FACTORED REINFORCEMENT LEARNING USING EXTENDED SEQUENCE TREES

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

BY

COŞKUN ŞAHİN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

AUGUST 2015

Approval of the thesis:

FACTORED REINFORCEMENT LEARNING USING EXTENDED SEQUENCE TREES

submitted by **COŞKUN ŞAHİN** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Gülbin Dural Ünver _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Adnan Yazıcı _____
Head of Department, **Computer Engineering**

Prof. Dr. Faruk Polat _____
Supervisor, **Computer Engineering Dept., METU**

Examining Committee Members:

Prof. Dr. İsmail Hakkı Toroslu
Computer Engineering Department, METU _____

Prof. Dr. Faruk Polat
Computer Engineering Department, METU _____

Prof. Dr. Kemal Leblebicioğlu
Electrical and Electronics Engineering Department, METU _____

Prof. Dr. Göktürk Üçoluk
Computer Engineering Department, METU _____

Assist. Prof. Dr. Mehmet Tan
Computer Engineering Department, TOBB ETU _____

Date: _____

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: COŞKUN ŞAHİN

Signature :

ABSTRACT

FACTORED REINFORCEMENT LEARNING USING EXTENDED SEQUENCE TREES

Şahin, Coşkun
M.S., Department of Computer Engineering
Supervisor : Prof. Dr. Faruk Polat

August 2015, 70 pages

Reinforcement Learning (RL) is an area concerned with learning how to act in an environment to reach a final state while gaining maximum amount of reward. Markov Decision Process (MDP) is the formal framework to define an RL task. In addition to different techniques proposed to solve MDPs, there are several studies to improve RL algorithms. Because these methods are often inadequate for real-world problems. Classical approaches require enumeration of all possible states to find a solution. But when states are described by a number of features in the environment, state space grows exponentially, which is known as *curse of dimensionality* [3]. It is possible to model environments more compactly by taking advantage of new representation. Factored Markov Decision Processes (FMDPs) are used for this purpose and on top of this structure, Factored Reinforcement Learning (FRL) methods are applied to utilize new structured representation. Furthermore, this approach may not be sufficient for large scale problems. Since there are a huge number of states and actions to

consider, learning process requires more time and resources. In this thesis, we propose a compact factored structure to solve this problem.

Automatic detection and use of temporal abstractions during learning is proven to be an effective way to increase learning speed. Repeating patterns are found in different parts of the problem and a common sub-policy is used for all of them without exploring the solution again and again. Extended Sequence Tree (EST) algorithm is an automatic temporal abstraction detection technique that uses history of states and actions to store frequently used patterns in a structured manner and offers alternative actions to the underlying RL algorithm. In this work, we propose a factored automatic temporal abstraction method based on extended sequence tree by taking care of state differences via state variable changes in successive states. The aim is to store useful history portions more compactly to avoid excessive memory usage. The proposed method has been shown to provide significant memory gain on selected benchmark problems.

Keywords: Reinforcement Learning, Markov Decision Process, Factored Markov Decision Process, Learning Abstractions, Extended Sequence Tree, Factored Extended Sequence Tree

ÖZ

BÖLÜMLENMİŞ GENİŞLETİLMİŞ DİZİ AĞAÇLARIYLA TAKVİYELİ ÖĞRENME

Şahin, Coşkun
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Faruk Polat

Ağustos 2015 , 70 sayfa

Takviyeli öğrenme, bir problem ortamında maksimum ödülü almaya çalışırken bitiş durumuna ulaşmak için nasıl davranılması gerektiğini öğrenme ile ilgilenen bir alandır. Takviyeli öğrenmede ortamı tanımlamak için en yaygın kullanılan yapı Markov Karar Süreci'dir (MKS). MKS'leri çözmek için önerilen değişik tekniklere ek olarak takviyeli öğrenme metotlarını farklı yönlerden geliştirmek için birçok araştırma yapılmıştır. Çünkü bu teknikler çoğu zaman gerçek dünya problemlerini çözmede yetersiz kalırlar. Klasik yaklaşımlar, bir çözüm üretebilmek için bütün olası durumların hesaplanmasını gerektirirler. Fakat durumlar, ortamdaki özniteliklerin değerlerinin listesiyle ifade edilebildiğinde, durum uzayı öznitelik sayısına bağlı olarak üstel olarak büyür. Buna *çok boyutluluğun laneti* denir. Yeni ifade biçiminin avantajlarını kullanarak ortam durumunu daha kompakt bir şekilde modellemek mümkündür. Bölümlenmiş Markov Karar Süreçleri (BMKS) bu amaç için kullanılır ve bu yapıya Bölümlenmiş Takviyeli Öğrenme

metotları uygulanarak yeni modelin faydalarından istifade edilir. Fakat bu yaklaşım da büyük ölçekli problemler için yeterli olmayabilir. Değerlendirilmesi gereken durum ve eylemlerin adedinin çok büyük olması nedeniyle öğrenme süreci fazla zaman ve kaynak gerektirir. Bu çalışmada, bu sorunun çözümü için bölümlenmiş kompakt bir yapı önerilmektedir.

Zamansal soyutlamaların otomatik olarak tespit edilip kullanılmasının öğrenme hızını artırdığı ispatlanmıştır. Bu yolla, problemin farklı bölümlerinde tekrar eden şablonlar bulunup hepsinde ortak bir hareket tarzı uygulanarak aynı çözümü tekrar tekrar hesaplamanın önüne geçilmektedir. Genişletilmiş Dizi Ağaçları algoritması, durum ve eylemlerin tarihçelerini, sıklıkla kullanılan şablonları yapısal bir şekilde kaydedip alt katmanda çalışan takviyeli öğrenme algoritmasına alternatif eylemler öneren bir otomatik geçici soyutlama tespit tekniğidir. Bu çalışmada, genişletilmiş dizi ağaçlarına dayanan, birbirlerini takip eden durumlardaki değişken değerlerinin farklılıklarını kullanan, bölümlenmiş bir otomatik geçici soyutlama metodu önerilmektedir. Çalışmadaki amaç, durum ve eylem tarihçelerini daha kompakt bir şekilde saklayıp büyük hafıza kullanımından kaçınmaktır. Metodun önemli ölçüde hafıza kazanımı sağladığı yaygın bir şekilde kabul gören problemler üzerinde gösterilmiştir.

Anahtar Kelimeler: Takviyeli Öğrenme, Markov Karar Süreci, Bölümlenmiş Markov Karar Süreci, Soyutlamaları Öğrenme, Genişletilmiş Dizi Ağacı, Bölümlenmiş Genişletilmiş Dizi Ağacı

to my brother Cihan

ACKNOWLEDGEMENTS

Firstly, I would like to thank my supervisor, Prof. Faruk Polat, who introduced me with the world of Artificial Intelligence. Through my M.Sc. years, I enjoyed taking his courses and discussing the ideas about my thesis with him. I appreciate his full support and patience during my research. I have learned so much from his informational and motivational talks.

I would like to thank Dr. Erkin Çilden for his friendly attitude and great ideas that directed my research. He had never hesitated to help me in all the ways he can do. He has taught me many things about the academic world. This work would have been never done without his guidance.

I would like to thank my family and friends for never leaving me alone through these years. I also appreciate the support of my colleagues who excuse my absence at work while doing my research.

This work is partially supported by the Scientific and Technological Research Council of Turkey (TUBITAK) under Grant No. 113E239 and TUBITAK BIDEB MS scholarship (2210) program.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
ML	Machine Learning
MDP	Markov Decision Process
DP	Dynamic Programming
RL	Reinforcement Learning
TD	Temporal Difference
FRL	Factored Reinforcement Learning
FMDP	Factored Markov Decision Process
DBN	Dynamic Bayesian Network
CPT	Conditional Probability Table
SDP	Structured Dynamic Programming
SVI	Structured Policy Iteration
SPI	Structured Value Iteration
HRL	Hierarchical Reinforcement Learning
EST	Extended Sequence Tree
Factored-EST	Factored Extended Sequence Tree

CHAPTER 1

INTRODUCTION

Learning is the ability to benefit from past experiences while performing new actions. Many researches have been conducted on examining and adapting different learning models on agents so that they can make intelligent decisions. Although definition of *intelligence* depends on the context, it can be generalized as behaving in a reasonable way with limited time and resources, interpreting and using feedbacks to build better plans, adapting them to new domains.

An agent can be anything that acts in an environment, such as a human, a robot, a factory or a vending machine. Artificial Intelligence (AI) is an area concerned with building intelligent agents. It is used in game playing, natural language processing, planning, speech recognition and many other fields. AI problems are defined by means of what to achieve, instead of the way of solving them. Degree of intelligence can be measured considering accuracy of actions, learning speed, resource usage efficiency and other aspects depending on the application. During learning, the agent uses a set of data provided at the beginning or received while execution continues. The data contain the characteristics of the system and outcomes of the interactions with the environment. Unlike human beings, a computer agent mostly requires a structured and organized representation scheme for defining problems and finding solutions.

An *action* is reaction of the agent to its current perception of the environment. Set of these perceptions is called a *state*. It is the snapshot of the environment including information that the agent needs to know what to do next. Communication between environment and agent is carried out through sensors and

actuators (Figure 1.1). After executing an action, the agent receives immediate reward (or punishment) signals indicating how good selecting that action was. The main concern of the agent is to select reasonable actions in each state. The meaning of *reasonable action* depends on the expectations of the agent. It may prefer actions bringing high immediate rewards or the ones that seem bad in the short term, but lead to better states in the future.

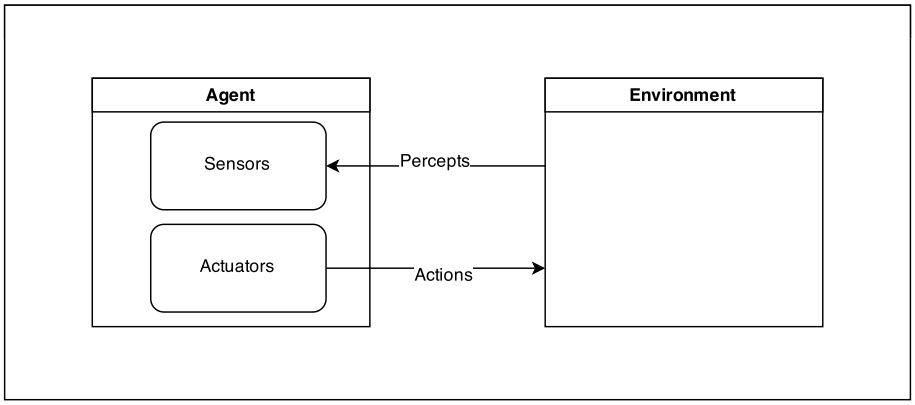


Figure 1.1: Interaction of agents with environments

Features of the environment are important while choosing an appropriate learning method. In *deterministic environments*, performing an action in a particular state always returns the same feedbacks. In other words, next state is certain given the current state and the action fired. Most of the time, this is not the case in real-world problems. There are some factors changing the effects of an action and making the environment *non-deterministic*. So, depending on a probability distribution, the agent can perceive different states and rewards after an action is applied in identical situations. If the probability distributions do not change in time, the environment is said to be *stationary*. Content of the feedbacks also varies among tasks. Through this thesis, we will assume that environments are *fully observable*, meaning that the agent knows the exact state that it is located in. In *partially observable* tasks, sensors cannot receive all state features because of different reasons. For example there may be some noise or simply problem definition relies on partial observability.

Machine Learning (ML) is an area under AI that aims to learn automatically

using experiences. Based on when the data is provided, ML is divided into two categories. In *supervised learning*, the agent is given a training data set at the beginning that will be used to learn and find solutions to unseen cases. For example, in a classical e-mail spam filtering task, e-mails that are known to be spam are used to classify new e-mails. However, in *unsupervised learning*, the agent is guided with rewards and punishments during the learning process. RL is an online ML method, where the agent tries to learn what to do in an environment in order to reach some goal while maximizing the number of rewards[19]. RL methods are more appropriate than supervised learning in interactive environments, where feedbacks are received after each action execution. An internal structure is built with these feedbacks so that the agent can decide next actions using past experiences.

RL tasks are usually defined via MDPs. MDP is a framework that is used to model sequential decision problems with uncertainty. The goal of solving an MDP is to find a *policy* for the agent to perform optimal actions in all states. When state transitions and reward function are known, the agent can find an optimal policy using dynamic programming (DP) approaches. Otherwise, online methods are used to solve the problem. But in both cases, classical RL algorithms are inefficient for problems with large state space. The internal structure grows exponentially with increasing number of state variables which is known as *curse of dimensionality*. In most of the learning problems, states can be defined with a feature set of the environment. In large domains containing a huge number of features, using classical MDP model leads to excessive memory usage and computation cost. FMDPs [5] take the advantage of new state representation to define tasks in a more structured and compact way. In this approach, state transition function is represented via Dynamic Bayesian Networks (DBNs) [7] and reward function is a decision tree. The new model makes it possible to define functions for a subset of states, instead of expressing them for all states one by one. Factored counterparts of the methods built on MDPs can be constructed with the new representation, too. Similar to FMDPs, these methods use decision trees while expressing other functions, such as policies and values of states.

Decision trees are generally used to test instances of a set of variables in order to find the corresponding values in the leaf nodes. Given a list of variable instances, starting from the root node, path to be followed is determined by the value of variable located in the current node. In FMDPs, decision tree nodes store the state variables while edges are labeled with values of corresponding variables. Leaves show the actual data, such as probability distribution, reward values etc., which is assigned to all the states in the state set of the path. For example, in an environment with 16 binary features (meaning that each of them takes *true* or *false* values), there are 2^{16} possible states. If a policy function that is found by following a path from the root to one of the leaves contains 10 variables, the action in the leaf node is independent of the remaining 6 variables. Thus, it is defined for $2^6 = 64$ states in the given policy. Grouping elements having common features prevents generating all data during computation, resulting in less computation time and memory usage. A DBN is a graphical representation of conditional dependencies of a set of variables. In FMDPs, a new DBN is constructed for each action. DBN of action a shows which variables affect the next value of each variable when action a is performed. In the new model, computations are based on the variables instead of states.

During the learning process, the agent may encounter some sub-tasks which are similar or exactly the same. Each of them may repeat many times at different regions of the solution space. Although all instances of these tasks have similar or exactly the same solutions, the agent attempts to solve them separately without any knowledge of previous achievements. This makes it difficult for the agent to converge to an optimal policy in a reasonable time [12]. Temporal abstractions are used to share solutions among similar sub-tasks and thus improve learning performance. *Options framework* [20] is a widely accepted formalism aiming to extend RL with temporal abstractions. Options can be provided in the problem definition or found during learning. Specifying them in advance becomes hard when state and action spaces are large. But discovering them automatically is more challenging and interesting issue.

Extended Sequence Tree (EST) is one of the abstraction methods based on options framework. It attempts to automatically detect and employ useful sub-

policies during RL [12]. It is a tree-based approach that makes use of history of states and actions. EST builds a suffix tree representing common useful parts of state-action-reward sub-sequences in a unified and compact form and eventually uses these abstract skills to speed up learning. Periodically, latest history is provided to EST structure and after extracting possibly frequently used sub-histories, they are added to the tree with some attributes to keep eligibilities of corresponding states and actions. After getting the feedback from the environment, an alternative action is provided to the learning algorithm and one of the proper paths in EST is followed to detect successful past action sequences. In order to decrease memory consumption, useless action sequences are pruned and the structure is updated dynamically. EST is an effective solution for classical MDP tasks. It can also be used in FMDPs without any modification, but for large domains, the structure may use huge amount of memory.

There are some temporal abstraction techniques used in factored environments. Some of them focuses on the model and try to decompose FMDP into smaller sub-tasks that can be solved more easily. In this thesis we introduce an extended version of EST, *Factored Extended Sequence Tree* (Factored-EST), that exploits the factored structure in a way that options are based on the state variables instead of states [16]. The main objective of the approach is minimizing the memory usage of EST in factored models in order to solve the curse of dimensionality problem in huge domains without affecting learning performance. Factored-EST relies on the assumption that each action changes values of only a small subset of variables. So, history of events can be represented more compactly by considering just modified variables in successive steps instead of storing all of them. New history encoding stores all variable values of initial states, changing variables in each step and corresponding actions and immediate rewards. Internal EST node structure is modified so that set of initial states are managed in a decision tree. Differences between subsequent states kept in a directed acyclic graph with a pointer to the initial state of the history to calculate values of all variables when necessary. We conducted experimental study to show effectiveness of factored-EST compared to EST and HEXQ algorithms.

The rest of the thesis is organized as follows. Chapter 2 gives some background

information about how to represent and solve learning problems. In addition, some temporal abstraction methods on MDPs and FMDPs are discussed here. After that we present our approach to build a memory efficient options discovery framework for factored environments in Chapter 3. Chapter 4 contains experimental study to justify our contribution. Conclusion and future research directions are highlighted in Chapter 5.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Markov Decision Processes

MDP is a framework for decision-theoretic planning. It is used to model decision making in environments with *Markov property*, which means that next state only depends on the current state and the action being performed. In other words, state transitions are independent of the history of states that is observed until reaching the current state. Markov property makes it easier to model and solve learning problems. MDP is a tuple $\langle S, A, T, R \rangle$ where:

- S : A finite set of states,
- A : A finite set of actions,
- T : Transition function $T : S \times A \times S \rightarrow [0, 1]$,
- R : Reward function $R : S \times A \rightarrow \Re$.

T defines the transition probabilities between states based on the selected action. Having property $\forall s \in S$, $\forall a \in A$, $\sum_{s' \in S} T(s, a, s') = 1$, $T(s, a, s')$ is the probability of reaching to state s' when action a is fired in state s. In episodic tasks, there is a subset of S containing *terminal (or goal) states*, in which taking an action will not change the state. In such tasks, learning starts from the beginning when one of the goal states is reached. $R(s, a)$ is the immediate reward gained by the agent after taking action a in state s. Figure 2.1 shows a simple MDP with 4 states and 4 actions. The arrows show transitions between states.

If state s has an outgoing link to state s' with an edge labeled with (a, p), then the transition function gives $T(s, a, s') = p$, where $0 \leq p \leq 1$. For example, if we choose action a_2 in state s_1, the agent will go to s_2 with 0.1 probability, to s_4 with 0.6 probability and s_3 with 0.3 probability. For the sake of simplicity, transitions with 0 probability are not shown in the figure. There is only one terminal state, which is s_4, with no outgoing links. Note that, performing an action does not have to change current state, which is the case in s_1 for action a_1 with probability 0.6.

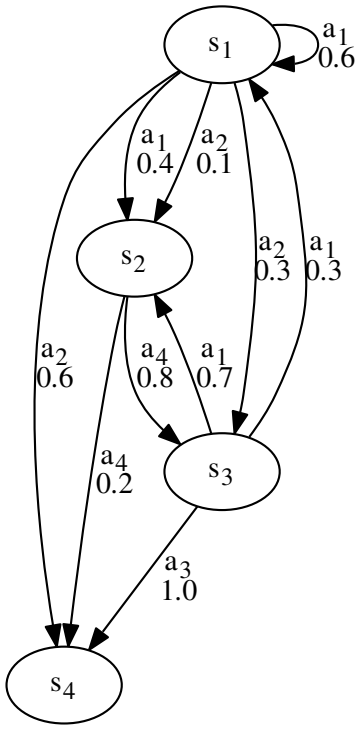


Figure 2.1: An example of Markov Decision Process

In learning tasks, agents can partially control the current state by taking actions. After an action is performed, the agent observes the outcomes of the action and receives some feedback from the environment. Assuming *fully observability*, this feedback includes the next state and an immediate reward. In *partially observable MDPs*, agent may not receive the exact effects of the action. In

addition, some problems require infinitely many states or actions which are out of scope of this thesis.

In MDP tasks, the goal is to find an optimal policy π^*, which maximizes the total expected reward received by the agent. A policy $\pi : S \times A \rightarrow [0, 1]$ is a mapping that defines the probability of selecting an action in a particular state. If $\forall s{\in}S$, $\exists a_s{\in}A$ such that $\pi(s, a_s) = 1$ and $\forall a{\in}A$, where $a{\neq}a_s$, $\pi(s, a) = 0$, the policy is called *deterministic policy*. In *stationary policies*, mappings between states and actions are independent of the history of the agent and time. Thus, stationary policies satisfy Markov property, too.

The value of a policy π, denoted as $V^\pi(s)$, gives the expected cumulative reward that can be collected by the agent after following policy π in state s. π is said to be better than policy π' if $\forall s{\in}S$, $V^\pi(s) > V^{\pi'}(s)$. Thus, optimal solutions can be determined by comparing values of the policies. The simplest way of finding the best policy in a deterministic world is trying all the possibilities, which is an exponential operation with $|S|^{|A|}$ possible policies. In most cases, it will be impractical to consider all candidate policies. When reward and transition functions are known, an optimal policy can be found using classical dynamic programming techniques [19]. These techniques calculate value of each state for a given policy with the formula:

$$V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} T(s, a, s')[R^a_{ss'} + \gamma V^\pi(s')]. \tag{2.1}$$

Equation 2.1, known as *Bellman equation* [2], defines the relationship between value of current state and next states. $R^a_{ss'}$ is the immediate reward of taking action a in state s and making transition to s' and $T(s, a, s')$ gives the probability of ending up in state s' after applying the action. Value of a state depends on the immediate reward and value of the next state discounted by a factor $\gamma \in [0, 1]$. Similarly, state-action value function (Q function) [22], denoted as $Q^\pi(s, a)$, estimates the value of taking action a in state s and following policy π till the end. It can be expressed as:

$$Q^{\pi}(s,a) = \sum_{s'} T(s,a,s')[R^{a}_{ss'} + \gamma V^{\pi}(s')] \tag{2.2}$$

Assuming that $Q^{*}(s,a)$ is the state-action value function in an optimal policy, it is found by

$$Q^{*}(s,a) = \sum_{s'} T(s,a,s')[R^{a}_{ss'} + \gamma V^{*}(s')]. \tag{2.3}$$

$V^{*}(s)$ is determined by finding the best expected value of states greedily as

$$V^{*}(s) = max_{a} Q^{*}(s,a), \tag{2.4}$$

and thus the optimal policy π^{*} is

$$\pi^{*}(s) = argmax_{a} Q^{*}(s,a). \tag{2.5}$$

The way of selecting a learning method for a given task is related with the characteristics of the task, like in real-world problems. Learning a programming language most probably requires different skills and road maps than learning how to swim or play the piano. Thus, effectiveness of solution methods highly depends on the application context. Problem definition, characteristics of feedbacks received from the environment, existence of any noise and definition of a successful solution should be taken into consideration while determining the approach. One of the main steps of selecting an approach is deciding if learning should be online or offline. In online methods, agent observes the outcomes after an action is fired. But in offline methods, it is assumed that transition and reward functions are provided in the problem definition. So, agent knows the possible outcomes before applying an action.

2.1.1 Dynamic Programming for MDPs

There are two different approaches to solve an MDP when model of the environment is provided to the agent, known as *policy iteration* and *value iteration*.

Policy iteration algorithm starts with an arbitrary policy and improves it iteratively. In each iteration, *policy evaluation* and *policy improvement* operations are performed until the current policy becomes stable. The algorithm is provided in Algorithm 1. In policy evaluation part, value of each state is calculated until it converges to some extent. The algorithm uses a threshold θ as a stopping criteria of this stage. Policy improvement part tries to find an alternative action for each state in the policy, such that it brings more expected cumulative reward. Alternative actions are found with a greedy approach, using calculated state values in policy evaluation.

Value iteration [2] starts with arbitrary initial values for all states and updates them iteratively. Independent of the initial values, it converges to final values after sufficient number of computations. Unlike policy iteration algorithm, a deterministic policy is calculated after the convergence. Algorithm 2 shows steps of value iteration. Value update part actually uses action-value function $Q(s, a)$ and follows greedy approach to calculate the value function $V(s)$.

2.2 Reinforcement Learning

RL is an online learning method that learns to map situations to actions such that long-term accumulated reward gained by the agent is maximized. A reward, $r \in \Re$, is a signal received after each move, which guides the agent through the achievement of goal. Thus, rewards are expected to be provided in a way that they will not be misleading [19]. The agent uses trial-and-error method since model of the environment is not provided. An action is chosen and performed, as a result next state and immediate reward is observed. Use of feedbacks depends on type of the RL method. Model of the environment is specified by the transition and reward function. RL approaches can be categorized as *model-free* and *model-based* algorithms based on the need of an environment model [14].

Algorithm 1 Policy iteration algorithm

procedure POLICY-ITERATION(S, A, R, θ)
Require: S is the set of states
Require: A is the set of actions
Require: R is the reward function
Require: θ is a threshold to stop policy evaluation
    Fill $V(s) \in \Re$ and $\pi(s) \in A(s)$ arbitrarily $\forall s \in S$ ▷ Initialization
    $stable \leftarrow false$
    while *stable* is $false$ **do**
        $stable \leftarrow true$
        $\Delta \leftarrow infinite$
        while $\Delta > \theta$ **do** ▷ Policy evaluation
            $\Delta \leftarrow 0$
            for $s \in S$ **do**
                $v \leftarrow V(s)$
                $V(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R_{ss'}^{\pi(s)} + \gamma V(s')]$
                $\Delta \leftarrow max(\Delta, |v - V(s)|)$
            end for
        end while
        for $s \in S$ **do** ▷ Policy improvement
            $a' \leftarrow \pi(s)$
            $\pi(s) \leftarrow argmax_a \sum_{s'} PT(s, a, s')[R_{ss'}^{a} + \gamma V(s')]$
            if $a' \neq \pi(s)$ **then**
                $stable \leftarrow false$
            end if
        end for
    end while
    return $\pi(s), V(s)$
end procedure

In *model-free algorithms*, the agent is not concerned with the problem dynamics and tries to find an optimal policy by using values of states. Observations gathered after an action is fired are used to update value of the current state for the corresponding action. After a number of incremental update operations, the value function converges to an optimal solution. Temporal difference learning [17] and adaptive heuristic critic [1] methods are examples of this approach. But *model-based algorithms* first build a model for the environment from the feedbacks. An optimal solution is computed using the resulting model. For example, Dyna [18] uses one of the TD-learning methods after the model is constructed. Model-based methods are preferable in domains where real world interaction is expensive.

Algorithm 2 Value iteration algorithm

procedure VALUE-ITERATION(S, A, R, θ)
Require: S is the set of states
Require: A is the set of actions
Require: R is the reward function
Require: θ is a threshold to stop value evaluation
    Fill $V(s) \in \Re$ arbitrarily $\forall s \in S$ ▷ Initialization
    $\Delta \leftarrow infinite$
    while $\Delta > \theta$ **do** ▷ Until values converges more than a threshold θ in each step
        $\Delta \leftarrow 0$
        for $s \in S$ **do**
            $v \leftarrow V(s)$
            $V(s) \leftarrow \sum_{s'} P_{ss'}^{\pi(s)}[R_{ss'}^{\pi(s)} + \gamma V(s')]$ ▷ Update value of state s
            $\Delta \leftarrow max(\Delta, |v - V(s)|)$
        end for
    end while
    for $s \in S$ **do** ▷ Policy calculation
        $\pi(s) \leftarrow argmax_a \sum_{s'} P_{ss'}^{a}[R_{ss'}^{a} + \gamma V(s')]$
    end for
    return $\pi(s), V(s)$
end procedure

DP methods suffer from time and space requirements while storing and processing transition matrices [6]. Unlike these methods, in RL, agent learns by interacting with the environment, collects statistics about effects of acting and uses them to improve the current policy. One of the key concepts in RL is to keep a balance between *exploitation* and *exploration*. Exploitation is using past experiences while deciding new actions to perform. Depending on the learning rate, exploitation may result in successful and unsuccessful attempts. Exploration is for searching unexplored features of the environment by choosing different actions than the ones that seem to be optimal. This is commonly referred to action selection in RL. Most popular of them are *ϵ-greedy selection* and *Boltzman selection*. ϵ-greedy approach generates a random number $r \in [0, 1]$ and next action is selected randomly if $r < \epsilon$, otherwise best action is chosen. In Boltzman selection mechanism, in state s, action a can be selected with probability calculated by formula in Equation 2.6.

$$p(a) = \frac{e^{Q(s,a)/\tau}}{\sum_{b} e^{Q(s,b)/\tau}}, \tag{2.6}$$

where τ is *temperature* and determines how randomly the actions are chosen. When τ is very high, all actions have equal probabilities while small τ values gives priority to actions with greater values.

2.2.1 Temporal Difference Learning

TD-learning methods approximate optimal policy and value functions without requiring an environment model. After each step, value of the current state is estimated based on the learned values. Assume that rewards that are collected after k steps are $\{r_1, r_2, r_3, , \cdots, r_k\}$. A good estimation of expected value, V_k, is found by taking average of these results,

$$\begin{aligned} V_k &= (r_1 + r_2 + r_3 + \cdots + r_k)/k, \\ kV_k &= (r_1 + r_2 + r_3 + \cdots + r_{k-1}) + r_k. \end{aligned} \tag{2.7}$$

Since the first expression on the right hand side can be rewritten using V_{k-1}, the equation can be rewritten as

$$kV_k = (k-1)V_{k-1} + r_k. \tag{2.8}$$

Dividing both sides by k gives

$$\begin{aligned} V_k &= ((k-1)/k)V_{k-1} + r_k/k, \\ &= (1 - (1/k))V_{k-1} + r_k/k. \end{aligned} \tag{2.9}$$

If α is taken as $1/k$, the equation becomes

$$\begin{aligned} V_k &= (1-\alpha)V_{k-1} + r_k\alpha, \\ &= V_{k-1} + \alpha(r_k - V_{k-1}). \end{aligned} \tag{2.10}$$

It is very common to take α as independent of k. Because, when step number increases, the effect of current observation decreases in equation 2.10. In order to favor the latest data, it can be taken constant.

Equation 2.11 shows the update formula for value of state s_t in TD learning after adding value of next state s_{t+1}.

$$V(s_t) = V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)], \tag{2.11}$$

where $\alpha \in (0, 1)$ is *learning rate* and $\gamma \in (0, 1)$ is *discount factor*. Since current estimated value is used instead of the one in the current policy, like DP approaches, TD-learning is a bootstrapping method. SARSA uses the current policy that is being followed for updating the values while Q-learning [23] is independent of the policy.

Q-learning algorithm is an off-policy, model-free method that updates state-action value, $Q(s, a)$, after performing action a in state s and receiving the feedback. Learning process consists of a set of episodes and at each step in an episode, action selection mechanism decides to the next move and Q-table is updated afterwards. Basic Q-learning algorithm is given in Algorithm 3, where $\alpha \in [0, 1]$ is the learning rate and $\gamma \in [0, 1]$ is the discount factor. When α becomes close to 0, value of $Q(s, a)$ is updated more slowly, while greater α results in more exploration.

2.3 Temporal Abstraction in Reinforcement Learning

Temporal abstractions are used to increase the learning speed in RL by introducing new abstract actions into the system. These actions, lasting more than one time step, are the solutions to some sub-MDPs of the main problem. After finding sub-policies of the common tasks in a domain, the agent does not have to solve them again. When an already-solved task is encountered, the abstract actions containing the sequence of primitive actions in the corresponding sub-policy is used and therefore learning rate is improved. The techniques in

Algorithm 3 Q-learning algorithm

procedure Q-LEARNING(S, A, α, γ)
Require: S is the set of states
Require: A is the set of actions
Require: α is the learning rate
Require: γ is the discount factor
    Initialize Q(s,a) for each $s \in S$ and $a \in A$ arbitrarily
    for each episode **do**
        $s \leftarrow initialstate$
        for each step **do**
            Choose action to apply with an action selection mechanism
            Observe next state s' and reward r
            $Q(s,a) = (1-\alpha)Q(s,a) + \alpha(r + \gamma max_{a' \in A} Q(s', a'))$ ▷ Q-table update
            $s \leftarrow s'$
        end for
    end for
end procedure

temporal abstraction are mainly different from each other by the way of introducing abstract actions. Sometimes, abstractions are provided in the problem definition. This requires extensive understanding of the problem domain which is not possible most of the time. Because, defining abstractions manually for every task that is intended to be solved becomes tedious, especially in large domains. This prevents us from developing a flexible algorithm for general use. The alternative approach is to discover them automatically. At the beginning of learning, MDP is divided into sub-MDPs and all of them are solved separately. Then, the solutions are merged to find actual solutions to the task. HEXQ algorithm [13] is an example of this idea. Some prefers learning abstractions during learning by saving useful sub-policies and using them later when necessary, like EST [12] which will be explained in the next section.

2.3.1 Extended Sequence Tree Algorithm

Extended Sequence Tree (EST) is a sequence-based automatic temporal abstraction method, which uses sequences of state-action-reward tuples to identify sub-tasks that the agent tries to solve multiple times during learning. It stores useful parts of the history in a tree structure and makes use of them to suggest options to underlying RL algorithm using conditional branching. The tree is

dynamically updated after receiving new set of histories. The algorithm tries to use the similarities between the sequences to construct a compact structure and favors the portions that are more common among them. In order to do that, each state and action in the tree stores an attribute showing the usage frequency of the element. The ones having values lower than a given threshold are pruned in order to filter the uncommon parts and thus memory usage is decreased.

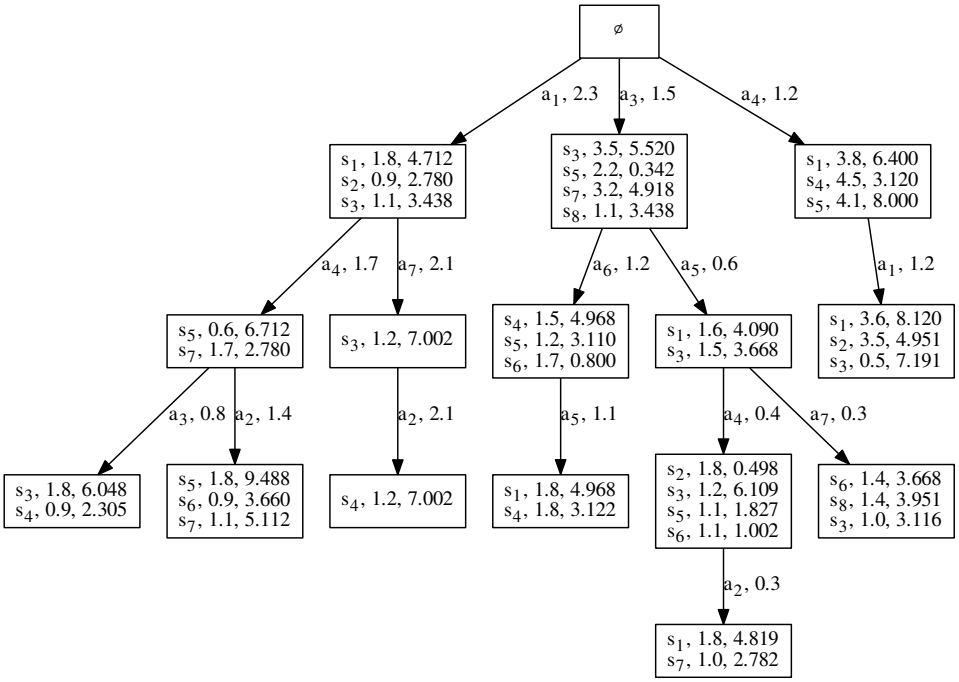


Figure 2.2: An example Extended Sequence Tree structure

EST is a tuple $\langle N, E \rangle$ where N is the set of nodes and E is the set of edges. Each node is a list of tuples $\langle s_i, \xi_i, R_i \rangle$, called *continuation set*, having a state s_i, its expected cumulative reward R_i and eligibility value ξ_i. Edges are labeled as $\langle a, \psi \rangle$, where a is the action that can be performed on the states included in the child EST node. Each path connecting root node to one of the other nodes represents a unique action sequence, in other words, an option. If the action sequence of node q can be obtained by appending action a to action sequence of node p, node p is connected to node q with an edge labeled as $\langle a, \psi \rangle$. ψ is the eligibility value of the edge indicating how frequently action sequence of q

is executed. Action a is allowed to be applied if the current state s is included in the continuation set of node q. Eligibility value ξ_s of state s shows how frequently the action a is selected in this state. Figure 2.2 is an example of an EST constructed using a list of histories. The root node, denoted as $\emptyset$, is the starting point of each option and has no elements in its continuation set. Note that states and actions can appear multiple times in different parts of the tree. The structure makes it possible to follow different options by comparing the attribute values of the elements.

Algorithm 4 Reinforcement Learning with Extended Sequence Tree

Require: T is an EST with only root node
Require: A is the set of actions
repeat
  Let *current* is active node of T
  $current = root$ ▷ Initially current node is the root
  Let s be the current state
  $h = s$ ▷ History includes only initial state at the beginning
  $active = false$ ▷ Initially EST is not active
  repeat
    a_{next} = SELECT-ACTION ▷ Select the next action
    Take action a_{next} and observe reward R and next state s'
    Update state-action values using underlying RL algorithm
    Append r, a_{next}, s' to h
    $s = s'$
  until s is a terminal state
  UPDATE-TREE(T, h) ▷ Update tree with the new history
until a termination condition is satisfied

Algorithm 4 shows the general usage of EST structure. Until learning rate reaches to an acceptable level, multiple episodes are run on the problem. The action selection mechanism of the underlying RL algorithm is modified, in a way that it takes into consideration the actions that EST recommends. In each episode, starting from the root node, the structure tries to find a meta-action that is suitable for the current state of the agent. If RL algorithm uses the action offered by EST, it becomes active and tries to follow a path in the tree using expected cumulative values of states. The process terminates when a terminal state is reached or EST cannot find an action for the current state. In the next step, it start from the beginning. When EST is not active, RL algorithm uses its own action selection method.

In specified periods, history of state-action-reward tuples are collected and given to the EST. The structure is dynamically updated after a set of sub-histories are generated from the given history. Assuming h is the corresponding history where

$$h = s_1 a_1 r_2 \cdots s_i a_i r_{i+1} s_{i+1} a_{i+1} r_{i+2} \cdots s_{i+k} a_{i+k} r_{i+k+1} s_{i+k+1} a_{i+k+1} r_{i+k+2} \cdots r_t s_t,$$

the total discounted cumulative reward of h is $r_2 + \gamma r_3 + \cdots + \gamma^{t-2} r_t$. For each s appearing in h, we can find a path from s to the terminal state s_t, which is possibly a sub-history of the optimal policy π^*. The idea is to explore the starting points of frequently encountered sub-problems in the history. Some of the states may be encountered multiple times, which gives multiple sub-histories after the extraction. For example, if $s_i = s_{i+k}$, two alternative histories for s_i are:

$$h_{i_1} = s_i a_i r_{i+1} s_{i+1} a_{i+1} r_{i+2} \cdots s_{i+k} a_{i+k} r_{i+k+1} s_{i+k+1} a_{i+k+1} r_{i+k+2} \cdots r_t s_t$$

$$h_{i_2} = s_{i+k} a_{i+k} r_{i+k+1} s_{i+k+1} a_{i+k+1} r_{i+k+2} \cdots r_t s_t.$$

In order to decide which one is better, discounted cumulative rewards of these histories are compared, and the one with the greater value is selected, which is called π^*-sub-history candidate. After all of them are found, they are inserted into the tree. Beginning from the root node, at each step i, if active node has an edge with label $\langle a_i, \psi_i \rangle$,

- ψ_i is incremented to indicate that up to this step, action sequence is used frequently.
- If the child node n where the edge is pointing to contains a tuple $\langle s_i, \xi_i, R_{s_i} \rangle$, ξ_i is incremented and R_{s_i} is updated with α rate of R_i, which is the discounted cumulative reward of s_i in the history . Otherwise a new tuple $\langle s_i, 1, R_i \rangle$ is added to n. Afterwards, n becomes the new active node.

If active node has no edge for action a_i, a new one labeled as $\langle a_i, 1\rangle$ and a new node n pointed by this edge is created. Tuple $\langle s_i, 1, R_i\rangle$ is inserted to the child node n and it becomes the active node.

Following history addition operation, eligibility values in the whole structure is decreased and the elements having values lower than the given threshold is pruned in order to decrease memory consumption. Since the aim of EST is to maintain only the common sub-problems in the task, rarely used ones can be ignored. If an edge in the EST has lower eligibility value than $\psi_{threshold}$, it is deleted with its child EST node recursively. In an EST node n, if a tuple for a state has lower eligibility than $\xi_{threshold}$, it is deleted from the set of tuples, and if the corresponding set becomes empty, n is deleted with its children and parent edge from the tree. Sequence of steps followed while updating an EST is provided in Algorithm 5.

Algorithm 5 Algorithm to update extended sequence tree T

```
1: procedure UPDATE-TREE(T, e)
Require: T is the EST
Require: e is the history of events
2:     H = GENERATE-PROBABLE-HISTORIES(e)
3:     for all h ∈ H do
4:         ADD-HISTORY(h, T)                    ▷ Add the history to EST
5:     end for
6:     UPDATE-NODE(n)                           ▷ Update root node of T
7: end procedure
```

2.4 Factored Reinforcement Learning

The methods proposed for solving MDPs in previous sections are not adaptable to large scale domains. Because, they enumerate all states and actions while modeling and solving the tasks. In a problem with a huge state and action space, this approach is inadequate. Factored Markov Decision Processes (FMDPs) [4] are used to handle this issue by defining the problems in a structured and more compact way. Factored Reinforcement Learning (FRL) algorithms use this framework and take the advantage of new representation. In this section, FMDPs and some algorithms that are constructed on top of this model will be

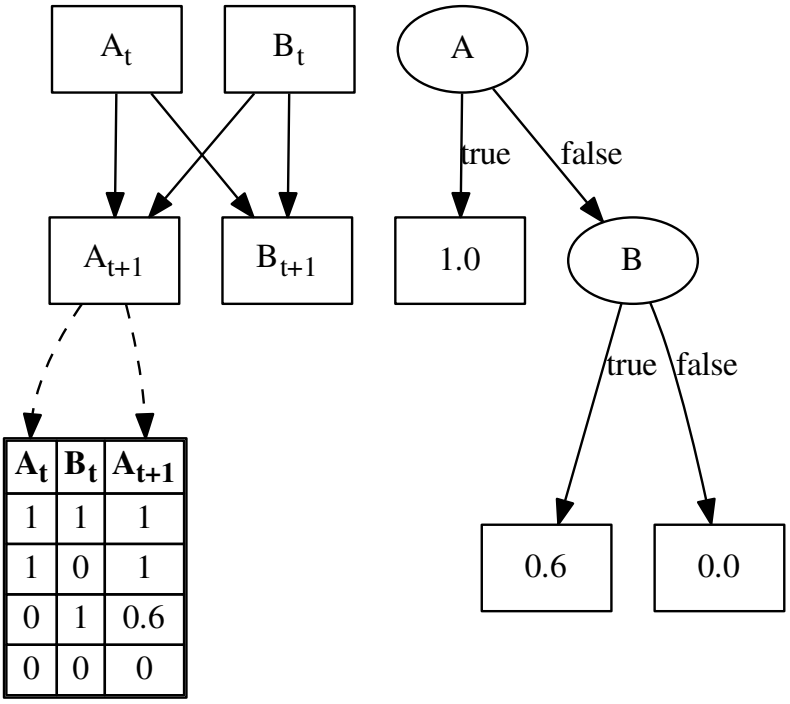
explained.

2.4.1 FMDPs

In most of the learning problems, in fact, states are combination of features of the domain. So, it is possible to define states as a vector of these features. However, state space grows exponentially by the number of features. FMDPs are used to build more compact transition, policy and reward functions by exploiting the new state representation. In factored model, a state is characterized using variables $x_1, \cdots, x_n$ as $s = \langle x_1, x_2, \cdots, x_n \rangle$, where each variable x_i can take finite possible values from its domain $Dom(x_i)$. By this way, it is possible to use similarities between states and group the ones having the same feature values while defining and solving the problem.

Dynamic Bayesian Networks (DBNs) are used to model transition probabilities and conditional dependencies between variables. A DBN for action a is a directed acyclic graph showing the effects variables on other variables when a is performed. Since all actions cause different changes, DBNs are action-specific. In this work, we assume that, there are no *synchronic arcs* in DBNs. It means, value of a variable at time t cannot affect another variable at time t. Only pre-action and post-action nodes are connected. From a given DBN, for each variable x, we can find the list of variables determining the next value of x. Then, for each possible value combinations of the variable in the list, probability distribution of x can be defined in a tabular form, called *conditional probability table* (CPT). However, using such a structure can be disadvantageous since its size grows exponentially in number of variables. A better and more compact way is to use a tree structure that exploits common parts of different combinations having the same probability distributions.

Figure 2.3 contains a transition probability function example for a specific action in a domain with two binary variables, A and B. According to the DBN, current values of both A and B are used to decide their next values. The CPT lists the probabilities of A being *true* at time $t + 1$ based on the current situation. For example, when A is 0 and B is 1, with 0.6 probability, A will be 1 after the



A_t	B_t	A_{t+1}
1	1	1
1	0	1
0	1	0.6
0	0	0

Figure 2.3: Transition function example in factored form

action is applied. It also says that A_{t+1} will be *true* with probability 1.0 in states $(A_t = 1, B_t = 1)$ and $(A_t = 1, B_t = 0)$. So, we can determine the value of A_{t+1} without knowing anything about B. In a decision tree, these states are merged such that probability distribution depends only on variable A as shown in the figure. More formally, while constructing transition probability tree for a variable x, if $\forall y_k \in Dom(y)$ these probabilities do not change, y can be omitted from the tree. The effectiveness of this structure is more visible in larger domains. Figure 2.4 shows examples of reward and policy functions represented in the same tree structure. Policy tree requires 3 nodes instead of 4 entries in the tabular form. Similarly, immediate reward depends only on the value of A, which is expressed in the tree form more compactly.

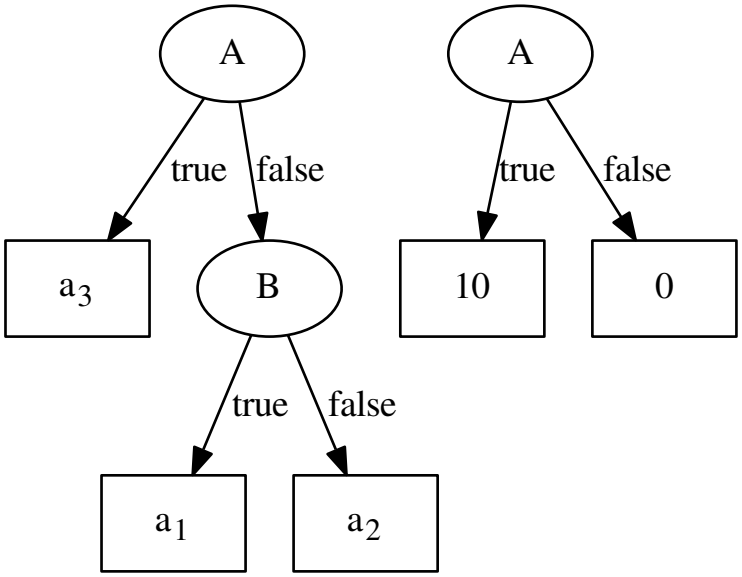


Figure 2.4: Policy and reward function examples in factored form

2.4.2 Solving FMDPs

Similar to classical MDP counterparts, when transition and reward functions are provided, Structured Dynamic Programming (SDP) approaches, such as Structured Value Iteration (SVI) and Structured Policy Iteration (SPI) [5], can be used to produce an optimal policy in a tree form. These methods use tree manipulation techniques to append, merge and simplify decision trees during computations. SDYNA [8] is a FRL method that can be applied when perfect problem structure is not known. The algorithm consists of three phases: *acting, learning* and *planing*. It iteratively constructs reward and transition functions by modifying them after each feedback. Based on the new structure, policy and value functions are updated and the next action is chosen accordingly. SPITI is an instance of SDYNA that uses ϵ-greedy algorithm for action selection, Incremental Tree Induction (ITI) [21] in learning phase and SVI as planning method.

In addition to SDP and FRL methods, there exist some techniques to define

temporal abstractions in structured worlds. These methods use pre-solved sub-policies in order to decrease time and space complexities in large domains. HEXQ [13] and TeXDYNA [15] are examples of Hierarchical Reinforcement Learning (HRL) algorithms [11] applied on such problems.

2.4.2.1 HEXQ

Hierarchical MDPs are constructed by dividing a given (factored) MDP into smaller sub-tasks that can be solved more easily. This hierarchical structure also makes it possible to reuse MDP solutions of lower levels in the higher levels so that the agent does need to handle common parts of the problem separately in different regions. HEXQ is one of the algorithms that decomposes a given FMDP into sub-problems in a hierarchical way and solves each level independently. Unlike MAXQ [9] algorithm, in which structure of the hierarchy is defined by the user, HEXQ is an automatic decomposition technique. The algorithm is based on the assumption that, in state transitions, value of each variable changes with a different frequency. Without any prior knowledge about transition probabilities, the algorithm performs random actions in the environment and observes variable value changes in subsequent states. Variable with the highest frequency is located in the lowest level of the hierarchy. Starting with this variable, value changes of the variables are observed between subsequent states and a directed graph (DG) is constructed using transitions between values. The nodes are connected to each other with the action that causes value change. Then, set of state-action pairs, called *exits*, are found to determine stopping criteria of levels in the hierarchy. In an exit $p = \langle s_i, a_i \rangle$, taking exit action a_i in exit state s_i means either the value of the next variable in the ordering changes or the task terminates. For example, in a maze with multiple rooms, location of the agent in a room changes more frequently than the room number. The exits in DG of *location* variable is the set of cells connecting neighbor rooms and corresponding actions.

After finding the relations between variable value changes and actions, MDPs for the corresponding level is constructed. In order to determine MDP regions,

DG of the variable is divided into blocks such that in each block, each exit state can be reached using non-exit actions. As a result, we have a set of sub-MDPs that contain exits, transitions and an entry point of the sub-task. The exits are used as the termination conditions of the corresponding MDP. The same decomposition procedure is applied to the value transition graph of variables in each level. But primitive actions are only used in the lowest one, since variable value transitions in the remaining levels are defined in terms of exits in the lower level. For this reason, exits of a level becomes abstract actions of the next level in the hierarchy. Applying an abstract action means solving related MDP in lower level. For example, in the maze problem, reaching a cell in a different room first requires finding an exit from the current room towards the destination point. The agent chooses the abstract action that changes value of room number which means finding the path to the exit state in the current room.

Algorithm 6 Algorithm to execute HEXQ after building hierarchical MDP structure

1: **procedure** EXECUTE-HEXQ(e, s, a)
Require: e is the current level in the hierarchy
Require: s is the current state
Require: a is the selected action
2: **if** $e = 0$ **then** ▷ a is an primitive action
3: Apply primitive action a and observe reward r and new state
4: **return**
5: **end if**
6: $m \leftarrow$ sub-MDP associated with action a
7: **repeat** ▷ a is an abstract action
8: Choose an action a' by using m and an action selection mechanism
9: $ls^e \leftarrow$ abstraction of state s in level e
10: EXECUTE-HEXQ($e - 1, s, a'$)
11: Update $E_m(ls^e, a')$ by using immediate reward r and value coming from m
12: **until** $p = \langle s, a \rangle$ is an exit
13: **end procedure**

Algorithm 6 shows the execution of HEXQ after constructing hierarchical MDPs. Starting from the highest level and initial state, it applies given action in a way that depends on the location in the hierarchy. If the lowest level is reached, which means one of the primitive actions is chosen, the agent interacts with the environment and performs the action. Otherwise, action a is abstract and it requires solving corresponding MDP in the lower level until finding an exit point.

Afterwards, action-value function E is updated by using the feedbacks. Action selection can be done in different ways. Constructing a reward function while taking random actions at the beginning of the algorithm can make it possible to apply DP methods to solve sub-MDPs. Alternatively, an online approach, like Q-learning, can also be used.

2.4.2.2 TeXDYNA

TeXDYNA is a sub-task discovery method that manages incremental hierarchical decomposition of the problem while learning the structure of it. It combines HRL and FMDP methods to solve learning problems with unknown structure. The algorithm extracts options from the global transition function and uses an existing FRL method to learn by using the options. Since it is an online learning algorithm, options are dynamically updated and improved. Algorithm 7 shows execution of TeXDYNA.

Algorithm 7 TeXDYNA algorithm

```
1: procedure EXECUTE-TEXDYNA(F, O)
Require: F FMDP structure of the problem
Require: O Option hierarchy
2:     F ← UPDATE-FMDP(F)                          ▷ Update global transition model
3:     O ← UPDATE-OPTIONS(O, F)                    ▷ Update options using new FMDP model
4:     UPDATE-POLICY(O, F)                         ▷ Update hierarchical policy with option set O
5:     MODIFIED-SPITI(O, F)                        ▷ Choose one of the options by using SPITI algorithm
6: end procedure
```

The algorithm consists of learning and planning phases. The first two lines of Algorithm 7 correspond to the learning phase. Transition function which is represented by decision trees is updated and used to build options in a hierarchical structure. Each option has its own model and local policy. An option o is a tuple $\langle I, \pi, e \rangle$, where I is a subset of states in which option o can be initiated, π is the local policy of o and e is the *exit* that is defined similar to the one in HEXQ algorithm. Each exit consists of four items:

- v : variable whose value changes in the exit
- a : action that is executed in the exit

- v_{ch} : a pair of values $\langle v_t : v_{t+1} \rangle$ where v_t is the value of v before applying a and v_{t+1} is the value after applying a
- c : a set of criteria that makes the exit available, containing variable label - value pairs in the form of $v_x : value$.

Each primitive action is also defined as an option with an empty initiation set and exit so that SPITI can choose one of them in planning phase. In TeXDYNA, SPITI is modified in a way that it chooses an option instead of a primitive action in its planning phase.

CHAPTER 3

FACTORED EXTENDED SEQUENCE TREE METHOD FOR FACTORED REINFORCEMENT LEARNING

Extended Sequence Trees are proved to be an effective way of increasing learning rate in MDPs. The same structure can be used in FMDPs by using set of variables in state representations. By this way, it is possible to obtain solutions in fewer number of steps compared to classical RL algorithms. But, the problem is about the size of the domain. Without making any modifications on the current structure, the algorithm will suffer from excessive memory usage. This contradicts with the idea of factored model, which generalizes the structure of problem definitions and solutions by using the similarities between the elements in terms of variable values. In this way, we can build more compact models that require less resources during computations. Similar to the new approaches in FMDP and FRL, EST can be modified in a way that it takes the advantage of factored model and uses a new representation for useful state-action sub-sequences. The aim is to find optimal solutions more quickly for the tasks with huge state and action spaces while minimizing memory requirements of EST structure.

The idea of factored-EST is based on the assumption that each action affects only a small subset of variables. Thus, after an action is performed, it is sufficient to focus on the variables whose values changed with the corresponding action. In other words, there is no need to maintain variables that have the same value in successive steps. Depending on the number of affected variables, new representation will consume less memory compared to its classical counterpart.

The first step while building the new structure is to modify a given history of events in accordance with this observation. Assuming that h is a history of an agent where

$$h : s_1\, a_1\, r_2\, s_2\, a_2\, r_3 \cdots s_{i-1}\, a_{i-1}\, r_i\, s_i$$

for a classical MDP task, it can be represented in a factored form after transforming state labels into vector of variables as

$$h' :$$
$$(x_{1_1}, \cdots, x_{k_1})\, a_1\, r_2\, (x_{1_2}, \cdots, x_{k_2})\, a_2\, r_3 \cdots (x_{1_{i-1}}, \cdots, x_{k_{i-1}})\, a_{i-1}\, r_i\, (x_{1_i}, \cdots, x_{k_i}),$$

where s_i is the state at time i, x_{i_j} is the value of i^{th} variable at time j, a_i is the action performed at time i and r_{i+1} is the reward received after applying action a_i. Assuming that $\mathit{diff}\,(t)$ function gives the difference of state variables between s_{t-1} and s_t, we can redefine the history as

$$h'' : (x_{1_1}, \cdots, x_{k_1})\, a_1\, r_2\, (\mathit{diff}\,(2))\, a_2\, r_3 \cdots (\mathit{diff}\,(i-1))\, a_{i-1}\, r_i\, (\mathit{diff}\,(i)).$$

Note that $\mathit{diff}\,(i)$ is the empty set when action a_{i-1} has no effect on the current state s_{i-1}.

Algorithm 8 Algorithm to construct a factored history from a given history

1: **procedure** HISTORY-DIFF(H)
Require: H is the history of events in the form of
$(x_{1_1}, \cdots, x_{k_1})\, a_1\, r_2 \cdots (x_{1_{i-1}}, \cdots, x_{k_{i-1}})\, a_{i-1}\, r_i\, (x_{1_i}, \cdots, x_{k_i})$
Ensure: F is the modified history
2: $F = (x_1 : x_{1_1}, \cdots, x_k : x_{k_1})$ ▷ Append the initial state to the new history form
3: **for** $a = 2..i$ **do** ▷ For all states except for the first one
4: $D_a = \emptyset$ ▷ Initialize difference set
5: **for** $b = 1..k$ **do** ▷ For all variables in the domain
6: **if** $x_{b_a} \neq x_{b_{a-1}}$ **then**
7: $D = D \cup \{(x_b : x_{b_a})\}$ ▷ Append the difference to D
8: **end if**
9: **end for**
10: $F = F \cup \{(a_{i-1}, r_i, D_a)\}$ ▷ Append the event tuple to new history form
11: **end for**
12: **return** F
13: **end procedure**

Algorithm 8 gives the whole procedure of constructing a factored history of events. The algorithm first appends the initial state of the history in the form of variable-value tuples to a new list. Then, in each step, a list of modified variables are constructed and added to the new history with action and reward values. An example of factored representation of a state history is shown for an agent in taxi domain in Figure 3.1. In this problem, the agent is in a 5×5 grid world with some obstacles preventing the taxi from moving. It tries to pickup the passenger from a predefined location and deliver him to another one. Successful delivery gives +20 reward while wrong *pick up* and *put down* actions result in -10 reward.

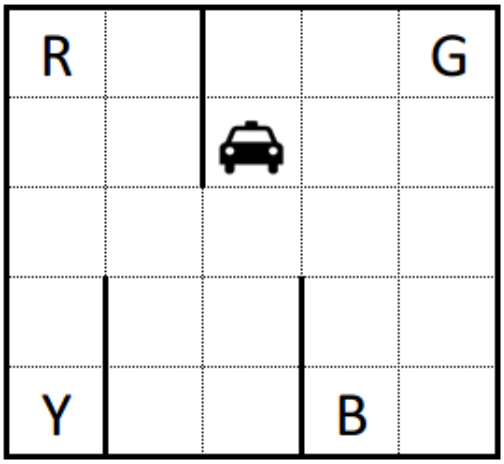


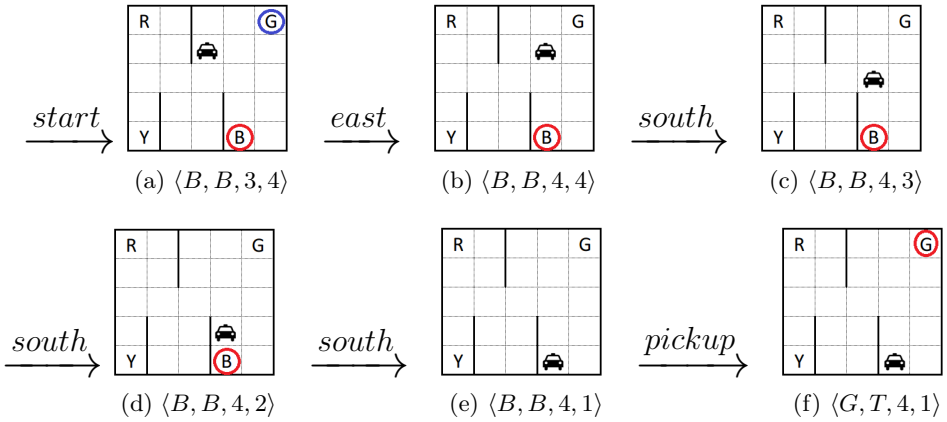
Figure 3.1: Taxi domain

Variables used to define a state in taxi domain are listed below.

- $Dest$: Destination cell of the agent. $Dom(Dest) = \{R, G, B, Y\}$.
- $Pass$: Current passenger location. $Dom(Pass) = \{R, G, B, Y, T\}$.
- x: Current horizontal location of the agent. $Dom(x) = \{1, 2, 3, 4, 5\}$.
- y: Current vertical location of the agent. $Dom(y) = \{1, 2, 3, 4, 5\}$.

The value "T" in $Dom(Pass)$ indicates that the passenger is in the taxi. In the examples through this thesis, we will express the states with these variables in this order, i.e., $\langle Dest, Pass, x, y \rangle$. Assume that the bottom leftmost cell is point $(1, 1)$, the agent starts at $(3, 4)$, the passenger is initially located at cell

B and the final destination is G. Figure 3.2 shows performing a list of actions successively and their effects on the state variables.



(a) $\langle B, B, 3, 4\rangle$ (b) $\langle B, B, 4, 4\rangle$ (c) $\langle B, B, 4, 3\rangle$

(d) $\langle B, B, 4, 2\rangle$ (e) $\langle B, B, 4, 1\rangle$ (f) $\langle G, T, 4, 1\rangle$

Figure 3.2: An example of taxi domain state history

In Figure 3.2, the agent first goes to *east* which changes only variable x. After that, it applies *south* action three times and reaches to the passenger. These moves affect only y coordinate of the taxi, while other variables stay unchanged. Picking up the passenger in the next move updates passenger position and the current destination of the agent. Note that the environment is deterministic in this example. In classical notation, this sub-history can be shown as

$$\langle B, B, 3, 4\rangle\, east \langle B, B, 4, 4\rangle\, south\, \langle B, B, 4, 3\rangle south \langle B, B, 4, 2\rangle south \langle B, B, 4, 1\rangle pickup \langle G, T, 3, 4\rangle,$$

which is transformed into

$$\langle B, B, 3, 4\rangle\, east \langle x : 4\rangle\, south\, \langle y : 3\rangle south \langle y : 2\rangle south \langle y : 1\rangle pickup \langle Dest : G, Pass : T\rangle$$

in the factored notation by Algorithm 8.

From the example, it can be concluded that, in a sub-history, if the initial state is known, all successive states can also be determined from the state difference sets. It means that, the same amount of information can be expressed with less data. This new factored representation forms a basis to Factored Extended Sequence Tree (Factored-EST) algorithm. The nodes of the classical EST algorithm are modified in a way that they can be used to store initial states and state differences of a factored history model. The internal structure of tree nodes depends

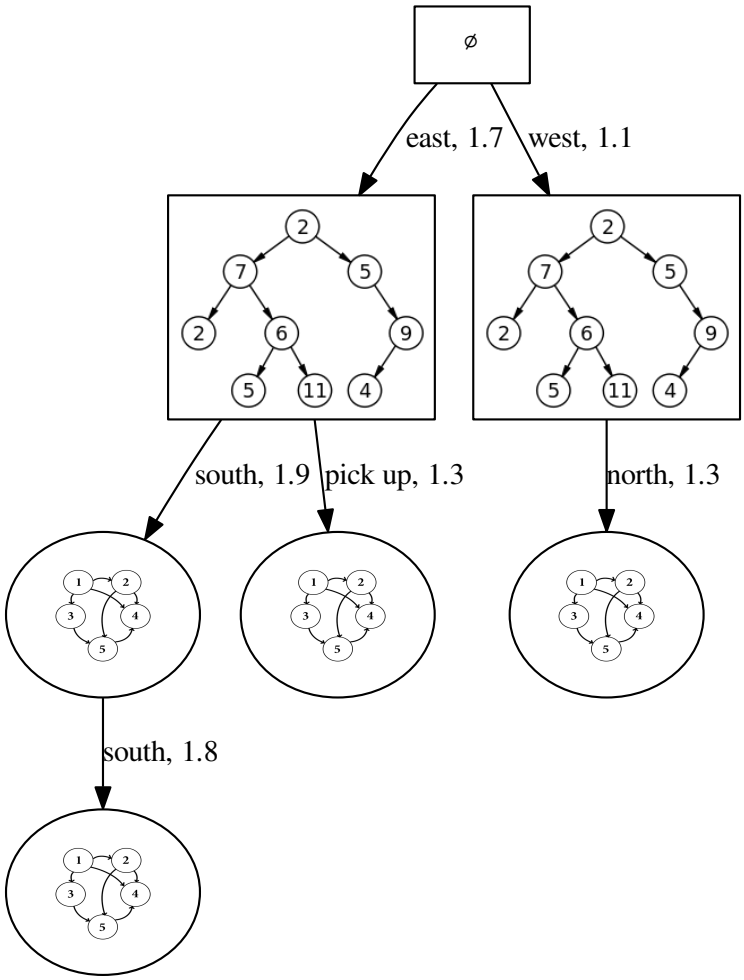


Figure 3.3: An example factored-EST structure

on the type of the state that is being stored. We use decision trees to store initial states and a graph structure for differences between successive states. Therefore, any child of root node of an factored-EST contains a decision tree while all other nodes have state difference graphs. Since action representation does not change in the factored model, it is reasonable to use the original EST edge notation in the new model. The structure of a factored-EST is shown in Figure 3.3.

3.1 Decision Trees

In FMDPs, decision trees are used to define reward and value functions more compactly, by taking advantage of the similarities between states. State variables are kept in the nodes of the tree, while edges are the decision points showing the paths to be followed depending on the value of the corresponding variable. Starting from the root node, each path to one of the leaf nodes defines a set of criteria and all the states satisfying every criteria in this set is said to be an element of the path's state set. Leaf nodes store the actual data which can be used for each state in the state set of the path. The type of the data being expressed depends on the context. For reward function, it is immediate rewards of the states while in value function, it shows the expected state values. Policy function can also be represented in a decision tree where leaf states give the actions to be performed.

In a similar way, initial states of a set of factored histories can be maintained in a decision tree in order to reuse the same variable instances in different states. Therefore, it is more compact than listing all states as in classical EST structure. Leaf nodes contain eligibility values and discounted cumulative rewards of the states. Figure 3.4(a) shows a simple decision tree of initial states of a set of histories for taxi problem.

From time to time, factored-EST is fed with a given history in order to keep it up-to-date. The very first step of adding a new history to the factored-EST structure is inserting the initial state s_1 to one of the child nodes of the root. Assuming a_1 is the first action, the algorithm selects the node D which is connected to the root by an edge labeled with $\langle a_1, \cdot \rangle$. It is guaranteed that at most one decision tree can meet this criteria. If there does not exist such a tree, a new one is created which is initialized with state s_1. But inserting s_1 to an existing decision tree requires tree traversal operation. Initially the root is selected as the current node. At each step, the value of the current variable is searched in the edges of the corresponding tree node. If such an edge e is found, current node becomes the node that is pointed by e and process continues with the next variable. Otherwise, a new node is created, which is selected as the

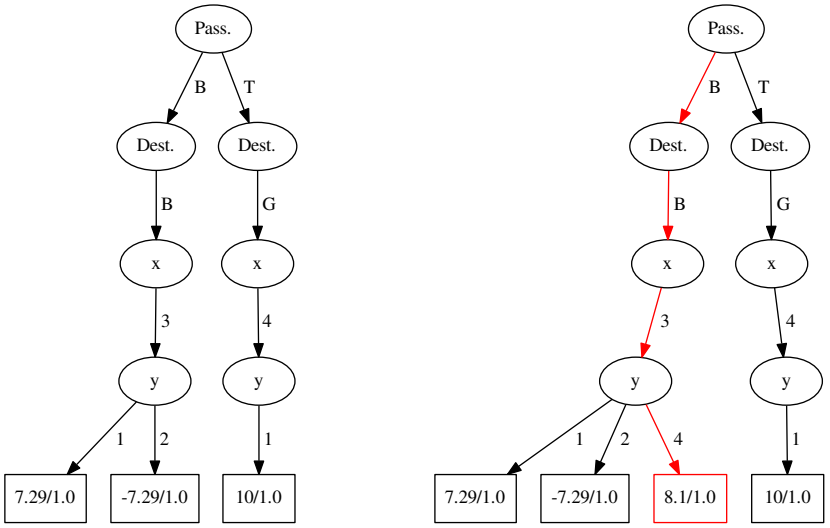


Figure 3.4: Taxi domain decision trees (a) initial structure (b) after adding state $\langle B, B, 3, 4\rangle$

current node for next step. If the state has already been inserted to the tree, its eligibility value is incremented to indicate that it can be a starting point of a frequently used pattern. Additionally, existing reward value R is updated in the same way provided in Algorithm 12.

Figure 3.4(b) visualizes the resulting decision tree after adding initial state s_1 : $\langle B, B, 3, 4\rangle$ to the one on the left. The nodes show the variable labels and edges indicates the variable values. Since current destination, passenger location and x coordinate of the location of the taxi are already included in the existing tree, only a leaf node for y variable is created with an edge showing that value of y is 4. Discounted cumulative reward of s_1 and initial eligibility value (1.0) are attached to the new node. These leaf nodes are used to point to the corresponding histories in state difference graphs. It is important to note that variable order is the same in decision tree nodes and factored history representation.

While updating the entire factored-EST structure, all eligibility values are multiplied by a given factor and states having less eligibility values than a given threshold are deleted from the decision tree. State deletion is handled by the

classical tree node deletion algorithm. It starts from the leaf node of a given state and deletes all the nodes until a parent node with more than one child is encountered. Then it terminates. If it is the last state located in the decision tree, the tree is also deleted with its child nodes and parent edge.

3.2 State Difference Graphs

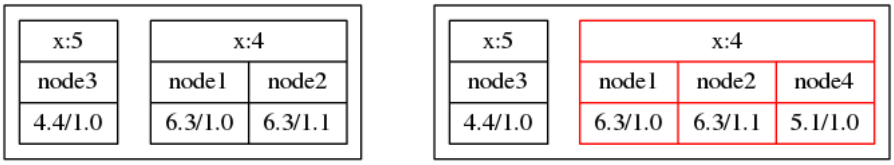
In order to store variable value differences between successive steps, factored-EST uses state difference graphs. It is a directed acyclic graph where some of the nodes have termination conditions for related difference sets to indicate that a given difference set is contained in the graph or not. A graph node g is a tuple $\langle x_i, v_i, D_i \rangle$. x_i is the variable label, v_i is its value and D_i is a list of tuples (l_m, R_m, ξ_m) containing a link l_m to the first state of corresponding history, reward value R_m and eligibility value ξ_m for the state. If a graph node g_a has an outgoing edge to node g_b, it is guaranteed that the variable label of g_a, which is x_{g_a}, comes before the variable label of g_b, which is x_{g_b}, in the variable order of the problem definition. This property ensures that there are no cycles in the graph.

Assume that a history h is given with an initial state s_h and a link l_h pointing to the leaf node of s_h in the decision tree. If $diff(i)$ is $\{x_i : v_i, x_k : v_k\}$, existence of $diff(i)$ in the $(i+1)^{th}$ level node of a factored-EST is checked as follows:

- There should be nodes g_i and g_k in the difference graph with labels $\langle x_i, v_i, D_i \rangle$ and $\langle x_k, v_k, D_k \rangle$
- g_i must have an outgoing link to g_k
- D_k must contain a tuple with values $(l_h, \cdot, \cdot)$.

Differently from the original EST node structure where each state has one reward and eligibility value, each variable difference set has multiple reward and eligibility values in factored-EST. This is the reason of having a list of tuples in D_i instead of storing a single tuple. Because, given two histories, if the

initial states are different, $diff(t)$ values may be the same for both of them although s_{i-1} and s_i values are different. For example the difference set of state $\langle B, B, 3, 2\rangle$ with respect to $\langle B, B, 4, 3\rangle$ has the same elements of the difference set of state $\langle G, T, 3, 2\rangle$ with respect to $\langle G, T, 5, 5\rangle$, which is $\{x : 3, y : 2\}$. Most probably, these sets will have different reward and eligibility values, although both of them have identical graph paths. For this reason, the entries should be separated from each other in the graph. In addition, there is a possibility that the action performed does not have any effect on the current state. In this case, state difference set is empty, which is handled by creating a node with empty label.



(a) (b)

Figure 3.5: State difference graphs (a) initial form, (b) after adding set {x:4} for link *node*4

Figure 3.5(a) shows an example of state difference graph. By looking at the node counts, it can be concluded that at least three different histories are introduced to the system. Also, graph labels show that the corresponding action has affected only the variable x. In one of the histories, new value of x is 5, while in other two it becomes 4. Similar to decision trees, adding a new difference set to a given graph requires graph traversal operation. Entire process is given in Algorithm 9.

Given a graph node G, a difference set S in the form of $\{var_1 : val_1, var_2 : val_2, \cdots, var_k : val_k\}$, a link L to the leaf node of decision tree for the current history and reward of the state R, at each step i the algorithm operates as follows:

- If the graph has a node g labeled with $(var_i : val_i)$ set it as the current node

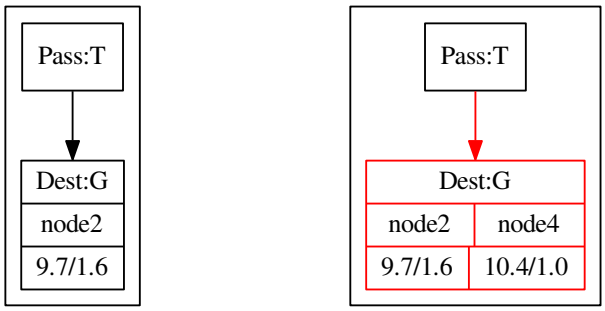
Algorithm 9 State difference set addition to a given graph

1: **procedure** ADD-STATE-DIFF(G, S, L, R)
Ensure: G is the state difference graph where S will be added
Ensure: S is a set of variables whose values are changed after the action is executed in the form of $\{var_1 : val_1, var_2 : val_2, \cdots, var_k : val_k\}$
Require: L is a link to the decision tree leaf node storing the initial state of processed history
Require: R is discounted cumulative reward for the state
2: $current \leftarrow nil$ ▷ current graph node
3: **for** $i \leftarrow 1..k$ **do**
4: **if** $\exists$ a node n with label $var_i : val_i$ **then**
5: $current \leftarrow n$
6: **else**
7: Add new node $n(var_i : val_i)$ to G.
8: $current \leftarrow n$
9: **end if**
10: **if** $i < k - 1$ and $\nexists$ a neighbor of $current$ with label $var_{i+1} : val_{i+1}$ **then**
11: Add a neighbor with label $var_{i+1} : val_{i+1}$ ▷ Add the label of the next difference to the neighbor set
12: **end if**
13: **end for**
14: **if** $current$ is nil **then** ▷ state difference set is $\emptyset$
15: Add new node n with label $\emptyset$ to G
16: $current \leftarrow n$
17: **end if**
18: Add tuple $\langle L, R, 1.0 \rangle$ to D of $current$
19: **end procedure**

- Otherwise, create a new one
- If this is not the last element of the difference set, add the label of the next difference (var_{i+1}, val_{i+1}) to the neighbor list of g
- Otherwise, add tuple $(L, R, 1.0)$ to D_i

In Figure 3.5, the variable differences between initial state and the next state given in Figure 3.2(a) and (b) are added to an existing state difference graph. The initial state was inserted into a decision tree in Figure 3.4(b), whose leaf node is marked as $node4$. Assuming that the graph a in figure 3.5(a) is connected to the corresponding tree with an edge labeled with $\langle west, \cdot \rangle$, which is the first action of the sub-history, the variable change $\{x : 4\}$ should be added to a. Note that connections between graph nodes indicate that an action affected all variables in the path. Thus, it is reasonable to have no connections between the nodes of this graph. Because there is no way of assigning multiple values to x

at the same time. It is also possible to have the same state in subsequent steps, which means that the applied action has no effect on the variables. In this case, a graph node labeled with $\emptyset$ is used to store the values.



(a) (b)

Figure 3.6: Graphs with connections (a) initial form, (b) after adding set {Pass:T, Dest:G} for link *node*4

Figure 3.6 shows the necessity of keeping attribute values separately for histories with different initial states. In the example, the last difference set in Figure 3.2 is inserted into the graph. Note that, the graph already includes a node for both difference tuples, but the attributes in the node labeled with $Dest : G$ belong to another history. In order not to overwrite existing values, a new set is created for *node*4.

Each graph node of factored-EST is also updated like all the other elements in the structure. Algorithm 10 gives the entire procedure. A list of links pointing to the leaves of initial states that are marked to be deleted is provided to the algorithm. Initially, the attributes related with these initial states are deleted from all graph nodes. In addition, eligibility values of other attributes are decremented by a given factor $\xi_{decay} \in (0, 1)$ and the ones having lower eligibility values than the threshold are also eliminated. Then, the graph is traversed to find the isolated nodes. A graph node is determined to be deleted if it satisfies three criteria listed below:

Algorithm 10 Algorithm for updating a factored-EST graph node

```
1: procedure UPDATE-GRAPH(LS)
Require: LS is set of decision tree nodes which will be deleted from the decision tree.
2:     E ← ∅                                              ▷ Set of graph nodes to be deleted
3:     Let N be the graph to be updated
4:     for all n = ⟨x, v, D⟩ ∈ G do
5:         for all d = (l, R, ξ) ∈ D do
6:             if d ∈ LS then
7:                 remove d from D
8:             end if
9:             ξ ← ξ × ξ_decay                            ▷ Update eligibility value
10:            if ξ < ξ_threshold then
11:                remove d from D
12:            end if
13:        end for
14:        Let H be the graph nodes pointed by n
15:        if D = ∅ and H = ∅ then                        ▷ If there is nothing pointed out by n
16:            E = E ∪ {n}
17:        end if
18:    end for
19:    while E is updated do
20:        for all n = ⟨x, v, D⟩ ∈ G do
21:            if n ∈ E or D ≠ ∅ then   ▷ Node is already marked to be deleted or there are
   leaf values in it
22:                continue
23:            end if
24:            Let H be the graph nodes pointed by n
25:            if H − E = ∅ then                          ▷ All of the neighbor nodes will be deleted
26:                E = E ∪ {n}
27:            end if
28:        end for
29:    end while
30:    for all n ∈ E do
31:        remove n from N
32:    end for
33: end procedure
```

- All eligibility values are below the predefined threshold value.
- All decision trees that are pointed by the node are marked to be deleted.
- There are no outgoing links to other nodes.

All isolated nodes are collected in a set E and the remaining nodes are checked if there is at least one neighbor node which is not an element of E or eligibility and reward values of some valid initial states are stored in the current node. If

this is not the case, the current node is also added to E. This process continues until E cannot be extended. Finally, all nodes in E are deleted. If the graph becomes empty after this update operation, corresponding factored-EST node is also removed with its parent edge and child nodes.

3.3 Factored Extended Sequence Tree

By combining the structures and algorithms explained in previous sections, Factored Extended Sequence Tree (Factored-EST) algorithm can be constructed. A factored-EST is a tuple $\langle D, N, E \rangle$ where D is the set of decision trees, N is the set of state difference graphs and E is the set of edges. Decision tree set D includes the root node and its children while all the other nodes are managed in the graph set N. Apart from the root node, the elements of D are used to keep initial states of the histories. In addition, they provide links to the nodes in N to be able to associate reward and eligibility values with the histories, as pointed out in section 3.2. In order to analyze a history, the initial values of each variable has to be known, so that, it will be possible to determine exact states in any step by using the differences between states.

There are three types of edge connections in factored-EST:

- Root node to decision tree connection,
- Decision tree to state difference graph connection, and
- State difference graph to state difference graph connection.

Although each edge connects different types of nodes, they all have the same information: *candidate action* and *its eligibility value*. Figure 3.3 contains examples of each edge type.

Algorithm 11 Algorithm to generate probable π^*-histories from a given history h

1: **procedure** GENERATE-PROBABLE-HISTORIES(h)
Require: h is a history of events in the form of $s_1 a_1 r_2 s_2 a_2 r_3 \cdots r_t s_t$
2: $best[s_{t-1}] = s_{t-1} a_{t-1} r_t s_t$ ▷ $best[s]$ stores the current π^*-history candidate of s
3: $R[s_{t-1}] = r_t$ ▷ R[s] stores the cumulative discounted reward for state s
4: **for** $i \leftarrow (t-2)..1$ **do**
5: **if** $R[s_i]$ is not set or $r_{i+1} + \gamma R[s_{i+1}] > R[s_i]$ **then** ▷ If s is not encountered before or can be improved
6: $best[s_i] = s_i a_i r_{i+1} + best[s_{i+1}]$ ▷ Create or update probable π^*-history for s_i
7: $R[s_i] = r_{i+1} + \gamma R[s_{i+1}]$
8: **end if**
9: **end for**
10: **return** best
11: **end procedure**

Periodically, history of events experienced by the agent are provided to factored-EST to extract useful sub-histories. The method that is used to process this list of events is the same as the one in the original EST algorithm. Algorithm 11 generates sub-histories that are probably sub-policies in an optimal solution. For each state s in the history, a sequence of events connecting s to the terminal state s_t is found. During this process, some of the states may be encountered more than once. In this case, the algorithm updates the path if expected reward in the new path is higher (line 5 in the algorithm).

After extracting a set of histories from the given history, each of them is inserted into the factored-EST structure by using Algorithm 12. The algorithm starts by transforming candidate histories into factored history of events representation. Then, for each step of the histories, discounted cumulative rewards are calculated. Depending on the type of the current factored-EST node, the algorithm calls corresponding state addition or update methods for the state currently being processed. If it is the initial state, decision tree state addition process is applied while in other states, difference set is added to the current graph.

Algorithm 12 Algorithm to add given probable π^*-history to factored-EST T

procedure FACTORED-EST-ADD-HISTORY(T, H)
Require: T is extended sequence tree
Require: H is a history of events in the form of
$(x_{1_1}, \cdots, x_{k_1})\, a_1\, r_2 \cdots (x_{1_{i-1}}, \cdots, x_{k_{i-1}})\, a_{i-1}\, r_i\, (x_{1_i}, \cdots, x_{k_i})$
Let h = HISTORY-DIFF(H) ▷ History is converted into factored form
$(x_{1_1}, \cdots, x_{k_1})\, a_1\, r_2\, (\mathit{diff}(2))\, a_2\, r_3 \cdots (\mathit{diff}(t-1))\, a_{t-1}\, r_t\, (\mathit{diff}(t))$
$R[t] = r_t$
for $i = (t-1)$ *to* 1 **do**
$R[i] = \gamma R[i+1] + r_i$
end for
current = root node of T ▷ Active node is initially the root node
if ∃ a decision tree d in the children of *current* pointed by an edge $\langle a_1, \psi_1 \rangle$ **then**
Increment ψ_1
if d contains state $(x_{1_1}, \cdots, x_{k_1})$ with leaf l **then**
Increment ξ_{s_1}
$R_{s_1} = R_{s_1} + \alpha(R[1] - R_{s_1})$
else
Add s_1 to d with leaf node $l = (R_1, 1.0)$
end if
else
Create an empty decision tree d
Add s_1 to d with leaf node $l = (R_1, 1.0)$
Connect d to *current* by an edge labeled with $\langle a_1, 1.0 \rangle$
end if
$L = l$ ▷ The link to the corresponding leaf node of decision tree d
current $= d$
for $i = 2..t-1$ **do**
if ∃ a graph n in the children of *current* pointed by an edge $\langle a_i, \psi_i \rangle$ **then**
Increment ψ_i
if n contains $diff(i)$ and tuple $l, \xi_{s_i}, R_{s_i} \rangle$ in the leaf node **then**
Increment ξ_{s_i}
$R_{s_i} = R_{s_i} + \alpha(R[i] - R_{s_i})$
else
ADD-STATE-DIFF($n, diff(i), l, R_{s_i}$)
end if
else
Create a new state difference graph n
ADD-STATE-DIFF($n, diff(i), l, R_{s_i}$)
Connect *current* to n by an edge labeled with $\langle a_i, 1.0 \rangle$
end if
current $= n$ ▷ Child node becomes new active node.
end for
end procedure

The complete algorithm for running factored-EST with an underlying RL method

in an episodic task is provided in Algorithm 13. Since no experience is available at the beginning, factored-EST is passive. In each episode, starting with the initial state, the algorithm selects an action, performs it, observes the outcomes and updates state-action values based on these results. Each state-action-reward tuple is stored in a history h, which is used to generate probable π^*-histories. After each episode, these histories are forwarded to factored-EST to update the structure.

In the action selection phase, there are two main methods to decide the action to be performed:

- If factored-EST is active
  - A set of factored-EST nodes containing the current state (or state difference set) of the agent is found in the child nodes of current factored-EST node.
  - If there exist such nodes, the one having the maximum expected reward for the current state, n_{max}, is chosen. Assuming that $\langle a_i, \cdot \rangle$ be the label of the edge connecting the current factored-EST node and n_{max} is returned.
  - Otherwise, factored-EST becomes passive and the root is assigned to be the current factored-EST node.
- If factored-EST is passive
  - Action is determined by factored ϵ-greedy algorithm.
  - If the meta-action of factored-EST is selected, it becomes active again. Corresponding child node of the root is the current factored-EST node.

Factored ϵ-greedy algorithm (Algorithm 14) enriches regular ϵ-greedy method by using the meta-action provided by factored-EST while selecting the action. The meta-action a_M is the one connecting the root to a decision tree containing current state s in one of its paths with maximum value among other trees containing s too. The chosen action is added to candidate actions list. Then,

Algorithm 13 Reinforcement learning with factored-EST algorithm

Require: T is a factored-EST with only root node
Require: A is the set of actions
procedure SELECT-ACTION(S)
Require: S is the state difference set between current state and previous state ▷ Empty if factored-EST is passive
    if *active* is true **then**
        Let $N = \{n_1, \cdots n_k\}$ be the set of child graphs of the current node containing state difference set S.
        if $N \neq \emptyset$ **then** ▷ There are actions that can be offered by factored-EST
            Select n_i with maximum reward
            Let $\langle a_i, \cdot \rangle$ be the label of the edge connecting n_i to the current node
            $current = n_i$
            return a_i
        else
            $active = false$
            $current = root$
        end if
    end if
    Let a_M be the action connecting root node to the children and can be used in state s
    Choose an action a based on the expected return values, using FACTORED-ϵ-GREEDY algorithm.
    if $a = a_M$ **then**
        $active = true$
        Let n be the child node of the root node connected with edge labeled with $\langle a_M, \cdot \rangle$
        $current = n$
        return a_M
    else
        return a
    end if
end procedure
repeat
    Let *current* is active node of T
    $current = root$ ▷ Initially current node is the root
    Let s be the current state
    $h = s$ ▷ History includes only initial state at the beginning
    $active = false$ ▷ Initially factored-EST is not active
    repeat
        Let a_{next} =SELECT-ACTION ▷ Select the next action
        Take action a_{next} and observe reward R and next state s'
        Update state-action values using underlying RL algorithm
        Append a_{next}, R, s' to h
        $s = s'$
    until s is a terminal state
    UPDATE-TREE(T, h)
until a termination condition is satisfied

with ϵ probability, next action is decided randomly while with $1-\epsilon$ probability the best action is selected (see Algorithm 14). In the worst case, RL algorithm chooses an action based on the current state-action values, which is updated after each observation.

Algorithm 14 Factored ϵ-Greedy algorithm to select an action to apply

procedure FACTORED-ϵ-GREEDY(s, T, Q, ϵ)
Require: s is a given state in the form of $(x_{1_1}, \cdots, x_{k_1})$
Require: T is the factored-EST
Require: Q is the estimated state-action value function
    $A' = A$ ▷ Set of states are copied into another list
    for all $a \in A$ **do**
        $value[a] = Q(s, a)$ ▷ Value of performing each action is $Q(s, a)$
    end for
    Let $N = \{n_1, \cdots, n_k\}$ ▷ N is the child decision trees of root containing s in their continuation sets
    if $N \neq$ **then**
        Select n_i with maximum expected reward
        Let $\langle a_{n_i}, \cdot \rangle$ be the label of the edge connecting root and n_i
        Let $\langle s, \cdot, R_{s,n_i}$ be the corresponding tuple in n_i
        $A' = A' \cup \{a_{n_i}\}$
        $value[a_{n_i}] = R_{s,n_i}$
    end if
    Pick a number $p \cup [0, 1)$ with uniform probability
    if $p < \epsilon$ **then**
        Return a randomly selected action from A'
    else
        Return action a with maximum value ▷ If there are multiple actions with maximum value select one of them randomly
    end if
end procedure

The last step of updating an EST instance (Algorithm 4) is updating the root node, which calls itself recursively for every child node. Our approach in factored-EST model for this operation is given in Algorithm 15. After decrementing eligibility values of the edges, the ones with low eligibility values are deleted from the tree with their child nodes. The same procedure is applied to the remaining child nodes. Then, the current node is updated depending on the type of the structure. In case of state difference graphs, Algorithm 10 is called with given to-be-deleted list of decision tree leaves, which is LS. If the current node is a decision tree, it is traversed to decrement eligibility values and find the states that should be deleted. If it is decided to do so, state leaf links are added to

LS. In factored-EST, every state difference graph G has only one predecessor tree node D whose type is decision tree. It is guaranteed that G cannot contain decision tree node links except for the ones in D. Thus, it is sufficient for G to know to-be-deleted decision tree nodes of D that are collected in LS. After all of the children of D are updated using these marked leaf nodes, every leaf in LS is deleted from D to preserve compactness.

Algorithm 15 Algorithm for updating a factored-EST node n

1: **procedure** UPDATE-NODE(n, LS)
Require: LS is set of decision tree nodes which will be deleted from the decision tree.
2: Let E be the set of outgoing edges of node n
3: **for all** $e = \langle n, n', \langle a_{n'}, {\psi'}_{n,n'} \rangle \rangle \in E$ **do**
4: ${\psi'}_{n,n'} = {\psi'}_{n,n'} * \psi_{decay}$
5: **if** ${\psi'}_{n,n'} < \psi_{threshold}$ **then**
6: Remove e and sub-tree starting with n'
7: **else**
8: UPDATE-NODE(n', LS)
9: **if** There are no states in n' **then**
10: Remove e and sub-tree starting with n'
11: **end if**
12: **end if**
13: **end for**
14: **if** n is a graph **then**
15: UPDATE-GRAPH(LS)
16: **else** ▷ Decision tree update
17: **for all** $l = \langle \xi_i, R_i \rangle \in L$ **do** ▷ For all leaf nodes
18: $\xi_i = \xi_i * \xi_{decay}$
19: **if** $\xi_i < \xi_{threshold}$ **then**
20: Remove l from L
21: $LS = LS \cup \{l\}$ ▷ Add leaf node to list to update graphs
22: **end if**
23: **end for**
24: **end if**
25: **end procedure**

CHAPTER 4

EXPERIMENTAL RESULTS

We have experimented factored-EST on widely accepted benchmark problems. The main goal of the experiments is to see the effectiveness of the method on memory usage. Unlike its new structure, since the algorithm of using factored-EST in learning is same as the one in the original EST, it is expected to see no difference in learning rate. In order to compare the new structure with the classical one, a plain EST implementation is used, which stores all variable values at each step. To show that factored-EST makes learning faster, we analyzed the number of convergence steps in factored-EST, plain EST and a classical RL algorithm that does not use EST at all. In addition, to compare different temporal abstraction methods in terms of learning speeds, factored-EST, which is a sequence-based method, is compared with hierarchical decomposition method HEXQ.

4.1 Settings

In order to minimize the effects of external factors in the experiments, the same underlying RL algorithm, Q-learning, is used in all tested method instances (EST, factored-EST and HEXQ). Unless otherwise stated, all experiments are conducted with the parameter values given in Table 4.1. For each benchmark problem, 100 experiments are run with 50 episodes and the average of the results are reported. For both EST versions, factored ϵ-Greedy method is used as action selection mechanism with a constant ϵ value. Resulting plots are smoothed for visual clarity. The results of different mechanisms are compared in terms of:

- **EST sizes**: Memory consumption of EST and factored-EST structures
- **EST update times**: Speed of updating EST and factored-EST structures
- **Number of steps to reach a goal**: Number of steps it takes to finish a task for EST, factored-EST and Q-learning algorithms
- **Number of active steps of EST**: Number of steps where EST and factored-EST decided the next action
- **EST execution times**: Average time elapsed when EST and factored-EST are active
- **Reward per step**: Average rewards gained per step for EST, factored-EST and Q-learning algorithms.

Table4.1: Learning settings

Parameter	**Value**
α	0.01
γ	0.9
λ	0.9
ϵ	0.1
ψ_{decay}	0.95
ξ_{decay}	0.99
$\psi_{threshold}$	0.01
$\xi_{threshold}$	0.01

4.1.1 Problems

Coffee-Robot Domain

In coffee-robot domain [5], a robot in the office tries to bring coffee from a coffee shop to its owner who is located in the office. The weather can be rainy, in this case, the robot should take an umbrella before leaving the office. Successful delivery while staying dry results in 1.0 reward, but if the agent is wet, it gets 0.9 reward. There are four actions which can be performed:

- **Move**: Go to the other location.
- **GetUmbrella**: Get the umbrella if you are at the office.
- **DeliverCoffee**: Deliver coffee to the owner if you are at the office.
- **BuyCoffee**: Buy coffee if you are at the coffee shop.

The environment is non-deterministic and delivering coffee is successful with probability 0.8. State variables are *HasOwnerCoffee (HOC), HasRobotCoffee (HRC), HasRobotUmbrella (U), IsRobotWet (W), Location (L), IsItRaining (R)*. Variable *Location* can take values in {*office, shop*}, while the other variables are binary. Initially, *IsItRaining* variable can take both values while *Location* is *office* and the domain for other variables is {*true, false*}. Reward tree for the domain and DBN of *DeliverCoffee* action for *HOC* variable is shown in Figure 4.1.

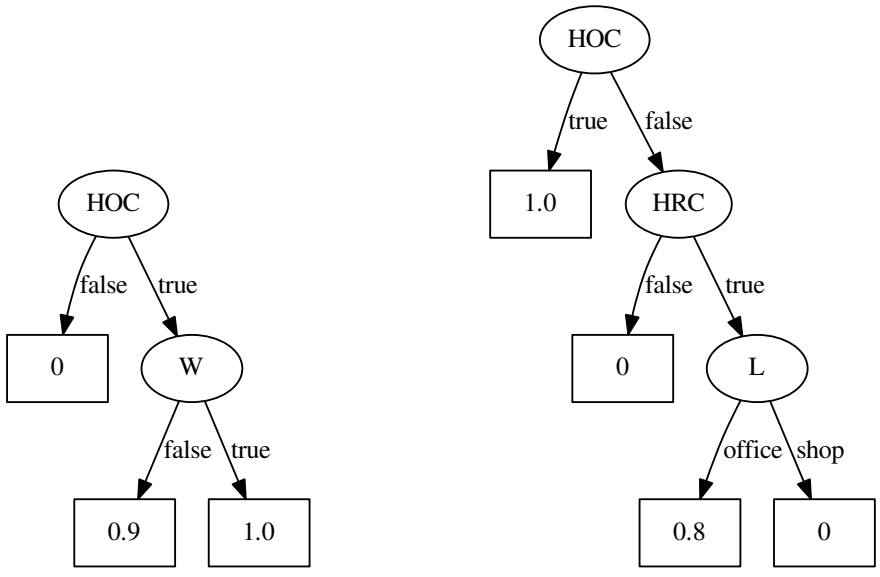


Figure 4.1: (a) Coffee domain - reward tree (b) *DeliverCoffee* DBN for *HOC*

Room Domain

Room domain is a 9×9 grid world divided into two equal rooms vertically with walls (Figure 4.2). There is only one cell that connects these rooms, which defines the bottleneck of the problem. The agent starts from top leftmost cell of the first room (S) and tries to reach to the bottom rightmost cell of the second room (G). It can move to *north, south, east* and *west*. There are only two state variables, x and y. They are used to describe coordinates on the grid with domain $\{1, 2, 3, \cdots, 9\}$. For example, in the problem S is $(x = 1, y = 1)$ and G is $(x = 9, y = 9)$. The environment is non-deterministic like coffee-robot problem. Attempting to move to a cell is successful with probability 0.8 and the agent will go to one of the diagonal cells otherwise. Every move causes -0.01 reward (punishment due to action cost) except for reaching to G, which gives 1.0 reward.

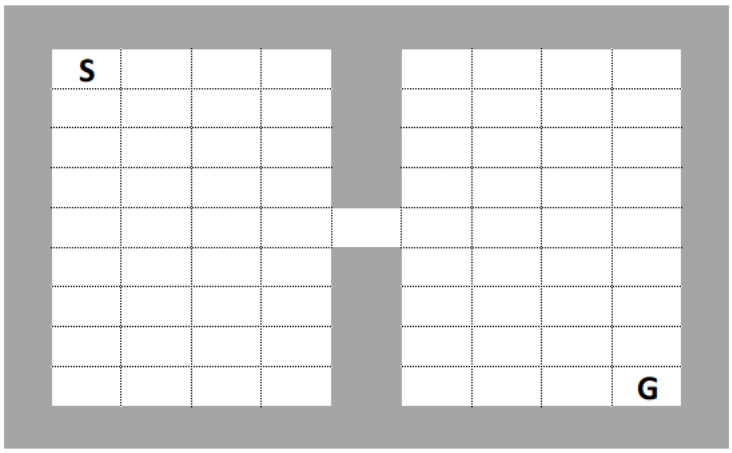


Figure 4.2: Room domain

Taxi Domain

The last benchmark problem is factored version of the taxi domain [10] (Figure 3.1). It is a task where a taxi driver moves in 5×5 grid world in four directions with some obstacles around to prevent him from moving. There are four predefined locations which are R, G, B, and Y. The goal is to pick up a passenger located in one of these four locations and deliver him to another one.

Possible actions are *north, south, east, west, pickup, putdown.* Invalid *pickup* or *putdown* action results in -10 reward. This contains cases like putting the passenger down in wrong cell, choosing *pickup* action when there is no passenger in the current location, choosing *putdown* action where there is no passenger in the taxi. Successful delivery gives 20 reward and completes the task. Concerning the state variables, x and y are used to specify coordinates (x, y) of the taxi, *passenger_location* and *destination* variables are needed to represent passenger location and current destination of the agent. The environment is deterministic and apart from *pickup* and *putdown* actions, all actions can only affect either x or y variables. Table 4.2 contains state variables, their domains and actions that cause changes in their values.

Table4.2: Taxi state variable domain sets and affecting actions

Variable	**Domain**	**Affecting Actions**
x	$1, 2, 3, 4, 5$	$north, south, east, west$
y	$1, 2, 3, 4, 5$	$north, south, east, west$
passenger_location	R, G, B, Y, T	$pickup, putdown$
destination	R, G, B, Y	$pickup$

In our instance of the problem, the agent start at cell $(1, 1)$ and picks up the passenger at G. Passenger destination is location R.

4.2 Results

4.2.1 Number of Steps to Reach a Goal State

In order to see the effects of temporal abstraction in FMDPs, the algorithms are tested in terms of speeds of convergence to an optimal policy. Figures 4.3, 4.4 and 4.5 show how many steps it takes to solve the tasks using Q-learning algorithm. In all of the problem instances, using one of the EST versions on top of the RL algorithm speeds up the learning rate. However, in coffee-robot domain (Figure 4.3), Q-learning also performs as good as other cases by the 7^{th} step. Because there are small number of states and actions to consider in the problem.

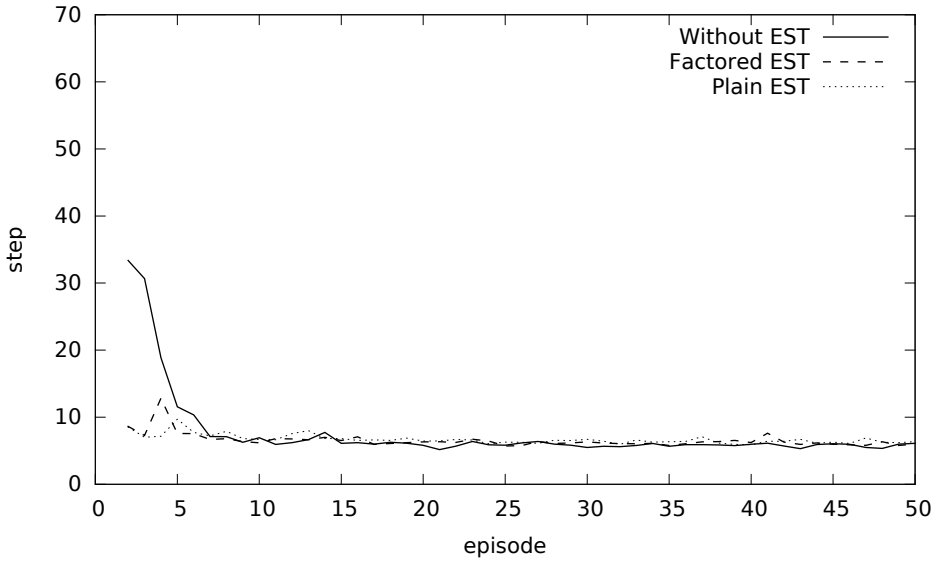


Figure 4.3: Comparison of number of steps to reach a goal in coffee-robot domain

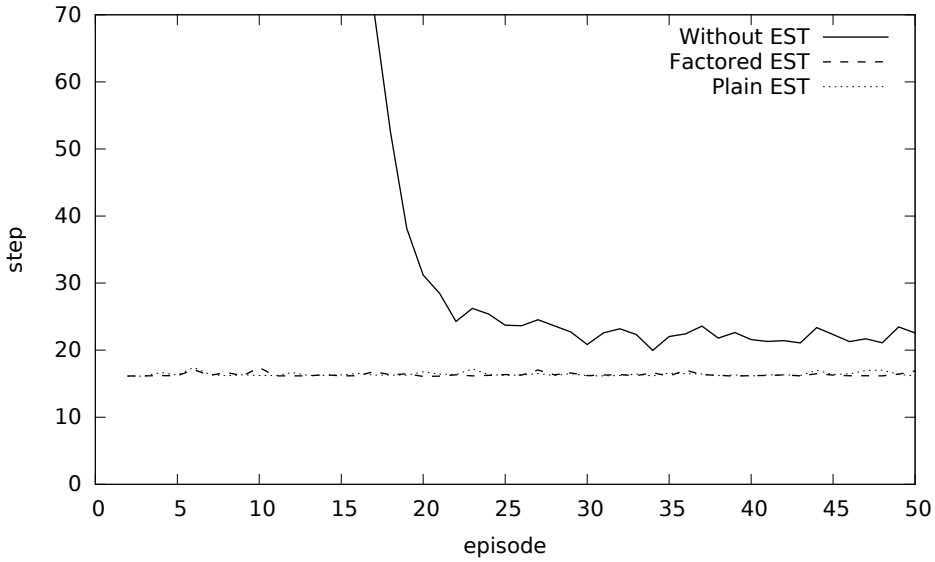


Figure 4.4: Comparison of number of steps to reach a goal in taxi domain

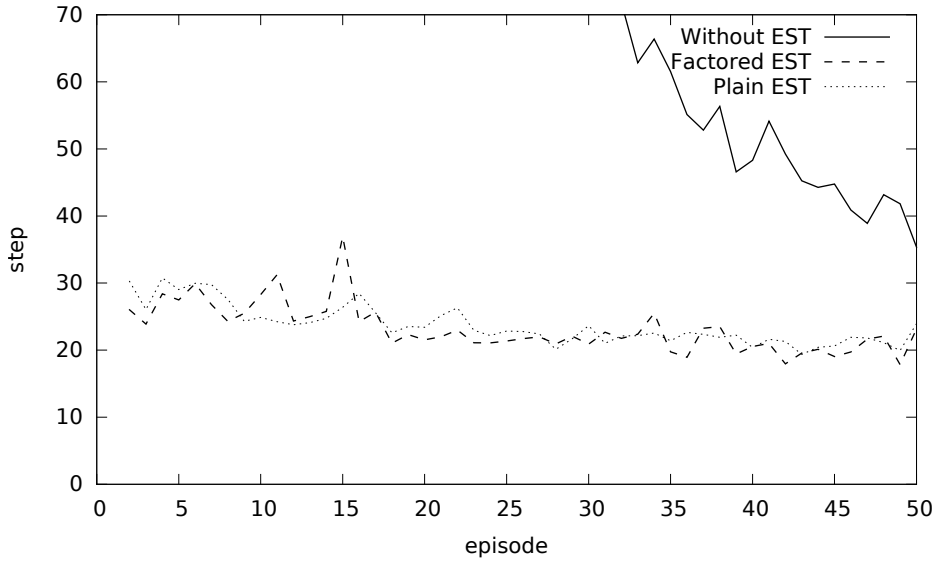


Figure 4.5: Comparison of number of steps to reach a goal in room domain

The results of taxi and room domains show the advantage of EST more clearly. Unlike coffee-robot task, these tasks have a bigger state space, which makes it difficult to find a good solution in a reasonable number of steps with a classical RL algorithm. It is important to point out that, determinism in taxi domain leads to finding an optimal policy in a very short time with EST and factored-EST, while Q-learning algorithm cannot achieve it even after 50^{th} episode. Similarly, the sharp changes in room domain is the result of non-determinism of transitions. For Q-learning, the number of steps in the early episodes of these tasks are huge compared to EST versions. Thus, in order to narrow down the scale of the results, they are ignored in the plots.

4.2.2 Average Tree Sizes

The average memory consumptions of EST and factored-EST are shown in Figures 4.6, 4.7 and 4.8. In coffee-robot domain, which is a slightly smaller problem compared to the others, factored-EST performs better than EST in all of the episodes. But in other two domains, factored-EST uses more memory in the early episodes. Because internal links of state difference graphs and decision

trees decreases memory efficiency. After pruning starts, unnecessary parts are deleted from the trees which causes a significant drop in both plots and factored-EST outperforms EST in a short time.

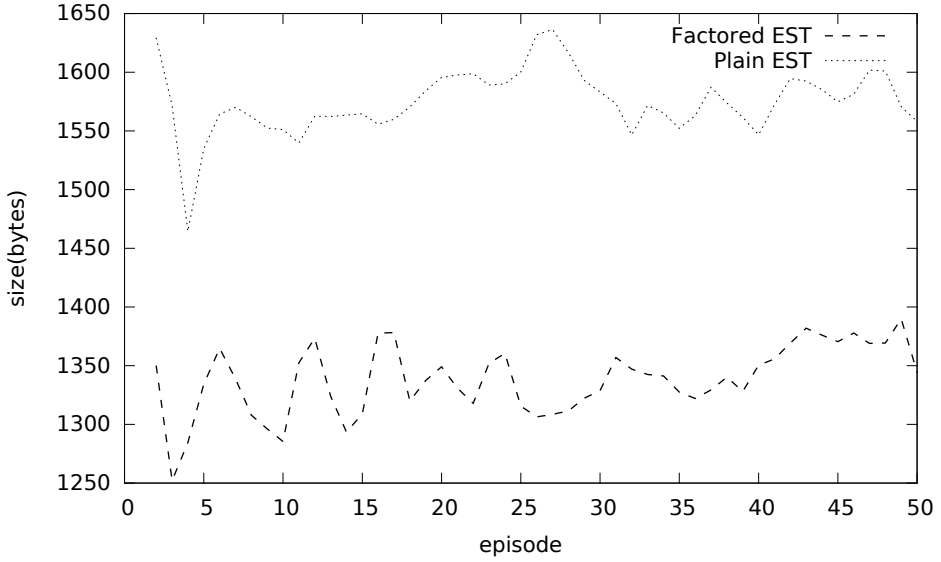


Figure 4.6: Comparison of factored-EST and EST sizes in coffee-robot domain

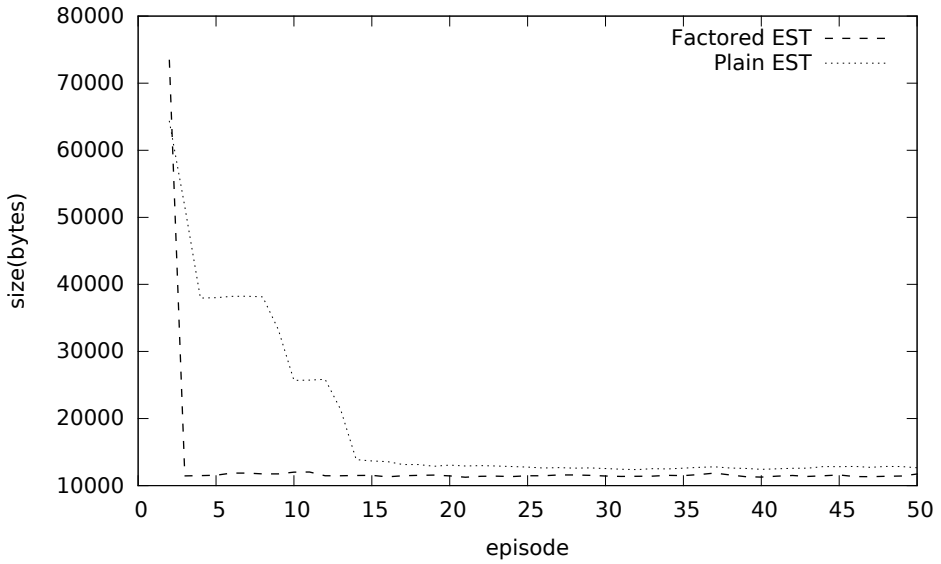


Figure 4.7: Comparison of factored-EST and EST sizes in taxi domain

The amount of pruning is directly related with the eligibility threshold values.

Higher values results in continuous pruning with small changes while for lower values sharp changes in the plots become more visible. The results of taxi domain show that, the structure of the factored-EST becomes stable before the 5^{th} episode. Since nearly optimal policies are found in the early stages of learning in both EST versions, only reward and eligibility values are updated in the structures, so, there are a small number of node/edge deletion/insertion operations. Although both factored-EST and EST use nearly the same amount of memory after 15^{th} step, it takes more time for EST to prune the tree. Because in factored-EST, if the initial state has been deleted, all the successor steps that have no common parts with other histories are also deleted immediately.

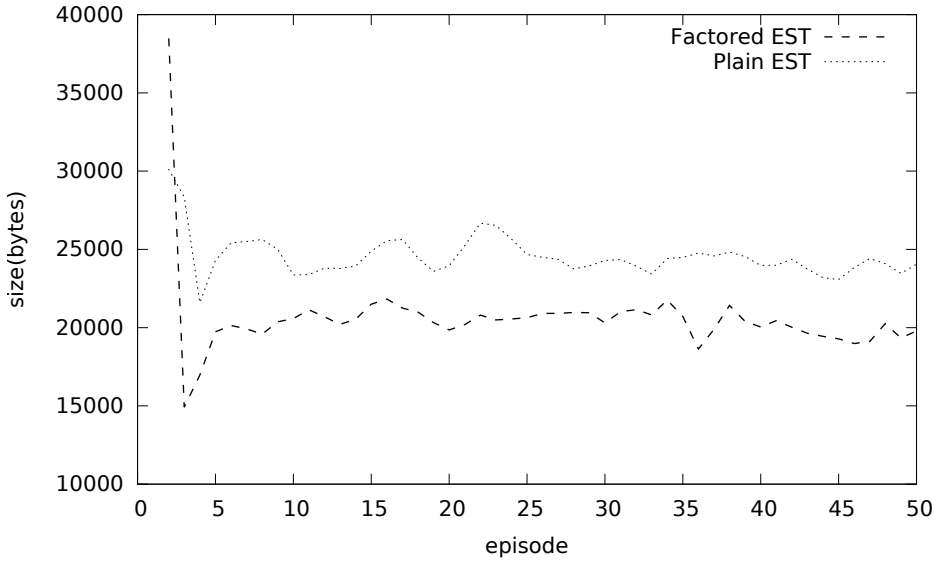


Figure 4.8: Comparison of factored-EST and EST sizes in room domain

4.2.3 Average CPU Times of Tree Updates

The gain in memory efficiency of factored-EST causes an overhead on tree update operation. Figures 4.9, 4.10 and 4.11 compare EST and factored-EST in terms of time consumed to add new histories and prune the parts with low eligibility values in the structures. The graph and tree data structures in the nodes of factored-EST requires additional search and comparison operations. In order the find out the existence of a given variable instance, the algorithm traverses

a decision tree or a graph. Moreover, each history is processed to find value differences between subsequent states. All these extra processes result in more time consumption than classical EST algorithm.

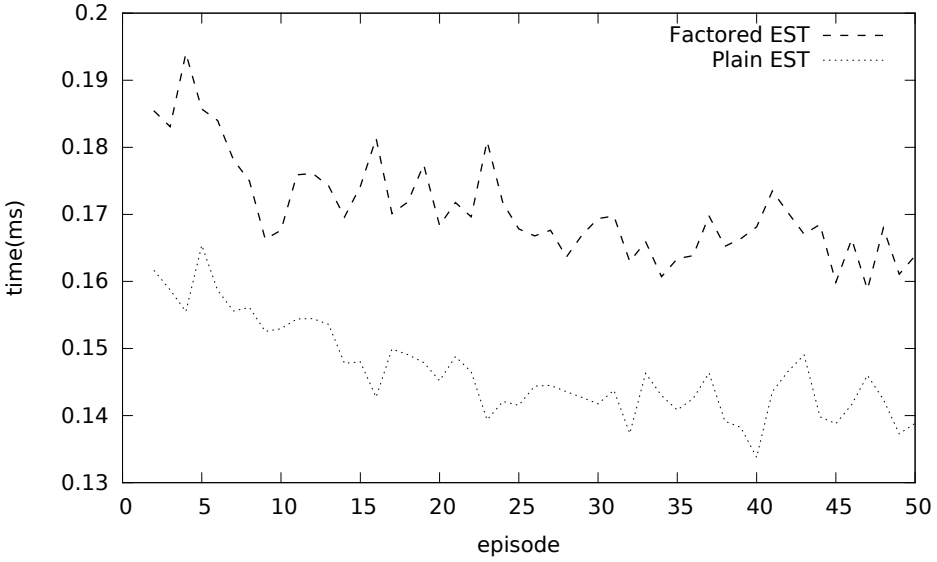


Figure 4.9: Comparison of factored-EST and EST structure update CPU times in coffee-robot domain

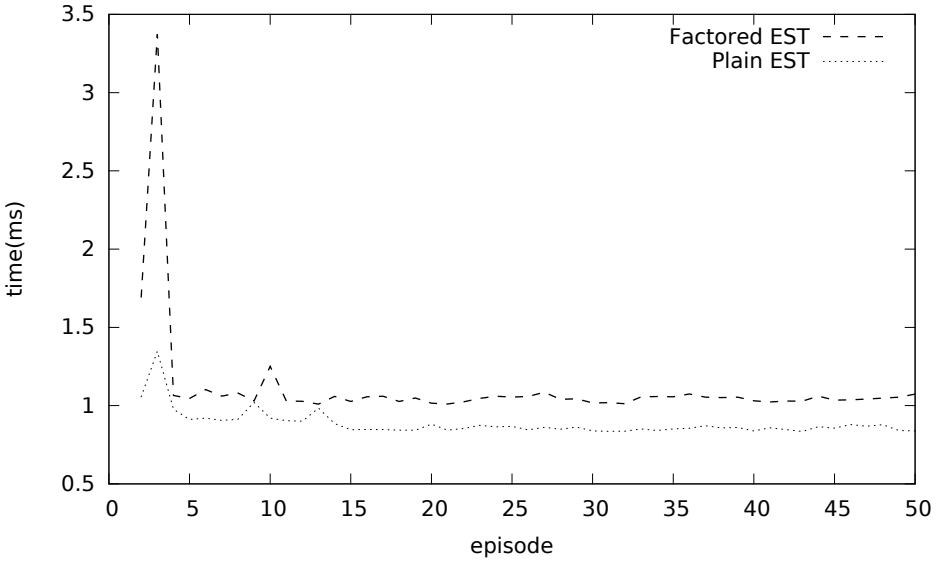


Figure 4.10: Comparison of factored-EST and EST structure update CPU times in taxi domain

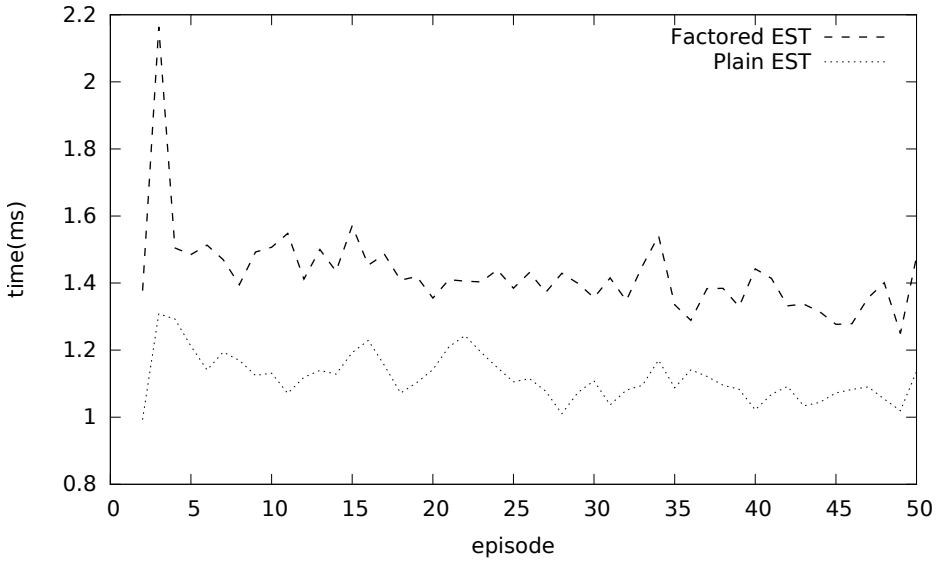


Figure 4.11: Comparison of factored-EST and EST structure update CPU times in room domain

As seen in the plots, time consumption is directly proportional to the size of the tree. In addition, there are some peaks showing that there is a high amount of pruning in that point. Especially in taxi and room domains, it is more obvious when the first pruning operation starts before the 5^{th} episode. Similar to the other plots, in taxi domain, in a short time, results of both versions become very similar.

4.2.4 Average Reward Per Step

Figures 4.12, 4.13 and 4.14 show the average amounts of rewards received by the agent in all benchmark problems. In the room domain, the cumulative reward is directly related with number of steps to reach the goal state. Because, each extra move brings -0.01 reward. Thus, finding a shorter path increases average rewards per step. In coffee-robot domain, Q-learning algorithm performs better than EST versions. One of the possible reasons is that EST versions first find a solution without taking the umbrella and the robot gets wet while delivering the coffee. It seems that Q-learning finds the optimal policy resulting in gaining more cumulative rewards. It is possible to get the same total reward with a

sub-optimal behavior in taxi domain. But, since EST and factored-EST finish the task in fewer number of steps, average reward per step results are better than Q-learning algorithm.

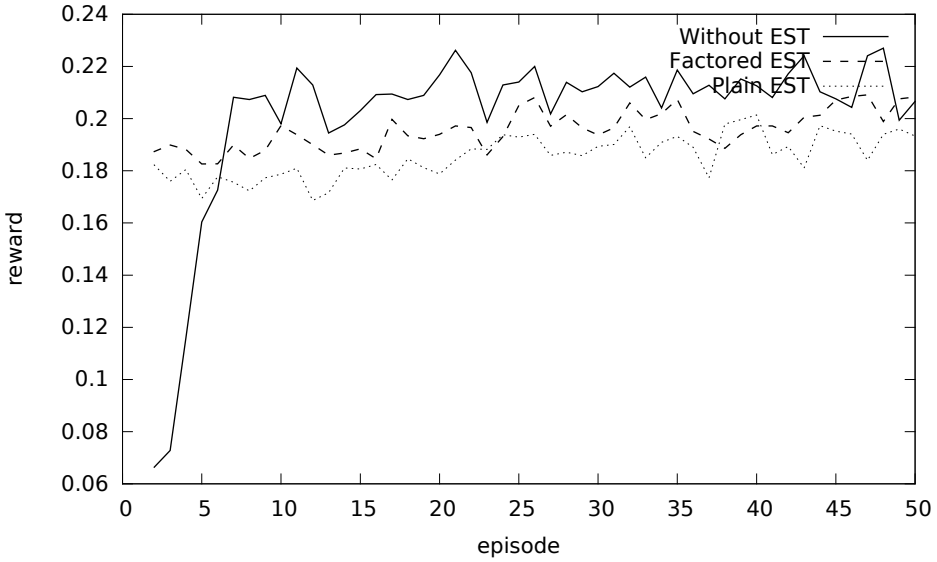


Figure 4.12: Comparison of average reward gained per step in coffee-robot domain

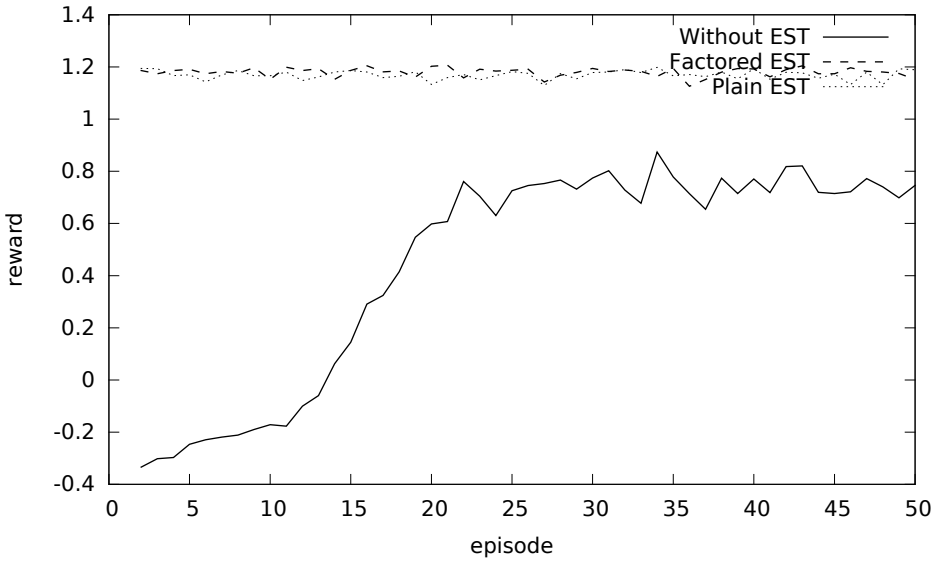


Figure 4.13: Comparison of average reward gained per step in taxi domain

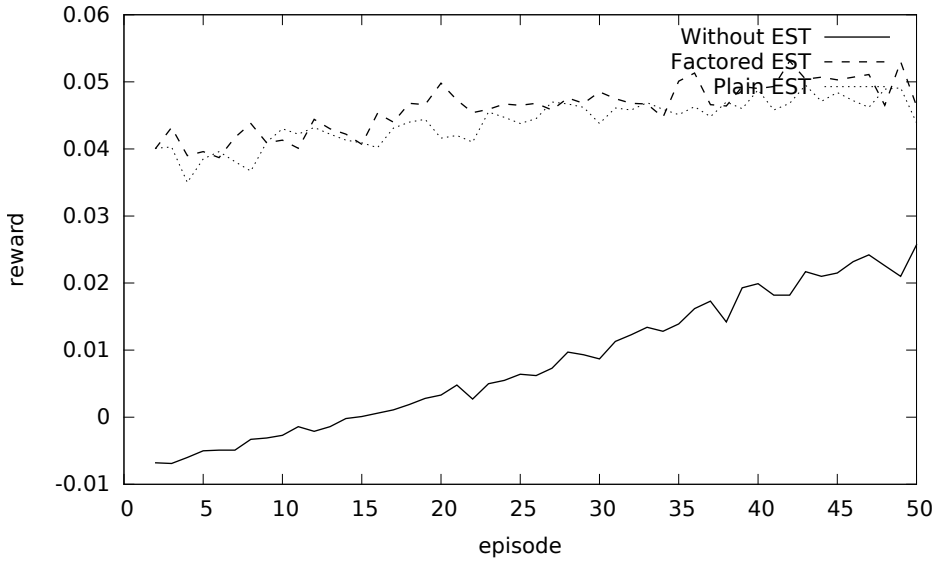


Figure 4.14: Comparison of average reward gained per step in room domain

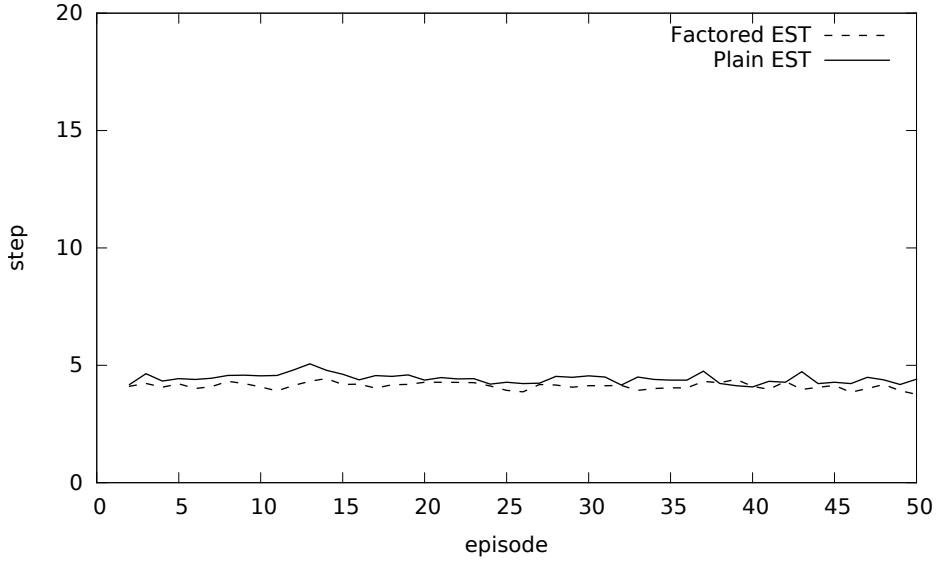


Figure 4.15: Comparison of average active steps of factored-EST and EST in coffee-robot domain

4.2.5 Other Experiments

In addition to the criteria given above, we have observed the number of steps in which EST and factored-EST are active as shown in Figures 4.15, 4.16 and

4.17. The results show how many actions are decided in an episode in average. In all of the problems, both EST and factored-EST contribute to the solutions in nearly the same number of steps. The comparison of the measurement of average milliseconds that EST and factored-EST are active in an episode given in Figures 4.18, 4.19 and 4.20 respectively.

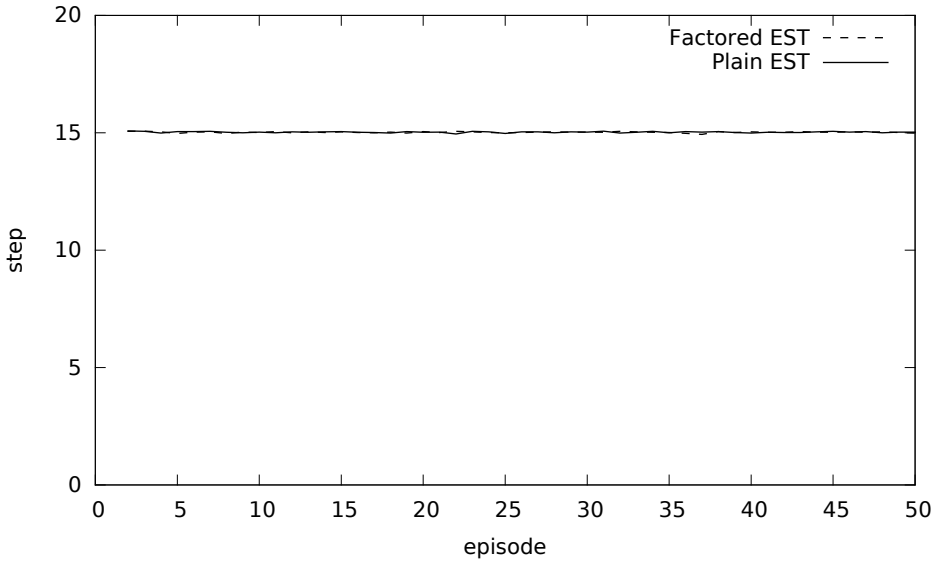


Figure 4.16: Comparison of average active steps of factored-EST and EST in taxi domain

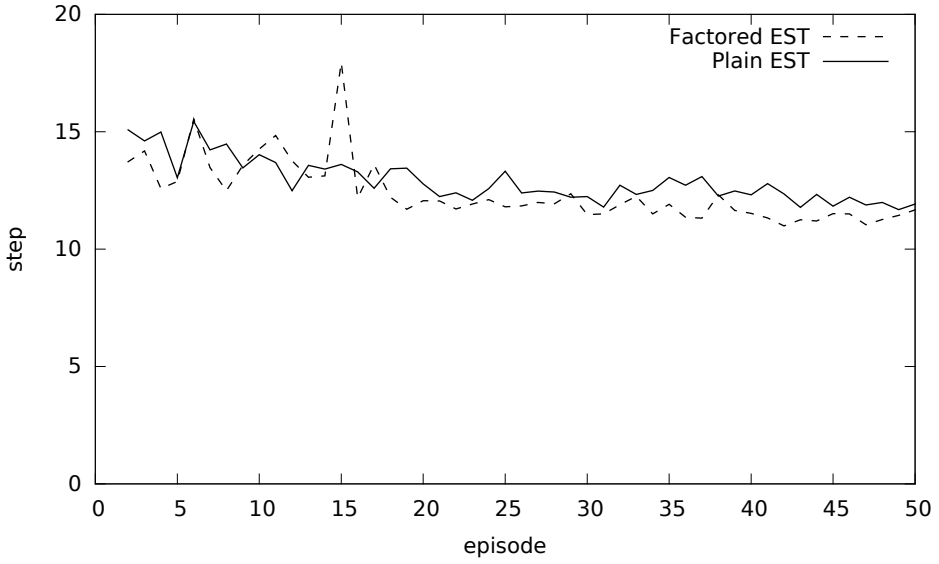


Figure 4.17: Comparison of average active steps of factored-EST and EST in room domain

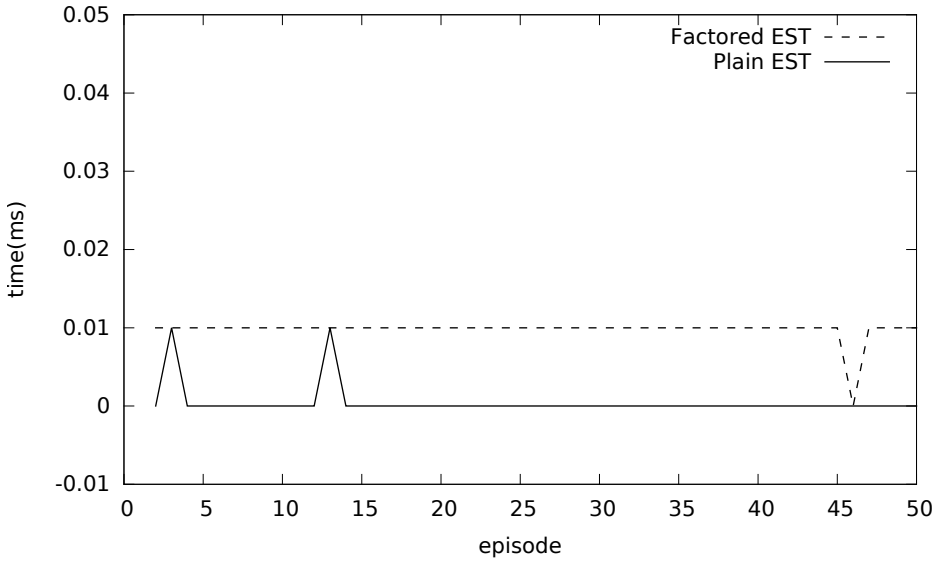


Figure 4.18: Comparison of average active milliseconds of factored-EST and EST in coffee-robot domain

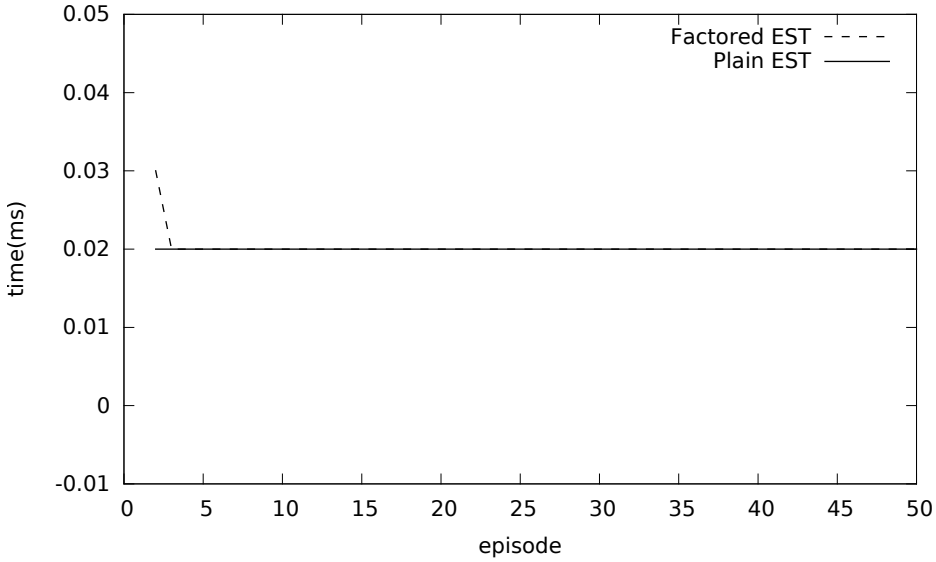


Figure 4.19: Comparison of average active milliseconds of factored-EST and EST in taxi domain

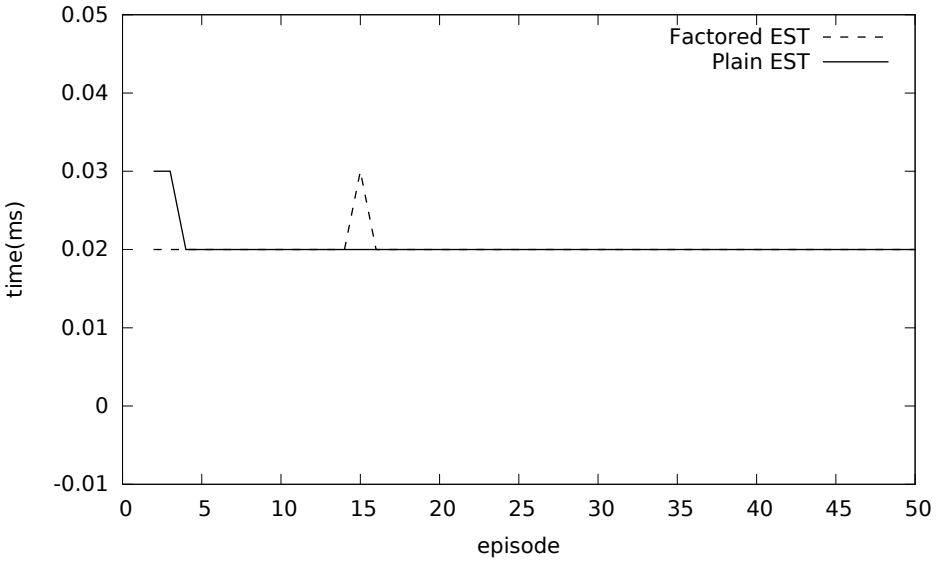


Figure 4.20: Comparison of average active milliseconds of factored-EST and EST in room domain

4.2.6 HEXQ/EST/Factored-EST Comparison

We have also done some experiments in order to compare the effects of temporal abstractions in sub-goal based methods and sequence-based methods in terms of learning rates. For this purpose, we selected HEXQ algorithm as an automatic hierarchy decomposition technique. Because, as it requires an underlying RL algorithm like factored-EST, we are able to compare the effectiveness of the structures constructed by both algorithms. Taxi domain (section 4.1.1) is chosen with some modifications in state variables. Instead of using x and y for expressing cell location, only one variable $loc \in [0, .., 24]$ is used. Actions are the same but, the transition function is changed according to the new state representation. Initially located in cell 5, the agent picks the passenger up from B and delivers it to Y. In HEXQ algorithm, after hierarchically decomposing the problem, Q-learning is used in order to solve constructed sub-MDPs.

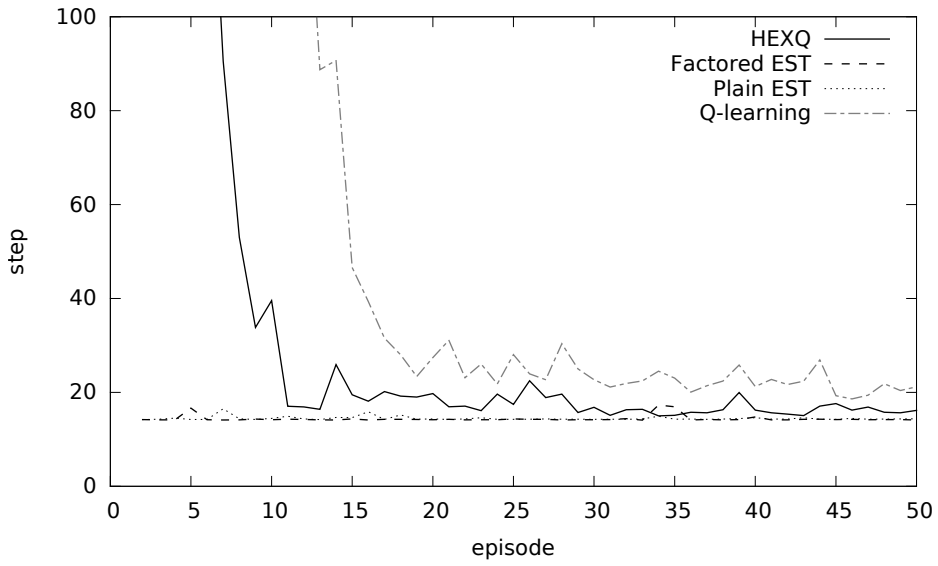


Figure 4.21: Comparison of learning speeds of different temporal abstraction techniques in taxi domain

Figure 4.21 shows the resulting learning speeds of EST, factored-EST and HEXQ algorithms compared to Q-learning algorithm. Although HEXQ performs better than Q-learning, since it still solves sub-MDPs in two levels, it takes approximately 10 episodes to approach optimal solution. However, EST and factored-EST eliminate the unnecessary steps from the first history and finds the desired solution in the first episode.

4.3 Discussion

In our experiments, firstly we have compared the proposed factored-EST model with the original one in terms of memory usage of the structures and time consumptions while updating the trees. The results show that, factored-EST outperforms EST in memory usage. The drawback of the algorithm appears in traversing decision trees and graphs which is done periodically during the learning. These operations are relatively more time consuming than the methods applied by EST algorithm. Additionally, we also observed number of steps to reach a final state for factored-EST, plain EST and Q-learning without EST to see that there is no negative effect of factored-EST implementation on learning

rate. Note that the results of the first episodes are omitted from the plots, since EST is not active until the first histories are generated and inserted into the system.

CHAPTER 5

CONCLUSION AND FUTURE WORK

This thesis proposes factored-EST algorithm that constructs a compact tree representation for maintaining useful experiences of a learning agent. EST aims to detect and store useful sub-sequences of experiences frequently encountered during the learning process and help the agent to use them later. It keeps some statistics of the history of states and actions in a tree structure. Throughout the learning process, these useful action sequences are fed to the underlying reinforcement learning algorithm. In order to keep memory usage in a reasonable level, the tree is updated dynamically during the learning process. It is an effective way to increase learning speed for problems defined with classical MDP framework.

In real-world problems, mostly, state and action spaces are huge and states are represented by a set of variables. There are some approaches to take the advantage of new state representation. These methods utilize the new structured model to define and solve learning problems in a more compact way to increase the learning rate and require less memory while finding a reasonable policy. EST is proved to be an effective method for the first objective. Although it is possible to use EST in this kind of tasks, learning mechanism suffers from excessive memory usage.

We have attacked the problem by exploiting the factored structure. Our modification on EST assumes that each action affects a small subset of variables. Thus, by only storing the differences between subsequent states in a history, we can construct a more memory efficient tree structure for maintaining options.

Before inserting into factored-EST structure, history of events containing state-action-reward tuples are transformed into the factored form. Following the initial state, at each step, applied action, variables that are affected by the action and immediate reward values are appended to the new history structure. Factored-EST model uses decision trees for representing initial states and directed acyclic graphs for maintaining variable value differences of successive states. Each state difference set contains a link to the corresponding initial state so that, complete state information can be computed when necessary. Differently from classical EST, factored-EST uses initial and previous states to find reward and eligibility values of current states.

Factored-EST is tested on three benchmark problems, which are coffee-robot, room and taxi domain. The results show that new algorithm requires less memory than the classical EST structure. The learning performance of both versions are very similar. The disadvantage of factored-EST is brought by graph and tree traversal methods for searching, updating and deleting the elements, which are slightly inefficient. In addition, to compare sequence-based EST and factored-EST algorithms with hierarchical abstraction approaches with respect to their performance of converging to optimal solutions, some experiments are conducted with HEXQ algorithm.

One of the most obvious future work is decreasing the complexity of update operations in factored-EST nodes while preserving memory gain. Moreover, we are planning to employ the factored-EST method for problems with partially observability.

REFERENCES

[1] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(5):835–846, September/October 1983.

[2] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.

[3] R. E. Bellman. *Adaptive control processes - A guided tour*. Princeton University Press, Princeton, New Jersey, U.S.A., 1961.

[4] C. Boutilier, R. Dearden, and M. Goldszmidt. Exploiting structure in policy construction. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1104–1111, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[5] C. Boutilier, R. Dearden, and M. Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1–2):49 – 107, 2000.

[6] E. Çilden. *Abstraction in Reinforcement Learning in Partially Observable Environments*. Ph.D. thesis, Middle East Technical University, 2014.

[7] T. Dean and K. Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence Journal*, 5(3):142–150, 1989.

[8] T. Degris, O. Sigaud, and P.-H. Wuillemin. Learning the structure of factored markov decision processes in reinforcement learning problems. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 257–264, New York, NY, USA, 2006. ACM.

[9] T. G. Dietterich. The maxq method for hierarchical reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 118–126. Morgan Kaufmann, 1998.

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

[11] J.-P. Forestier and P. Varaiya. Multilayer control of large markov chains. *Automatic Control, IEEE Transactions on*, 23(2):298–305, Apr 1978.

[12] S. Girgin, F. Polat, and R. Alhajj. Improving reinforcement learning by using sequence trees. *Machine Learning*, 81(3):283–331, 2010.

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

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

[15] O. Kozlova, O. Sigaud, and C. Meyer. Texdyna: Hierarchical reinforcement learning in factored mdps. In *From Animals to Animats 11, 11th International Conference on Simulation of Adaptive Behavior, SAB 2010, Paris - Clos Lucé, France, August 25-28, 2010. Proceedings*, pages 489–500, 2010.

[16] C. Sahin, E. Cilden, and F. Polat. Memory efficient factored abstraction for reinforcement learning. In *Cybernetics (CYBCONF), 2015 IEEE 2nd International Conference on*, pages 18–23, June 2015.

[17] R. S. Sutton. Learning to predict by the methods of temporal differences. In *Machine Learning*, pages 9–44. Kluwer Academic Publishers, 1988.

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

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

[20] 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.

[21] P. E. Utgoff, N. C. Berkman, J. A. Clouse, and D. Fisher. Decision tree induction based on efficient tree restructuring. In *Machine Learning*, pages 5–44, 1996.

[22] C. J. C. H. Watkins. *Learning from Delayed Rewards*. Ph.D. thesis, King's College, Cambridge, UK, May 1989.

[23] C. J. C. H. Watkins and P. Dayan. Technical note: q-learning. *Machine Learning*, 8(3-4):279–292, May 1992.