_{enn}^{[2:2]} \cup \sigma_{enen}^{[2:2]}) \circ \odot_2 \{\sigma_{enn}^{[3:]}, \sigma_{enen}^{[3:]}\}\} \\ &= \langle C_{\sigma_{ee,1}} \cup C_{\sigma_{enn,1}} \cup C_{\sigma_{enen,1}}, e \rangle \\ & \quad \circ \odot_1 \{ \begin{matrix} \langle C_{\sigma_{ee,2}}, e \rangle, \\ \langle C_{\sigma_{enn,2}} \cup C_{\sigma_{enen,2}}, n \rangle \end{matrix} \circ \odot_2 \{ \begin{matrix} \langle C_{\sigma_{enn,3}}, n \rangle, \\ \langle C_{\sigma_{enen,3}}, e \rangle \langle C_{\sigma_{enen,4}}, n \rangle \end{matrix} \}\} \end{aligned}$$

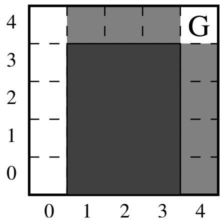


Figure 4.6: After first step, $\varsigma_{\sigma_{en},\sigma_{ee}}$ behaves like σ_{en} if current state is in the right light shaded area, behaves like σ_{ee} if it is in top light shaded area, and either as σ_{en} or σ_{ee} if it is in dark shaded area.

Note that $\varsigma_{\sigma_{ee},\sigma_{enn},\sigma_{enen}}$, and in general a S-CTS, also favors abstractions which last for longer duration by executing $\langle C_{\sigma_{ee},2}, e\rangle$ directly if the state s observed after the termination of $\langle C_{\sigma_{ee},1} \cup C_{\sigma_{enn},1} \cup C_{\sigma_{enen},1}, e\rangle$ is in $C_{\sigma_{ee},2}$ but not in $C_{\sigma_{enn},2} \cup C_{\sigma_{enen},2}$. Similarly, the S-CTS corresponding to the other branch is initiated at once if s is in $C_{\sigma_{enn},2} \cup C_{\sigma_{enen},2}$, but not in $C_{\sigma_{ee},2}$.

Given a S-CTS ς, its corresponding sequence tree can be constructed by using Algorithm 4, given next. The main function CREATE-SEQ-TREE creates a root node and then calls the auxiliary BUILD procedure to recursively construct the sequence tree representing ς. BUILD takes two parameters, a *parent* node and a S-CTS u. If u is a CTS of length one, then a new node with continuation set $init_u$ is created and connected to the parent by an edge with label $first\text{-}act_u$. If u is of the form $\sigma \circ \varsigma$, where σ is a CTS, then BUILD creates a new node, *child*, with continuation set $init_\sigma$, connects *parent* to *child* by an edge with label $first\text{-}act_\sigma$; *child* is connected to the sequence tree of ς if $|u| = 1$ or else to the sequence tree of $\varsigma^{[2:]} \circ \varsigma$. Otherwise, u is of the form $u = \odot_t\{\varsigma_1, \ldots, \varsigma_n\}$; for each ς_i BUILD calls itself recursively to connect *parent* to sequence tree of ς_i.

Note that if a S-CTS u is of the form $\odot_t\{\varsigma_1, \ldots, \varsigma_n\}$, then by definition each ς_i is either a CTS or of the form $\sigma_i \circ \upsilon_i$, where σ_i is a CTS. Therefore, at every call to BUILD, a new node representing an action choice is created either directly (lines 8 and 11) or indirectly (line 21). As a result, CREATE-SEQ-TREE requires linear time with respect to the total number of action sequences that can be generated by the S-CTS ς to construct the corresponding sequence tree.

Instead of creating more functional and complex S-CTSs from scratch, one can extend the union operation defined in Definition 4.1.3 for CTSs to combine behaviors of a conditional terminating sequence and a S-CTS. As we will show later, this also

Algorithm 4 Algorithm to construct the sequence tree corresponding to a given S-CTS.

```
function CREATE-SEQ-TREE($\varsigma$)    ▷ Returns the sequence tree of S-CTS $\varsigma$
    Create a new node root
    BUILD(root, $\varsigma$)
    return root
end function

procedure BUILD(parent,u)
    if $u = \langle I, a \rangle$ then
        Create a new node child with $init_{child} = I$
        Connect parent to child by an edge with label $a$
    else if $u = \sigma \circ \varsigma$ where $\sigma$ is a CTS then
        Create a new node child with $init_{child} = init_{\sigma}$
        Connect parent to child by an edge with label $first\text{-}act_{\sigma}$
        if $|\sigma| = 1$ then
            BUILD(child, $\varsigma$)
        else
            BUILD(child, $\sigma^2 \circ \varsigma$)
        end if
    else
        $u$ is of the form $\odot_t\{\varsigma_1, \ldots, \varsigma_n\}$
        for i=1 to n do
            BUILD(parent, $\varsigma_i$)
        end for
    end if
end procedure
```

enables to represent a set of CTSs as a single S-CTS. The extension is not trivial since one needs to consider the branching structure of a S-CTS. For this purpose we define a time dependent operator $\otimes_t$.

Definition 4.1.6 (Combination operator) *Let u be a CTS and v be a S-CTS*[1] *The binary operator $\otimes_t$, when applied to u and v, constructs a new syntactically valid S-CTS $u \otimes_t v$ that represents both u and v, and is defined recursively as follows, depending on the form of v:*

1. *If v is a CTS, then*
    - *If action sequence of u is a prefix of action sequence of v (or vice versa), then $u \otimes_t v = u \cup v$ (or $v \cup u$).*
    - *If first actions of u and v are different from each other, then $u \otimes_t v = \odot_t\{u, v\}$.*
    - *Otherwise, action sequences of u and v have a maximal common prefix of length $k-1$, and $u \otimes_t v = (u^{[1:k-1]} \cup v^{[1:k-1]}) \circ (\odot_{t+k}\{u^{[k:]}, v^{[k:]}\})$.*
2. *If $v = \sigma \circ \varsigma$, where σ is a CTS, then,*
    - *If the action sequence of u is a prefix of action sequence of σ, then $u \otimes_t v = (\sigma \cup u) \circ \varsigma$.*
    - *If action sequence of σ is a prefix of action sequence of u, then $u \otimes_t v = (\sigma \cup u^{[1:|\sigma|]}) \circ (u^{[|\sigma|+1:]} \otimes_{t+|\sigma|+1} \varsigma)$.*
    - *if first actions of u and σ are different from each other, then $u \otimes_t v = \odot_t\{u, v\}$.*
    - *Otherwise, action sequences of u and σ differ at a position $k \leq |\sigma|$, and $u \otimes_t v = (\sigma^{[1:k-1]} \cup u^{[1:k-1]}) \circ (\odot_{t+k}\{u^{[k:]}, \sigma^{[k:]} \circ \varsigma\})$.*
3. *if $v = \odot.\{\varsigma_1, \ldots, \varsigma_n\}$, then*

$$u \otimes_t v = \begin{cases} \odot_t\{\varsigma_1, \ldots, \varsigma_{i-1}, u \otimes_t \varsigma_i, \varsigma_{i+1}, \ldots, \varsigma_n\} & \text{if } first\text{-}act_u \in first\text{-}act_{\sigma_i} \\ \odot_t\{\varsigma_1, \ldots, \varsigma_n, u\} & \text{otherwise.} \end{cases} \qquad \blacksquare$$

---

[1] It is also possible to define a more general combination operator that acts on two S-CTS. However the definition is more complicated and the operator is not required for our purposes in this thesis, therefore we preferred not to include it.

The operator $\otimes_t$ combines u and v by either directly uniting u with a prefix of v, or creating a new branching condition or update an existing one depending on the action sequence of u and the structure of v. When v is represented using a sequence tree T, it can easily be extended to represent $u \otimes_t v$ by starting from the root node of the tree and following edges that matches the action sequence of u. Let *current* denote the active node of T, which is initially the root node. At step k, if there exists an edge with label $a_{u,k}$ connecting *current* to node n, then the k^{th} continuation set of u is added to the continuation set of n and *current* is set to n. Otherwise, there are three possible cases depending on the number of out-going edges of *current*. In all cases, a new sequence tree for $u^{[k:]}$ is created and connected to *current* by unifying the root node of the created tree with *current*. If *current* has a single out-going edge, then it becomes a decision point of order 2. If *current* is already a decision point, then its order increases by one. The construction of the sequence tree of $u \otimes_t v$ from the sequence tree of v is linear in the length of u and completes at most after $|u|$ steps.[2]

One important application of the $\otimes_t$ operator, as we show next, is that given a set of CTSs to be used in a reinforcement learning problem, by iteratively applying $\otimes_t$ one can obtain a single S-CTS which represents the given CTSs and extend their overall behavior to allow different action sequences be followed depending on the history of observed events. This operation is formally defined as follows:

Definition 4.1.7 (Combination of a set of CTSs) *Let $\Sigma = \{\sigma_1, \dots, \sigma_n\}$ be a set of CTSs and assume that the sequence to be initiated at state s is chosen on the basis of the probability distribution $P(s, \cdot)$ determined by a given function $P : S \times \Sigma \rightarrow [0, 1]$. The S-CTS $\prod \Sigma$ defined as*

$$\prod \Sigma = \begin{cases} \sigma & \text{if } \Sigma = \{\sigma\} \\ \sigma_1 \otimes_0 \prod\{\sigma_2, \dots, \sigma_n\} & \text{otherwise} \end{cases}$$

such that the branching function $\mu_{\{\varsigma_1,\dots,\varsigma_k\},t}$ at decision point $\odot_t\{\varsigma_1, \dots, \varsigma_k\}$ satisfies

$$\mu_{\{\varsigma_1,\dots,\varsigma_k\},t}(\eta, \varsigma) = \max\{P(s, \sigma_i) | \sigma_i \in \Sigma, A_{\sigma_i^{1,t-1}} = \Gamma_{\eta,t-1} \text{ and } a_{\sigma_i,t} \in \textit{first-act}_{\varsigma}\}$$

where $\Gamma_{\eta,t}$ is the sequence of actions taken during the last t steps of history $\eta \in \Omega$, is called the combination of CTSs in Σ. ■

---

[2] Proof is by induction on u.

Suppose that the CTS to be initiated at state s is chosen on $\prod\Sigma$ combines sequences in Σ one by one, and $\mu_{\cdot,t}$ selects a branch based on the initiation probability of CTSs that are compatible with the sequence of actions observed until time t. Suppose that $\prod\Sigma$ is initiated at state s, and let $s_1 = s, s_2, \ldots, s_k$ and $a_1, \ldots, a_{k-1}$ be the sequence of observed states and actions selected by $\prod\Sigma$ until termination, respectively. Then, by construction of $\prod\Sigma$, for each $i = 1, \cdots, k-1$ there exists a CTS $\sigma_i \in \Sigma$ such that $s_i \in C_{\sigma_i,i}$ and the action sequence of σ_i starts with $a_1 \ldots a_i$ (i.e., $A_{\sigma_i}^{1,i} = a_1 \ldots a_i$). Furthermore, one can prove that if $\sigma_\tau \in \Sigma$ is selected by P at state s and executed successfully $|\sigma_\tau|$ steps until termination, then initiated at s, $\prod\Sigma$ takes exactly the same actions as σ_τ, and exhibits the same behavior as we show next.

Theorem 4.1.8 *If $\tau \in \Sigma$ is selected by P at state s and executed successfully $|\tau|$ steps until termination, then initiated at s, $\prod\Sigma$ takes exactly the same actions as τ, and exhibits the same behavior.*

Proof. Let $s = s_1, s_2, \ldots, s_{|\tau|}$ be the sequence of observed states during the execution of τ. By definition, these states are members of the initiation sets of tuples in τ, i.e., for all $i = 1..|\tau|, s_i \in C_{\tau,i}$. Let u be a S-CTS, and a be an action in $first\text{-}act_u$. The behavior of u after selecting action a can be represented by a S-CTS, $u \rightarrow a$, defined as follows:

- If u is a CTS then $u \rightarrow a = u^{[2:]}$.
- If $u = \sigma \circ \varsigma$ where σ is a CTS then

$$u \rightarrow a = \begin{cases} \sigma^{[2:]} \circ \varsigma & \text{if } |\sigma| > 1 \\ \varsigma & \text{otherwise} \end{cases}$$

- If $u = \odot.\{\varsigma_1, \ldots, \varsigma_n\}$, then there exists a unique σ_i such that $a \in first\text{-}act_{\varsigma_i}$ and $u \rightarrow a = \varsigma_i \rightarrow a$.

Suppose that $\prod\Sigma$ chose actions $a_{\tau,1}, \ldots, a_{\tau,k-1}$ followed by $a' \neq a_{\tau,k}$. Let $\prod\Sigma^i$ denote the resulting S-CTS after selecting actions $a_{\tau,1}, \ldots, a_{\tau,i}$, i.e., $\prod\Sigma^i = \prod\Sigma \rightarrow a_{\tau,1} \rightarrow \ldots \rightarrow a_{\tau,i}$. By construction of $\prod\Sigma$, $s_k \in init_{\prod\Sigma^{k-1}}$ and $a_{\tau,k} \in first\text{-}act_{\prod\Sigma^{k-1}}$. Depending on the form of $\prod\Sigma^{k-1}$, we have the following cases:

- $\prod\Sigma^{k-1} = \sigma \circ \varsigma$, where σ is a CTS. Hence, $s_k \in init_\sigma$ and $a' = a_{\sigma,1} = a_{\sigma_\tau,k}$.$\perp$

- $\prod \Sigma^{k-1} = \odot_k\{\varsigma_1, \ldots, \varsigma_n\}$. Since $a_{\tau,k} \in \mathit{first\text{-}act}_{\prod \Sigma^{k-1}}$, by definition, there exists a S-CTS ς_ψ which contains $a_{\tau,k}$ in its first action set, i.e., $a_{\tau,k} \in \mathit{fist\text{-}act}_{\varsigma_\psi}$, and therefore s_k is in the initiation set of ς_ψ. Let X be the set of S-CTSs $\{\varsigma_1, \ldots, \varsigma_n\}$, which can continue from state s_k, i.e., $X = \{\varsigma_i : s_k \in \mathit{init}_{\varsigma_i}\}$. If $|X| = 1$, then $a' \in \mathit{first\text{-}act}_{\varsigma_\psi}$; but by the construction of a S-CTS $\mathit{first\text{-}act}_{\varsigma_\psi} = \{a_{\tau,k}\}$, and consequently $a' = a_{\tau,k}$. Otherwise, by definition, we have

  $$\mu_{X,k}(\eta, \varsigma_i) = \max\{P(s, \sigma_j) : \sigma_j^{1,k-1} = \tau^{1,k-1} \text{ and } a_{\sigma_j,k} \in \text{first-act}_{\varsigma_i}\}$$

  But, for all $\varsigma_i \in X$ other than ς_τ, we have $\mu_{X,k}(\eta, \varsigma_i) < \mu_{X,k}(\eta, \varsigma_\psi) = P(s, \sigma_\tau)$, since σ_τ is selected by P, and thus $a' \in \mathit{first\text{-}act}_{\varsigma_\psi} = \{a_{\sigma_\tau,k}\}$.$\perp$

Both cases lead to a contradiction, completing the proof. ■

Note that, the total number of action sequences in $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ is $\sum_{i=1}^{n} |\sigma_i|$ and hence it is possible to build the corresponding sequence tree for $\prod \Sigma$ in linear time.

4.1.4 Online Discovery of CTS based Abstractions

In a reinforcement learning problem, when the set of CTSs, Σ, is known in advance, one can construct a corresponding sequence tree and by employing the process described above utilize it instead of the CTSs whenever needed. However, determining useful CTSs is a process of increasing complexity requiring extensive domain knowledge and such an information may not be always available prior to learning. An alternative, and certainly more interesting for machine learning, approach is to discover useful CTSs on-the-fly and integrate them as the learning progresses. Learning macro-actions in the form of conditionally terminating sequences has been previously studied by McGovern and an algorithm named *acQuire-macros* is proposed [31]. In acQuire-macros algorithm, all state-action trajectories experienced by the agent are stored and a list of eligible sequences is kept. Periodically, such as at the end of each episode,

(i) using the stored trajectories frequent action sequences having a support over a given threshold are identified and added to the list of eligible sequences using a process that makes use of successive doubling starting from sequences of length 1 (which is equivalent to primitive actions),

(ii) running averages of identified sequences are incremented and if running average of a particular sequence is over a giv en threshold and the sequence passes a problem-specific static filter a new option is created for the action sequence, and

(iii) running averages of all eligible sequences are decayed.

Although empirically it is shown to be quite effective, this approach has several drawbacks;

- it requires all state-action trajectories since the beginning of the learning be stored in a database,
- identification of frequent sequences which is repeated at each step is a costly operation since it requires processing of the entire database, and
- a separate option is created for each sequence which necessitates problem-specific static filtering to prevent options that are "similar" to each other.

In order to overcome this shortcomings, we propose a novel approach which utilizes a single abstraction that is modified continuously and the agent executes it as an exploration policy, but does not maintain a value for it in the traditional sense. This single option is a combination of many action sequences and is represented as a modified version of a sequence tree. During execution, the choice among different branches is made using an estimate of the return obtained following their execution. Periodically, using the observed state-action trajectories this tree is updated to incorporate useful abstractions. We first start with the determination of valuable sequences.

Definition 4.1.9 (π-history) *A history that starts with state s and obtained by following a policy π until the end of an episode (or between designated conditions such as when a reward peak is reached) is called a π-history of s.* ■

Let π and π^* denote the agent's current policy and an optimal policy, respectively, and $h = s_1 a_1 r_2 \ldots r_t s_t$ be a π-history of length t for state s_1. Total cumulative reward of h is defined as

$$R(h) = r_2 + \gamma r_3 + \cdots + \gamma^{t-2} r_t$$

and reflects the discounted accumulated reward obtained by the agent upon following action choices and state transitions in h. Now, suppose that in h a state appears at

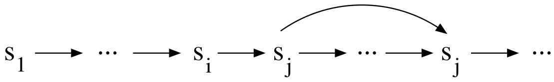


Figure 4.7: Two history alternatives for state s_1.

two positions i and j, i.e., $s_i = s_j, i \neq j$; and consider the sequence

$$h' = s_1 a_1 r_2 \ldots r_i s_i a_{j+1} r_{j+1} \ldots r_t s_t$$

where s_i and s_j are collapsed and the sequence in between is removed (Figure 4.7); one can observe the following:

Observation 4.1.10 *h' is also a (synthetic) π-history for state s_1 and could be a better candidate for being a π^*-history if $R(h') > R(h)$.*

Observation 4.1.11 *Every suffix of h of the form $h_i = s_i a_i r_{i+1} \ldots r_t s_t$ for $i = 2, \cdots, t-1$ is also a π-history of s_i.*

Algorithm 5 Algorithm to generate probable π^*-histories from a given history h.

```
function GENERATE-PROBABLE-HISTORIES($h$)
    $h$ is a history events of the form $s_1 a_1 r_2 \ldots r_t s_t$
    $best[s_{t-1}] = s_{t-1} a_{t-1} r_t s_t$    ▷ best holds current $\pi^*$-history candidates.
    $R[s_{t-1}] = r_t$    ▷ $R[s]$ holds the total cumulative reward for $best[s]$.
    for $i = t - 2$ down to 1 do    ▷ from rear to front
        if $R[s_i]$ is not set or $r_{i+1} + \gamma R[s_{i+1}] > R[s_i]$ then    ▷ if $s_i$ is either not encountered before or has a lower return estimate
            $best[s_i] = s_i a_i r_{i+1} \circ best[s_{i+1}]$    ▷ Create or update the candidate history corresponding to state $s_i$.
            $R[s_i] = R_{i+1} + \gamma R[s_{i+1}]$    ▷ Update maximum reward.
        end if
    end for
    return best
end function
```

Combining these two observations, we can generate a set of potential π^*-history candidates by processing h from rear to front. Let $best(s)$ denote the π-history for state s with maximum total cumulative reward, initially $best(s_{t-1}) = s_{t-1} a_{t-1} r_t s_t$.

For each $s_i, i = t-2, \cdots, 1$, if s_i is not encountered before (i.e., for all $j > i, s_j \neq s_i$) or $r_i + \gamma R(best(s_{i+1}))$ is higher than the total cumulative reward of the current $best(s_i)$, $R(best(s_i))$, then $best(s_i)$ is replaced by $s_i a_i r_{i+1} \circ best(s_{i+1})$, where $\circ$ is the concatenation operator and appends the history represented by $best(s_{i+1})$ to $s_i a_i r_{i+1}$. Finally, for each unique s_i in $(s_1, \ldots, s_t)$, the resulting $best(s_i)$ is used as a probable π^*-history for state s_i. The complete procedure is given in Algorithm 5.

As learning progresses, probable π^*-histories with action sequences that are part of useful CTSs (i.e. sub-policies) would appear more frequently in the recent episodes, whereas the occurrence rate of histories whose action sequences are representatives of CTSs with dominated sub-policies or that are less general would be low. Therefore, by keeping track of the generated histories and generalizing them in terms of continuation sets, one can identify valuable abstractions and utilize them to improve the learning performance. For efficiency and scalability, this must be accomplished without storing and processing all state-action trajectories since the beginning of learning. Also, note that between successive iterations existing abstractions are not expected to change drastically. Therefore, instead of explicitly creating CTSs first and then constructing the corresponding sequence tree each time, it is more preferable and practical to build it directly in an incrementally manner. For this purpose, we propose a modified version of the sequence tree.

Definition 4.1.12 (Extended Sequence Tree) *An* extended sequence tree *is a tuple $\langle N, E\rangle$ where N is the set of nodes and E is the set of edges. Each node represents a unique action sequence; the root node, denoted by $\emptyset$, represents the empty action set. If the action sequence of node q can be obtained by appending action a to the action sequence represented by node p, then p is connected to q by an edge with label $\langle a, \psi\rangle$; it is denoted by the tuple $\langle p, q, \langle a, \psi\rangle\rangle$. ψ is the eligibility value of the edge and indicates how frequently the action sequence of q is executed.*

Furthermore, q holds a list of tuples $\langle s_1, \xi_{s_1}, R_{s_1}, \ldots, \langle s_k, \xi_{s_k}, R_{s_k}\rangle\rangle$ stating that action a can be chosen at node p if current state observed by the agent is in $\{s_1, \ldots, s_k\}$. $\{s_1, \ldots, s_k\}$ is called the continuation set of node q and denoted by $cont_q$. R_{s_i} is the expected total cumulative reward that the agent can collect by selecting action a at state s_i after having executed the sequence of actions represented by node p. ξ_{s_i} is the eligibility value of state s_i at node q and indicates how frequently action a is actually selected at state s_i. ■

An extended sequence tree is basically an adaptation of sequence tree that contains *additional eligibility and reward attributes* to keep statistics about the represented abstractions; the additional attributes allows discrimination of frequent sequences with high expected reward.

A π-history $h = s_1 a_1 r_2 \ldots r_t s_t$ can be added to an extended sequence tree T by invoking Algorithm 6. Similar to the CONSTRUCT procedure presented in Algorithm 3, ADD-HISTORY starts from the the root node of the tree and follows edges according to the action sequence of the history. Initially, the active node of T is the root node. At step i, if the active node has a child node n which is connected by an edge with label $\langle a_i, \psi \rangle$ then

(i) ψ is incremented to reinforce the eligibility value of the edge,

(ii) if node n contains a tuple $\langle s_i, \xi_{s_i}, R_{s_i} \rangle$ then ξ_{s_i} is incremented to reinforce the eligibility value of the state s_i, and R_{s_i} is set to $R_i = r_{i+1} + \gamma r_{i+2} + \cdots + \gamma^{t-i-1} r_t$ if R_i is greater than the existing value. R_i denotes the discounted cumulative reward obtained by the agent upon following h starting from step i. Otherwise a new tuple $\langle s_i, 1, R_i \rangle$ is added to node n.

If the active node does not have such a child node, then a new node n containing the tuple $\langle s_i, 1, R_i \rangle$ is created and connected to the active node by an edge with label $\langle a_i, 1 \rangle$. In both cases, n becomes the active node. When h is added to the extended sequence tree, only the nodes representing the prefixes of the action sequence of h are modified and associated attributes are updated in support of observing such sequences.

In order to identify and store useful abstractions, based on the sequence of states, actions and rewards observed by the agent during a specific period of time (such as throughout an episode, or between reward peaks in case of non-episodic tasks), a set of probable π^*-histories are generated using Algorithm 5 and added to the extended sequence tree using Algorithm 6. Then, the eligibility values of edges are decremented by a factor of $0 < \psi_{decay} < 1$, and eligibility values in the tuples of each node are decremented by a factor of $0 < \xi_{decay} \leq 1$. For an action sequence σ that is frequently used, edges on the path from the root node to the node representing σ and tuples corresponding to the visited states in the nodes over that path would have higher eligibility values since they are incremented each time a π-history with action sequence σ is added to the tree; whereas they would decay to 0 for sequences that are used

Algorithm 6 Algorithm for adding a π-history to an extended sequence tree.

```
procedure ADD-HISTORY(h,T)
    $h$ is a $\pi$-history of the form $s_1 a_1 r_2 \ldots r_t s_t$.
    $R[t] = 0$    ▷ Calculate discounted cumulative rewards obtained by the agent.
    for $i = t - 1$ to 1 do
        $R[i] = r_i + \gamma R[i+1]$
    end for
    $current$ =root node of $T$    ▷ The active node is initially the root node.
    for $i = 1..t - 1$ do
        if ∃ a node $n$ such that $current$ is connected to $n$ by an edge with label $\langle a_i, \psi \rangle$ then
            Increment $\psi$.    ▷ Reinforce the eligibility value of the edge.
            if $n$ contains a tuple $\langle s_i, \xi_{s_i}, R_{s_i} \rangle$ then
                Increment $\xi_{s_i}$.    ▷ Reinforce the eligibility value of state $s_i$ at node $n$.
                $R_{s_i} = \max(R_{s_i}, R[i])$    ▷ Update the expected discounted cumulated reward.
            else
                Add a new tuple $\langle s_i, 1, R[i] \rangle$ to node $n$.
            end if
        else
            Create a new node $n$ containing the tuple $\langle s_i, 1, R[i] \rangle$.
            Connect $current$ to $n$ by an edge with label $\langle a_i, 1 \rangle$.
        end if
        $current = n$
    end for
end procedure
```

Algorithm 7 Algorithm for updating extended sequence tree T.

```
procedure UPDATE-SEQUENCE-TREE($T$,$e$)    ▷ $e$ is the history of events observed by the agent during a specific period of time.
    $H$=GENERATE-PROBABLE-HISTORIES($e$)
    for all $h \in H$ do    ▷ Add each generated $\pi$-history to $T$.
        ADD-HISTORY($h$,$T$)
    end for
    UPDATE-NODE(root node of $T$)    ▷ Traverse and update the tree.
end procedure

procedure UPDATE-NODE($n$)
    Let $E$ be the set of outgoing edges of node $n$.
    for all $e = \langle n, n', \langle a_{n'}, \psi_{n,n'} \rangle \rangle \in E$ do    ▷ for each outgoing edge
        $\psi_{n,n'} = \psi_{n,n'} * \psi_{decay}$    ▷ Decay the eligibility value of the edge.
        if $\psi_{n,n'} < \psi_{threshold}$ then    ▷ Prune the edge if its eligibility value is below $\psi_{threshold}$.
            Remove $e$ and the subtree rooted at $n'$.
        else
            UPDATE-NODE($n'$)    ▷ Recursively update the child node $n'$.
            if tuple list of $n'$ is empty then    ▷ Prune the edge if its continuation set is empty.
                Remove $e$ and the subtree rooted at $n'$.
            end if
        end if
    end for
    for all $t = \langle s_i, \xi_{s_i}, R_{s_i} \rangle$ in tuple list of $n$ do    ▷ for each tuple in $n$
        $\xi_{s_i} = \xi_{s_i} * \xi_{decay}$    ▷ Decay the eligibility value of the tuple.
        if $\xi_{s_i} < \xi_{threshold}$ then    ▷ Prune the tuple if its eligibility value is below $\xi_{threshold}$.
            Remove $t$ from the tuple list of $n$.
        end if
    end for
end procedure
```

less. This has an overall effect of supporting valuable sequences that are encountered frequently. If the eligibility value of an edge is very small (less than a given threshold $\psi_{threshold}$) then this indicates that the action sequence represented by the outgoing node is rarely executed by the agent and consequently the edge and the subtree below it can be removed from the tree to preserve compactness. Similarly, if the eligibility value of a tuple $\langle s, \xi, R \rangle$ in a node n is very small (less than a given threshold $\xi_{threshold}$) then it means that the agent no longer observes state s frequently after executing the action sequence on the path from the root node to node n. Such tuples can also be pruned to reduce the size of the continuation sets of the nodes. After performing these operations, the resulting extended sequence tree represents recent useful CTSs in a compact form. The entire process is presented in Algorithm 7.

An extended sequence tree is used to select actions similar to a sequence tree. However, a prior probability distribution to determine branching is not available as in the case of sequence trees. Therefore, when there are multiple viable actions, i.e. current state observed by the agent is contained in continuation sets of several child nodes of the active node of the tree, the edge to follow is chosen dynamically based on the properties of the child nodes. One important consequence of this situation is that, instead of a single CTS, the agent, in effect, starts with a set of CTSs which initially contains all those represented by the extended sequence tree; it selects a subset of CTSs from this set that are compatible with the observed history of events and follows them concurrently executing their common action. This enables the agent to broaden the regions of the state space where the CTSs are applicable and results in longer execution patterns than that may be attained by employing a single CTS (since it covers are smaller region of the state space). One possible option for branching, which we opted for the experiments presented in the next section, is to apply an ϵ-greedy method; with $1-\epsilon$ probability the agent selects the edge connected to the child node containing the tuple with highest discounted cumulative reward for the current state, otherwise one of them is chosen randomly.

Since the extended sequence tree, and the associated mechanism defined above to select which actions to execute based on its structure, is not an option in the traditional sense, but rather a single meta-abstraction that incorporates a set of evolving CTSs, it is not feasible to directly integrate it into the reinforcement learning framework by extending the value functions as in the case of SMDP and options framework.

Algorithm 8 Reinforcement learning with extended sequence tree.

```
$T$ is an extended sequence tree with $root$ node only.
repeat
    Let $current$ denote the active node of $T$.
    $current = root$    ▷ $current$ is initially set to the root node of $T$.
    Let $s$ be the current state.
    $h = s$    ▷ Episode history is initially set to the current state.
    repeat    ▷ for each step
        if $current \neq root$ then    ▷ Continue action selection from the current node of $T$.
            Let $N = \{n_1, \ldots, n_k\}$ be the set of child nodes of $current$ which contain $s$ in their continuation sets.
            Select $n_i$ from $N$ with sufficient exploration.
            Let $\langle a_{n_i}, \psi_{n_i} \rangle$ be the label of the edge connecting $current$ to $n_i$.
            $a = a_{n_i}$
            $current = n_i$    ▷ Advance to node $n_i$.
        else
            $current = root$
            Let $N = \{n_1, \ldots, n_k\}$ be the set of child nodes of the root node of $T$ which contain $s$ in their continuation sets.
            if $N$ is not empty and with probability $p_{sequence}$ then
                Select $n_i$ from $N$ with sufficient exploration.    ▷ Initiate action selection using the extended sequence tree.
                Let $\langle a_{n_i}, \psi_{n_i} \rangle$ be the label of the edge connecting $root$ to $n_i$.
                $a = a_{n_i}$
                $current = n_i$    ▷ Advance to node $n_i$.
            else
                Choose $a$ from $s$ using the underlying RL algorithm.
            end if
        end if
        Take action $a$, observe $r$ and next state $s'$
        Update state-action value function using the underlying RL algorithm based on $s, r, a, s'$.
```

```
        Append $r, a, s'$ to $h$.    ▷ Update the observed history of events.
        $s = s'$    ▷ Advance to next state.
        if $current \neq root$ then    ▷ Check whether action selection can continue from the active node or not.
            Let $N = \{n_1, \ldots, n_k\}$ be the set of child nodes of $current$ which contain $s$ in their continuation sets.
            if $N$ is empty then
                $current = root$    ▷ Action selection using the extended sequence tree cannot continue from the current state.
            end if
        end if
    until $s$ is a terminal state
    UPDATE-SEQUENCE-TREE($T, h$)
until a termination condition holds
```

Instead, a flat policy approach can be used and the proposed method can be integrated into any regular reinforcement learning algorithm such as Q-learning, $Sarsa(\lambda)$, etc, by triggering the action sequence of the extended sequence tree (and consequently represented CTSs) with a given probability, $p_{sequence}$, and reflecting the action selections to the underlying reinforcement learning algorithm for value function updates. This leads to the learning model given in Algorithm 8 that, based on the generated extended sequence tree and treating it as meta-heuristic to guide any underlying reinforcement learning algorithm, discovers and utilizes useful temporal abstractions during the learning process.

4.2 Experiments

We applied the method described in Section 4.1.4 to different reinforcement learning algorithms and compared its effect on performance on three test domains described in Chapter 3. The performance of the proposed approach is also compared with the acQuire-macros algorithm of McGovern [31, 32] on a simple grid world problem for which existing results are available.

After analyzing the outcomes of a set of initial experiments to determine the optimal values of parameters involved in the learning process, a learning rate of $\alpha =$

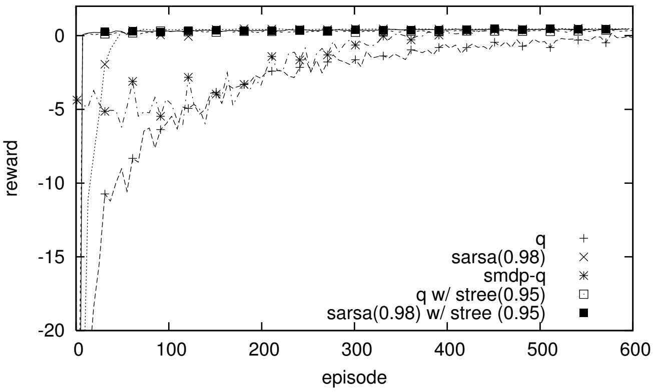


Figure 4.8: Results for the six-room maze problem.

0.125 is used, and λ is taken as 0.98 and 0.90 in the $Sarsa(\lambda)$ algorithm for the six-room maze and taxi problems, respectively. Initially, Q-values are set to 0 and ϵ-greedy action selection mechanism is used with $\epsilon = 0.1$, where action with maximum Q-value is selected with probability $1 - \epsilon$ and a random action is selected with probability ϵ. The reward discount factor is set to $\gamma = 0.9$. For the SMDP Q-learning algorithm [6], we implemented hand-coded macro-actions. In the six-room maze problem, these macro-actions move the agent from any cell in a room, except the bottom right one which contains the goal location, to one of two doorways that connect to neighboring rooms in minimum number of steps; in the taxi problem, they move the agent from any position to one of predefined locations in shortest possible way. In all versions of the taxi problem, the number of predefined locations was 4. Therefore, there were 4 such macro-actions each corresponding to one of these locations. All results are averaged over 50 runs. Unless stated otherwise, while building the sequence tree ψ_{decay}, ξ_{decay}, and eligibility thresholds are taken as 0.95, 0.99, and 0.01, respectively. The sequence tree is generated during learning without any prior training session, and at each time step processed with a probability of $p_{sequence} = 0.3$ to run the represented set of abstractions. At decision points, actions are chosen ϵ-greedily based on reward values associated with the tuples.

4.2.1 Comparison with Standard RL Algorithms

We first applied the sequence tree method to standard reinforcement learning algorithms on six-room maze and three different versions of the Taxi domain on 5×5, 8×8 and 12×12 grids. Figure 4.8 and Figure 4.9 show the progression of the reward

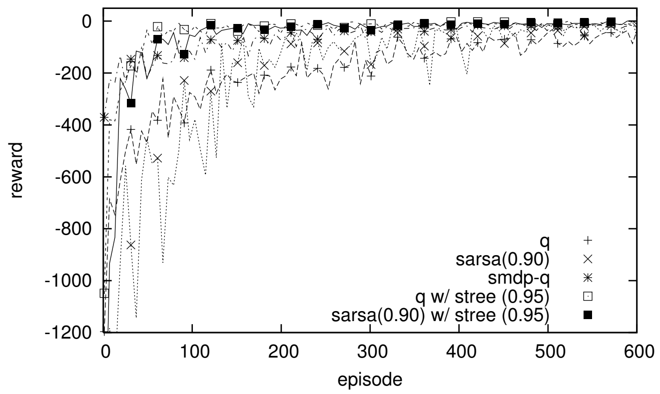


Figure 4.9: Results for the 5 × 5 taxi problem with one passenger.

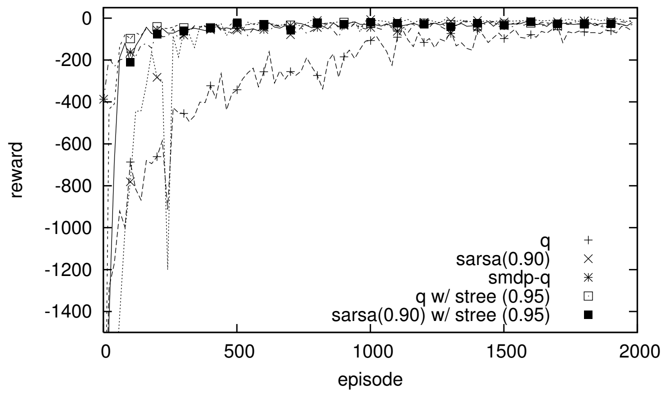


Figure 4.10: Results for the 8 × 8 taxi problem with one passenger.

obtained per episode for the six-room maze and 5 × 5 taxi problem with one passenger, respectively. As apparent from the learning curves, both $Sarsa(\lambda)$ and SMDP Q-learning converge faster compared to regular Q-learning. When the sequence tree method is applied to Q-learning and Sarsa(λ), the performance of both algorithms improve substantially. In SMDP Q-learning algorithm the agent receives higher negative rewards when options that move the agent away from the goal are erroneously selected at the beginning; SMDP Q-learning needs to learn which options are optimal. The effect of this situation can be seen in the six-room maze problem, where it causes SMDP Q-learning to converge slower compared to Sarsa(λ). On the contrary, algorithms that employ sequence tree start to utilize shorter sub-optimal sequences immediately in the initial stages of learning; this results in more rapid convergence with respect to hand-coded options.

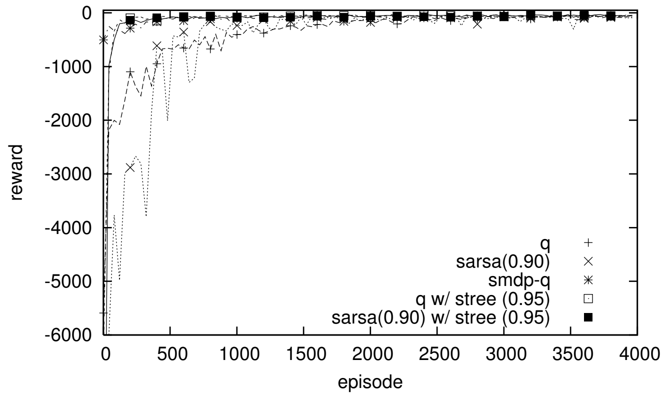


Figure 4.11: Results for the 12 × 12 taxi problem with one passenger.

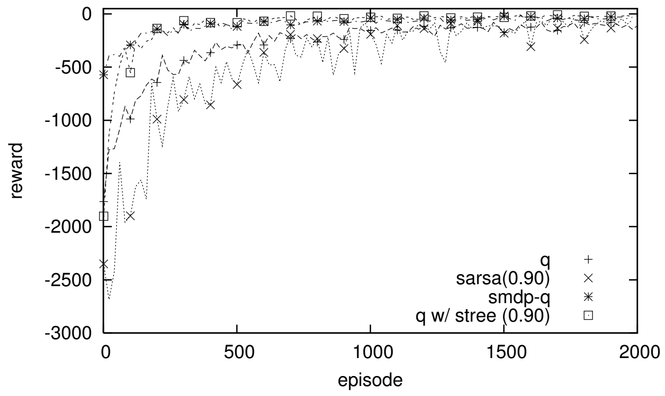


Figure 4.12: Results for the 5 × 5 taxi problem with two passengers.

4.2.2 Scalability

Results for the 8 × 8 and 12 × 12 taxi problems with one passenger, which have larger state spaces and contain more obstacles, are given in Figure 4.10 and Figure 4.11. In both cases, we observed similar learning curves as in the 5 × 5 version but performance improvement is more evident.

In the taxi problem, the number of situations to which subtasks can be applied increases with the number of passengers to be transported. This also applies to other problems; a new parameter added to the state representation leads to a larger (usually exponentially) state space, and consequently the number of instances of subtasks that involve only a subset of variables also increase. Therefore, more significant improvement in learning performance is expected when subtasks can be utilized effectively. Results for the 5 × 5 taxi problem with multiple passengers (from two up to four) are presented in Figure 4.12 and Figure 4.13. Note that the performance of the algorithms

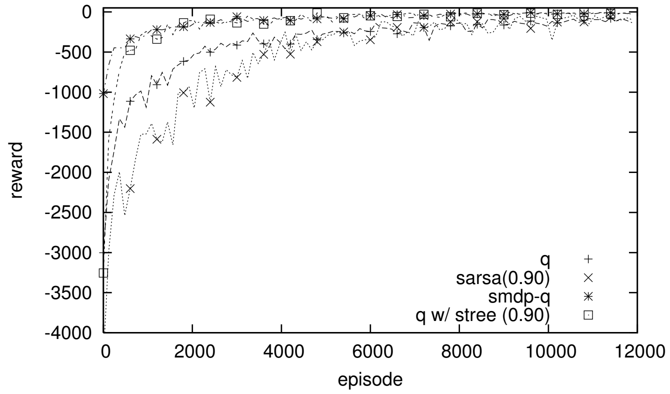


Figure 4.13: Results for the 5 × 5 taxi problem with four passengers.

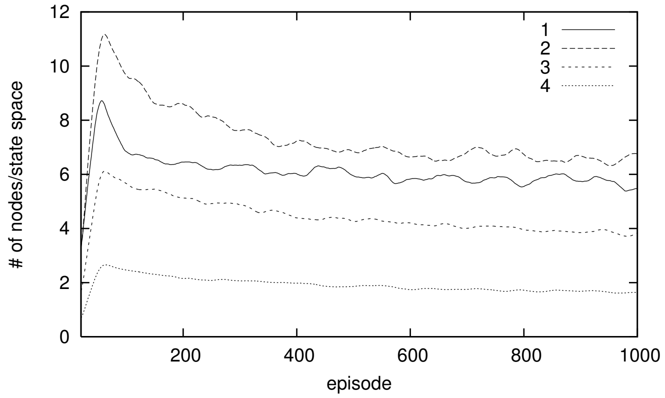
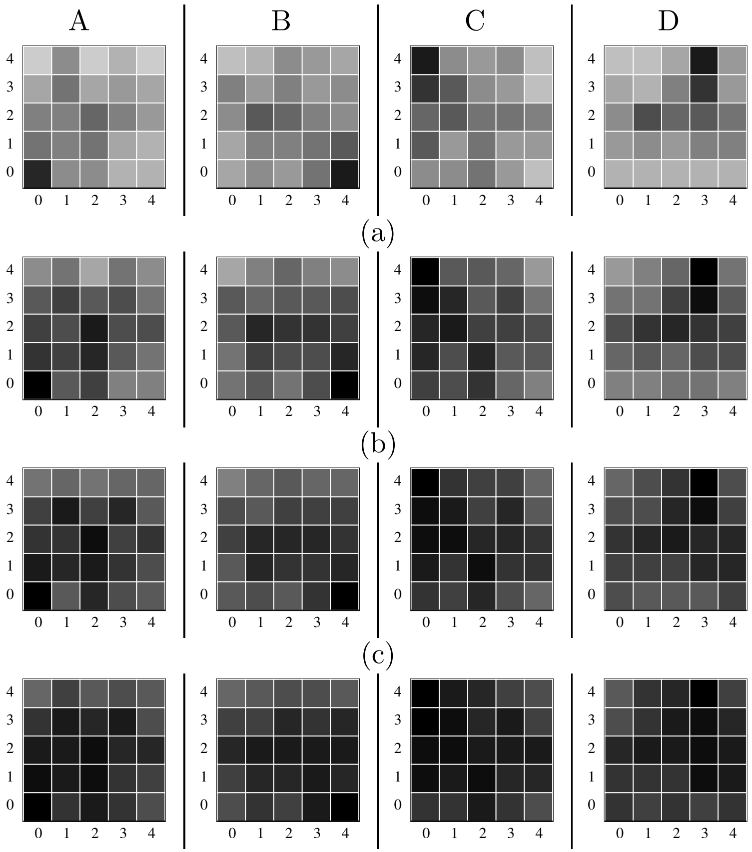


Figure 4.14: The average size of the sequence trees with respect to the size of state space for 5 × 5 taxi problem with one to four passengers.



(a) (b) (c) (d)

Figure 4.15: State abstraction levels for four predefined locations (A to D from left to right) in 5×5 taxi problem with one passenger after 50, 100, 150 and 200 (a-d) episodes. Darker colors indicate higher abstraction.

that do not make use of abstractions degrade rapidly as the number of passengers increases, and consequently common subtasks become more prominent. The results also demonstrate that the proposed method is effective in identifying solutions in such cases. Figure 4.14 shows the average size of the sequence tree (i.e., number of nodes) with respect to the size of the state space for different number of passengers in the 5×5 taxi problem. Note that as new passengers are added, the state space increases exponentially in the number of passengers, i.e., multiplies with the number of predefined locations, whereas the relative space requirement decreases indicating that the proposed method scales well in terms of space efficiency.

4.2.3 Abstraction Behavior

The content of the sequence tree is mostly determined by the π-histories generated based on the experiences of the agent. Due to the complex structure of the tree it is not feasible to directly give a snapshot of it to expose what kind of abstractions it contains. Rather, we opt to take a more comprehensible and qualitative approach, and at various stages of the learning process examined how successful the tree is in generating abstractions belonging to similar subtasks. Note that, in the taxi problem, the agent must first navigate to the location of the passenger to be picked-up independent of the passenger's destination; for the regions of the state space that differ only in the destination of the passenger the subtask to be solved is the same and one expects the agent to learn similar action sequences within these regions. In order to measure this, we conducted a set of experiments on 5×5 taxi problem with one passenger. At various stages of the learning process, we calculated the ratio of the number of co-occurrences of states that only differ in the variable corresponding to the destination of the passenger in the tuple lists of the nodes. For a given state, this ratio must be close to 1 if action sequences that involve that state are also applicable to other states having different destinations for the passenger. The results of the experiments are presented in Figure 4.15. Each of the four columns denote the case in which the passenger is at a specific predefined location from A to D, and each row shows the results for the sequence tree generated after a certain number of episodes (at every 50 episodes starting from 50 up to 200 episodes). The intensity of color in each cell indicates the ratio corresponding to the state in which the agent is positioned at that cell; black represents 1, white represents 0 and the intensity of intermediate values decrease linearly. One can observe that after 50 episodes all cells have non-white intensities which get darker with increasing number of episodes and eventually turn into black, i.e. the ratios converge to 1, which means that the sequence tree is indeed successful in identifying abstractions that cover multiple instances of the same or similar subtasks starting from early stages of the learning.

4.2.4 Effects of the Parameters

Apart from the performance, another factor that needs to be considered is the structure of the sequence tree, its involvement in the learning process, and their overall effect.

The structure of the sequence tree directly depends on the eligibility and decay parameters that regulate the amount of information to be retained in the sequence tree. We found out that from these parameters the most prominent one that causes

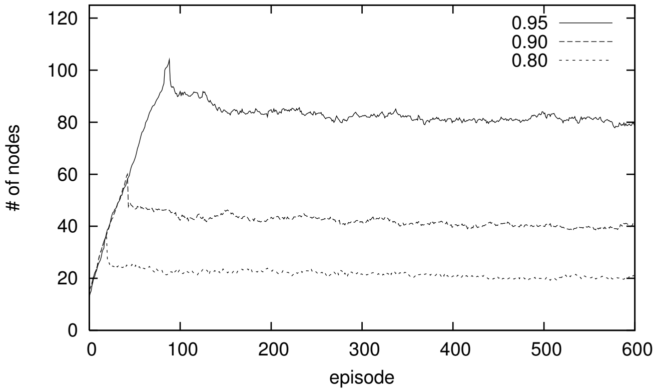


Figure 4.16: The average size of sequence trees for different ψ_{decay} values in six-room maze problem.

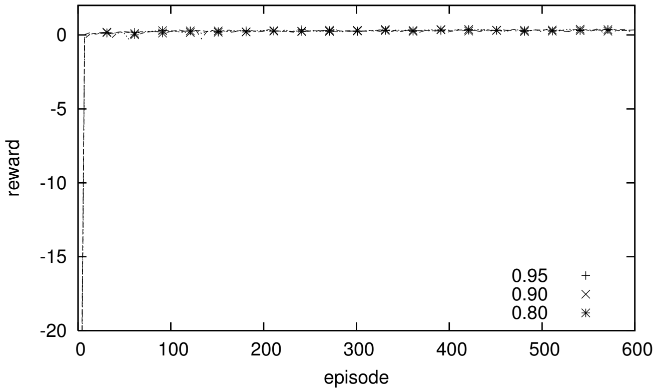


Figure 4.17: Results for different ψ_{decay} values in six-room maze problem.

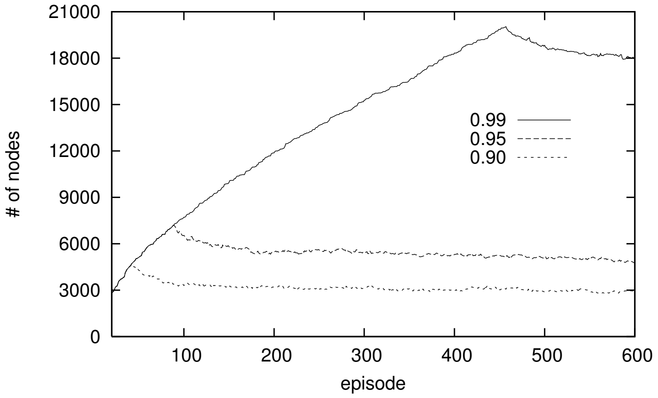


Figure 4.18: The average size of sequence trees for different ψ_{decay} values in 5×5 taxi problem with one passenger.

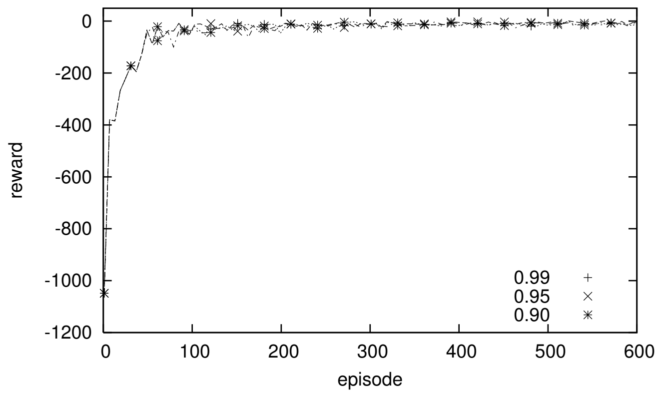


Figure 4.19: Results for different ψ_{decay} values in 5×5 taxi problem with one passenger.

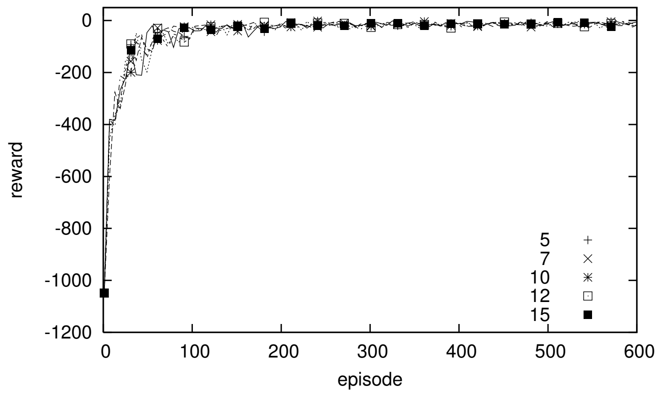


Figure 4.20: Q-learning with sequence tree for different maximum history lengths in the 5×5 taxi problem with one passenger.

the most significant difference is the edge eligibility decay, ψ_{decay}. The results for various ψ_{decay} presented in Figure 4.16 and Figure 4.18 show that the size of the sequence tree decreases considerably for both six-room maze and taxi problems as ψ_{decay} gets smaller. This is due to the fact that only more recent and commonly used sequences have the opportunity to be kept in the tree and others get eliminated. Note that, since such sequences are more beneficial for the solution of the problem, different ψ_{decay} values show almost indistinguishable behavior (Figure 4.17 and Figure 4.19). Hence, by selecting ψ_{decay} parameter appropriately it is possible to reduce memory requirements without degrading the performance.

Other than lowering ψ_{decay} value, another possible way to reduce the size of the extended sequence tree is to limit the length of the probable π^*-histories that are added to it. After generating probable histories using Algorithm 5, instead of the entire history $h = s_1 a_1 r_2 \ldots r_t s_t$, one can only process h up to l_{max} steps (i.e.,

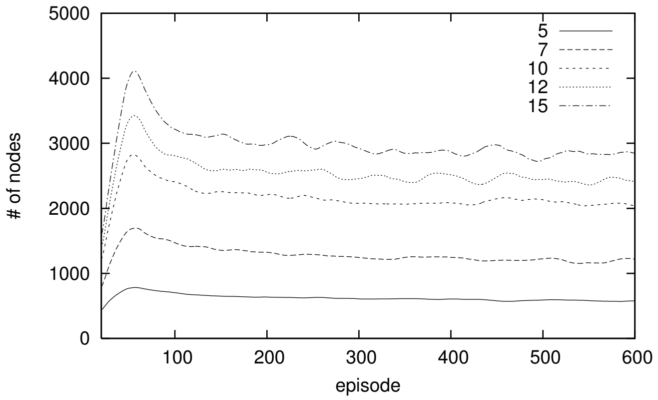


Figure 4.21: Average size of the sequence trees for different maximum history lengths in the 5×5 taxi problem with one passenger.

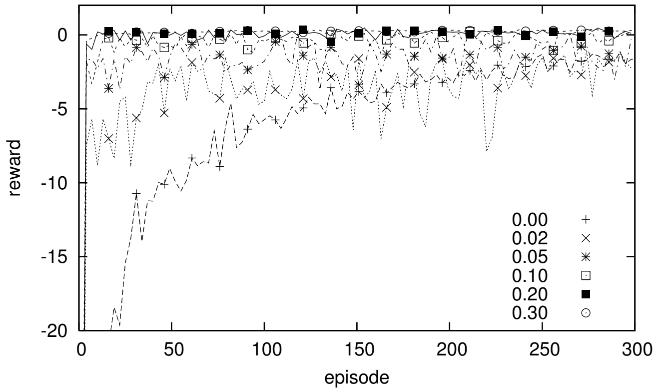


Figure 4.22: The effect of $p_{sequence}$ in the six-room maze problem.

$s_1 a_1 r_2 \dots r_{l_{max}+1} s_{l_{max}+1}$) in Algorithm 6 by omitting actions taken after l_{max} steps. This corresponds to having CTSs of length at most l_{max}, and therefore maximum depth of the extended sequence tree will be bounded by l_{max}. Since shorter abstractions are prefixes of longer abstractions, and applicability of abstractions decreases with the increase in length; the performance of the method is not expected to change drastically when pruning is applied. The learning curves of sequence tree based-Q-learning on 5×5 taxi problem for different l_{max} values is presented in Figure 4.20, which conforms to the expectation and demostrates that it is further possible to prune the sequence tree without compromising performance (Figure 4.21).

The involvement of the sequence tree in the learning process depends on the value of $p_{sequence}$, i.e. the probability that it is employed in the action selection mechanism. With decreasing $p_{sequence}$ values it is expected that the performance of the proposed method to degrade and converge to that of which does not make use of the sequence

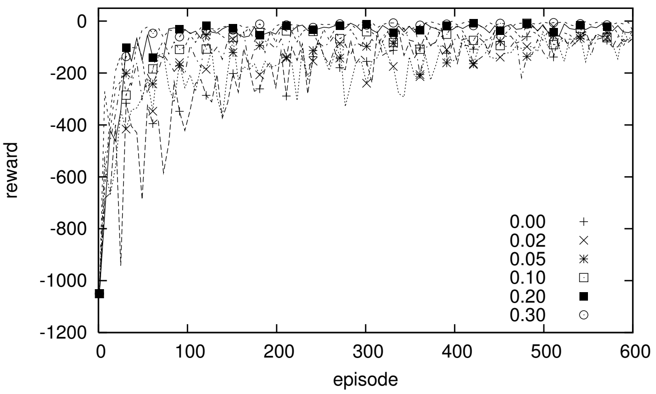
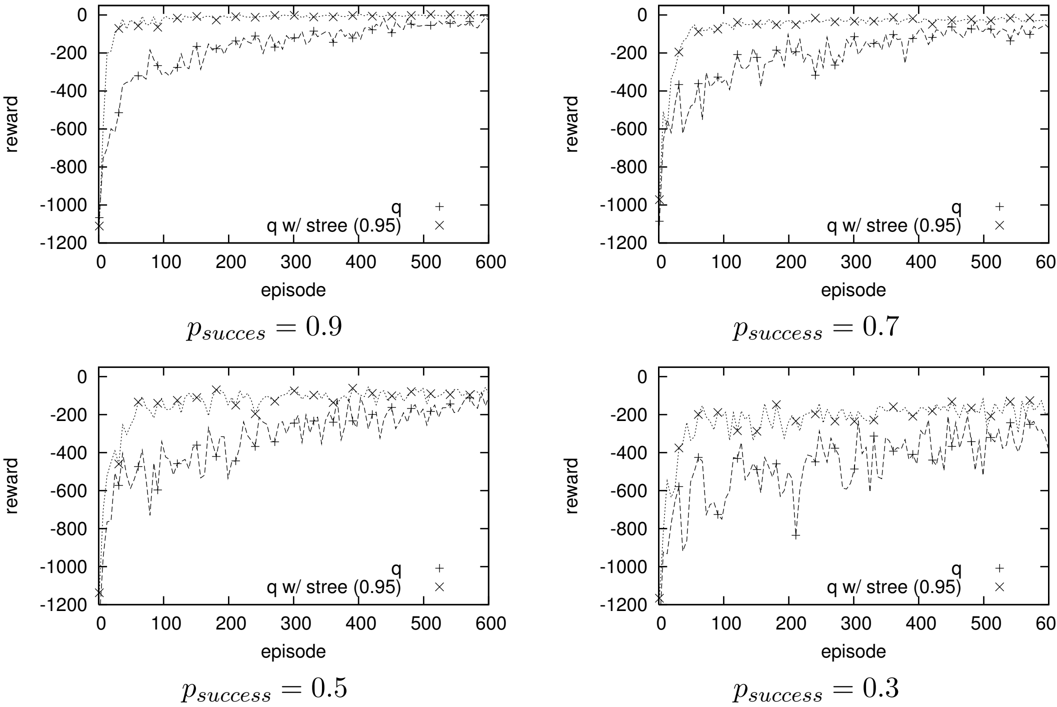


Figure 4.23: The effect of $p_{sequence}$ in the 5×5 taxi problem with one passenger.

tree. The results of experiments with different $p_{sequence}$ values given in Figure 4.22 and Figure 4.23 for both problems confirm this expectation. When $p_{sequence}$ is very low, extended sequence tree is not effective because the CTSs represented by the tree, and therefore discovered abstractions, are not utilized sufficiently. Experiments in largers taxi problems, which are not included here, also exhibit similar results. In general, $p_{sequence}$ must be determined in a problem specific manner to balance exploration of the underlying learning algorithm and exploitation of the abstractions. Lower $p_{sequence}$ values reduce exploitation; on the contrary high $p_{sequence}$ values hinder exploration. For the six-room maze and taxi problems (and also for the keepaway problem studied next), a moderate value of $p_{sequence} = 0.3$ appears to result in sufficient exploitation without significantly affecting the final performance. A similar effect is expected in most of the real world problems. Although not studied in this work, it is also possible to change $p_{sequence}$ dynamically and increase it as learning progresses in order to favor exploitation and further take advantage of useful abstractions.

4.2.5 Effect of Non-determinism in the Environment

In order to examine how the non-determinism of the environment affect the performance, we conducted a set of experiments by changing $p_{success}$, i.e. the probability that movement actions succeed, in the taxi domain. Non-determinism increases with decreasing $p_{success}$. The results are presented in Figure 4.24. Except more fluctuation is observed in the received reward due to increased non-determinism, the proposed method preserves its behavior and methods that employ sequence tree consistently learn in less number of steps compared to their regular counterparts.



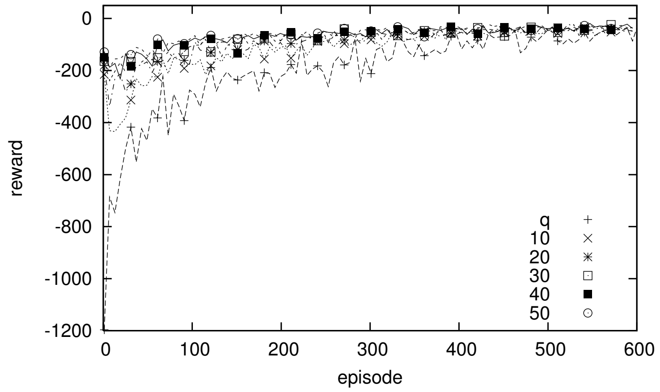
$p_{succes} = 0.9$ $p_{success} = 0.7$

$p_{success} = 0.5$ $p_{success} = 0.3$

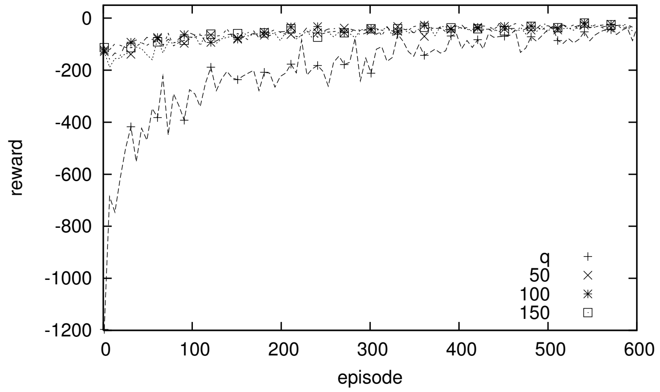
Figure 4.24: Results with different levels of non-determinism in actions.

4.2.6 Quality of the Discovered Abstractions

The results given so far demonstrate the on-line performance of the method, i.e. while it is continuously evolving and abstractions that it represent change dynamically. Since an off-line pre-learning stage is not involved, this is a more suitable and efficient approach considering that the aim of the whole process is to solve the problem and learn an optimal behavior. However, due to the fact that the abstractions represented by the sequence tree are not fixed, this makes it difficult to assess their quality and raises the question of whether the proposed method creates good policies efficiently or not. For this purpose, we let the sequence tree integrated into standard Q-learning algorithm to evolve for different number of episodes, and then used the resulting trees as a single option, i.e. primitive actions plus sequence tree as pseudo option, in a separate run using SMDP Q-learning algorithm. This allows us to isolate and observe the effect of discovered abstractions in a controlled manner. The results of the experiments for 5 × 5 taxi problem with one passenger are presented in Figure 4.25. The learning curves demonstrate that even sequence trees in their early stage of development accommodate useful abstractions. The sequence tree generated after 20 episodes is quite effective and leads to a substantial improvement. The performance of SMDP Q-



(a)



(b)

Figure 4.25: SMDP Q-learning algorithm when the previously generated sequence tree is employed as a single option. Each number in the key denotes the number of episodes used to generate the tree. (a) 10-50 episodes at every 10 episodes, and (b) 50-150 episodes at every 50 episodes.

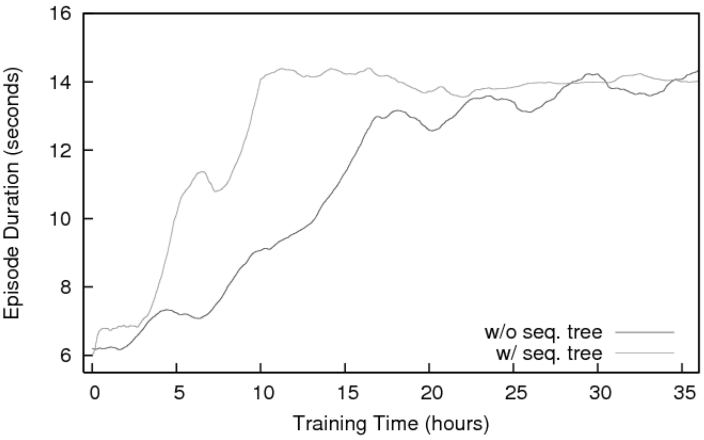


Figure 4.26: Results for the keepaway problem.

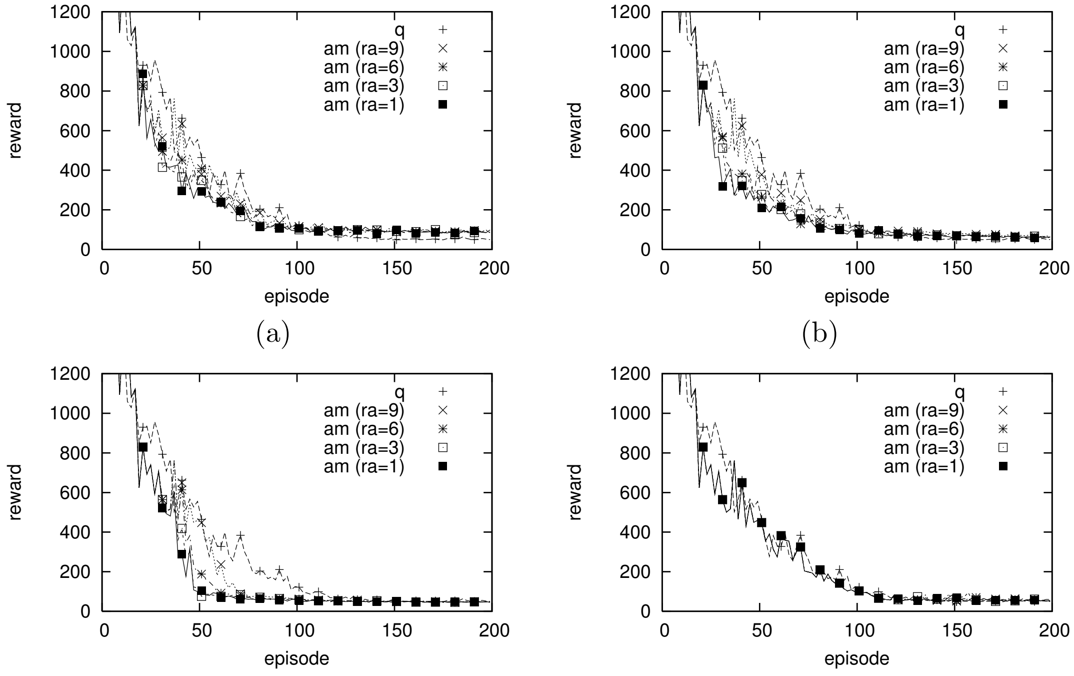
learning increases with the number of episodes used to generate the sequence tree and then saturates (Figure 4.25 (b)). This indicates that the sequence tree based approach is successful in finding meaningful abstractions, improving them and preserving the most useful ones.

4.2.7 Results for the Keepaway Problem

We have extensively analyzed different aspects of the proposed method under sample domains of moderate size. In order to test how our approach performs in a challenging machine learning task, we implemented the method described in [51] using the publicly available keepaway player framework [49] and integrated sequence tree to their algorithm. We conducted the experiments with 3 keepers and 2 takers playing in a $20m \times 20m$ region and the learners were given noiseless visual sensory information. The state representation used by the learner keeepers is a mapping of full soccer state to 13 continuous values that are computed based on the positions of the payers and center of the playing region. In order to approximate the table of Q-values 32 uniformly distributed tilings are overlaid which together form a feature vector of length 416 mapping continuous space into a finite discrete one. By using open-addressed hashing technique size of the state space is further reduced and only needed parts are stored. Since the feature vector spans an extremely large space and hashing does not preverve locality, i.e. states with similar feature vectors get mapped to unrelated addressed, an alternative discretization method is needed while generating the sequence tree. For this purpose, we chose 5 variables out of 13 state variables that are shown to display similar results to those obtained when all state variables are utilized. Each variable is then discretized into 12 classes and together used to represent states while building and employing the sequence tree. The immediate reward received by each agent after selecting a high level skill is the number of primitive time steps that elapsed while following the action. The takers use a hand-coded policy implemented in [49]: A taker either tries to (a) catch the ball if he is the closest or second closest taker to the ball or if no other taker can get to the ball faster than he does, or (b) position himself in order to block a pass from the keeper with the largest angle with vertex at the ball that is clear of takers. The learning parameters were as follows: $\gamma = 1.0$, $\alpha = 0.125$, $\epsilon = 0.01$, $\lambda = 0.9$, $p_{sequence} = 0.3$, $\psi_{decay} = 0.95$, $\xi_{decay} = 0.95$, and $\psi_{threshold} = \xi_{threshold} = 0.01$. The learning curves showing the progression of episode time with respect to training time are presented in Figure 4.26. SMDP Sarsa(λ) algorithm with sequence tree converges faster, achieving an episode time of around 14 seconds in almost one third of the time required by its regular counterpart. This data also supports that the proposed sequence tree based method is successful in utilizing useful abstractions and improve the learning performance in more complex domains.

4.2.8 Comparison with acQuire-macros Algorithm

Finally, we compared the performance of our method with the *acQuire-macros* algorithm of McGovern [31, 32], which is the starting point of the work presented in this manuscript; and it is also based on CTSs. For a fair comparison, the experiments are conducted on a 20×20 empty grid world problem, one room without any obstacles, which is the domain already studied by McGovern and empirical results are reported in the literature. In this problem, the agent is initially positioned at the lower left corner of the grid and tries to reach the upper right corner. The action set and dynamics of the environment are same as in the six-room maze problem. The agent receives an immediate reward of 1 when it reaches the goal cell, and 0 otherwise. The discount rate is set to $\gamma = 0.9$, so that the goal state must be reached in as few steps as possible to maximize the total discounted reward. In order to comply with the existing work, a learning rate of $\alpha = 0.05$ and ϵ-greedy action selection with $\epsilon = 0.05$ is used. In order to determine best paremeter setting, we applied acQuire-macros algorithm using various minimum support, minimum running average and minimum sequence length values. The results of the experiments show that as the minimum running



(a) (b)

(c) (c)

Figure 4.27: Results for the acQuire-macros algorithm using minimum support of (a) 0.1, (b) 0.3, (c) 0.6, and (d) 0.9 on 20 × 20 grid world problem. In each figure, the learning curves for different minimum running average values of 1, 3, 6 and 9 are plotted and compared with regular Q-learning. Minimum sequence length is taken as 4.

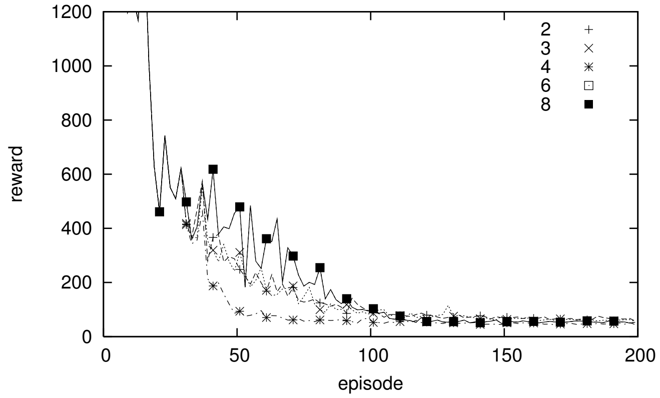


Figure 4.28: Results for the acQuire-macros algorithm using different minimum sequence lengths on 20 × 20 grid world problem. Minimum support and minimum running average are taken as 0.6 and 3, respectively.

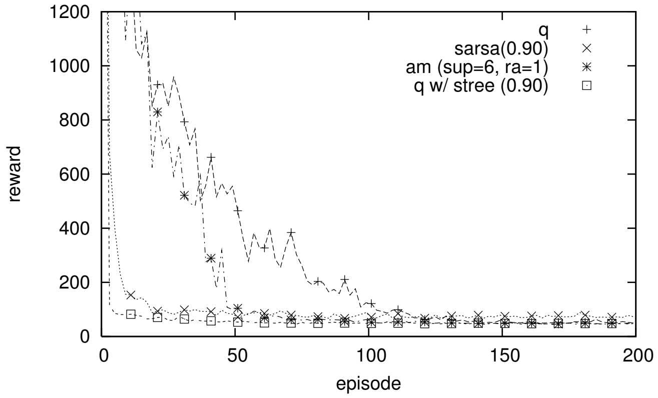


Figure 4.29: Results for the 20 × 20 grid world problem.

average gets smaller the acQuire-macros algorithm converges to optimal policy faster irrespective of the minimum support. A moderate minimum support of 0.6 performs better than a fairly high value of 0.9 that filters out most of the sequences. Lower minimum support values lead to high number of options, and in the extreme fails to converge to optimal behavior. Best result is achieved with minimum sequence length of 4. Figures showing the effect of parameters are presented in Figure 4.27 and 4.28).

Results for various learning algorithms on the 20×20 grid world problem are given in Figure 4.29. Although acQuire-macros performs better than regular Q-learning, it falls behind $Sarsa(\lambda)$. This is due to the fact that options are not created until sufficient number of instances are observed and there is no discrimination between options based on their expected total discounted rewards. The sequence tree based Q-learning algorithm, does not suffer from such problems and shows a fast convergence by making efficient use of abstractions in the problem.

CHAPTER 5

EMPLOYING STATE SIMILARITY TO IMPROVE REINFORCEMENT LEARNING PERFORMANCE

In this chapter, following MDP homomorphism notion proposed by Ravindran and Barto [41, 42], we propose a method to identify states with similar sub-policies without requiring a model of the MDP or equivalence relations, and show how they can be integrated into reinforcement learning framework to improve the learning performance [12, 16, 15]. As in Chapter 4, we first collect history of states, actions and rewards, from which traces of policy fragments are generated. These policy fragments are then translated into a tree structure to efficiently identify states with similar sub-policy. The number of common action-reward sequences is used as a metric to define the similarity between two states. Updates on the action-value function of a state are then reflected to all similar states, expanding the influence of new experiences. Similar to sequence tree based approach discussed in Chapter 4, the proposed method can be treated as a meta-heuristic to guide any underlying reinforcement learning algorithm. The effectiveness of the proposed method is demonstrated by reporting experimental results on two domains, namely six-room maze and various versions of taxi problem. Furthermore, it is compared with other RL algorithms, and a substantial level of improvement is observed on different test cases. Also, although the approaches are different, we present how the performance of our work compared to other option discovery algorithms.

This chapter is organized as follows: Our approach to reinforcement learning with equivalent state update is presented in Section 5.1 on an illustrative example. A method to find similar states during the learning process is described in Section 5.2. We present experimental results in Section 5.3. Finally, in Section 5.4 proposed method is combined with option discovery algorithm described in previous chapter

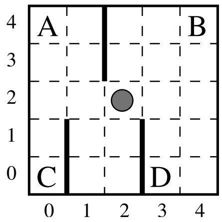
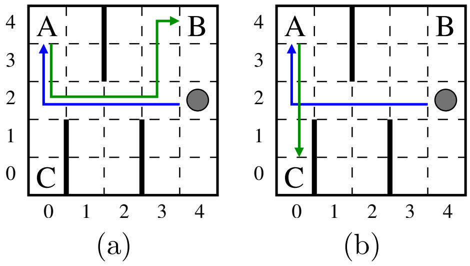


Figure 5.1: 5×5 taxi problem. Predefined locations are labeled with letters A to D.



(a) (b)

Figure 5.2: Two instances of the taxi problem. In both cases, the passenger is situated at location A, but the destinations are different: B in (a) and C in (b).

and the results are presented.

5.1 Reinforcement Learning with Equivalent State Update

Let $L = \{A, B, C, D\}$ be the set of possible locations on the 5×5 version of the Dietterich's taxi problem presented in Figure 5.1 (a). As described in Chapter 3, we represent each possible state by a tuple of the form $\langle r, c, l, d \rangle$, where $r, c \in \{0, 1, 2, 3, 4\}$ denote the row and column of the taxi's position, respectively, $l \in L \cup \{T\}$ denotes the location of the passenger (either one of predefined locations or in case of $l = T$ picked-up by the taxi), and $d \in L$ denotes the destination of the passenger. (r_l, c_l) is the position of the location $l \in L$ on the grid. Let $M_{taxi} = \langle S_{taxi}, A_{taxi}, \Psi_{taxi}, T_{taxi}, R_{taxi} \rangle$ be the corresponding Markov decision process, where S_{taxi} is the set of states, A_{taxi} is the set of actions, $\Psi = S_{taxi} \times A_{taxi}$ is the set of admissible state-action pairs, and $T_{taxi} : \Psi_{taxi} \times S_{taxi} \rightarrow [0, 1]$ and $R_{taxi} : \Psi_{taxi} \rightarrow \Re$ are the state transition and reward functions conforming to the description given above, respectively.

The taxi domain inherently contains subtasks that need to be solved by the agent. For example, consider an instance of the problem in which the passenger is situated at location A and is to be transported to location B as presented in Figure 5.2 (a). Starting at any position (r, c), the agent must first navigate to location A in order to pick up the passenger. Let $\langle r, c \rangle_{ld}$ denote the state tuple $\langle r, c, l, d \rangle \in S_{taxi}$. This navigation subtask can be modeled by using a simpler Markov decision process $M_A = \langle S_A, A_A, \Psi_A, T_A, R_A \rangle$, where

- $S_A = \{\langle r, c\rangle | r, c \in \{0, \dots, 4\}, \langle r, c\rangle \neq \langle r_A, c_A\rangle\} \cup \{s_\Upsilon\}$ is the set of states,
- $A_A = A_{taxi} \cup \{a_\Upsilon\}$ is the set of actions,
- $\Psi_A = \{(\langle r, c\rangle, a) | (\langle r, c\rangle_{AB}, a) \in \Psi_{taxi}, (r, c) \neq (r_A, c_A)\} \cup \{(s_\Upsilon, a_\Upsilon)\}$ is the set of admissible state-action pairs,
- $T_A : \Psi_A \times S_A \rightarrow [0, 1]$ defined as

  $$\begin{aligned} T_A(\langle r, c\rangle, a, \langle r', c'\rangle) &= T_{taxi}(\langle r, c\rangle_{AB}, a, \langle r', c'\rangle_{AB}) \\ T_A(\langle r, c\rangle, a, s_\Upsilon) &= T_{taxi}(\langle r, c\rangle_{AB}, a, \langle r_A, c_A\rangle_{AB}) \\ T_A(s_\Upsilon, a_\Upsilon, s_\Upsilon) &= 1 \end{aligned}$$

  is the state transition function, and
- $R_A : \Psi_A \rightarrow \Re$, such that

  $$\begin{aligned} R_A(\langle r, c\rangle, a) &= R_{taxi}(\langle r, c\rangle_{AB}, a) \\ R_A(s_\Upsilon, a_\Upsilon) &= 0 \end{aligned}$$

  is the expected reward function.

In M_A, the state s_Υ is an absorbing (in other words, sub-goal or termination) state, corresponding to state $\langle r_A, c_A\rangle_{AB}$ in M_{taxi}, with one action a_Υ that transitions to itself with probability one. Once control reaches s_Υ, it stays there. Note that, for every state $\langle r, c\rangle_{AB} \in S$ such that $r, c \in \{0, \dots, 4\}$ and $(r, c) \neq (r_A, c_A)$, there exists a state $\langle r, c\rangle \in S'$ with exactly the same state-transition and reward structure, and all other states in S are mapped to the absorbing state, s_Υ. By defining a surjection from Ψ_{taxi} to Ψ_A, it is possible to transform M_{taxi} to M_A, and such a transformation is called a *partial MDP homomorphism* [41].

Definition 5.1.1 *Let* $M = \langle S, A, \Psi, T, R\rangle$ *and* $M' = \langle S' \cup \{s_\Upsilon\}, A' \cup \{a_\Upsilon\}, \Psi', T', R'\rangle$, *such that* $s_\Upsilon \notin S'$ *and* $a_\Upsilon \notin A'$, *be two Markov decision processes.* M *is partially homomorphic to* M' *if there exists a surjection* $f : S \rightarrow S' \cup \Upsilon$ *and a set of state dependent surjections* $\{g_s : A_s \rightarrow A'_{f(s)} | s \in S\}$ *such that the following conditions hold:*

1. $\Psi' = \{(f(s), g_s(a)) | (s,a) \in \Psi\} \cup \{(s_\Upsilon, a_\Upsilon)\}$*,*

2. $\forall s \in f^{-1}(S'), s' \in S, a \in A_s$*,*

$$T'(f(s), g_s(a), f(s')) = \sum_{s'' \in \{t \in S | f(t) = f(s')\}} T(s, a, s'')$$

3. $T'(s_\Upsilon, a_\Upsilon, s_\Upsilon) = 1$*,*

4. $R'(f(s), g_s(a)) = R(s,a)$*,* $\forall s \in f^{-1}(S'), a \in A_s$*, and*

5. $g_s(a) = a_\Upsilon, \forall s \in f^{-1}(s_\Upsilon)$*.*

Condition (2) states that state-action pairs that have the same image under f *and* g_s *have the same block transition behavior in* M*, and condition (4) states that they have the same expected reward.* s_Υ *is an* absorbing*, or termination, state with a single admissible action* a_Υ *which transitions* s_Υ *back to itself. The surjection* $h : \Psi \rightarrow \Psi' \cup \{(s_\Upsilon, a_\Upsilon)\}$ *defined by* $h((s,a)) = (f(s), g_s(a))$ *is called a partial MDP homomorphism from* M *to* M'*, and* M' *is called the partial homomorphic image of* M *under* $h = \langle f, \{g_s | s \in S\}\rangle$*.* $\triangleleft$

Going back to our example, for M_A given above, the surjections $f : S_{taxi} \rightarrow S_A \cup \{s_\Upsilon\}$ and $\{g_s : A_s \rightarrow A'_{f(s)} | s \in S_{taxi}\}$ defined as

$$f(s) = \begin{cases} \langle r, c\rangle, & s = \langle r, c\rangle_{AB} \wedge (r,c) \neq (r_A, c_A) \\ s_\Upsilon, & \text{otherwise} \end{cases} \tag{5.1}$$

$$g_s(a) = \begin{cases} a, & s \in \{\langle r, c\rangle_{AB} | (r,c) \neq (r_A, c_A)\} \\ a_\Upsilon, & \text{otherwise} \end{cases} \tag{5.2}$$

satisfy the imposed conditions, and therefore M_A is a partial homomorphic image of M under $h = \langle f, \{g_s | s \in S_{taxi}\}\rangle$. Now, suppose that instead of location B, the passenger is to be transported to any other location d; for example C as in Figure 5.2 (b). Despite the change in destination, the agent must again first navigate to location A. Furthermore, for any state $s = \langle r, c\rangle_{Ad} \wedge (r,c) \neq (r_A, c_A)$ and admissible action $a \in A_s$, we have

$$\begin{aligned} R_A(\langle r, c\rangle, a) &= R_{taxi}(s, a) \\ T_A(\langle r, c\rangle, a, \langle r', c'\rangle) &= T_{taxi}(s, a, \langle r', c'\rangle_{Ad}) \\ &= T_{taxi}(s, a, \langle r', c'\rangle_{AB}) \end{aligned}$$

$$\begin{aligned} T_A(\langle r, c\rangle, a, s_\Upsilon) &= T_{taxi}(s, a, \langle r_A, c_A\rangle_{Ad}) \\ &= T_{taxi}(s, a, \langle r_A, c_A\rangle_{AB}) \end{aligned}$$

Hence, $h' = \langle f', \{g'_s | s \in S\}\rangle$ which extends h given in (5.1) and defined as

$$f'(s) = \begin{cases} \langle r, c\rangle, & s = \langle r, c\rangle_{A\cdot}, (r, c) \neq (r_A, c_A) \\ s_\Upsilon, & \text{otherwise} \end{cases}$$

$$g'_s(a) = \begin{cases} a, & s \in \{\langle r, c\rangle_{A\cdot} | (r, c) \neq (r_A, c_A)\} \\ a_\Upsilon, & \text{, otherwise} \end{cases}$$

is a partial MDP homomorphism from M_{taxi} to M_A. Let $s_1 = \langle r, c\rangle_{Ad_1}$ and $s_2 = \langle r, c\rangle_{Ad_2}$ be two states in S_{taxi} such that $(r, c) \neq (r_A, c_A)$ and $d_1 \neq d_2$. For any admissible action a, the image of the state-action pairs (s_1, a) and (s_2, a) under h', are the same, i.e. $(f'(s_1), g'_{s_1}(a)) = (f'(s_2), g'_{s_2}(a)) = (\langle r, c\rangle, a)$, which means that both states are *equivalent* with respect to state transition and reward structures.

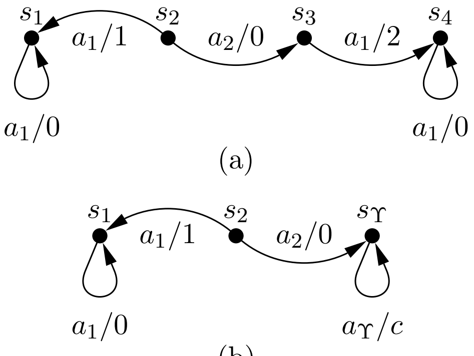
Definition 5.1.2 *Given an MDP $M = \langle S, A, \Psi, T, R\rangle$, state-action pairs (s_1, a_1) and $(s_2, a_2) \in \Psi$ are* equivalent *if there exists an MDP M' which is a partial homomorphic image of M under $h = \langle f, \{g_s | s \in S\}\rangle$ such that $f(s_1) = f(s_2)$ and $g_{s_1}(a_1) = g_{s_2}(a_2)$. States s_1 and s_2 are equivalent if $f(s_1) = f(s_2)$ and there exists a bijection $\rho : A_{s_1} \rightarrow A_{s_2}$ such that $g_{s_1}(a) = g_{s_2}(\rho(a)), \forall a \in A_{s_1}$.* ◁

The set of state-action pairs equivalent to (s, a), and the set of states equivalent to s are called the equivalence classes of (s, a) and s, respectively. Let $M' = \langle S', A', \Psi', T', R'\rangle$ be the image of $M = \langle S, A, \Psi, T, R\rangle$ under the partial MDP homomorphism $h = \langle f, \{g_s | s \in S\}\rangle$. Ravindra and Barto [41] proved that if h is a complete homomorphism, i.e. the set of absorbing states is empty, then an optimal policy π^* for M can be constructed from an optimal policy $\pi^*_{M'}$ for M' such that for any $a \in g_s^{-1}(a')$,

$$\pi^*(s, a) = \frac{\pi^*_{M'}(f(s), a')}{|g_s^{-1}(a')|} \tag{5.3}$$

This makes it possible to solve an MDP by solving one of its homomorphic images which can be structurally simpler and easier to solve. However, in general such a construction is not viable in case of partial MDP homomorphisms, since the optimal policies would depend on the rewards associated with absorbing states.

For example, let $M = \langle S, A, \Psi, P, R\rangle$ be an MDP with



(a)

(b)

Figure 5.3: Sample MDP M (a), and its homomorphic image M' (b). State transitions are deterministic and indicated by directed edges. Each edge label denotes the action causing the transition between connected states and the associated expected reward. Optimal policies are preserved only if $c \geq 1/9$.

- $S = \{s_1, s_2, s_3, s_4\}$,
- $A = \{a_1, a_2\}$,
- $\Psi = \{(s_1, a_1), (s_2, a_1), (s_2, a_2), (s_3, a_1), (s_4, a_1)\}$

where all state transitions are deterministic and expected rewards are as given in Figure 5.3(a). Suppose that discount factor, γ, is 0.9. Then, optimal policy at s_2 is to select action a_2 ($Q^*(s_2, a_1) = 1$, and $Q^*(s_2, a_2) = 1.8$). Consider deterministic MDP $M' = \langle S' \cup \{s_\Upsilon\}, A' \cup \{a_\Upsilon\}, \Psi' \cup (s_\Upsilon, a_\Upsilon), P', R' \rangle$ with

- $S = \{s_1, s_2\}$,
- $A = \{a_1, a_2\}$, and
- $\Psi' = \{(s_1, a_1), (s_2, a_1), (s_2, a_2)\}$

having state transitions and expected rewards given in Figure 5.3(b), where the expected reward for executing action a_Υ at the absorbing state is a constant c. M' is a partial homomorphic image of M under $h = \langle f, \{g_s | s \in S\} \rangle$ defined as

$$\begin{aligned} f(s_1) &= s_1 \\ f(s_2) &= s_2 \\ f(s_3) &= f(s_4) = s_\Upsilon \\ g_{s_1}(a_1) &= g_{s_2}(a_1) = a_1 \\ g_{s_2}(a_2) &= a_2 \\ g_{s_3}(a_1) &= g_{s_4}(a_1) = a_\Upsilon \end{aligned} \tag{5.4}$$

One can easily calculate that $Q^*(s_2, a_1) = 1$ and $Q^*(s_2, a_2) = 9 * c$. Accordingly, the optimal policy for M' at s_2 is to select a_1 if $c < 1/9$, and a_2 otherwise, which is different than the optimal policy for M.

As the example given above demonstrates, unless reward functions are chosen carefully, it may not be possible to employ the solutions of partial homomorphic images of a given MDP to solve it. However, if equivalence classes of state-action pairs and states are known they can be used to speed up the learning process. Let (s, a) and (s', a') be two equivalent state-action pairs in $M = \langle S, A, \Psi, T, R \rangle$ based on the partial homomorphic image $M' = \langle S' \cup \{s_\Upsilon\}, A' \cup \{a_\Upsilon\}, \Psi', T', R' \rangle$ of M under $h = \langle f, \{g_s | s \in S\} \rangle$. Suppose that while learning the optimal policy, the agent selects action a at state s which takes it to state t with an immediate reward r such that $f(t) \neq s_\Upsilon$, and consequently estimate of state-action value function $Q(s, a)$ is updated as described in Section 3.2. Let $t_{M'} = f(t)$ be the image of state t under h, and $\Omega = \{t' | f'(t') = t_{M'} \wedge P(s', a', t') > 0\}$ be the set of states that map to $t_{M'}$ under h. By definition, for any $t' \in \Omega$, the probability of experiencing a transition from s' to t' upon taking action a' is equal to that of from s to t upon taking action a. Also, since (s, a) and (s', a') are equivalent, we have $R(s, a) = R(s', a')$, i.e. both state-action pairs have the same expected reward, and, since t is non-absorbing state, independent of the reward assigned to the absorbing state, they have the same policy. Therefore, for any state $t' \in \Omega$, $o = \langle s', a', r, t' \rangle$ can be regarded as a virtual experience tuple and $Q(s', a')$ can be updated similar to $Q(s, a)$ based on observation o, i.e. pretending as if action a' transitioned the agent from state s' to state t' with an immediate reward r. In particular, for the Q-learning algorithm $Q(s', a')$ is updated using

$$Q(s', a') = (1 - \alpha)Q(s', a') + \alpha(r + \gamma \max_{a'' \in A_{t'}} Q(t', a''))$$

The complete Q-learning with equivalent state update is presented in Algorithm 9.

Note that, since partial MDP homomorphisms do not preserve state transition and reward behavior for states that map to absorbing states, special care must be taken when adapting on-policy methods to incorporate equivalent state update. In particular, consider the Sarsa algorithm that has the update rule

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(r + \gamma Q(s', a')) \tag{5.5}$$

where $\langle s, a, r, s' \rangle$ is the experience tuple and a' is chosen from s' using policy derived from the learning policy π_Q based on current Q-values, i.e. $a' = \pi_Q(s')$. Let (t, a_t)

Algorithm 9 Q-learning with equivalent state update.

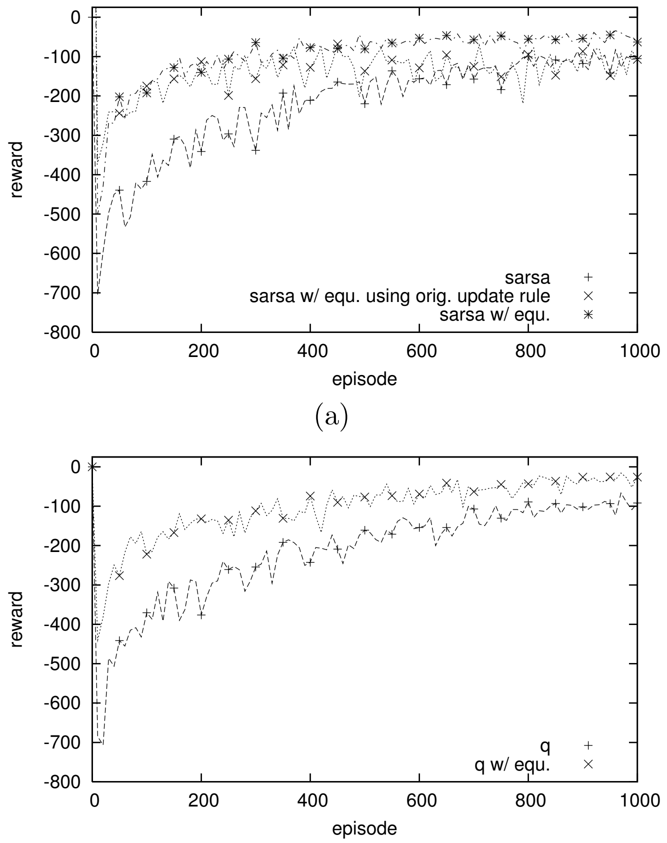
```
Initialize $Q$ arbitrarily (e.g., $Q(\cdot,\cdot) = 0$)
repeat
    Let $s$ be the current state
    repeat                                              ▷ for each step
        Choose $a$ from $s$ using policy derived from $Q$ with sufficient exploration
        Take action $a$, observe $r$ and the next state $s'$
        $Q(s,a) = (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a'' \in A_{s'}} Q(s',a''))$
        for all $(t, a_t)$ equivalent to $(s,a)$ under a partial MDP homomorphism $h = \langle f, \{g_s | s \in S\}\rangle$ do
            Let $t'$ be a state in $f^{-1}(f(s'))$
            $Q(t,a_t) = (1-\alpha)Q(t,a_t) + \alpha(r + \gamma \max_{a_{t'} \in A_{t'}} Q(t',a_{t'}))$
        end for
        $s = s'$
    until $s$ is a terminal state
until a termination condition holds
```

and $(t', a_{t'})$ be equivalent to (s,a) and (s',a') under a partial MDP homomorphism, such that the image of s', and consequently t', is the absorbing state of the homomorphic image. Then, if the update rule of Sarsa is applied using virtual experience $\langle t, a_t, t', a_{t'}\rangle$, $Q(t,a_t)$ is not guaranteed to converge to optimal value since $(t', a_{t'})$ may not be a greedy action selection [47]. In order to overcome this problem, updates can be restricted to those that transitioned to non-absorbing states, or alternatively the action $\pi_Q(t')$ imposed by the learning policy at t' can be used instead of $a_{t'}$, leading to the Sarsa with equivalent state update given in Algorithm 10.

Figure 5.4(a) compares the performance of regular Q-learning and Q-learning with equivalent state update on 5×5 taxi problem when equivalence classes of state-action pairs are calculated based on four partial homomorphic images each corresponding to navigating to four predefined locations. It shows the number of steps taken by the agent until passenger is successfully delivered to its destination. The initial Q-values are set to 0, the learning rate and discount factor are chosen as $\alpha = 0.05$, and $\gamma = 0.9$, respectively. In order to determine these learning parameters, we conducted a set of initial experiments that cover a range of possible values and picked the ones that

Algorithm 10 Sarsa with equivalent state update.

```
Initialize $Q$ arbitrarily (e.g., $Q(\cdot, \cdot) = 0$)
repeat
    Let $s$ be the current state
    repeat                                                        ▷ for each step
        Choose $a$ from $s$ using policy derived from $Q$ with sufficient exploration
        Take action $a$, observe $r$ and the next state $s'$
        Choose $a'$ from $s'$ using policy derived from $Q$
        $Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(r + \gamma Q(s', a'))$
        for all $(t, a_t)$ equivalent to $(s, a)$ under partial MDP homomorphism $h = \langle f, \{g_s | s \in S\}\rangle$ do
            Let $t'$ be a state in $f^{-1}(f(s'))$ such that $P(t, a_t, t') > 0$
            Choose $a_{t'}$ from $t'$ using policy derived from $Q$
            $Q(t, a_t) = (1 - \alpha)Q(t, a_t) + \alpha(r + \gamma Q(t', a_{t'}))$
        end for
        $s = s'$
        $a' = a$
    until $s$ is a terminal state
until a termination condition holds
```



(a)

(b)

Figure 5.4: (a) Q-learning vs. Q-learning with equivalent state update, and (b) Sarsa vs. Sarsa with equivalent state update on 5×5 taxi problem with one passenger using partial homomorphic images corresponding to navigation to four predefined locations. The figure shows number of steps to successful transportation averaged over 30 runs.

perform best overall on regular RL algorithms (i.e. without equivalent state update). ϵ-greedy action selection mechanism is used with $\epsilon = 0.1$. Starting state which includes the initial position of the taxi agent, location of passenger and its destination are selected randomly with uniform probability. The results are averaged over 30 runs. As expected, even though we do not take advantage of all possible state-action equivalences in the domain, the taxi agents learns and converges to optimal policy faster when equivalent state-action pairs are also updated. Sarsa with equivalent state update given in Algorithm 10 also shows similar results, outperforming regular Sarsa as presented in Figure 5.4(b). When equivalent state update is applied to Sarsa using the update rule of Equation 5.5, we observe a steep learning curve during the initial episodes of the task, but then due to broken links of virtual experiences, i.e. transitions to absorbing states, the convergence rate falls drastically below that of Algorithm 10. In more complex 12×12 version of the problem with larger state space, Figure 3.2, it even fails to converge to optimal policy, whereas Q-learning and Sarsa with equivalent state update outperform their regular counterparts (Figure 5.5).

The method described above assumes that equivalence classes of states and corresponding partial MDP homomorphisms are already known. If such information is not available prior to learning, then for a restricted class of partial MDP homomorphisms it is still possible to identify states that are similar to each other with respect to state transition and reward behavior as learning progresses based on the collected history of events, as we show in the next section. Experience gathered on one state, then, can be reflected to similar states to improve the performance of learning.

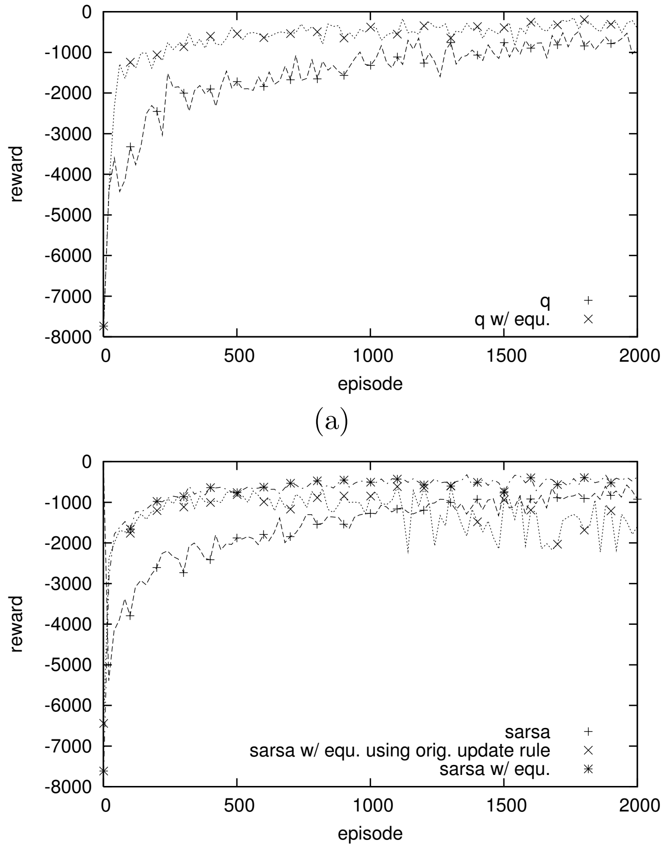
5.2 Finding Similar States

For the rest of this chapter, we will restrict our attention to *direct* partial MDP homomorphisms $h = \langle f, \{g_s | s \in S\} \rangle$, where for each state s that maps to a non-absorbing state, g_s is the identity function, i.e. $g_s(a) = a$, and for the sake of simplicity, f will be used to denote $h = \langle f, \{g_s | s \in S\} \rangle$.

Let $M = \langle S, A, \Psi, T, R \rangle$ be a given MDP. As previously given in Definition 4.1.9, starting from state s, a sequence of states, actions and rewards

$$\sigma = s_1, a_2, r_2, \ldots, r_{n-1}, s_n$$

such that $s_1 = s$ and each a_i, $2 \leq i \leq n-1$, is chosen by following a policy π, i.e.,



(a)

(b)

Figure 5.5: (a) Q-learning vs. Q-learning with equivalent state update, and (b) Sarsa vs. Sarsa with equivalent state update on 12 × 12 taxi problem with one passenger using partial homomorphic images corresponding to navigation to four predefined locations.

based on $\pi(s_i)$, is called a π-*history* of s with length n [53]. $AR_\sigma = a_1 r_1 a_2 \dots a_{n-1} r_{n-1}$ is the action-reward sequence of σ, and the restriction of σ to $\Sigma \subseteq S$, denoted by σ_Σ, is the longest prefix $s_1, a_2, r_2, \dots, r_{i-1}, s_i$ of σ, such that for all $j = 1..i, s_j \in \Sigma$ and $s_{i+1} \notin \Sigma$.

Suppose that (s, a) and (s', a') are two state-action pairs in M that are equivalent to each other based on partial homomorphic image $M' = \langle S' \cup \{s_\Upsilon\}, A \cup \{a_\Upsilon\}, \Psi', T', R' \rangle$ of M under direct partial MDP homomorphism f. Consider a π-history of state s, $\sigma = s_1, a_2, r_2, \dots, r_{n-1}, s_n$. Let $\Sigma_f \subseteq S$ be the inverse image of S' under f, and $\sigma_{\Sigma_f} = s_1, a_2, r_2, \dots, r_{k-1}, s_k$ where $k \leq n$ be the restriction of σ on Σ_f. The image of σ_{Σ_f} under f, denoted by $F(\sigma_{\Sigma_f})$, is obtained by mapping each state s_i to its counterpart in S', i.e. $F(\sigma_{\Sigma_f}) = f(s_1), a_2, r_2, f(s_2), a_3, \dots, r_{k-1}, f(s_k)$. By definition, $F(\sigma_{\Sigma_f})$ is a π-history of $f(s)$ in M', and since s and s' are equivalent, there exists a π-history σ' of s' such that the image of its restriction to Σ_f under f is equal to $F(\sigma_{\Sigma_f})$, i.e. $F(\sigma'_{\Sigma_f}) = F(\sigma_{\Sigma_f})$, and furthermore $AR_{\sigma'_{\Sigma_f}} = AR_{\sigma_{\Sigma_f}}$. Therefore, if two states are equivalent under the direct partial MDP homomorphism f, then the set of images of π-histories restricted to the states that map to non-absorbing states under f, and consequently, the list of associated action-reward sequences are the same. This property of equivalent states leads to a natural approach to calculate the similarity between any two states based on the number of common action-reward sequences.

Given any two states s and s' in S, let $\Pi_{s,i}$ be the set of π-histories of state s with length i, and $\varsigma_i(s, s')$ calculated as

$$\varsigma_i(s, s') = \frac{\sum_{j=1}^{i} |\{\sigma \in \Pi_{s,j} | \exists \sigma' \in \Pi_{s',j}, AR_\sigma = AR_{\sigma'}\}|}{\sum_{j=1}^{i} |\Pi_{s,j}|} \tag{5.6}$$

be the ratio of the number of common action-reward sequences of π-histories of s and s' with length up to i to the number of action-reward sequences of π-histories of s with length up to i. One can observe the following:

- $\varsigma_i(s, s')$ will be high, close to 1, if s' is similar to s in terms of state transition and reward behavior, and low, close to 0, in case they differ considerably from each other.
- Even for equivalent states the action-reward sequences will eventually deviate

and follow different courses as the subtask that they are part of ends. As a result, for i larger than some threshold value, ς_i would inevitably decrease and no longer be a permissible measure of the state similarity.

- On the contrary, for very small values of i, such as 1 or 2, ς_i may over estimate the amount of similarity since number of common action-reward sequences can be high for short action-reward sequences.

Combining these observations, and also since optimal value of i depends on the subtasks of the problem, in order to increase robustness it is necessary to take into account action-reward sequences of various lengths. Therefore, the maximum value or weighted average of $\varsigma_i(s, s')$ over a range of problem specific i values, k_{min} and k_{max}, can be used to combine the results of evaluations and approximately measure the degree of similarity between states s and s', which we will denoted by $\varsigma(s, s')$. Once $\varsigma(s, s')$ is calculated, s' is regarded as equivalent to s if $\varsigma(s, s')$ is over a given threshold value $\tau_{similarity}$. Likewise, by restricting the set of π-histories to those that start with a given action a in the calculation of $\varsigma(s, s')$, similarity between state-action pairs (s, a) and (s', a) can be measured approximately.

If the set of π-histories for all states are available in advance, then using Equation 5.6 similarities between all state pairs can be calculated and equivalent states can be identified prior to learning. However, in most of the RL problems, the dynamics of the system is not known in advance and consequently such information is not available. Therefore, using the history of observed events (i.e. sequence of states, actions taken and rewards received), the agent must incrementally store the π-histories of length up to k_{max} during learning and enumerate common action-reward sequences of states in order to calculate similarities between them. For this purpose, we propose an auxiliary structure called *path tree*, which stores the prefixes of action-reward sequences of π-histories for a given set of states.

Definition 5.2.1 *A path tree $P = \langle N, E \rangle$ is a labeled rooted tree, where N is the set of nodes, such that each node represents a unique action-reward sequence; and $e = (u, v, \langle a, r \rangle) \in E$ is an edge from u to v with label $\langle a, r \rangle$, indicating that action-reward sequence v is obtained by appending a, r to u, i.e., $v = uar$. The root node represents the empty action sequence. Furthermore, each node u holds a list of $\langle s, \xi \rangle \in S \times \mathcal{R}$ tuples, stating that state s has one or more π-histories starting with action sequence*

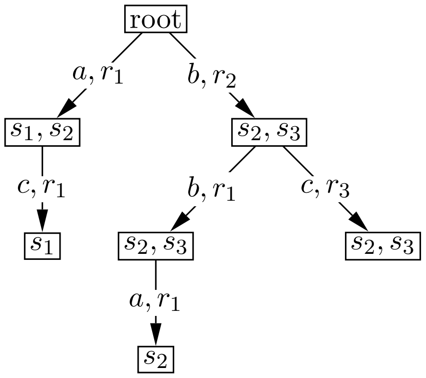


Figure 5.6: Path tree for π-histories $\{s_1ar_1{\cdot}cr_1{\cdot}, s_2ar_1{\cdot}, s_2br_2{\cdot}br_1{\cdot}ar_1{\cdot}, s_2br_2{\cdot}cr_3{\cdot}, s_3br_2{\cdot}cr_3{\cdot}, s_3br_2{\cdot}br_1{\cdot}\}$. $\cdot$ represents intermediate states, i.e. states of the π-history except the first one. Eligibility values of edges and tuples in the nodes are not displayed.

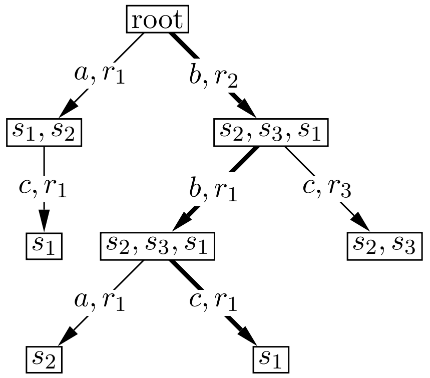


Figure 5.7: After adding π-history $s_1br_2{\cdot}br_1{\cdot}cr_1{\cdot}$ to the path tree given in Figure 5.6. Thick edges indicate the affected nodes.

σ_u. ξ is the eligibility value *of σ_u for state s, representing its occurrence frequency. It is incremented every time a new π-history for state s starting with action sequence σ_u is added to the path tree, and gradually decremented otherwise.* $\triangleleft$

A sample path tree for a given set of π-histories is presented in Figure 5.6.

A π-history $h = s_1a_2r_2 \ldots r_{k-1}s_k$ can be added to a path tree by starting from the root node and following edges according to their label. Let $\hat{n}$ denote the active node of the path tree, which is initially the root node. For $i = 1..k-1$, if there is a node n such that $\hat{n}$ is connected to n by an edge with label $\langle a_i, r_i\rangle$, then either ξ of the tuple $\langle s_1, \xi\rangle$ in n is incremented or a new tuple $\langle s_1, 1\rangle$ is added to n if it does not exist, and $\hat{n}$ is set to n. Otherwise, a new node containing tuple $\langle s_1, 1\rangle$ is created, and $\hat{n}$ is connected to this node by an edge with label $\langle a_i, r_i\rangle$. The new node becomes the active node (Algorithm 11).

The state of the sample path tree after adding a new π-history $s_1br_2{\cdot}br_1{\cdot}cr_1{\cdot}$ using

Algorithm 11 Algorithm for adding a π-history to a path tree.

```
procedure ADD-HISTORY(h,T)
$h$ is a $\pi$-history of the form $s_1 a_2 r_2 \ldots r_{k-1} s_k$.
    Let $\hat{n}$ be the root node of $T$.
    for $i = 1$ to $k - 1$ do
        if $\exists$ node $n$ such that $\hat{n}$ is connected to $n$ by an edge with label $\langle a_i, r_i \rangle$ then
            if $\exists$ a tuple $\langle s_1, \xi \rangle$ in $n$ then
                Increment $\xi$.
            else
                Add a new tuple $\langle s_1, 1 \rangle$ to $n$.
            end if
        else
            Create a new node $n$ containing tuple $\langle s_1, 1 \rangle$.
            Connect $\hat{n}$ to $n$ by an edge with label $\langle a_i, r_i \rangle$.
        end if
        $\hat{n} = n$
    end for
end procedure
```

the algorithm described above is given in Figure 5.7.

Algorithm 12 Algorithm for generating π-histories from a given history of events and adding them to the path tree.

```
procedure UPDATE-PATH-TREE(H,T,$k_{max}$)
    $H$ is a sequence of tuples of the form $\langle s_1, a_1, r_1\rangle \dots \langle s_n, a_n, r_n\rangle$
    for $i = 1$ to $n$ do
        $h = s_i a_i r_i$                          ▷ $h$ is a $\pi$-history of state $s_i$.
        for $j = i + 1$ to $min(n, i + k_{max} - 1)$ do
            Append $s_j a_j r_j$ to $h$
        end for
        ADD-HISTORY($h, T$)
    end for
end procedure
```

At each time step the agent keeps track of the current state, action it chooses and the reward received from the environment in return, and stores them as a sequence of tuples $\langle s_1, a_1, r_1\rangle, \langle s_2, a_2, r_2\rangle \dots$. After each episode or for non-episodic tasks when a specific condition holds, such as executing a fixed number of steps or when a reward peak is reached, the following steps are executed:

1. This sequence is processed from front to back; for each tuple $\langle s_i, a_i, r_i\rangle$ using the next $k_{max} - 1$ tuples $\langle s_{i+1}, a_{i+1}, r_{i+1}\rangle \dots \langle s_{i+k_{max}-1}, a_{i+k_{max}-1}, r_{i+k_{max}-1}\rangle$ a new set π-history is generated and added to the path tree using the algorithm described above (Algorithm 12). Once this is done, the sequence of tuples is no longer needed and can be disposed.

2. The eligibility values of tuples in the nodes of the tree are decremented by a factor of $0 < \xi_{decay} < 1$, called *eligibility decay rate*, and tuples with eligibility value less than a given threshold, $\xi_{threshold}$, are removed to keep the tree in a manageable size and focus the search on recent and frequently used action-reward sequences.

Using the generated path tree, $\varsigma(s, s')$ can be calculated incrementally for all $s, s' \in S$ by traversing it in breadth-first order and keeping two arrays $\kappa(s)$ and $K(s, s')$. $\kappa(s)$ denotes the number of nodes containing s, and $K(s, s')$ denotes the number of

Algorithm 13 Calculating state similarities using the generated path tree.

```
procedure CALCULATE-SIMILARITY(T, $k_{min}$, $k_{max}$)
    Initialize $\varsigma(u, v), \kappa(u)$, and $K(u, v)$ to 0
    $level = 1$
    $currentNodes = \langle$ root node of T$\rangle$ ▷ $currentNodes$ contains the list of nodes in the curent level.
    while $currentNodes \neq \emptyset$ and $level \leq k_{max}$ do
        $nextNodes = \langle\rangle$
        for all $node \in currentNodes$ do    ▷ for each node in the current level
            for all $u$ in the tuple list of $node$ do
                Increment $\kappa(u)$
                for all $v \neq u$ in the tuple list of $node$ do
                    Increment $K(u, v)$    ▷ Increment number of common sequences.
                end for
            end for
            Append children of $node$ to $nextNodes$
        end for
        if $level \geq k_{min}$ then ▷ Update similarity values if level of $k_{min}$ is reached.
            for all $\langle u, v\rangle$ such that $K(u, v) > 0$ do
                $\varsigma(u, v) = \max(K(u, v)/\kappa(u), \varsigma(u, v))$
            end for
        end if
        $currentNodes = nextNodes$    ▷ Next level consists of the children of the nodes in the current level.
        $level = level + 1$
    end while
end procedure
```

nodes containing both s and s' in their tuple lists. Initially $\varsigma(s, s')$, $K(s, s')$, and $\kappa(s)$ are set to 0. At each level of the tree, the tuple lists of nodes at that level are processed, and $\kappa(s)$ and $K(s, s')$ are incremented accordingly. After processing level i, $k_{min} \leq i \leq k_{max}$, for every s, s' pair that co-exist in the tuples list of a node at that level, $\varsigma(s, s')$ is compared with $K(s, s')/\kappa(s)$ and updated if the latter one is greater (Algorithm 13). Note that, since eligibility values stored in the nodes of the path tree is a measure of occurrence frequencies of corresponding sequences, it is possible to extend the similarity function by incorporating eligibility values in the calculations as normalization factors. This would improve the quality of the metric and also result in better discrimination in domains with high degree of non-determinism, since the likelihood of the trajectories will also be taken into consideration. In order to simplify the discussion, we opted to omit this extension. Once ς values are calculated, equivalent states, i.e. state pairs with ς greater than $\tau_{similarity}$, can be identified and incorporated into learning. Note that, since similarities of states are not expected to change drastically each episode this last step, i.e. calculation of similarities, can be executed at longer intervals. The overall process described above leads to the general learning model given in Algorithm 14.

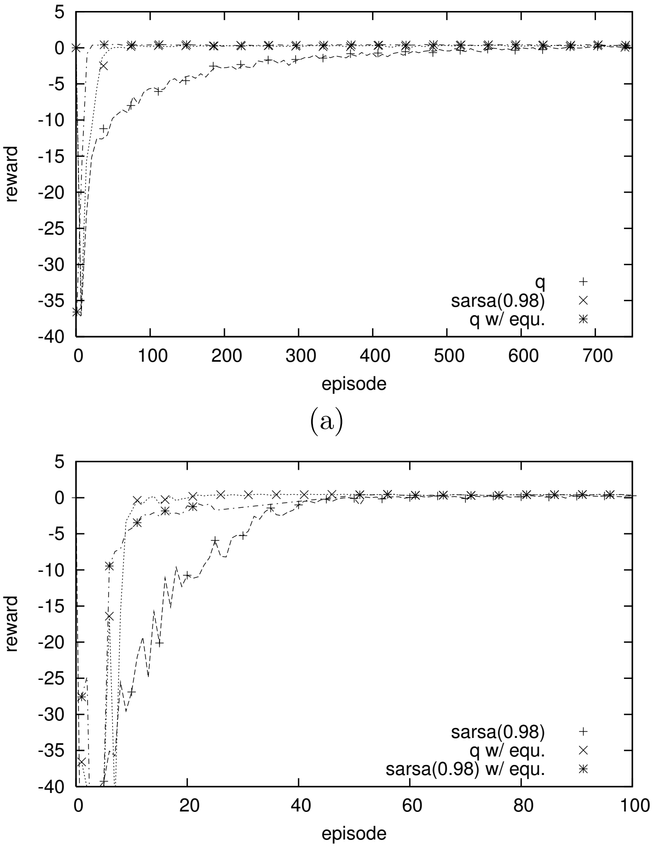
5.3 Experiments

We applied the similar state update method described in Section 5.2 to Q-learning and compared its performance with different RL algorithms on two test domains: a six-room maze and various versions of the taxi problem. Also, its behavior under various parameter settings, such as maximum length of π-histories and eligibility decay rate, are examined.

In all test cases, the initial Q-values are set to 0, and ϵ-greedy action selection mechanism, where action with maximum Q-value is selected with probability $1-\epsilon$ and a random action is selected otherwise, is used with $\epsilon = 0.1$. The results are averaged over 50 runs. Unless stated otherwise, the path tree is updated using an eligibility decay rate of $\xi_{decay} = 0.95$ and an eligibility threshold of $\xi_{threshold} = 0.1$, and in similarity calculations the following parameters are employed: $k_{min} = 3$, $k_{max} = 5$, and $\tau_{similarity} = 0.8$. As will be seen from the results presented in the rest of this section, these values are found to perform well both in terms of learning performance

Algorithm 14 Reinforcement learning with equivalent state update.

```
Initialize $Q$ arbitrarily (e.g., $Q(\cdot,\cdot) = 0$).
$T = \langle root \rangle$        ▷ Initially the path tree contains only the root node.
repeat        ▷ for each episode
    Let $s$ be the current state which is initially the starting state.
    $H = \emptyset$        ▷ Initially episode history is empty.
    repeat        ▷ for each step
        Choose $a$ from $s$ using using the underlying RL algorithm.
        Take action $a$, observe $r$ and the next state $s'$.
        Update $Q(s, a)$ based on $s, a, r, s'$.
        for all $t$ similar to $s$ do
            if probability of receiving reward $r$ at $t$ by taking action $a > 0$ then
                Update $Q(t, a)$ based on $t, a, r, \varsigma(s, t)$.
            end if
        end for
        Append $\langle s, a, r \rangle$ to $H$.
        $s = s'$
    until $s$ is a terminal state
    UPDATE-PATH-TREE($H, T, k_{max}$)
    Traverse $T$ and update eligibility values.
    if time to re-calculate similarities of states then        ▷ ex. every $n$ episodes.
        CALCULATE-SIMILARITY($T, k_{min}, k_{max}$)
    end if
until a termination condition holds
```



(a)

(b)

Figure 5.8: (a) Q-learning with equivalent state update vs. Q-learning and Sarsa(λ), and (b) Q-learning and Sarsa(λ) with equivalent state update vs. Sarsa(λ) in six-room maze problem.

and the size of the path tree. Since our aim is to analyze the effect of the proposed method when it is applied to an existing algorithm, other related learning parameters (ex. learning rate or λ in Sarsa(λ)) are chosen in such a way that best overall results are achieved on the existing algorithms. For this purpose, we conducted a set of initial experiments and tested a range of values for each parameter. In both of the test domains, an episode is terminated automatically if the agent could not successfully complete the given task within 20000 time steps.

5.3.1 Comparison with Standard RL Algorithms

Figure 5.8 shows the total reward obtained by the agent until goal position is reached in six-room maze problem when equivalent state update is applied to Q-learning and Sarsa(λ) algorithms. Based on initial testing, we used a learning rate and discount factor of $\alpha = 0.125$, and $\gamma = 0.90$, respectively. For the $Sarsa(\lambda)$ algorithm, λ is

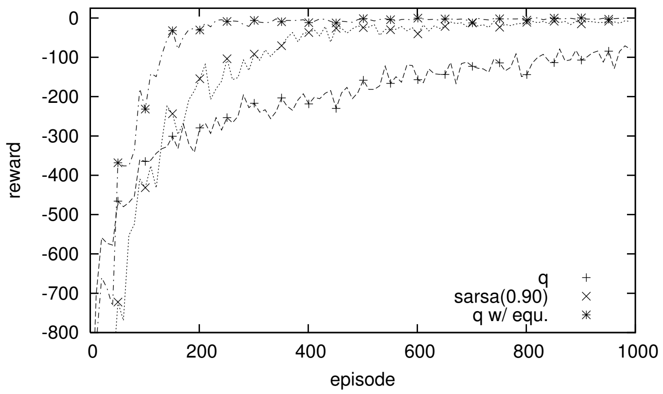


Figure 5.9: Q-learning with equivalent state update vs. Q-learning and Sarsa(λ) in 5×5 taxi problem with one passenger.

taken as 0.98. The path tree is updated and state similarities are calculated after each episode. As expected, due to backward reflection of received rewards, Sarsa(λ) converges much faster compared to Q-learning. The learning curve of Q-learning with equivalent state update indicates that, starting from early states of learning, the proposed method can effectively utilize state similarities and improve the performance considerably. Since convergence is attained in the in less than 20 episodes, the result obtained using Sarsa(λ) with equivalent state update is almost indistinguishable from that of Q-learning with equivalent state update as plotted in Figure 5.8 (b).

The results of the corresponding experiments in 5×5 taxi problem showing the total reward obtained by the agent until passenger is successfully delivered to its destination is presented in Figure 5.9. At each episode, the starting state (i.e. the initial position of the taxi agent, location of the passenger and its destination) is selected randomly with uniform probability. Learning rate, α, is set to 0.05, and λ is taken as 0.9 in Sarsa(λ) algorithm. The path tree is updated after each episode and state similarities are computed every 5 episodes starting from the 20^{th} episode in order to let agent gain experience for the initial path tree. Similar to six-room maze problem, Sarsa(λ) learns faster than regular Q-learning. The learning curves of algorithms with equivalent state update reveals that the proposed method is successful in identifying similar states which leads to an early improvement in performance, and consequently allows the agent to learn the task more efficiently. Q-learning with equivalent state update converges to optimal behavior after 200 episodes, Sarsa(λ) reaches the same level after 1000 episodes, and regular Q-learning falls far behind of both algorithms

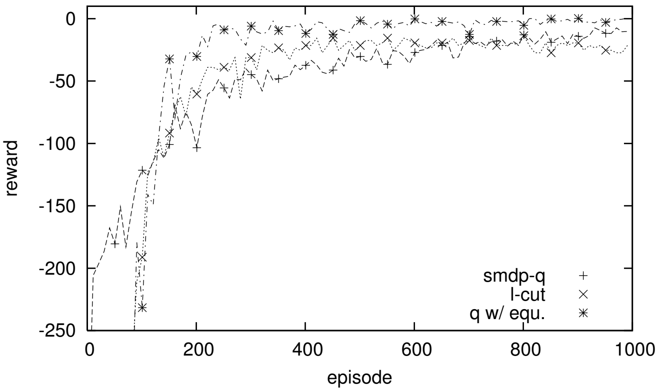


Figure 5.10: Q-learning with equivalent state update vs. SMDP Q-learning and L-Cut on 5 × 5 taxi problem with one passenger.

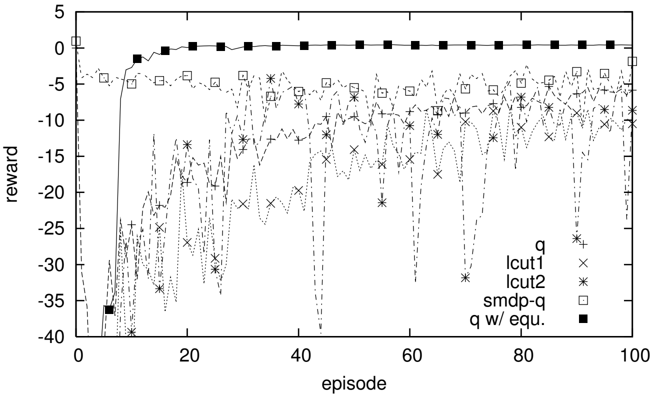


Figure 5.11: Q-learning with equivalent state update vs. SMDP Q-learning and L-Cut on six-room maze problem.

(average reward of -100 after 1000 episodes).

Figure 5.10 and 5.11 compare Q-learning with equivalent state update in six-room maze and taxi domains with more advanced RL methods that make use of temporal abstractions. In SMDP Q-learning [7], in addition to primitive actions, the agent can select and execute macro-actions, or options. For the six room maze problem, we defined hand-coded options which optimally, i.e. using minimum number of steps, move the agent from its current position to one of door-ways connecting to a neighboring room. Similarly, for the taxi domain, an option is defined for each of the predefined locations which moves the agent from its current position to the corresponding predefined location as soon as possible. The L-Cut algorithm of Simsek and Barto [46], instead of relying on user defined abstractions, automatically finds possible sub-goal states as the learning progresses and generates and utilizes options for solving them. The subgoal states of the problem are identified by approximately

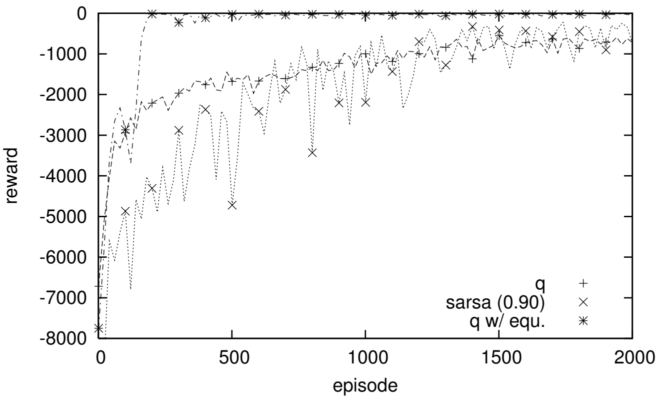


Figure 5.12: Q-learning with equivalent state update vs. Q-learning and Sarsa(λ) in 12×12 taxi problem with one passenger.

partitioning a state transition graph generated using recent experiences of the agent, such that the probability of transition is low between states in different blocks but high within the same partition. Then, options are created by executing action replay on the neighborhood of the sub-goal states using artificial rewards. It is one of the recent methods proposed for option discovery in the field of hierarchical reinforcement learning, and follows the tradition of a different approach, which tries to identify sub-goals or sub-tasks of the problem explicitly instead of implicitly making use of relations between states as proposed in this work. In the experiments with the taxi domain, we run L-Cut algorithm using the parameters as specified in [46]. As presented in Figure5.10, although SMDP Q-learning has a very steep learning curve in the initial stages, by utilizing abstractions and symmetries more effectively both Q-learning with equivalent state update and L-Cut perform better in the long run and converge to optimal policy faster. However, in the six-room maze problem, despite the fact that we tested under various parameter settings two of which are plotted[1], L-cut algorithm is found to perform poorly, possibly due to generating options that move the agent to a door-way away from the goal since number of transitions between rooms would be high both ways during the initial stages of the learning, compared to Q-learning with equivalent state update which exhibits rapid improvement.

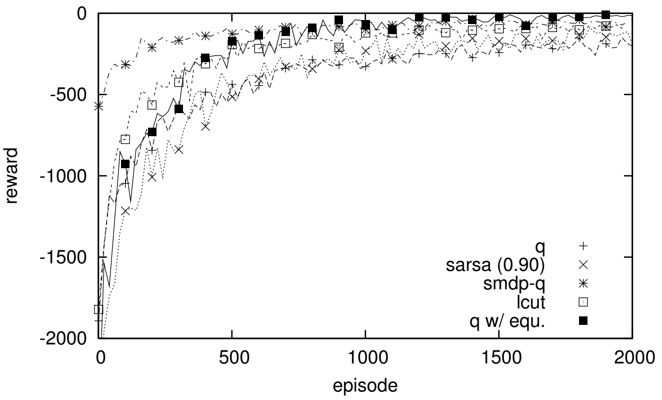


Figure 5.13: 5 × 5 taxi problem with two passengers.

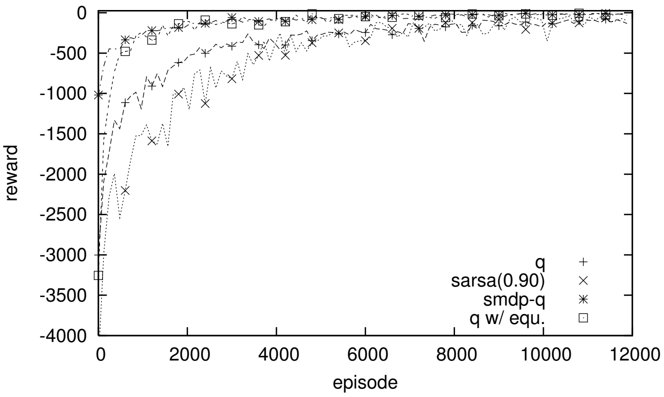


Figure 5.14: 5 × 5 taxi problem with four passengers.

5.3.2 Scalability

The results for the larger 12 × 12 taxi problem presented in Figure 5.12 demonstrates that the improvement becomes more evident as the complexity of the problem increases. In this case, both regular Q-learning and Sarsa(λ) fail to converge to optimal behavior within 2000 episodes (average reward of -600), whereas Q-learning when proposed method is applied successfully converges to optimal behavior after 250 episodes.

In taxi problem, the number of equivalent states increases with the number of passengers to be transported. Therefore, more improvement in learning performance is expected when they can be utilized effectively. In order to test the performance of the proposed method, we also run experiments on different versions of the taxi problem with multiple passengers, where the passengers must be successively transported to their destinations one by one. Note that, the ordering of how passengers are to be transported is also important and the problem involves additional complexity

---

[1] lcut1: parameters as specified in [46], lcut2: $t_c = 0.1, h = 300, t_o = 5, t_p = 0.1$

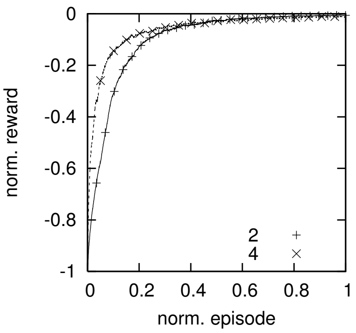
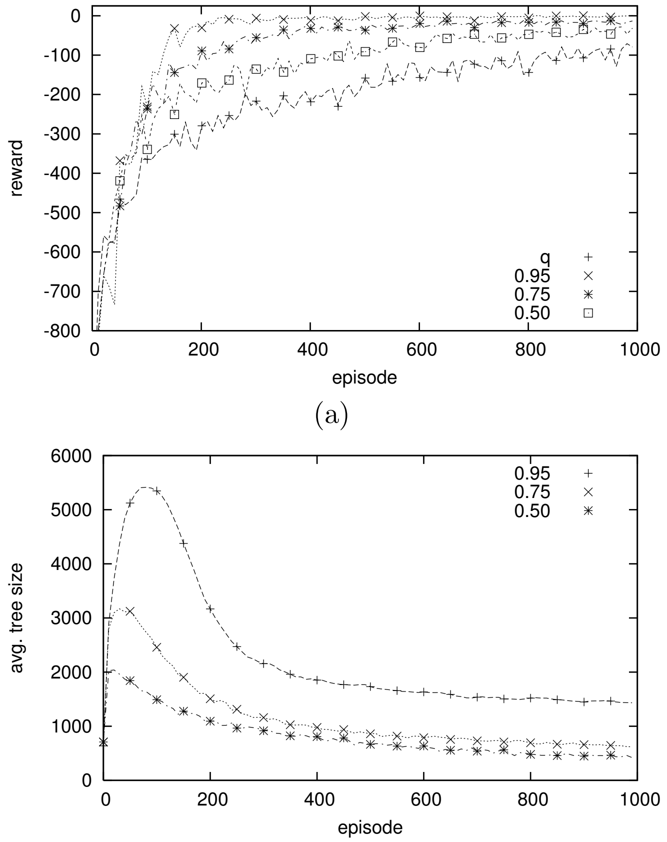


Figure 5.15: Convergence rate of 5 × 5 taxi problem with two passengers vs. four passengers.

and optimization in this respect. Results for the 5 × 5 taxi problem with two and four passengers are presented in Figure 4.12 and Figure 4.13, respectively. Note that the convergence rate of the regular algorithms decrease as the number of passengers increases, and common subtasks, consequently equivalent states, become more prominent. The results demonstrate that the proposed method is effective in identifying solutions in such cases. Figure 5.15 shows the convergence rate of Q-learning with equivalent state update for the case of two and four passengers. The x-axis (number of episodes) is normalized based on the time then the optimal policy is attained, and y-axis (reward) is normalized using the minimum and maximum rewards received by the agent. As can be seen from Figure 5.15, the proposed method has a steeper learning curve and converges relatively faster when there are more passengers indicating that the equivalent states are utilized effectively.

5.3.3 Effects of the Parameters

In order to analyze how various parameter choices of the proposed method affect the learning behavior, we conducted a set of experiments under different settings. The results for various ξ_{decay} values are presented in Figure 5.16. As the eligibility decay decreases, the number of π-histories represented in the path tree also decrease which considerably reduces the execution time of the algorithm (Figure 5.18(a)). On the other hand, this also causes recent π-histories to dominate over existing ones, less number of equivalent states can be identified and the performance of the method also converges to that of regular Q-learning as ξ_{decay} gets smaller. Figure 5.17 shows how the length of π-histories affect the performance in the taxi domain. Different k_{max} values are found to display almost indistinguishable behavior, even though the path



(a)

(b)

Figure 5.16: Effect of ξ_{decay} on 5×5 taxi problem with one passenger. (a) Reward obtained, and (b) average size of the path tree for different ξ_{decay} values.

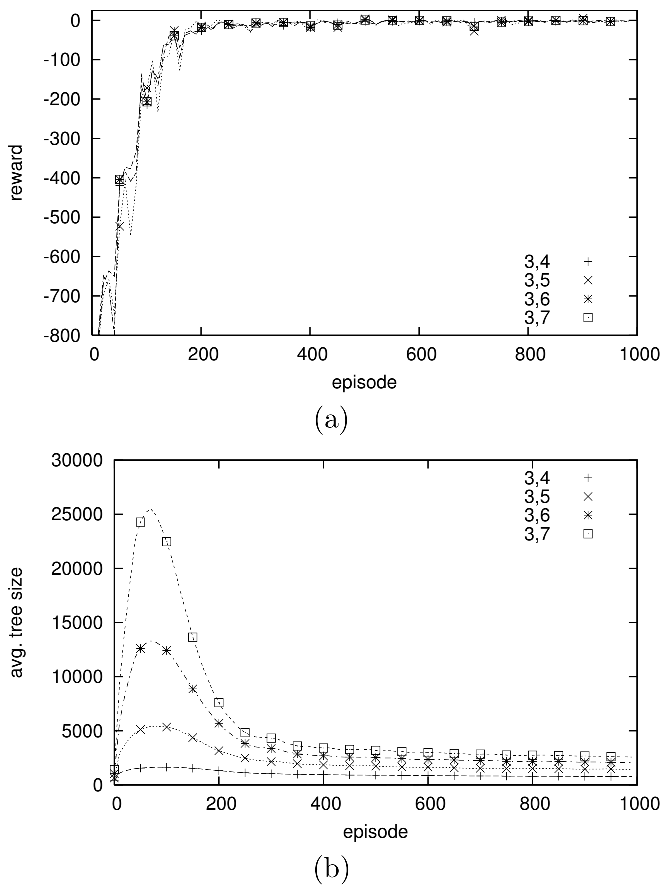


Figure 5.17: Effect of k_{min} and k_{max} on 5×5 taxi problem with one passenger. (a) Reward obtained, and (b) average size of the path tree.

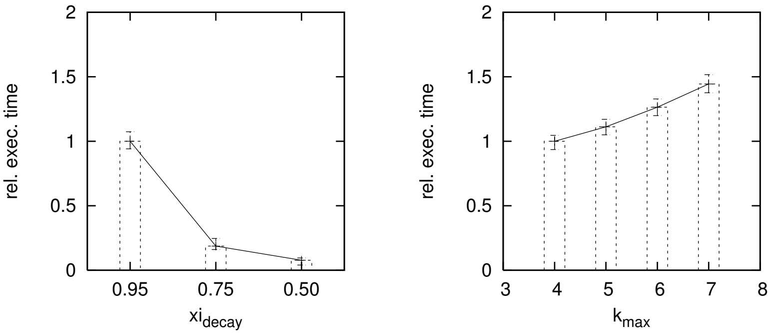
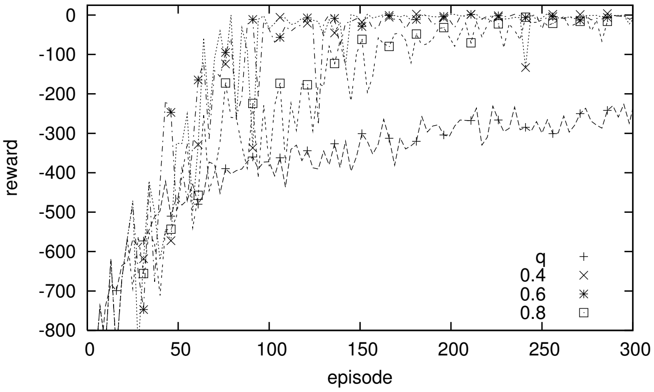
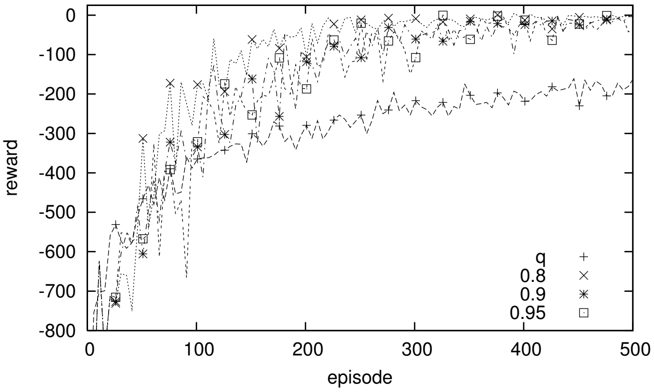


Figure 5.18: Execution times in 5×5 taxi problem with one passenger for different values of (a) ξ_{decay}, and (b) k_{max}.



(a)

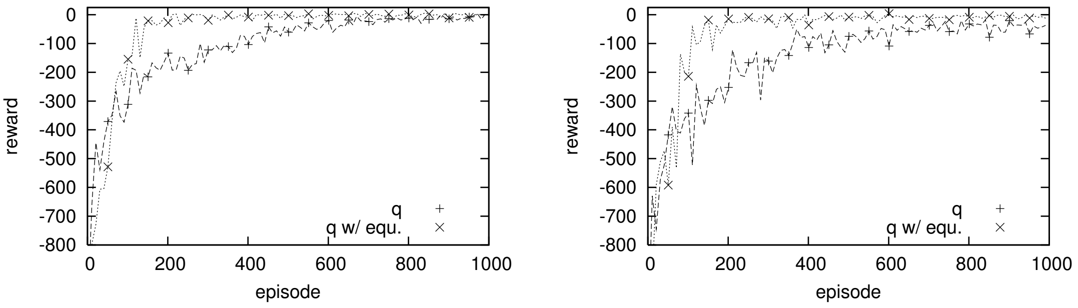


(b)

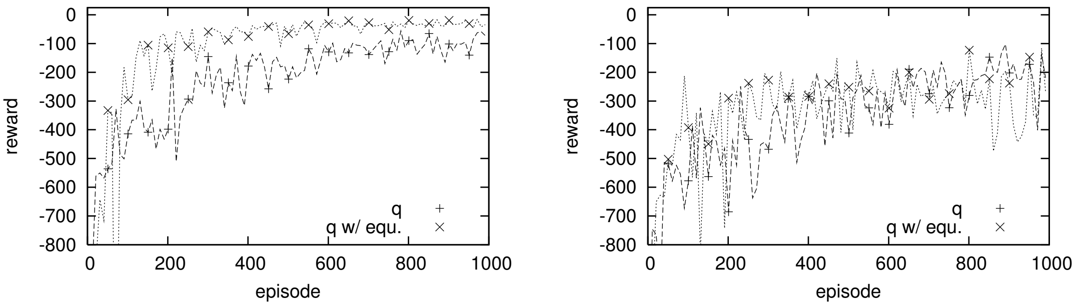
Figure 5.19: Effect of $\tau_{similarity}$ on 5×5 taxi problem with one passenger.

tree shrinks substantially as k_{max} decreases which directly affects the execution time (Figure 5.18(b)). Although that minimum and maximum π-history lengths are inherently problem specific, in most applications, k_{max} near k_{min} is expected to perform well as restrictions of partial MDP homomorphisms to smaller state sets will also be partial MDP homomorphisms themselves.

Figure 5.19 show the results obtained using different similarity threshold values, i.e. $\tau_{similarity}$. As $\tau_{similarity}$ increases, less number of states are regarded as equivalent; this leads to a decrease in the number of state-action value updates per time step and therefore the convergence rate also decreases. On the contrary, the range of the reflected experience expands as $\tau_{similarity}$ decreases which improves the performance and speeds up the learning process. The learning curves of small $\tau_{similarity}$ values also indicate that the similarity function can successfully separate equivalent and non-equivalent states with a high accuracy.



$p_{success} = 0.9$ $p_{success} = 0.7$



$p_{success} = 0.5$ $p_{success} = 0.3$

Figure 5.20: Results with different levels of non-determinism in actions on 5 × 5 taxi problem with one passenger.

5.3.4 Effect of Non-determinisim in the Environment

In order to examine how the non-determinism of the environment affect the performance, we conducted a set of experiments by changing $p_{success}$, i.e. the probability that movement actions succeed, in the 5×5 taxi problem. Non-determinism increases with decreasing $p_{success}$. The results are presented in Figure 5.20. Although more fluctuation is observed in the received reward due to increased non-determinism, for moderate levels of non-determinism when the proposed method is applied to Q-learning it consistently learns in less number of steps compared to its regular counterpart. When the amount of non-determinism is very high in the environment, as in case of $p_{success} = 0.3$, path tree cannot capture meaningful information and therefore the performance gain is minimal.

5.3.5 Comparison with Experience Replay

The proposed idea of using state similarities and then updating similar states leads to more state-action value, i.e. Q-value, updates per experience. It is known that remembering past experiences and reprocessing them as if the agent repeatedly experienced

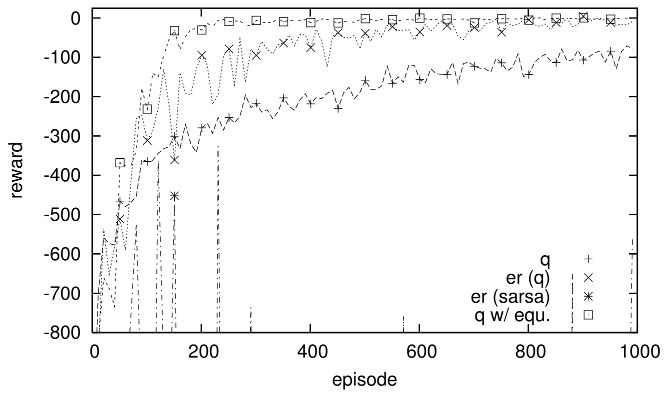


Figure 5.21: Q-learning with equivalent state update vs. experience replay with same number of state updates on 5 × 5 taxi problem with one passenger.

what it has experienced before, which is called *experience replay* [25], speed up the learning process by accelerating the propagation of rewards. In general, a sequence of experiences are replayed in temporally backward order to increase the effectiveness of the process. Experience replay also results in more updates per experience. In order to test whether the gain of the equivalent state update method in terms of learning speed is simply due to the fact that more Q-value updates are made or not, we compared its performance to experience replay using the same number of updates. The results obtained by applying both methods to regular Q-learning algorithm on the 5 × 5 taxi problem are presented in Figure 5.21. Note that in on-policy algorithms (such as Sarsa) or when function approximation is used to store Q-values, only experiences involving actions that follow the current policy of the agent must be replayed for experience replay to be useful. Otherwise, utilities of some state-action pairs may be underestimated[2], causing the process not to converge to optimal policy. We can clearly observe this behavior in Figure 5.21 when Sarsa update rule is used with experience replay; it fails to learn a meaningful policy. When past experiences are replayed using the update rule of Q-learning, the performance of learning improves, but falls short of Q-learning with equivalent state update (optimal policy is attained after 1000 episodes compared to 200). This indicates that in addition to number of updates, how they are determined is also important. By reflecting updates in a semantically rich manner based on the similarity of action-reward patterns, rather than neutrally as in

---

[2] Replaying bad actions repeatedly will disturb the sampling of the current policy in case of on-policy algorithms, and have side effects on the function approximators. More detailed discussion can be found in [25].

experience replay, the proposed method turns out to be more efficient. Furthermore, in order to apply experience replay, the state transition and reward formulation of a problem should not change over time as past experiences may no longer be relevant otherwise. The dynamic nature of the sequence tree allows the proposed method to handle such situations as well.

5.4 Combining State Similarity Based Approach with Option Discovery

The two methods described in this chapter and Chapter 4 are both based on collected state-action(-reward) sequences and applicable to problems containing subtasks. Also, both are meta-level in the sense that they can be applied to any underlying reinforcement learning algorithm. However, they focus on different kinds of abstraction; one of them tries to discover useful higher-level behavior, whereas the other one tries to increase the range of gained experiences. We would expect first method to be more successful in domains with less number of subtasks with large number of instances, since sequence tree allows efficient clustering of such subtasks. On the other hand, the second method is expected to work better on domains containing a large number of simple subtasks, due to the fact that state similarities can be found more efficiently and accurately in that case. Therefore, it is possible to cascade the proposed approaches together in order to combine their impact and improve the performance of learning. The actions to be executed are determined using the extended sequence tree as described in Section 4.1.4, and then the experience is reflected to all similar states using the path tree based approach as described in Section 5.2. This leads to the learning model given in Algorithm 15 that both discovers useful temporal abstractions and utilizes state similarity to improve the performance of learning online, i.e. during the learning process.

We applied the method which combines two proposed approaches to the Q-learning algorithm and compared its effect on different versions of the taxi problem described in Chapter 3.

Initially, Q-values are set to 0 and ϵ-greedy action selection mechanism is used with $\epsilon = 0.1$, where action with maximum Q-value is selected with probability $1 - \epsilon$ and a random action is selected with probability ϵ. The reward discount factor is

Algorithm 15 Reinforcement learning with extended sequence tree and equivalent state update.

```
$T_{seq}$ is an extended sequence tree with $root$ node only.
$T_{path} = \langle root \rangle$    ▷ Initially the path tree contains only the root node.
repeat
    Let $current$ denote the active node of $T_{seq}$.
    $current = root$    ▷ $current$ is initially set to the root node of $T_{seq}$.
    Let $s$ be the current state.
    $h = s$    ▷ Episode history is initially set to the current state.
    repeat    ▷ for each step
        if $current \neq root$ then    ▷ Continue action selection from the current node of $T_{seq}$.
            Let $N = \{n_1, \ldots, n_k\}$ be the set of child nodes of $current$ which contain $s$ in their continuation sets.
            Select $n_i$ from $N$ with sufficient exploration.
            Let $\langle a_{n_i}, \psi_{n_i} \rangle$ be the label of the edge connecting $current$ to $n_i$.
            $a = a_{n_i}$
            $current = n_i$    ▷ Advance to node $n_i$.
        else
            $current = root$
            Let $N = \{n_1, \ldots, n_k\}$ be the set of child nodes of the root node of $T_{seq}$ which contain $s$ in their continuation sets.
            if $N$ is not empty and with probability $p_{sequence}$ then
                Select $n_i$ from $N$ with sufficient exploration.    ▷ Initiate action selection using the extended sequence tree.
                Let $\langle a_{n_i}, \psi_{n_i} \rangle$ be the label of the edge connecting $root$ to $n_i$.
                $a = a_{n_i}$
                $current = n_i$    ▷ Advance to node $n_i$.
            else
                Choose $a$ from $s$ using the underlying RL algorithm.
            end if
        end if
        Take action $a$, observe $r$ and next state $s'$
        Update $Q(s, a)$ based on $s, a, r, s'$.
```

```
        for all $t$ similar to $s$ do
            if probability of receiving reward $r$ at $t$ by taking action $a > 0$ then
                Update $Q(t, a)$ based on $t, a, r, \varsigma(s, t)$.
            end if
        end for
        Append $r, a, s'$ to $h$.    ▷ Update the observed history of events.
        $s = s'$    ▷ Advance to next state.
        if $current \neq root$ then    ▷ Check whether action selection can continue from the active node or not.
            Let $N = \{n_1, \ldots, n_k\}$ be the set of child nodes of $current$ which contain $s$ in their continuation sets.
            if $N$ is empty then
                $current = root$    ▷ Action selection using the extended sequence tree cannot continue from the current state.
            end if
        end if
    until $s$ is a terminal state
    UPDATE-SEQUENCE-TREE($T_{seq}, h$)
    UPDATE-PATH-TREE($H, T_{path}, k_{max}$)
    Traverse $T_{path}$ and update eligibility values.
    if time to re-calculate similarities of states then    ▷ ex. every $n$ episodes.
        CALCULATE-SIMILARITY($T_{path}, k_{min}, k_{max}$)
    end if
until a termination condition holds
```

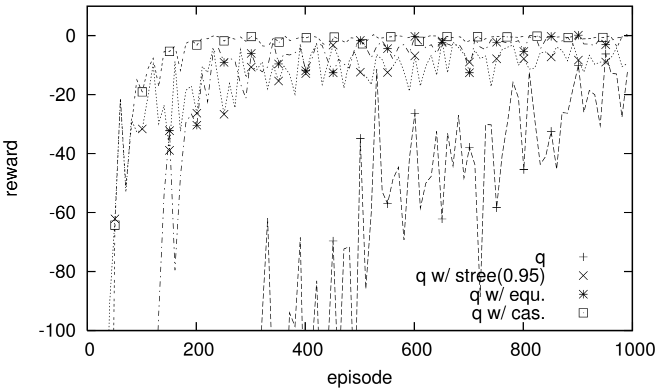


Figure 5.22: Results for the 5 × 5 taxi problem with one passenger.

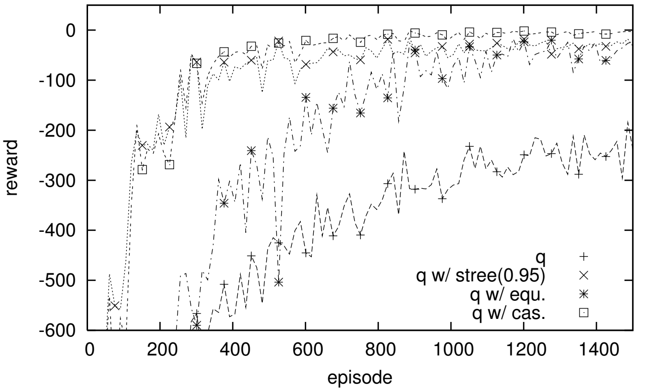


Figure 5.23: Results for the 5 × 5 taxi problem with two passengers.

set to $\gamma = 0.9$ and a learning rate of $alpha = 0.05$ is used. All results are averaged over 50 runs. While building the sequence tree in option discovery approach, ψ_{decay}, ξ_{decay}, and eligibility thresholds are taken as 0.95, 0.99, and 0.01, respectively. The sequence tree is generated during learning without any prior training session, and at each time step processed with a probability of $p_{sequence} = 0.3$ to run the represented set of abstractions. At decision points, actions are chosen ϵ-greedily based on reward values associated with the tuples. While employing state similarity based approach, the path tree is updated using an eligibility decay rate of $\xi_{decay} = 0.95$ and an eligibility threshold of $\xi_{threshold} = 0.1$. In similarity calculations, we set $k_{min} = 3$, $k_{max} = 5$, and $\tau_{similarity} = 0.8$. The path tree is updated and state similarities are calculated after each episode. An episode is terminated automatically if the agent could not successfully complete the given task within 20000 time steps.

We first applied the combined method to 5 × 5 taxi problem with one passenger. Figure 5.22 shows the progression of the reward obtained per episode. As it can be

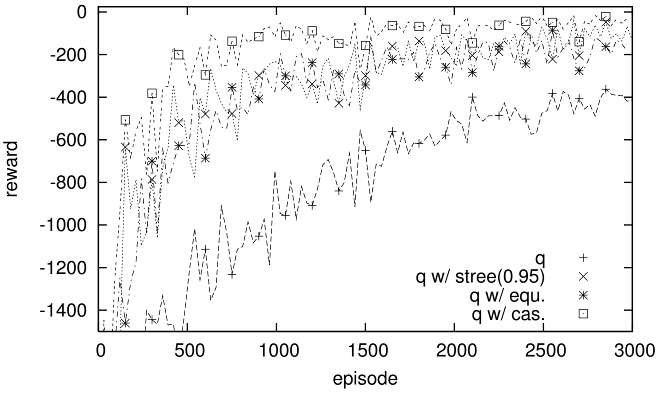


Figure 5.24: Results for the 5 × 5 taxi problem with four passengers.

seen from the learning curves, when both methods are used together, the agent learns faster compared to the situation in which they are used individually. Furthermore, the fluctuations are smaller which means that online performance is also more consistent. The results show that two methods do not affect each other in a negative way.

Results for the 5 × 5 taxi problem with two and four passengers are presented in Figure 5.23 and Figure 5.24. In both cases, we observed an improvement when the combined method is used. As the number of passengers increase, i.e. the complexity of the problem increases, the improvement becomes more evident.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis, we first proposed and analyzed the interesting and useful characteristics of a tree-based learning approach that utilizes stochastic conditionally terminating sequences. We showed how such an approach can be utilized for better representation of temporal abstractions. First, we emphasized the usefulness of discovering Semi-Markov options automatically using domain information acquired during learning. Then, we demonstrated the importance of constructing a dynamic and compact sequence-tree from histories. This helps to identify and compactly represent frequently used sub-sequences of actions together with states that are visited during their execution. As learning progresses, this tree is constantly updated and used to implicitly locate and run the appropriately represented options. Experiments conducted on three well-known domains – with bottleneck states, repeated subtasks and continuous state space, and macro-actions, respectively – highlighted the applicability and effectiveness of utilizing such a tree structure in the learning process. The reported test results demonstrate the advantages of the proposed tree-based learning approach over the other learning approaches described in the literature.

Secondly, we demonstrated a novel approach, which during learning, identifies states similar to each other with respect to action-reward patterns and based on this information reflects state-action value updates of one state to multiple states. Experiments conducted on two well-known domains highlighted the applicability and effectiveness of utilizing such a relation between states in the learning process. The reported test results demonstrate that experience transfer performed by the algorithm is an attractive approach to make learning systems more efficient.

In this work we mainly focused on single-agent reinforcement learning, in which other agents in the system are treated as a part of the environment. In the case of multiple learning agents co-existing in the same environment this is problematic since the environment is no longer stationary; the observations and actions of other agents must also be taken into account while building policies. This necessitates the use of game theoretic concepts, and in particular notion of stochastic games is needed to model the interaction between learning agents and the environment [26, 21, 22, 5, 27, 18, 55]. Also, directly extending the state-action(-reward) histories, which both methods are based upon, and related algorithms to include all such information from each agent would not suffice or produce expected results, since, depending on the agent's situation and subtask to be solved, the behavior of the agent may depend only a subset of other agents in the environment. The agent will additionally be faced with the decision of selecting which history streams from other agents to consider. This would require more complex metrics to be devised then those currently used. Thesis work will be extended to handle multi-agent cooperative learning.

Our future work will first examine guaranteed convergence of the proposed methods and their adaptation to more complex and realistic domains which would require the use of function approximators, such as neural networks, to represent larger state and action spaces.

REFERENCES

[1] Robocup official site. http://www.robocup.org.

[2] E. Alonso, M. d'Inverno, D. Kudenko, M. Luck, and J. Noble. Learning in multiagent systems. *Knowledge Engineering Review*, 16(3):277–284, 2001.

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

[4] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[5] M. Bowling and M. M. Veloso. An analysis of stochastic game theory for multiagent reinforcement learning. Technical report CMU-CS-00-165, Computer Science Department, Carnegie Mellon University, 2000.

[6] S. J. Bradtke and M. O. Duff. Reinforcement learning methods for continuous-time Markov decision problems. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 393–400. The MIT Press, 1994.

[7] S. J. Bradtke and M. O. Duff. Reinforcement learning methods for continuous-time markov decision problems. In *Advances in Neural Information Processing Systems 7*, pages 393–400, 1994.

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

[9] B. Digney. Learning hierarchical control structure for multiple tasks and changing environments. In *SAB '98: Proceedings of the Fifth Conference on the Simulation of Adaptive Behavior*, 1998.

[10] S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy of autonomous agents. In *ECAI '96: Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages*, pages 21–35, London, UK, 1997. Springer-Verlag.

[11] S. Girgin and F. Polat. Option discovery in reinforcement learning using frequent common subsequences of actions. In *IAWTIC '05: International Conference on Intelligent Agents, Web Technologies and Internet Commerce*, pages 371–376, Los Alamos, CA, USA, 2005. IEEE Computer Society.

[12] S. Girgin, F. Polat, and R. Alhajj. Effectiveness of considering state similarity for reinforcement learning. In *IDEAL '06: International Conference on Intelligent Data Engineering and Automated Learning*. Springer Verlag, 2006.

[13] S. Girgin, F. Polat, and R. Alhajj. Improving reinforcement learning by using stochastic conditionally terminating sequences. Submitted for Journal Review, 2006.

[14] S. Girgin, F. Polat, and R. Alhajj. Learning by automatic option discovery from conditionally terminating sequences. In *ECAI '06: Proceedings of the 17th European Conference on Artificial Intelligence*, 2006.

[15] S. Girgin, F. Polat, and R. Alhajj. Positive impact of state similarity on reinforcement learning performance. IEEE Transactions on System, Man, and Cybernetics Part B: Cybernetics, 2007.

[16] S. Girgin, F. Polat, and R. Alhajj. State similarity based approach for improving performance in rl. In *IJCAI '07: Proceedings of the 20th International Joint Conference on Artificial Intelligence*, 2007.

[17] R. Givan, T. Dean, and M. Greig. Equivalence notions and model minimization in markov decision processes. *Artificial Intelligence*, 147(1-2):163–223, 2003.

[18] A. Greenwald and K. Hall. Correlated-q learning. In *ICML '03: Proceedings of the Twentieth International Conference on Machine Learning*, pages 242–249, Washington DC, 2003. AAAI Press.

[19] M. E. Harmon and S. S. Harmon. Reinforcement learning: A tutorial. http://www-anw.cs.umass.edu/ mharmon/rltutorial/, 1996.

[20] M. Hauskrecht, N. Meuleau, L. P. Kaelbling, T. Dean, and C. Boutilier. Hierarchical solution of Markov decision processes using macro-actions. In *Uncertainty in Artificial Intelligence*, pages 220–229, 1998.

[21] J. Hu and M. P. Wellman. Multiagent reinforcement learning: Theoretical framework and an algorithm. In *ICML '98: Proceedings of the Fifteenth International Conference on Machine Learning*, pages 242–250, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[22] J. Hu and M. P. Wellman. Nash q-learning for general-sum stochastic games. *Journal of Machine Learning Research*, 4:1039–1069, 2003.

[23] L. Kaelbling, M. Littman, and A. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[24] G. Kuhlmann and P. Stone. Progress in learning 3 vs. 2 keepaway. In D. Polani, B. Browning, A. Bonarini, and K. Yoshida, editors, *RoboCup-2003: Robot Soccer World Cup VII.* Springer Verlag, Berlin, 2004.

[25] L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3-4):293–321, 1992.

[26] M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *ICML '94: Proceedings of the Eleventh International Conference on Machine Learning*, pages 157–163, New Brunswick, 1994.

[27] M. L. Littman. Friend or foe q-learning in general-sum markov games. In *ICML '01: Proceedings of the Eighteenth International Conference on Machine Learning*, pages 322–328, June 2001.

[28] P. Maes. Artificial life meets entertainment: Life like autonomous agents. *Communications of ACM*, 38(11):108–114, 1995.

[29] S. Mahadevan, N. Marchallek, T. K. Das, and A. Gosavi. Self-improving factory simulation using continuous-time average-reward reinforcement learning. In *ICML '97: Proceedings of the 14th International Conference on Machine Learning*, pages 202–210. Morgan Kaufmann, 1997.

[30] S. Mannor, I. Menache, A. Hoze, and U. Klein. Dynamic abstraction in reinforcement learning via clustering. In *ICML '04: Proceedings of the 21st International Conference on Machine Learning*, pages 71–78, New York, NY, USA, 2004. ACM Press.

[31] A. McGovern. acquire-macros: An algorithm for automatically learning macro-actions. In the Neural Information Processing Systems Conference (NIPS'98) workshop on Abstraction and Hierarchy in Reinforcement Learning, 1998.

[32] A. McGovern. *Autonomous Discovery of Temporal Abstractions From Interactions With An Environment.* PhD thesis, University of Massachusetts Amherts, May 2002.

[33] A. McGovern and A. G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *ICML '01: Proceedings of the 18th International Conference on Machine Learning*, pages 361–368, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[34] A. McGovern and R. S. Sutton. Macro-actions in reinforcement learning: An empirical analysis. Technical Report 98-79, University of Massachusetts, Department of Comp. Science, 1998.

[35] I. Menache, S. Mannor, and N. Shimkin. Q-cut - dynamic discovery of sub-goals in reinforcement learning. In *ECML '02: Proceedings of the 13th European Conference on Machine Learning*, pages 295–306, London, UK, 2002. Springer Verlag.

[36] T. M. Mitchell. *Machine Learning.* McGraw Hill, 1997.

[37] I. Noda, H. Matsubara, K. Hiraki, and I. Frank. Soccer server: A tool for research on multiagent systems. *Applied Artificial Intelligence*, 12(2–3):233–250, 1998.

[38] R. Parr and S. Russell. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems 10*, pages 1043–1049, 1998.

[39] R. E. Parr. *Hierarchical Control and learning for Markov decision processes.* PhD thesis, University of California at Berkeley, 1998.

[40] D. Precup, R. S. Sutton, and S. P. Singh. Theoretical results on reinforcement learning with temporally abstract options. In *ECML '98: Proceedings of the 10th European Conference on Machine Learning*, pages 382–393, 1998.

[41] B. Ravindran and A. G. Barto. Symmetries and model minimization in markov decision processes. Technical Report 01-43, University of Massachusetts, Amherst, 2001.

[42] B. Ravindran and A. G. Barto. Model minimization in hierarchical reinforcement learning. In *SARSA '02: Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation*, pages 196–211, London, UK, 2002. Springer-Verlag.

[43] B. H. Roth. Architectural foundations for real-time performance in intelligent agents. *Real-Time Systems*, 2(1-2):99–125, 1990.

[44] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, 2003.

[45] O. Simsek and A. G. Barto. Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *ICML '04: Proceedings of the 21st International Conference on Machine Learning*, Banff, Canada, 2004. ACM.

[46] O. Simsek, A. P. Wolfe, and A. G. Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *ICML '05: Proceedings of the 22nd International Conference on Machine Learning*, pages 816–823. ACM, 2005.

[47] S. Singh, T. Jaakkola, M. L. Littman, and C. Szepesvŕi. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 38(3):287–308, 2000.

[48] M. Stolle and D. Precup. Learning options in reinforcement learning. In *Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation*, pages 212–223, London, UK, 2002. Springer Verlag.

[49] P. Stone, G. Kuhlmann, M. E. Taylor, and Y. Liu. Keepaway soccer: From machine learning testbed to benchmark. In I. Noda, A. Jacoff, A. Bredenfeld, and Y. Takahashi, editors, *RoboCup-2005: Robot Soccer World Cup IX.* Springer Verlag, Berlin, 2006. To appear.

[50] P. Stone and R. S. Sutton. Scaling reinforcement learning toward RoboCup soccer. In *ICML '01: Proceedings of the Eighteenth International Conference on Machine Learning*, pages 537–544. Morgan Kaufmann, San Francisco, CA, 2001.

[51] P. Stone, R. S. Sutton, and G. Kuhlmann. Reinforcement learning for RoboCup-soccer keepaway. *Adaptive Behavior*, 2005. To appear.

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

[53] R. S. Sutton, D. Precup, and S. Singh. Between MDPs and semi-MDPs: a framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211, 1999.

[54] C. J. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3/4):279–292, 1992.

[55] M. Weinberg and J. S. Rosenschein. Best-response multiagent learning in non-stationary environments. In *The Third International Joint Conference on Autonomous Agents and Multiagent Systems*, New York, July 2004. To appear.

[56] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[57] M. Zinkevich and T. R. Balch. Symmetry in markov decision processes and its implications for single agent and multiagent learning. In C. E. Brodley and A. P. Danyluk, editors, *ICML '01: Proceedings of the 18th International Conference on Machine Learning*, pages 632–. Morgan Kaufmann, 2001.

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Girgin, Sertan

Nationality: Turkish (TC)

Date and Place of Birth: 19 October 1979, Ankara

Marital Status: Single

email: sertan@ceng.metu.edu.tr

EDUCATION

Degree	Institution	Year of Graduation
MS	METU Computer Engineering	2003
BS	METU Computer Engineering	2000
BS	METU Mathematics (Double Major)	2000
High School	Gazi Anatolian High School, Ankara	1996

WORK EXPERIENCE

Year	Place	Enrollment
2005-2006	University of Calgary, Canada	Researcher
2000-2005	METU Department of Computer Engineering	Research Assistant
1996-2000	TUBITAK Bilten	Part-time Researcher

FOREIGN LANGUAGES

Advanced English, Intermediate German

PUBLICATIONS

1. Girgin S., Polat F., and Alhajj R., Positive Impact of State Similarity on Reinforcement Learning Performance, IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics, 2007

2. Girgin S., Polat F., and Alhajj R., State Similarity Based Approach for Improving Performance in RL, Proceedings of the 20th International Joint Conference

on Artificial Intelligence (IJCAI), 2007

3. Girgin S., Polat F., and Alhajj R., Effectiveness of Considering State Similarity for Reinforcement Learning, Proceedings of the 7th International Conference on Intelligent Data Engineering and Automated Learning (IDEAL), 2006
4. Sahin E., Girgin S., and Ugur E., Area Measurement of large closed regions with a mobile robot, Autonomous Robots, Vol. 21, No. 3, pp 255-266, November 2006
5. Girgin S., Polat F., and Alhajj R., Learning by Automatic Option Discovery from Conditionally Terminating Sequences, Proceedings of the 17th European Conference on Artificial Intelligence (ECAI), 2006
6. Girgin S., and Polat F., Option Discovery in Reinforcement Learning using Frequent Common Subsequences of Actions, Proceedings of International Conference on Intelligent Agents, Web Technologies and Internet Commerce (IAWTIC), 2005
7. Sahin E., Girgin S., and Ugur E., Area Measurement of large closed regions with a mobile robot, METU-CENG-TR-2005-06, Technical Report, 2005
8. Polat F., Cosar A., Girgin S., Tan M., Cilden E., Balci M., Gokturk E., Kapusuz E., Koc V., Yavas A., and Undeger C., Küçük Ölçekli Harekatın Modellenmesi ve Simülasyonu, USMOS'05 I. Ulusal Savunma Uygulamaları Modelleme ve Simülasyon Konferansı, 2005, Ankara, Turkey
9. Girgin S., and Sahin E., Blind area measurement with mobile robots, Proceedings of the 8th Conference on Intelligent Autonomous Systems (IAS 8), 2004, Amsterdam, The Netherlands, (Also published as Technical report CMU-RI-TR-03-42, Carnegie Mellon University, Pittsburgh, USA, October 2003.)
10. Undeger C., Balci M., Girgin S., Koc V., Polat F., Bilir S. and Ipekkan Z., Platform Optimization and Simulation for Surviellance of Large Scale Terrains, Proceedings of Interservice/Industry Training, Simulation and Education Conference, 2002, Orlando, FL

ACADEMIC GRANTS

- The Scientific and Technical Research Council of Turkey, December 2005 - December 2006, Discovery of Temporal Abstraction and Hierarchical Structure in Reinforcement Learning, Project No. 105E181(HD-7)

PERSONAL ACHIEVEMENTS

- Best Graduation Project Reward, Comparison of Genetic Operators in OBDD Variable Ordering Problem, Department of Computer Engineering, Middle East Technical University, 2000
- 2nd Place, II. Programming Contest, ACM-SIGART and Computer Society of Bilkent University, 1996
- Member of National Team, Balkan Olympiad in Informatics, 1995
- Bronze Medal, II. National Olympics in Informatics, TUBITAK, 1994
- 1st Place, Golden Diskette'94 Programming Contest, PCWORLD/Turkey Magazine, 1994
- 1st Place, Golden Diskette'93 Programming Contest, PCWORLD/Turkey Magazine, 1993

HOBBIES

Outdoor sports (XC Skiing, Orienteering, Running, Bicycling, Canoeing), Amateur Radio Operator (TA2MFB, 1999-), Amateur Seaman (2004-)