MOTURU. KRISHNA PRIYA DARSINI

Application of Reinforcement learning algorithms to software verification.

Mémoire présenté
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de maîtrise informatique
pour l'obtention du grade de Maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES ET DE GÉNIE
UNIVERSITÉ LAVAL
QUÉBEC

2006

Résumé

Cette thèse présente une forme nouvelle de vérification de systèmes probabilistes en utilisant des algorithmes d'apprentissage par renforcement. Le développement de très grands et très complexes systèmes logiciels est souvent l'aboutissement d'un travail d'équipe. L'objectif est de satisfaire le client en lui livrant le produit spécifié, sans erreurs et à temps. Des erreurs humaines sont toujours faites lors du développement de tels systèmes, mais elles sont d'autant plus faciles à corriger si elles sont détectées tôt dans le processus de production. Pour ce faire, on a recours à des méthodes de vérification et de validation. Dans cette thèse, nous avons réussi à adapter des techniques d'apprentissage supervisé pour solutionner des problèmes de vérification de logiciels. Cette approche nouvelle peut-être utilisée, même si le modle complet n'est pas disponible, ce qui est une nouveauté en théorie de la vérification probabiliste. Dans cette thèse, nous ne nous intéressons pas seulement à vérifier si, oui ou non, un système se comporte exactement comme ses spécifications, mais aussi, à trouver, dans la négative, à quel point il s'en écarte.

Abstract

This thesis presents a novel form of system verification through reinforcement learning algorithms. Large and complex software systems are often developed as a team effort. The aim of the development is to satisfy the customer by delivering the right product, with the right quality, and in time. Errors made by developers will always occur when a system is developed, but their effect can be reduced by removing them as early as possible. Software verification and validation are activities that are conducted to improve product quality. In this thesis we will adapt the techniques used in reinforcement learning to Software verification to verify if implemented system meets its specifications. This new approach can be used even if the complete model of the system is not available, which is new in probabilistic verification. This thesis main aim is not only to answer the question whether the system behaves according to its specifications but also to find the degree of divergence between the system and its specifications.

Acknowledgements

This thesis would not have been possible without the support and encouragement of my supervisor Dr. François Laviolette, under whose supervision I chose this topic and began the thesis. His guidance , patience, kindness, the time he invested throughout my years as a graduate student and through the rough spots of the thesis are much appreciated. His insights were very valuable to me during the writing and editing of the thesis. The door to Prof. François Laviolette's, office was always open whenever I ran into a trouble spot or had a question about my research or writing. One simply could not wish for a better and friendlier supervisor.

I am indebted to my many student colleagues for providing a stimulating and fun environment in which to learn and grow. Most of all I would like to thank Sami Zhioua, for many interesting ideas and great conversations, which helped to form this thesis.

I express my sincere thanks to my parents, brother Madhu and sister Surekha and their families. My Brother has been an inspiration throughout my life. He has always supported my dreams and aspirations. I'd like to thank him for all he is, and all he has done for me.

Finally, I really have to thank my best friend Niranjan for standing by me in good and bad times.

To my nephew Rajeev and niece Manasa

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software is now used in various products, from small hearing aids to large super tankers. The hardware used for the software products becomes smaller and cheaper with an increased performance each year. The software products and systems have at the same time become more complex and include more functionality. The users of the software products expect the systems to function without problems even though the systems become more complex.

There is always a risk, however, that a software does not behave as expected. The system can return the wrong value, break down, or perform something else that the user does not expect. If it is a safety critical system then wrong behavior can be devastating. For personal computers, the wrong behavior might lead to that the program has to be restarted. The wrong behavior is often caused by human errors during software development.

The task of developing fault-free software systems is not easy. As the number of lines of code increase and the number of interaction possibilities with the user, the environment, or other systems increase, humans have difficulties of over-looking and foreseeing all possible outcomes of the program execution.

The human errors can occur in several ways. For example, the designer of the system might have misunderstood the way the system is expected to be used by the customers, or the programmers might have misunderstood the designers of the system. A logical fault could also have been implemented by mistake of a programmer.

These kinds of mistakes will always occur but their effects can be reduced before the system is delivered to the customers with, for example, efficient verification and

validation activities.

Motivated by the importance of fault-free software, we introduce in this thesis new kind of evaluation metric, based more precisely on trace equivalence divergence, to verify the conformance between the implemented system and its pre-established specifications. To do so we adopt the techniques of reinforcement learning.

The introduction of this thesis is structured according to the following. The next section presents research context and focus. Section 2 gives structure of the thesis.

1.1 Research Context and Focus

System verification refers to the activity of evaluating the system against specification of the system, i. e. answering the question: Is the system right? System validation refers to the activity of evaluating the system with respect to the customer's expectations, i. e answering the question: Is it the right system? Verification can thus be described as evaluating whether the system behaves according to the specifications, while validation can be described as evaluating whether the system behaves according to the user's expectations.

In this thesis we consider only one activity i.e., System verification to check if implemented system meets its specifications. There are many notions available to verify the equivalence between two processes in order to check if an implemented system confirms its specifications. For example we have trace, failure, ready, simulation and bisimulation equivalences. In this thesis we are interested in probabilistic trace equivalence notion.

If two processes (one process being the implemented system and another being the specification) are trace equivalent we say that the system behaves according to the specifications. One possible approach of interest to us from which this problem can be tackled is through the use of techniques from artificial intelligence (AI), particularly reinforcement learning (RL). Reinforcement Learning has two key advantages: the potential of learning how to verify a larger system in a short time and the ability to do so with or without a model of the system. Reinforcement learning concerns the problem of a learning agent interacting with its environment to achieve a goal (Sutton [19]). Instead of being given explicit examples of desired behavior, the learning agent must discover by guided trial and error how to behave to get the most reward (or reinforcement signal). Reinforcement learning has become popular as an approach to artificial intelligence because of its algorithms and mathematical foundations (Sutton

and Barto[18], Bertsekas and Tsitsiklis[1]) and also because of a series of successful applications (Tesauro[23]; Crites and Barto[4]; Zhang and Dietterich [26]). In addition, the basic advantage of RL compared to other learning approaches is that it requires no information about the environment except for the reinforcement signal (Narendra and Thathachar [13]). Reinforcement learning theory will be discussed in detail in chapter 3.

The model used in verification theory to represent the process is called Labelled Markov Process (LMP). This model is an oriented graph where the vertices represent the states of the process and the outgoing edges are associated with possible actions. The choice of a specific edge given an action is determined by a stochastic probability distribution. RL is generally used to solve the so-called Markov decision processes(MDPs). In other words, the problem of finding the difference between any two LMPs RL methods must therefore be translated into a problem of finding the optimal policy on some particular MDP. MDPs are very similar to LMPs.However with LMPs,the goal is to find all the possible ways to get the system into a dead lock and for MDPs the goal is usually to find an assignment of labels to states such that the total (discounted) reward obtained is maximized. In other words, although MDPs can be viewed roughly as LMPs with rewards, the goal in solving them is very different: in the first one, we want to find a way of behaving, or policy, which yields a maximal amount of reward, and in the second one, we are interested in questions regarding which states are reachable, which states will be revisited etc.

Blute, Desharnais, Edalat and Panangaden [2] introduced the notion of bisimulation for Labelled Markov Processes by extending the Larsen and Skou's [11] notion of bisimulation for probabilistic transition systems. This current notion of bisimulation distance is based on knowing the models of the two LMPs. Intuitively the smaller the distance between the two processes the more alike their behavior; in particular, they showed that states are at zero distance just in case they are bisimilar. On the other hand the RL approach that we develop in this thesis is particularly effective even when a model of the system is not available, but interaction with the system is permitted. This is exactly the situation here as we do not have any knowledge of the process that represents the system implementation but we are only able to interact with it. Once we construct the MDP from the two LMPs(One representing the implemented system and another representing the specifications), whose divergence needs to be verified, we apply the RL algorithms on this newly obtained MDP to find a policy, whose value determines the degree of divergence between the two initial LMPs.

1.2 Structure of the thesis

This section provides the structure of this thesis.

Chapter 2 reviews the theory of probabilistic verification. Here we recall the definitions of both probabilistic and non-probabilistic labelled transition systems. We also present the definition of Labelled Markov Process which is a particular type of probabilistic labelled transition system. We give the intuition behind trace and bisimulation equivalence by providing their formal definitions and modal logic. Finally, we give the metric semantics for trace and bisimulation equivalence. Since the results presented in this thesis depend on the definitions and the theory of trace equivalence we highly encourage the reader to read this chapter with much attention.

Chapter 3 describes the notations of reinforcement learning and discusses the central issues of reinforcement learning, including the trading off between exploration and exploitation in solving Markov decision processes. Different reinforcement learning techniques will be explained for example dynamic programming, monte carlo, and temporal difference methods. In this thesis we have chosen temporal difference algorithm, more precisely, Q-learning algorithm in the process of implementing our model.

Chapter 4 presents our model for checking trace equivalence between labelled Markov processes using reinforcement learning algorithms. Two approaches for MDP construction have been proposed. The first approach that we developed gives some information about trace equivalence but did not characterize it. Counter examples are provided in the chapter. However, with collaboration from Josée Desharnais, my director François Laviolette and Sami Zhioua we developed a second approach that gives rise to the notion of divergence that we were looking for. In this chapter we give the MDP construction that leads to this notion and also give empirical results related to it.

Chapter 5 concludes this thesis with a summary of the main contributions, and suggestions for future work.

Chapter 2

Probabilistic Verification

Formulating suitable models for the formal description, specification and analysis of concurrent systems is an important topic of study in theoretical computer science. A *concurrent* system is one where programs communicate among themselves by some well-defined mechanism and also continuously interact with the environment. Some systems can be designed in such a way that they are guaranteed to behave correctly under all circumstances, like total breakdown of the system or its physical environment. For example, one can design integrated circuits or write computer programs in such a way that they will perform exactly as intended, if the power supply is present and the computer is not physically damaged.

On the other hand, for many types of systems this guarantee of correctness cannot be achieved either because it is impractically expensive, or because the system includes intrinsically unreliable components. Examples of these systems include telecommunication systems and computer networks, distributed systems built over these networks, and complex software-controlled physical systems such as industrial plants, electrical power stations and transportation systems.

When a system cannot be guaranteed to exhibit the desired behavior under all circumstances, it becomes important to characterize the likelihood of undesirable behaviors. For these systems, the concept of unconditional correctness is substituted by bounds on the probability that certain behaviors occur. Using probabilities will also offer a method of telling how good or bad certain systems are: if the probabilities of errors occurring are very low, the system can be deemed useful, whereas if the probabilities of errors are very high, then clearly the system will have no practical applications.

A general model of above mentioned systems is given by so called labelled transi-

tion systems (LTSs for short), which capture the notion of states and their changes by performing transitions-which are labelled by actions. We begin this chapter by introducing the basic notions studied in this thesis. We define labelled transition systems together with the notion of a process, based on which probabilistic transition systems (exact definition to follow) called Labelled Markov Processes(LMPs) were developed. The results presented in this thesis depend on the definitions and the theory of trace equivalence and bisimulation. So the reader is highly encouraged to read this chapter with much attention. We recall the definitions of labelled transition systems and probabilistic labelled transition systems in section 1 and 2 respectively. In section 3, we present the definition of labelled Markov process along with some examples. In section 4, we give the intuition behind trace and bisimulation equivalence. We then give the formal definition of trace equivalence and bisimulation. In section 5, we give modal logic for both trace equivalence and bisimulation. Finally, we give the metric semantics for trace and bisimulation equivalence.

2.1 Non-probabilistic Labelled Transition Systems

A process is the behavior of a system. The system can be a machine, an elementary particle, a communication protocol, a chess player. Perhaps the most abstract process behavior can be described as follows: a process p performs an action from several possible actions and becomes a process p'. Processes are considered as agents that can execute actions in order to communicate with their environment. These actions can be observed by an external observer and determine the visible behavior of the process.

This simple idea is formally captured by the notion of *labelled transition systems*(LTS for short)[16]. A LTS is a rooted directed graph where each edge is labelled with an *action*. Each vertex of the graph is a distinct *state* of the process, and each edge represents a *transition* between states, with transition labels determining the interaction between the process and its environment(or with other processes)

Formally, a labelled transition system consists of a set of *states*(processes), a set of *labels*(actions), and a transition relation $\longrightarrow$ describing a change of a process state: if a process p can perform an action a and become a process p', we write $p \xrightarrow{a} p'$.

Definition 1. *A labelled transition system T is a tuple $(S, i, \mathcal{A}, \longrightarrow)$ where*

- *S is a finite or countable set of states(or processes),*
- *$i \in S$ is an initial state,*

- $\mathcal{A}$ *is a set of labels (or actions),*
- $\longrightarrow \subseteq S \times \mathcal{A} \times \mathcal{S}$ *is a transition relation,*

Example 1. *Let us start with the classical example of a tea/coffee vending machine. The very simplified behavior of the process which determines the interaction of the machine with a customer can be described as follows. From the initial state representing the situation "waiting for a request" (let us call the state p), two actions are enabled. Either the tea button or the coffee button is pressed (the corresponding action tea or coffee is executed) and the state of the control process of the machine changes accordingly to p_1 or p_2. Formally, this can be described by the transitions*

$$p \xrightarrow{tea} p_1 \quad and \quad p \xrightarrow{coffee} p_2$$

Now the customer is asked to insert the corresponding amount of money, let us say one dollar for a cup of tea and two dollars for a cup of coffee. This is reflected in the control state of the vending machine with corresponding changes. It can be modelled by the transitions

$$p_1 \xrightarrow{1\$} p_3 \quad and \quad p_2 \xrightarrow{2\$} p_3$$

Finally, the drink is collected and the machine returns to its initial state p, ready to accept another customer. This corresponds to the transition

$$p_3 \xrightarrow{collect} p$$

We shall often use a graphical representation of labelled transition systems. The following picture represents the tea/coffee machine described above.

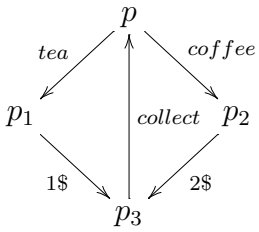


Figure 2.1: graphical representation of tea/coffee machine

From now on, the notion of a process will be equivalent to a rooted labelled transition system: the labelled transition system describes the process behavior and the root (a selected node of the transition system) represents the initial state of the process.

One of the first questions to be answered is which transition systems should be considered equivalent? Should we consider the two processes of figure 2.2 equivalent? The

notion of process equivalence is an important question in the theory of concurrency. The equivalence notion between two processes is used to compare the intended specification of the system to its actual implementation and equivalence between them is verified. In other words, we describe both specification and implementation as labelled transition systems, and then we verify if the two derived processes are equivalent. The problem of equivalence between processes is complicated by the presence of non-determinism. For better understanding of *non-determinism*, consider the two processes of figure 2.2 . The process s_0 is deterministic, whereas the process t_0 is non-deterministic. The reason being that, from the state t_0 there exists two different transitions on executing the same action. Starting from state t_0, system can make the transition either to t_1 or to t_2.

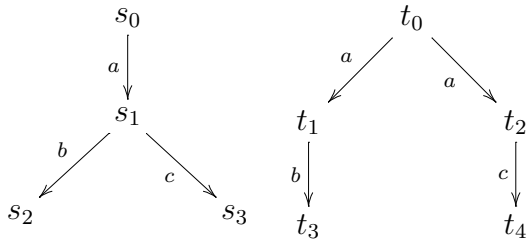


Figure 2.2: Non-deterministic process

2.2 Bisimulation: a game characterization

In general a process has many different possible behaviors, and we do not know in advance which traces will be generated by a particular execution. However we can determine in advance the set of all possible traces of a process P. The traces of a process are the possible sequences of actions that the process can perform. This set is written as *traces(P)*. The trace equivalence is denoted by $=_T$.

Definition 2. *(Trace equivalence)*$P =_T Q$*, if and only if traces(P) = traces(Q). Two processes are trace equivalent if they have the same observable behavior, as measured by traces.*

Bisimulation equivalence has many equivalent definitions, here we will give one in terms of games [22] because it is elegant and easy to understand, and therefore provides an intuitive understanding of the notion. A bisimulation game on a pair of processes p and q is a two-player game between an 'attacker' and a 'defender'. The game is played in rounds. In each round the players change the current processes p and q according to the following rule.

- The attacker chooses either p or q and performs an action a from the selected process.
- The defender has to perform the same action a from the other process.

The players reach a new pair of processes $p^{'}$ and $q^{'}$ and the game continues with another round from the current processes $p^{'}$ and $q^{'}$.

A *play* is a maximal sequence of pairs of states formed by the players according to the rule described above, and starting from the initial processes p and q. The defender is the winner in every infinite play. A finite play is lost by the player who is stuck. Note that the attacker gets stuck in current processes $p^{'}$ and $q^{'}$ if and only if no actions are enabled from these two processes. If both players get struck in the same round,then the defender is the winner.

The following standard fact highlights the connection between the winning strategies in bisimulation games and bisimulation equivalence: processes p or q are bisimilar iff the defender has a winning strategy(and non bisimilar iff the attacker has a winning strategy).

Example 2. *In this example we shall demonstrate that trace equivalence is not the same as bisimulation equivalence. Let us consider the following processes s_0 and t_0.*

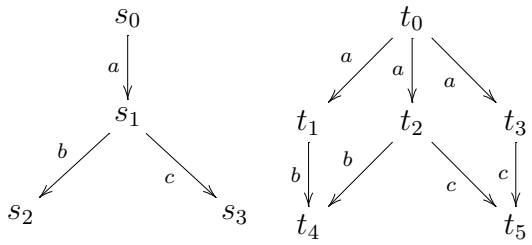


Figure 2.3: Non-bisimilar processes

Both processes can only perform the traces ab and ac, and hence are trace (linear time)equivalent. We will define a winning strategy for the attacker in order to show that s_0 and t_0 are not bisimilar. In the first round attacker chooses the action a in the process t_0 by taking the left-most transition. The defender has only one possible answer from s_0 thus reaching a process s_1. In the second round the attacker switches the process and plays from s_1 under the action c. the defender does not have the action c enabled in the other process and hence he loses.

Park and Milner [14][12] proposed the notion of bisimilarity for LTSs. This notion asserts that two processes are *bisimilar* iff any action by either can be matched with the same action by the other, and the resulting processes are also bisimilar.

Definition 3. *(Bisimulation equivalence)Let* $(P, i, \mathcal{A}, \longrightarrow)$ *and* $(Q, i, \mathcal{A}, \longrightarrow)$ *be two labelled transition systems. A relation* $\mathcal{R} \subseteq P \times Q$ *is a bisimulation if* $(p, q) \in \mathcal{R}$ *implies that for all* $a \in \mathcal{A}$*,*

- *if* $p \xrightarrow{a} p^{'}$ *then* $\exists q^{'} \in Q$ *such that* $q \xrightarrow{a} q^{'}$ *and* $(p^{'}, q^{'}) \in \mathcal{R}$*; and*
- *if* $q \xrightarrow{a} q^{'}$*, then* $\exists p^{'} \in P$ *such that* $p \xrightarrow{a} p^{'}$ *and* $(p^{'}, q^{'}) \in \mathcal{R}$*.*

Two states p, q are bisimilar if there exists a bisimulation $\mathcal{R}$ *such that* $(p, q) \in \mathcal{R}$*. Two LTSs are bisimilar if their initial states are bisimilar.*

2.2.1 Logical Characterization of Bisimulation

There exists a nice correspondence between bisimilarity and a well-known modal logic of Hennessey and Milner [10]. Two processes are bisimilar if and only if they satisfy the same formulas of the logic introduced below. It is sufficient to find a formula that distinguishes two processes in order to prove that they are not bisimilar. Hennessy-Milner logic formula has the following syntax:

$$\Phi := \top \quad | \quad \neg\Phi \quad | \quad \Phi_1 \wedge \Phi_2 \quad | \quad \Phi_1 \vee \Phi_2 \quad | \quad \langle a \rangle \, \Phi \quad | \quad [a] \, \Phi,$$

The interpretation is as follows. $\top$ is the constant true formula and is satisfied by every state. Negation, disjunction and conjunction are defined in the obvious way. Formula $\langle a \rangle \, \Phi$ is satisfied by a state if this state can become a state that satisfies ϕ by carrying out a-transition. Formula $[a]\Phi$ is satisfied by a state if every state it evolves to after performing a-transition satisfies formula Φ.

2.3 Probabilistic Labelled Transition Systems

In this section we present probabilistic analogs of labelled transition systems introduced by Larsen and Skou [11]. The natural extension of transition systems to probabilistic case is *probabilistic transition systems* (PLTS for short), also known as *Markov chains* or *discrete-time Markov process*. Probabilistic transition systems can be regarded as a specialization of non probabilistic transition systems where probabilities are used to

resolve non-determinism. In non-probabilistic transition systems, the possible steps from states to successor state are described by a transition relation $\longrightarrow \subseteq S \times \mathcal{A} \times S$ (where $\mathcal{A}$ stands for the underlying set of actions)i.e., state changes are associated with action labels. Intuitively $s \xrightarrow{a} t$ asserts that, in state s , it is possible to perform action a and to reach state t afterwards. the probability of choosing one particular transition is unspecified. In contrast, probabilistic transition systems quantify the probability of each transition by means of transition probability function P, where $P(s, a, t)$ determines the probability of performing action a from state s and to reach t in doing so.

Definition 4. *A probabilistic transition system is a tuple* $(S, \mathcal{A}, \mathcal{P})$*, where*

- S *is a set of states (or processes)*
- $\mathcal{A}$ *is a set of actions (or labels)*
- $P \in S \times \mathcal{A} \times S \longrightarrow$*[0, 1] a transition probability function satisfying:* $\sum_{s' \in S} P(s, a, s') = 0 \quad or \quad 1. \quad \forall a \in \mathcal{A} \quad \forall s \in S.$

We will represent probabilistic labelled transition systems as transition graphs whose edges are labelled with an action and a probability. We will label the edge of the graph as $a[p]$, where a is an action and p is the probability. We will just write a and drop the probability when $p = 1$.

There are two different models for probabilistic transition systems introduced by van Glabbeek et al. [25], namely *reactive* and *generative*. In the reactive model, the model selected for consideration in this thesis, the environment is allowed to offer processes one action at a time and if a process can perform this action a probabilistic choice is made between the transitions associated with this action. The result is that, for any action a process can perform, the total probability of the process performing transitions associated with this action is required to be 1. Moreover, we can consider this model as having both external and internal probabilistic choice: an external (deterministic) choice made by the environment as to which action, a process is allowed to perform, and an internal probabilistic choice as to which transition associated with this action the process subsequently performs. On the other hand, the generative model allows the environment to offer more than one action and processes then make probabilistic choices between transitions associated with these actions. Hence this model represents a type of external probabilistic choice and allows no further form of choice.

To illustrate the difference between these models consider the following example

(where + denotes probabilistic choice):

$$\frac{1}{4}a.E_1 + \frac{3}{4}a.F_1 + \frac{2}{3}b.E_2 + \frac{1}{3}b.F_2 \quad \text{and} \quad \frac{1}{8}a.E + \frac{1}{2}b.F + \frac{3}{8}c.G.$$

First, if we consider the behavior of the reactive process on the left, we note that if the environment offers the action a then the process will perform an a transition and behave as E_1 with probability $\frac{1}{4}$ and F_1 with probability $\frac{3}{4}$. Similarly, if the process is offered a b, it will perform a b transition and then behave as E_2 with probability $\frac{2}{3}$ and F_2 with probability $\frac{1}{3}$. Now the generative process on the right (recall that in the generative model the environment is allowed to offer more than one action at a time), when offered the actions a, b and c, it will choose the a transition with probability $\frac{1}{8}$, the b transition with probability $\frac{1}{2}$ and the c transition with probability $\frac{3}{8}$. If, however, the environment offers the actions a and b, then the process will perform the a transition with probability $\frac{1}{5}$ and the b transition with probability $\frac{4}{5}$. Note that these values are reached by normalizing the probabilities over the possible choices allowed, that is, over $\frac{1}{8}+\frac{1}{2}$. Similar calculations can be made for other actions being performed; in particular, if one of the a, b and c is offered, the process will choose the associated transition with probability 1.

2.3.1 Equivalences of Probabilistic Processes

In this subsection we discuss equivalences over probabilistic processes. We will restrict our attention to models containing internal probabilistic choice(for example, any model of reactive probabilistic processes), as this is the model considered in this thesis.

Definition 5. *Let $S = (S, \mathcal{A}, \mathcal{P})$ and $T = (T, \mathcal{A}, \mathcal{P})$ be two probabilistic transition systems. Then S is said to be trace equivalent to T if they accept any given trace with the same probability.*

Probabilistic bisimulation is an extension of bisimulation, described in Subsection 1.1.2, to allow for probabilities. Formally we can define a probabilistic bisimulation $\equiv_p$ over the set of processes of a probabilistic transition system, S say , as follows:

Definition 6. *Let $S = (S, \mathcal{A}, \mathcal{P})$ be a probabilistic transition system. Then a probabilistic bisimulation $\equiv_p$ is an equivalence on S such that, whenever $s \equiv_p t$, the following holds:*

$$\sum_{s' \in A} P(s, a, s') = \sum_{s' \in A} P(t, a, s') \quad \forall a \in \mathcal{A}, \quad \forall A \in S/\equiv_p.$$

$S/\equiv_p$ denotes the set of equivalence classes of S under $\equiv_p$. Then two probabilistic processes s and t are said to be probabilistic bisimilar in the case that (s,t) is contained in some probabilistic bisimulation.

As mentioned earlier every edge of a PLTS is labelled with an action and a probability. We must observe here that probabilities are not just another label and both must not be matched. Intuitively, we say that two states are bisimilar if by adding up the transition probabilities to all the states in an equivalence class of bisimilar states results in the same probability. For example, consider the following picture.

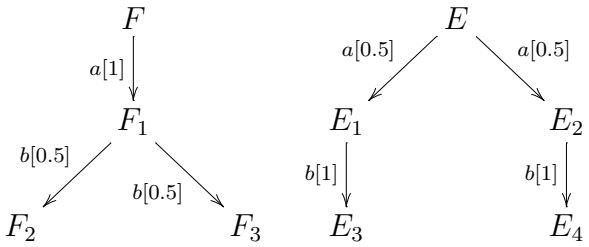


Figure 2.4: Probabilistic bisimilar processes

If one tried to match the label with the probabilities, then F and E are not bisimilar because F can jump to F_1 with probability 1 whereas E can not jump to any state with probability 1. However, we expect that states F and E are bisimilar because both can jump with probability one to respectively the state F_1 and the states E_1, E_2, which are again all bisimilar because all of them make a transition with probability 1 to states F_2, F_3 and E_3, E_4 which are also bisimilar.

2.4 Labelled Markov Processes

Labelled Markov processes (LMPs for short) provide a simple operational model of reactive probabilistic systems. They were first introduced by Blute, Desharnais, Edalat and Panangaden[2] and Desharnais [5]. An LMP can be viewed as a probabilistic transition systems(see Section 2.3), except that its state can be infinite(even continuous) and that the transition functions can be sub-probability transition function (i.e., might not sum to 1). A Markov process is a transition system with the property that the transition probabilities depend only on the current state and not on the past history of the process. Transitions are labelled with a set of actions to describe the processes interaction with its environment. In an LMP, they are governed by a probabilistic law.

Unlike in traditional Markov processes, where the probability distributions always sum up to 1, the sum of the probability distributions in LMPs can be less than 1. If a particular action can not be performed from a state then a transition with this action will have probability 0. This kind of no transitions would be represented with a transition back to itself in case of concurrency theory. However, while modeling the systems interactions with its environment it is important to distinguish between a state that can make a given transition and one that cannot. The sum of the probability distribution at any state must be between 0 and 1.

As mentioned earlier, since the state space of LMPs can be continuous we can not just specify transitions by giving, for each label, a probability for going from one state to another because in many interesting systems all such transition probabilities would be zero. Instead we must give the probability of going from a state s to a *set* of states A. Thus to deal with continuous state spaces, probability distributions are replaced with probability measures. In brief, a labelled Markov process consists of a measurable space (S, Σ) of states, a family Act of actions, and, for each $a \in Act$, a transition probability function $\mu_{-,a}$ that given a state $s \in S$, yields the probability $\mu_{s,a}(A)$ that the next state of the process will be in the measurable set $A \in \Sigma$ after performing action a.

Definition 7. *A Labelled Markov process is a triple (S, Σ, μ) consisting of a set S of states, a Σ-field Σ on S, and a transition probability function $\mu : S \times Act \times \Sigma \to [0, 1]$ such that*

- *for all $s \in S$ and $a \in Act$, the function $\mu_{s,a}(\cdot) : \Sigma \to [0, 1]$ is a subprobability measure, and*
- *for all $a \in Act$ and $A \in \sum$, the function $\mu_{-,a}(A) : S \to [0, 1]$ is a measurable.*

The function $\mu_{-,a}$ describes processes reaction to the action a selected by the environment. This represents a reactive model of probabilistic processes. Note that we consider subprobability measures, i. e. positive measures with total sum no greater than 1, to allow for the possibility that the process may refuse an action. The probability of refusal of an action a given the process is in state s is $1 - \mu_{s,a}(S)$

2.4.1 Bisimulation for Labelled Markov Processes

Blute et al.[2] introduced the notion of bisimulation for labelled Markov processes by extending the Larsen and Skou's [11] notion of bisimulation for probabilistic transition

systems. Desharnais et al.[6] have adapted this notion to the continuous case by taking measurability into account. They rather demand that equivalent states have equal probability of making transitions to any measurable set of equivalence classes of states. Instead of considering sets of equivalence classes the notion of $R - closed$ sets were considered. Let R be a relation on set S. A set $A \subseteq S$ is $R - closed$ if $x \in A$ and xRy implies $y \in A$.

Definition 8. *(Bisimulation relation) Let* $\mathcal{S} = (X, \Sigma, \mu)$ *and* $\mathcal{S}' = (X', \Sigma', \mu')$ *be two labelled Markov processes, then a bisimulation relation is an equivalence relation* R *on* $X \cup X'$ *such that for* $x \in X$ *and* $x' \in X'$*, with* xRx'*, for every* R-closed *set* $A \subseteq X \cup X'$ *such that* $A \cap X \in \sum$ *and* $A \cap X' \in \sum'$ *we have*

$$\mu_{x,a}(A \cap X) = \mu'_{x',a}(A \cap X')$$

for every $a \in Act$*. Two states are bisimilar if they are related by a bisimulation relation.*

2.4.2 Modal Logic for Bisimulation

Desharnais et al.[6] extend the logical characterization of probabilistic bisimilarity to labelled Markov processes. The modal logic proposed by Larsen and Skou contained a weak form of negation and relied on minimal deviation assumption. The work of Desharnais et al. is definitely an improvement over that of Larsen and Skou because their logic does not contain any negation.They have proposed five modal logics to characterize bisimulation for labelled Markov processes. We assume that there is a fixed set of "labels" or "actions". The letters a or b are used to represent actions. Among the five modal logics, $\mathcal{L}$ is the simplest and has the following formulas as syntax:

$$\top | \phi_1 \wedge \phi_2 | \langle a \rangle_q \phi$$

where a is an action from the fixed set of actions $\mathcal{A}$ and q is a rational number. The fact that a state s satisfies a formula ϕ is represented as $s \models \phi$. The formula $\top$ is satisfied by every process. The definition of conjunction is obvious. We say a state s satisfies the formula $\langle a \rangle_q \phi$ if and only if there exists $A \in \Sigma$ such that for all $s' \in A$, we have $s' \models \phi$ and $\mu_{a,s}(A) > q$. In other words, being in state s the system can make an a-transition to a next state, that satisfies ϕ, with probability greater than q.

The four additional logics that are syntactical extensions of $\mathcal{L}$ are defined below:

$$\begin{aligned} \mathcal{L}_{\vee} &:= \mathcal{L} \mid \phi_1 \vee \phi_2 \\ \mathcal{L}_{\Delta} &:= \mathcal{L} \mid \Delta_a \\ \mathcal{L}_{\neg} &:= \mathcal{L} \mid \neg\phi \\ \mathcal{L}_{\bigwedge} &:= \mathcal{L}_{\neg} \mid \bigwedge_{i \in N} \phi_i \end{aligned}$$

Given a labelled Markov process (S, Σ, μ) we interpret the above formulas as follows:

$$\begin{aligned} s \models \phi_1 \vee \phi_2 &\quad \text{means that } s \models \phi_1 \text{ or } s \models \phi_2; \\ s \models \Delta_a &\quad \text{means that } \mu_{a,s}(S) = 0; \\ s \models \neg\phi &\quad \text{means that } s \not\models \phi; \\ s \models \bigwedge_{i \in N} \phi_i &\quad \text{means that } s \models \phi_i \text{ for all } i \in N. \end{aligned}$$

Two processes are said to be bisimilar if they satisfy the same formulas of that logic. Desharnais [5], proved that logic $\mathcal{L}$ characterizes bisimulation and therefore that negation and infinite branching are not needed for the logic to characterize bisimulation between two LMPs.

2.4.3 Metrics for Labelled Markov Processes

In this section we recall the definition of a metric for approximate bisimilarity due to Desharnais et al.[7] As we have seen in earlier sections the notion of bismulation is used to check the equivalences between the processes. One drawback of this notion is that a small difference in probabilities may result in non-bisimilar processes. To overcome this, *metric* was introduced which refined the view of processes.This metric will assign a number to every pair of processes, giving so an indication of how far they are from each other.If the metric distance is 0, then the two processes will turn out to be bisimilar, and conversely. Processes that are very "close"to being bisimilar will get smaller distance than processes that are "far"from being bisimilar. The number itself will not be of great importance, as is usually the case with metrics. It is the relative distance that will be of importance.

2.4.4 Testing Labelled Markov Processes

Van Breugel et al.[9] have characterized bisimilarity on a LMP as a testing equivalence. They have extended the Larsen and Skou's [11] result that probabilistic bisimilarity coincides with testing equivalence to the more general setting of labelled Markov processes. The idea is to specify an interaction between experimenter and a process; the way a process responds to the various kinds of tests determines a simple and intuitive behavioral semantics.

The main idea is that a process is a black box whose only interface to its environment consists of buttons (corresponding to actions). The most basic kind of test is to try and press one of the buttons: either the button will go down and the process will make an invisible state change (corresponding to a labelled transition), or the button will not go down (corresponding to the absence of a labelled transition). The set $\mathcal{T}$ of tests for bisimulation is given by the following syntax:

$$t := \omega | a.t | (t_1, ..., t_n)$$

where $a \in Act$. The test ω does nothing but successfully terminate, while $a.t$ specifies the test: press the a-button and in case of success proceed with t and in the case of bisimulation t can be in the form of $b \wedge c$ i.e., from a state there may be two possible actions possible. $(t_1, \ldots, t_n)$ specifies the test: make n copies of (the current state of) the process and perform the test t_i on the i^{th} copy for $i = 1, \ldots, n$. Finally, test $a(b \wedge c)$ specifies that in case of success with the a-button try pressing the buttons b and c. With each test t we associate a set O_t of possible observations as follows

$$O_\omega = \{\omega^\surd\} \quad O_{a.t} = \{a^\times\} \cup \{a^\surd.e | e \in O_t\} \quad O_{(t_1,\ldots,t_n)} = O_{t_1} \times \ldots \times O_{t_n}$$

The only observation of the test ω is successful termination: $\omega^\surd$. Upon performing $a.t$ one possibility, denoted by $a^\times$, is that the a-button fails to go down (and so the test terminates unsuccessfully). Otherwise, the a-button goes down and we proceed to observe e by running t in the next state; this is denoted by $a^\surd.e$.

For a given test t, each state s of a labelled Markov process $\langle S, \Sigma, \mu \rangle$ induces a probability distribution $P_{t,s}$ on O_t. The definition of $P_{t,s}$ is by structural induction on t as follows.

$$
\begin{aligned}
P_{\omega,s}(\omega^{\surd}) &= 1 \\
P_{a.t,s}(a^{\times}) &= 1-\mu_{s,a}(X) \\
P_{a.t,s}(a^{\surd}.e) &= \int (P_t,(e))d\mu_{s,a} \\
P_{(t_1\ldots t_n)},s\,(e_1,...,e_n) &= P_{t_1,s}(e_1)\times\ldots\times P_{t_n,s}(e_n)
\end{aligned}
$$

The following theorem proposed by Worrell et. al. shows how bisimilarity may be characterized using the testing framework outlined above.For each test t there is a distinguished observation,denoted $t^{\surd}$, representing complete success: no action is refused.For instance, if $t = a.(b, c)$ then the completely successful observation is $a^{\surd}(b^{\surd}, c^{\surd})$.

Theorem 1. *(Probabilistic Bisimulation) Let $\langle S, \Sigma, \mu\rangle$ be a labelled Markov process. Then $x, y \in S$ are bisimilar iff $P_{t,x}(t^{\surd}) = P_{t,y}(t^{\surd})$ for all tests t.*

The idea is that for any test t and $E \subseteq O_t$, the probability of observing E can be expressed in terms of the probabilities of making completely successful observations on all the different subsets of t using the principle of inclusion-exclusion. For example, if $t = a.(b, c)$; then the probability of observing $a^{\surd}(b^{\surd}, c^{\times})$ in state x is equal to $P_{t_1,x}(t_1^{\surd}) - P_{t,x}(t^{\surd})$ where $t_1 = a.b$.

We can do the same in the case of trace-equivalence. A *trace* is a sequence of possible actions. Test syntax for trace equivalence is same as bisimulation except that the traces can only be linear. For example we can accept the traces which are in the form of *aba* or *abcba* but not $a(b, a)$. To each trace a probability distribution on observation is associated. For example, in process F of figure 2.4, the observations related to trace *ab* have the distribution $p_{a^{\times}}$=0, $p_{a^{\surd}b^{\times}}$=0 and $p_{a^{\surd}b^{\surd}}$=1. Based on this setting, we have the straightforward result [11]:

Theorem 2. *(Probabilistic trace-equivalence) Two processes are trace-equivalent iff they yield the same probability distribution on observations from every trace.*

The fact that tests in trace-equivalence setting are "traces "(not the case for bisimulation) can therefore be compatible with the necessity of sample of execution related to the RL-methods.

Chapter 3

Reinforcement Learning

The purpose of this chapter is to set out the background to the notations of reinforcement learning systems. It therefore discusses the central issues of reinforcement learning, including trading off exploration and exploitation, Markov decision processes, and reinforcement learning techniques. In section 3.1 we start by introducing reinforcement learning and the description of the basic reinforcement-learning framework. Section 3.2 explains the trade-off between exploration and exploitation. Section 3.3 introduces Markov Decision Processes. Section 3.4 presents dynamic programming. Section 3.5 presents Monte carlo methods. Section 3.6 considers classic model-free algorithm for reinforcement learning from delayed reward: Q-learning.

3.1 Introduction

Reinforcement learning (RL) [18] is a framework for computational learning agents that use experience from their interaction with an environment to improve performance over time. Reinforcement learning systems have been studied since the early days of cybernetics and work in statistics, psychology, neuroscience, and computer science. RL has a solid theoretical foundation for its approaches, based on the theory of Markov Decision Processes(MDPs). Most RL methods rely on the computation of value functions(exact definition to follow), which evaluate the long-term performance of the agent.

There is no explicit teacher to guide a learning agent, instead the agent must learn behavior through trial-and-error interactions with an unknown environment. A Reinforcement Learning agent senses a world, takes actions in it, and receives numeric

rewards(pay offs) and punishments from some reward function based on the consequences of the actions it takes. Positive payoffs are rewards and negative payoffs are punishments. The agent must learn to choose actions so as to maximize a long term sum or average of the future payoffs it will receive. Reinforcement learning theory comprises a formal framework for describing an agent interacting with its environment, which will be described in the following sub section.

3.1.1 Reinforcement-Learning Framework

In the standard reinforcement-learning framework, an agent is connected to its environment via action, as depicted in Figure 3.1

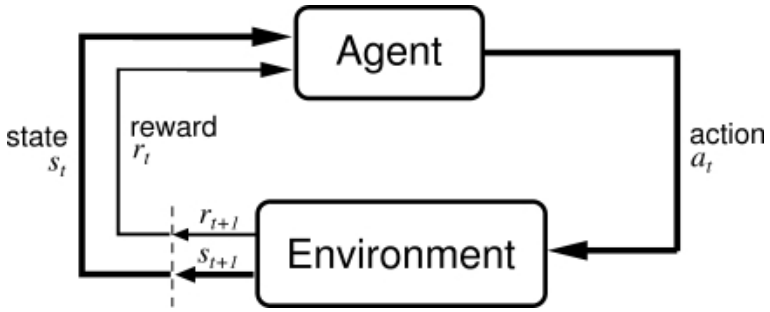


Figure 3.1: The agent-environment interaction in reinforcement learning taken from the book [18]

The agent and environment interact over a potentially infinite sequence of discrete time steps t=1, 2, 3, . . . , though the time units do not need to correspond to fixed interval of real time. Time steps can be determined by events happening within the system, such as the environment's change of state or the moment when a new action has to be taken by the agent. At each time step t, the reinforcement learning agent receives some representation of the environment's current *state*, $s_t \in S$, where S is the set of possible states, the agent then chooses an *action*, $a_t \in A(s_t)$, where $A(s_t)$ is the set of actions that can be executed in state s_t. The action changes the state of the environment, and the value of this state transition is communicated to the agent through a scalar *reward signal*, r. Thus the agent receives a *reward*, r_{t+1}, a real number, and finds itself facing a new state, $s_{t+1} \in S$. The numerical reward signal that the environment provides is the primary means for the agent to evaluate its performance. In other words its a scalar value which represents the degree to which a state or action is desirable. Reward functions can be used to specify a wide range of planning goals (e.g. by penalizing every non-goal state, an agent can be guided towards learning the fastest route to the final state). It is the means by which the designer of an RL system

can tell the agent *what* it is supposed to achieve, but not *how*. The reward function must necessarily be unalterable by the agent. This constitutes a major difference from supervised learning systems where examples of the desired response are provided by a teacher. The RL agent should choose actions that tend to increase the long-run sum of values of the reward signal. It can learn to do this over time by systematic trial and error, guided by a variety of algorithms that are the subject of later sections of this chapter.

The agent's job is to find a *policy*, the rule the agent uses to select actions, it is a function, often denoted π, that for each state assigns a probability to each possible action: for all $s \in S$ and all $a \in A(s)$, $\pi(s, a)$ is the probability that the agent executes a when in state s. We expect, in general, that the environment will be non-deterministic; that is, that taking the same action in the same state on two different occasions may result in different next states and/or different rewards. While interacting with its environment, a reinforcement learning agent adjusts its policy based on its accumulating experience to improve the amount of reward it receives over time. More specifically, it tries to maximize the *reward* it receives after each time step.

One can distinguish two main types of the RL tasks: *episodic* and *continuing tasks*. In the case of episodic tasks, the interaction with the environment is naturally divided into independent episodes. "Independent" means here that the performance in each episode depends only on the actions taken during that episode. In this thesis we will only work with episodic tasks. Here there is a terminal state, where an episode ends. The system then can be reinitialized to some starting conditions and a new episode begins. Thus at the end of an episode the learning agent is guaranteed that there will be no further delayed effects of its previous actions. Continuing tasks or infinite horizon problems, on the other hand, consist of just one infinite sequence of state changes , actions and rewards.

In general, a return represents a cumulative function of the reward sequence. For an episodic task, for example, it is the sum of all rewards received from the beginning of an episode until its end: $R(s_0) = \sum_{k=0}^{T} r_{k+1}$, where T is the terminal state where episode ends and s_0 is the starting state. In the case of continuing tasks, there are many problems where one would value rewards obtained in the near future more than those received later. In this case future rewards are discounted by a factor γ and the return is defined as:

$$\sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k} \tag{3.1}$$

where $\gamma \in [0, 1)$ is the *discount factor* and t is the time step. Thus it is not *immediate* reward that is to be maximized, but the agent must take into account the delayed conse-

quences of its actions. Rewards or punishments do not necessarily have to immediately follow the actions that have caused them, they may be received several steps later. This is referred to as learning with *Delayed reinforcement*([21], [24]). The discount factor γ adjusts the relative importance of the long-term consequences of actions versus the short term ones; for $0 < \gamma < 1$ reinforcement values distant in time are weighted less than those received soon. It corresponds to the common sense idea that punishments are less deterrent and rewards are less attractive if they are received in remote future. Different γ's may yield different optimal policies. In particular, if $\gamma = 0$, the agent is only concerned with maximizing immediate rewards: its objective would be to learn how to act at each time step t so as to maximize only r_{t+1}. But in general, acting to maximize immediate reward can reduce access to future rewards so that a longer-term return may actually be reduced. As γ approaches one, the agent takes future rewards into account more strongly: the agent becomes more far-sighted.

3.2 Exploration-Exploitation dilemma

The exploration-exploitation dilemma, which is an important problem frequently encountered in reinforcement learning, is defined as follows. When an agent is faced with a state of the environment, it has to choose between two options: (i) exploration and (ii) exploitation. The agent can choose to (i) explore its environment and try new actions in search for better ones to be adopted in the future , or (ii) exploit already tested actions and adopt them. When opting to explore new actions, the agent is considering its long term performance whereas when opting to exploit tested actions, the agent is guaranteeing its short term performance. In order to highlight the problems of exploration, we treat a very simple case in this section.

The simplest possible reinforcement learning problem is known as *n-armed bandit problem*. The agent is in a room with a collection of k gambling machines(each called a "one-armed bandit" in colloquial English). The agent is permitted a fixed number of pulls, h. Any arm may be pulled on each turn. The machines do not require a deposit to play; the only cost is in wasting a pull playing a suboptimal machine. When arm i is pulled, machine i pays off a numerical reward, 1 or 0, according to some underlying probability parameter p_i, where payoffs are independent events and the p_i's are unknown. What should the agents strategy be? This problem illustrates the fundamental tradeoff between exploration and exploitation. The agent might believe that a particular arm has a fairly high payoff probability; should it choose that arm all the time, or should it choose another one that it has less information about, but seems to be worse? Of course, answers to these questions depend on the estimated life span of

the agent. The shorter its life span, the more inclined is the agent to improve its short term performances; subsequently, the more inclined is the agent to limit exploration.

Formally, the agent has to resolve two subproblems. The first subproblem consists of choosing an exploration method. The exploration can be either directed or undirected. The exploration is directed when the choice is based on the acquired knowledge whereas it is undirected when the choice is random. The second subproblem consists of identifying a method that switches the agents mode between exploration and exploitation according to the state of the agent and the state of its environment. The two subproblems are important since they influence the learning speed, the performance and the actions of an agent. This influence is more critical when the agents environment is dynamic -which is the case of many reinforcement learning systems. Balancing exploration with the exploitation of current knowledge is a subtle problem that has been extensively studied. In principle it is possible to optimally balance exploration and exploitation by solving an MDP, which will be introduced in the section 3.3, whose states summarize the agent's entire history of observations and actions.

Several simple heuristic exploration methods are usually used in applications of reinforcement learning. In the simplest explore/exploit strategy is where, the agent selects $\epsilon - greedy$ actions. This means that with probability $1 - \epsilon$, the agent exploits its current knowledge by selecting a greedy action, that is, an action that is optimal given its current value estimates, and with probability ϵ, it selects an action at random, uniformly independently of its current value estimates. It is very easy to implement and enable the system to learn continually. Although $\epsilon - greedy$ action selection is an effective means of balancing exploration and exploitation in reinforcement learning its drawback is that when it explores it chooses equally among all actions. This means that it is likely to choose the worst-appearing action as it is to choose the next-to-best action. However, in *stationary* problems (i. e. problem in which its parameters do not change in time) the continual exploration may lead to suboptimal results, even if the optimal solution has already been learned.

The most obvious solution is the *soft max* action selection method. It chooses action a from possible actions at time t with probability.

$$\frac{e^{Q_{t(a)}/\tau}}{\sum_{b=1}^{n} e^{Q_{t(b)}/\tau}} \tag{3.2}$$

where τ is a positive parameter called the *temperature*. and $Q_{t(a)}$ is the value of the state-action pair. Parameter τ controls the amount of exploration (the probability of

executing actions other than the one with the highest Q-value). Q is the value of the state-action pair (exact definition to follow). If τ is high, or if Q-values are all the same, then a random action will be picked. If τ is low and Q-values are different, it will tend to pick the action with the highest Q-value. In the limit $\tau \to 0$, the softmax selection becomes the $\epsilon - greedy$ action selection. One problem of the softmax action selection mechanism is to determine the value of τ, which depends on the task.

3.3 Markov Decision Processes

A great part of the work done on reinforcement learning, in particular that on convergence proofs, assumes that the interaction between the agent and the environment can be modeled as a discrete-time finite Markov decision process(MDP)(state and actions spaces are finite). In the general case of the reinforcement learning problem, the agent's action determine not only its immediate reward, but also (at least probabilistically) the next state of the environment. Such environments can be thought of as networks of bandit problems, but the agent must take into account the next state as well as the immediate reward when it decides which actions to take. The model of long-run optimality the agent is using determines exactly how it should take the value of the future into account. The agent must learn from delayed reinforcement: it may take a long sequence of actions, receiving insignificant reinforcement, then finally arrive at a state with high reinforcement. The agent must be able to learn which of its actions are desirable based on reward that can take place far in future. Problems with delayed reinforcement are well modeled as *Markov decision processes*.

3.3.1 Basic Definitions

Markov Decision Processes(MDPs) are a standard, general formalism for modeling stochastic, sequential decision problems [17]. A Markov decision process consists of a tuple $< S, A, P, R >$. We describe these in detail below.

- S is a set of states. Each state must satisfy the *Markov property*: which means that environment's state at any time step $t > 0$ provides the same information about what will happen next as would the entire history of the process up to step t. Unless stated otherwise, we assume that states are finite set, and the agent is able to determine the state it is in.

- A is a set of actions available to the agent. We will assume that this set is finite, and that all actions are available to the agent at each state.
- $P^a_{ss'} = Pr\left\{s_{t+1} = s^{'} | s_t = s, a_t = a\right\}$ is a state transition function that defines the probability of transitioning to state $s^{'}$ at time $t+1$ after action a is taken when agent is in state s at time t. Because of the Markov property, the probability of a transition to state $s^{'}$ only depends on the prior state and action taken.
- $R^a_{ss'}$ is a reward function that determines the probability of receiving reward after choosing action a in state s and going to the next state $s^{'}$.

We will use the symbols s_t, a_t, and r_t to denote the state, action and actual reward at time step t.

3.3.2 Markov Property

From the above definitions it follows that both state transitions and rewards in an MDP depend solely on the current state and current action. It is important to emphasize that there is no dependence on previous state, action, and reinforcements. This is referred to as the *Markov property*, and is crucial for all reinforcement learning algorithms we are going to consider.

Formally the Markov Property holds if,

$$Pr\left\{s_{t+1} = s^{'} | s_t, a_t\right\} = Pr\left\{s_{t+1} = s^{'} | s_t, a_t, s_{t-1}, a_{t-1}, ...s_0, a_0\right\} \tag{3.3}$$

holds. That is to say that the probability distribution over states entered at $t+1$ is conditionally independent of the events prior to (s_t, a_t) - the next state and immediate reward depend only on the current state and action. In reinforcement learning , we also assume the same with the reward function,

$$Pr\left\{r_{t+1} = r | s_t, a_t\right\} = Pr\left\{r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}...s_0, a_0, r_1\right\} \tag{3.4}$$

The Markov property is a simplifying assumption which makes it possible to reason about optimality and proofs in a more straightforward way. For more detailed account of MDPs refer to Puterman's book [17]. In the following sections , the terms *process* and *environment* will be used interchangeably under the assumption that the agent's environment can be exactly modeled as a discrete finite Markov process.

3.3.3 Policies and value Functions

Conventionally, an agent starts off not knowing the state-transition probabilities or reward distributions. The goal of the agent is to find a *policy*, that maximizes the cumulative reward over time. A *policy* in an MDP is a function $\pi : S \rightarrow A$ that with each state $s \in S$ associates an action $a \in A$ to be performed in that state. To follow a policy π means to execute in each state s the action $\pi(s)$. A policy may be deterministic, which will always choose the same action in a state, or it may specify a distribution over actions, In later case π will be of the form $\pi(s, a) = Pr(s = s_t | a = a_t)$, which denotes the probability that the agent takes action a when the environment is in state s. Once we have established a policy, we can ask how much *return* this policy generates from any given state in the process.

The goal of an RL algorithm is either to evaluate performance of a given policy(prediction problem) or to find an optimal policy(control problem). The most commonly studied reinforcement learning algorithms are based on estimating *value functions*, which are scalar functions of states, or of state-action pairs, that tell how good it is for the agent to be in a state, or to take an action in a state. The notion of "how good" is the return expected to accumulate over the future, which is well-defined if the Markov property holds and the agent's policy is specified.

If the agent uses policy π, then the state value function V^π gives the *value*, $V^\pi(s)$, of each $s \in S$, which is the return expected to accumulate over the time period after visiting s , assuming that actions are chosen according to π. For the discounted infinite horizon problem, the value of state s is

$$V^\pi(s) = E_\pi \{R_t | s_t = s\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s \right\} \tag{3.5}$$

where E_π is the expected value given that policy π is followed.

Similarly, the *action value* of taking action a in state s under policy π, denoted $Q^\pi(s, a)$, is the expected return starting from s, taking the action a, and thereafter following policy π :

$$Q^\pi(s, a) = E_\pi \{R_t | s_t = s, a_t = a\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a \right\}. \tag{3.6}$$

There is one fundamental property of value functions that makes them valuable for solving RL tasks. The state-value function satisfies a recursive equation that, in the discounted case, has the following form:

$$V^{\pi}(s) = E_{\pi}\left\{R_t | s_t = s\right\} = \sum_{a} \pi\left(s,a\right) \sum_{s'} P^{a}_{s,s'} \left[R^{a}_{s,s'} + \gamma V^{\pi}\left(s'\right)\right] \tag{3.7}$$

where $s_{t+1} = s'$. This is the Bellman equation and represents the relationship between the value of a state and the value of its successors. This system of equations has a unique solution , which is state-value function for policy π. A similar equation is satisfied by the action-value function.

A policy π is considered to be better than or equal to another policy π' if its expected return is greater than or equal to the one of π'. In other words $\pi \geq \pi'$ if and only if $V^{\pi}\left(s\right) \geq V^{\pi'}\left(s\right)$, $\forall s \in S$. The *optimal policy* is the policy which is better than or equal to all other policies [18]. which is denoted by π^*. The *optimal state-value function*, $V^*(s)$, is the return expected after visiting state s assuming that actions are chosen optimally, i.e., it is the largest expected return possible after visiting state s. The *optimal action value* of taking action a in state s, denoted $Q^*(s,a)$, is the largest expected return starting from state s, taking the action a, and thereafter following an optimal policy. The optimal state-value function satisfies the Bellman optimality equation:

$$V^{*}(s) = \max_{a \in A(s)} Q^{\pi^{*}(s,a)} = \max_{a \in A(s)} \sum_{s'} P^{a}_{s,s'} \left[R^{a}_{s,s'} + \gamma V^{*}\left(s'\right)\right]. \tag{3.8}$$

Value functions are useful because of several properties of MDPs. If V^* is known, optimal policies can be found by looking ahead only one time step. That is, if s_t is the state at step t, then an optimal action is any $a \in A(s_t)$that maximizes the expected value of $r_{t+1} + \gamma V^*(s_{t+1})$. Thus given V^* and an accurate model of the immediate effects on the environment of all the actions, acting optimally does not require deep lookahead because V^* summarizes the effects of future behavior. If Q^* is known, then finding optimal actions is even easier. An optimal action at step t is any action that maximizes $Q^*(s_t, a)$. In this case, it is not necessary to look ahead one step, so that no model is needed of the effect of actions on the environment. This what makes reinforcement learning algorithms that use action-value functions a popular choice in many applications. Any such one-step ahead maximizing action for a state value function, or a maximizing action for an action-value function, is called a *greedy* action with respect to that function.

3.3.4 Practical Issues

In real tasks it may be sometimes difficult to comply with all the requirements imposed by the definition of an MDP. The most common cases are that:

- the state space of a task is infinite.
- the Markov property is not preserved.

The first problem is usually dealt with by artificially transforming the state space to a finite, discretized form, through quantizing appropriately continuous state variables. Unfortunately, such a transformation usually results in violating the Markov property, since new discretized states do not contain enough information.

As far as the Markov property is concerned , it is useful to distinguish two kinds of its violation. A *strong* violation is when there exists no optimal policy for a task: the state representation does not contain enough information to make optimal decisions. A *slight* violation is when , despite the fact that the Markov property is not held true, for a particular initial state of the task there exists an optimal, though it may be difficult or impossible to identify it for the agent solving the task. At first it may not be clear how it is possible. Basically, any violation of the Markov property means that, due to incomplete state information, the same state may have different consequences(i. e successor state and reinforcement values)each time it is visited, though always the same action is performed. In the case of a slight violation, when some arbitrary policy is followed, each visit to a state may yield different consequences. Nevertheless, for some fixed initial state or a set of initial states, and some fixed (e. g. optimal)policy , it is possible that each visit to a state yields (probabilistically)exactly the same consequences, provided that the same action is executed.

Before we consider algorithms for learning to behave in MDP environments, we will explore methods for determining the optimal policy. which will be discussed in the following sections.

In chapter 4 we will have to restrict ourself to tree-like Labelled Markov Processes in order not to violate the Markov property.

3.4 Dynamic Programming

Dynamic programming(DP) methods can be used to solve a Markov Decision Process,i.e., to find an optimal policy, if the full knowledge of the model is available. In particular, all transition probabilities and reward expectations must be known. Rather than solving the Bellman equations directly, DP methods treat them as recursive update rules. DP algorithms are *bootstrapping* they update the estimates of state values based on the estimates of the values for the successor states.

There are two basic DP methods used for computing an optimal policy in an MDP: *policy iteration* and *value iteration.* we will present a brief overview of these methods.

Policy Iteration

Starting from some arbitrary initial policy π_0 , policy iteration forms a sequence of policies $\pi_0, \pi_1, ...\pi_k, ...$,where each π_{k+1} is an improvement over π_k. To achieve this, the value function V^{π_k} is computed for π_k (the policy evaluation phase) and then π_{k+1} is taken to be greedy with respect to V^{π_k} (the policy improvement phase). Either π_{k+1} is an improvement over π_k or they are both optimal policies.

Policy evaluation task is concerned with computing the state-value function V^π for an arbitrary policy π. The agent starts with some arbitrary initial approximation of the state-value function, V_0^π and uses the Bellman equation for the state-value function as a recursive update rule to improve the approximation:

$$\begin{aligned} V_{k+1}(s) &= E_\pi \left\{ r_{t+1} + \gamma V_k(s_{t+1}) | s_t = s \right\} \\ &= \sum_a \pi(s,a) \sum_{s'} P^a_{s,s'} \left[R^a_{s,s'} + \gamma V_k\left(s'\right) \right] \end{aligned} \tag{3.9}$$

This is an iterative algorithm for policy evaluation, where an iteration consists of the updates being made to all states. Clearly, $V_k = V^\pi$ is a fixed point for this update rule because the Bellman equation for V^π assures us of equality in this case. Policy evaluation can be shown to converge in the limit to the correct V^π under the same conditions that guarantee the existence of V^π.

Estimating value functions is particularly useful for finding better policies. The Policy improvement algorithm uses the action-value function to improve the current policy. If $Q^{\pi}(s,a) \geq V^{\pi}(s)$ for some $a \neq \pi(s)$, then it is better to select the action a at the state s than to select $\pi(s)$. The Policy improvement algorithm states that for any pair of deterministic policies π and $\pi^{'}$, such that $\forall s \in S$, $Q^{\pi}(s,a) \geq V^{\pi}(s)$, policy $\pi^{'}$ must be as good as or better than π. Policy improvement thus must give us a strictly better policy except when the original policy is already optimal. In this manner we can construct a new improved policy $\pi^{'}$, which is greedy with respect to V^{π}:

$$\begin{aligned} \pi^{'}(s) &= argmax_{a \in A(s)} Q^{\pi}(s,a) \\ &= argmax_{a \in A(s)} \sum_{s^{'}} P^{a}_{s,s^{'}} \left[R^{a}_{s,s^{'}} + \gamma V_{\pi}\left(s^{'}\right) \right] \end{aligned} \tag{3.10}$$

Policy evaluation and Policy improvement can be interleaved to construct a sequence of successively improving policies. This algorithm is known as Policy iteration, which finds an optimal policy by systematically improving an initial policy until no changes are made. Once a policy π has been improved using V^{π} to yield a better policy $\pi^{'}$, we can then compute $V^{\pi^{'}}$ and improve it again to yield an even better $\pi^{''}$. We can thus obtain a sequence of improving policies and value functions:

$$\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V^{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} ... \xrightarrow{I} \pi^{*} \xrightarrow{E} V^{*},$$

where $\xrightarrow{E}$ denotes a policy *evaluation* and $\xrightarrow{I}$ denotes a policy *improvement*. Policy iteration converges to the optimal policy , because there is a finite number of policies in a finite MDP and each new policy is better than the previous one. A complete policy iteration algorithm is given in Figure 1:

Value Iteration

In the value iteration method an optimal policy is not computed directly. The optimal value function is computed instead, and then a greedy policy with respect to that function is an optimal policy. The algorithm is given below:

Contrary to policy iteration, value iteration does not converge in a finite number of steps. It produces successive approximations of the optimal value function, more

Algorithm 1 Policy iteration(using iterative policy evaluation) for V^*

Initialization
$V(s)$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$
Policy Evaluation
repeat
  $\Delta \leftarrow 0$
  For each $s \in S$:
  $v \leftarrow V(s)$
  $V(s) \leftarrow \sum_{s'} P_{ss'}^{\pi(s)} \left[R_{ss'}^{\pi(s)} + \gamma V\left(s'\right) \right]$
  $\Delta \leftarrow \max\left(\Delta, |v - V(s)|\right)$
until $\Delta < \theta$(a small positive number)
Policy Improvement
$policy - stable \leftarrow true$
For each $s \in S$:
$b \leftarrow \pi(s)$
$\pi(S) \leftarrow argmax_a \sum_{s'} P_{ss'}^{\pi(s)} \left[R_{ss'}^{\pi(s)} + \gamma V\left(s'\right) \right]$
if $b \neq \pi(s)$, **then**
  $policy - stable \leftarrow false$
end if
if policy-stable, **then**
  stop;
else
  go to Policy Evaluation
end if

Algorithm 2 Value iteration

initialize V arbitrarily, e. g. , $V(s) = 0$, for all $s \in S^+$
repeat
  $\Delta \leftarrow 0$
  For each $s \in S$:
  $v \leftarrow V(s)$
  $V(s) \leftarrow max_a \sum_{s'} P_{ss'}^{a} \left[R_{ss'}^{a} + \gamma V\left(s'\right) \right]$
  $\Delta \leftarrow \max\left(\Delta, |v - V(s)|\right)$
until $\Delta < \theta$(a small positive number)
Output a deterministic policy, π, such that
$\pi(s) = argmax_a \sum_{s'} P_{ss'}^{a} \left[R_{ss'}^{a} + \gamma V\left(s'\right) \right]$

and more accurate. It is stopped when the change introduced by backups becomes sufficiently small, i.e. less than some fixed θ.

Of both kinds of DP methods, value iteration is presently better understood and more widely used. The fact that the practical implementation of policy iteration involves a kind of value iteration phase may also play some role.

3.5 Monte Carlo Methods

DP methods can only be used when a model of the system(transition probabilities and expected rewards)is available. Of course, an agent can learn a model and then use it in DP methods. However, learning a value function directly from interaction with the environment can be more efficient. Monte Carlo(MC) methods estimate value functions directly based on the experience of the agent. By experience we mean sample sequences of states, actions and rewards from an on-line or simulated interaction with the environment. MC methods estimate the state or action values based on averaging sample returns observed during the interaction of the agent with its environment. Since samples of complete returns can be obtained only for finite tasks, MC methods are defined for episodic tasks. For each state(or state-action pair)a sample return is the sum of the rewards received starting from the occurrence of the state(or state-action pair)and until the end of an episode. As more samples are observed, their average convergence to the true expected value of the return under the policy used by the agent for generating the sample sequences.

One can design a policy iteration algorithm, where the policy evaluation step estimates the value function using MC methods. There is one complication, however, that did not arise in DP. If the agent adopts a deterministic policy π, then the experience generated by its interaction with the environment contains samples only for actions suggested by policy π. The values for other actions will not be estimated and there will be no information on which to base the policy improvement step. Therefore maintaining sufficient exploration is key for the success of policy iteration using MC methods. One solution is for the agent to adopt a stochastic policy with non-zero probabilities of selecting all actions from all states: a *soft stochastic policy*, such that $\pi(s,a) > 0, \forall s \in S, \forall a \in A(s)$. There are different ways to implement this approach.

In the case of on-policy methods, the agent uses a soft stochastic policy when it interacts with the environment to generate experience, and evaluates its performance under this policy. Another approach is off-policy learning: the agent uses one policy to

interact with the environment and generate experience(behavior policy), but estimates the value function for a different policy(estimation policy). In particular, an agent can try to learn the value of the optimal policy while following an arbitrary stochastic policy.

Policy iteration with MC-based policy evaluation converges in the limit to the optimal policy(both for the on-policy and off-policy learning) as long as every state-action pair is visited infinitely often. But in practice we encounter the same problem as for DP-methods-one can not wait forever until the policy evaluation step converges. When stopping after some finite number of observations , we are dealing with an approximate version of the algorithm.

3.6 Temporal Difference Learning

A combination of the ideas from DP and MC methods yields temporal difference(TD) learning. Similarly to the MC method, this approach allows learning directly from the on-line simulated experience without any prior knowledge of the system's model. The feature shared by TD and DP methods is that they both use bootstrapping for estimating the value functions. The fact that the agent need not know the dynamics of the environment is crucial in any non-trivial environment(since all models of the real world are imperfect) and is the soul of 'learning from experience'. In TD methods, learning takes place after every 'time step'(i.e. every action), which is beneficial as it makes for efficient learning-the agent can revise its policy after every action and state it experiences.

TD algorithms make updates of the estimated values based on each observed state transition and on the immediate reward received from the environment on this transition. The most basic TD algorithm called *tabular* TD(0), estimates V^{π} while the agent is behaving according to π and is applicable when the state set is small enough to store the state values in a lookup table with a separate entry for the value of each state. Suppose the agent is in state s, executes action a, and then observes the resulting reward r and the next state $s^{'}$. TD(0) updates the current estimate of the value of state s, V(s), using the following update:

$$V(s) \leftarrow (1-\alpha)\, V(0) + \alpha \left(r + \gamma V(s^{'} \right) \tag{3.11}$$

where α is positive step-size parameter. TD algorithms are based on the consistency condition expressed by the bellman equations. This TD algorithm is designed to move the term $r+\gamma V(s^{'})-V(s)$, called the TD *error*, toward zero for every state. An update of this general form is often called a *backup* because the value of a state is moved toward the current value of a successor state, plus any reward that is received on the transition.

TD(0) method bootstraps, but it uses sample updates instead of full updates as in the case of DP. Only one successor state observed during the interaction with the environment, is used to update V instead of using values of all the possible successors and weighting them accordingly to their probabilities. For any fixed policy π, the one-step TD algorithm converges in the limit to V^{π} [20]. The one -step TD method can be used for the policy evaluation step of the policy iteration algorithm. As with MC methods, sufficient exploration in the generated experience must be ensured in order to find the optimal policy. Again, one can use either on-policy or off-policy approaches to ensure an adequate exploration.

A very popular representation of the off-policy approach is the *Q-Learning* Algorithm, was proposed by Watkins in 1989. This approach has been used in this thesis to conduct the experiments described in chapter 5. This algorithm directly estimates Q^{*} without relying on the policy improvement property. It works as follows. Suppose the agent is in state s, executes action a, and then observes the resulting reward r and the next state $s^{'}$. The Q-Learning algorithm updates the action value estimate $Q(s,a)$, of the pair (s, a) using the following backup:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a^{'}} Q(s^{'},a^{'}) - Q(s,a) \right] \tag{3.12}$$

Where α is a positive step-size parameter. The Q-learning algorithm is shown in procedural form here 3. If α decreases appropriately with time and each state-action pair would be visited infinitely often in the limit, then this algorithm converges to $Q^{*}(s,a)$ for all $s \in S$ and $a \in A(s)$ with probability one. Unless it is known that the environment is deterministic, the "infinitely often"requirement is necessary for this kind of strong convergence of any method that is based , as this one is, on sampling environment state transitions and rewards. Letting the agent sometimes select actions randomly from a uniform distribution is one simple way to help the agent maintain enough variety in its behavior in order to try to satisfy this condition. Otherwise, the agent executes actions that are greedy with respect to its current estimates of Q^{*}. Q-learning is known as an *off-policy* method, since any suitable policy (e. g. ϵ -greedy, softmax)may be used for training purposes. Closely related to Q-Learning is the *Sarsa*

Algorithm 3 Q-learning: An off-policy TD control algorithm.

initialize Q(s, a) arbitrarily
Repeat (for each episode):
initialize s
Repeat (for each step of episode):
Choose a from s using policy derived from Q(e. g. , ϵ-greedy)
Take action a , observe r, $s^{'}$
$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s^{'}, a^{'}) - Q(s,a) \right]$
$s \leftarrow s^{'}$
Until s is terminal

algorithm. Suppose the agent is in state s, executes action a, observes reward r and the next state $s^{'}$, and then executes action $a^{'}$. then the Sarsa update is

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q(s^{'}, a^{'}) - Q(s,a) \right] \tag{3.13}$$

which is the same as the Q-learning update 3.12 except that instead of taking the maximum over the actions available in $s^{'}$, it uses the action, $a^{'}$, which was actually executed . (this requirement of s, a, r, $s^{'}$, $a^{'}$ is what accounts for the algorithm's name.)Notice that if actions are always greedy with respect to the current estimate of Q^*, then Sarsa is the same as Q-learning. Despite this similarity, Sarsa and Q-learning have somewhat different properties(see [18]). Q-learning converges to Q^* independently of the agent's behavior(as long as the conditions for convergence are satisfied), whereas Sarsa converges to an action-value function that is optimal given the agent's mode of exploration.

TD algorithms are closely related to dynamic programming algorithms, which also use back up operations derived from Bellman equations. There are two main differences. First, a dynamic programming backup computes the expected value of successor states using the state-transition distribution of the MDP, whereas a TD backup uses a sample from this distribution. (TD backups are sometimes called *sample backups*, in contrast to the *full backups* of dynamic programming.)A second difference is that dynamic programming uses multiple exhaustive "sweeps" of the MDP's state set, whereas TD algorithms operate on states as they occur in actual or simulated experiences. These differences make it possible to use TD algorithms on problems for which it is not feasible to use dynamic programming.

Chapter 4

Trace Equivalence between LMPs

In this chapter we will present our model for checking trace equivalence between labelled Markov processes using reinforcement learning algorithms. In the literature available to the date, a lot of research has already been done on equivalence notions between labelled Markov processes, but in this thesis we are going to analyse the problem of divergence between the LMPs from a different perspective by combining which I consider the two most interesting disciplines of computer science (i.e., system verification and reinforcement learning).

Our main aim in this thesis is to answer the question: is the implemented system behaves according to its specifications? To answer this question we consider two LMPs, one representing the system specifications and another representing the implemented system. Then with the help of reinforcement learning algorithms we verify if these two labelled Markov processes(LMPs) are trace equivalent. More precisely , in this thesis we consider probabilistic trace equivalence(details to follow). If the algorithm shows that they are trace equivalent the answer to the above question is yes and vice versa.

We assume that we have the complete model of only one LMP i.e., the one representing system specifications(LMPSpec for short). The second LMP(LMPImpl) is available in the form of a black box to which we feed actions as input and the output behavior will be observed. This represents the typical scenario in physical system verification against a set of specifications. Unlike in the non-probabilistic trace equivalence notion, where it is sufficient that two systems accept the same set of traces, probabilistic trace equivalence requires not only that both systems accept same set of traces but also with the same probability. Using our model we not only verify if two LMPs are trace equivalent but also find the degree of divergence between them, as it is very important in the case of probabilistic systems where a slight difference in the probabilities may cause

the two systems to be non-trace equivalent as we can see in the following example:

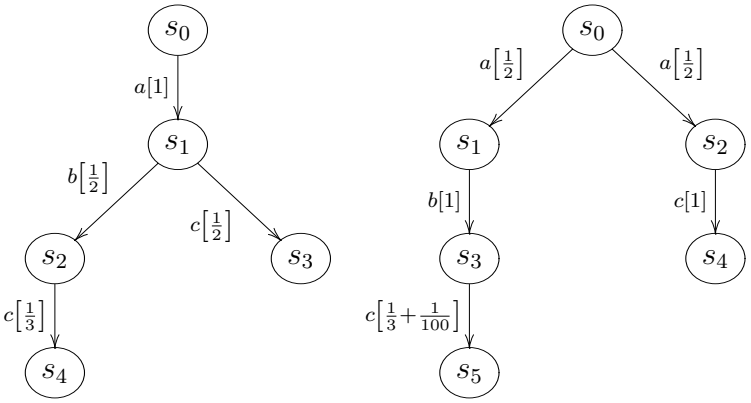


Figure 4.1: Trace equivalent but not probabilistic trace equivalent LMPs

The two LMPs in the above figure accept the same set of traces they are: $\{\epsilon, a, ab, ac, abc\}$ which means that they are trace equivalent. Except for the trace abc, they both accept rest of the traces with equal probabilities. The trace abc is accepted in the first LMP with probability $\frac{1}{6}$ and in the second LMP with probability $\frac{1}{3} + \frac{1}{100}$. Thus the LMPs in Figure 4.1 are not probabilistic trace equivalent. This difference is caused by a divergence in a single probability value i.e., probability value of action c which is $\frac{1}{3}$ in the first LMP and $\frac{1}{3} + \frac{1}{100}$ in the second. This is the reason why we considered to find the degree of divergence between the LMPs instead of simply checking if they are trace equivalent. For simplicity sake, probabilistic trace equivalence will be called trace equivalence for the reminder of this thesis.

Here we adopt the approach of testing. A test specifies an algorithm for an observer who shall experiment on these two processes. A test is nothing but a series of actions. A test is executed on both processes and then the resulting observations will be analyzed. In [11], Larsen and skou have shown that:

> ≪To be trace equivalent two processes must have exactly same probability distribution on the observation set of any test. ≫

If an action runs successfully in the LMPSpec while fails in LMPImpl shows the possible difference between them. Since our goal is to show the difference between the two LMPs under consideration, we encourage such situations in our RL approach by giving high positive reward. On the other hand, the presence of probabilities introduce entropy in LMPSpec which makes the job more challenging. Entropy means the uncertainty of an outcome, in other words the same process may accept an action in one run while it refuses it in another. In our model we give negative reward to such situations.

To recognize such entropy we consider a third process in addition to LMPSpec and LMPImpl called LMPClone, which is simply a clone of LMPSpec, available in the form of a black box just like LMPImpl.

Hence, the following three processes are available at our disposal:

- A complete model of the process representing system specifications, which we call *LMPSpec*.
- A second process representing implemented system in the form of a black box, which we call *LMPImpl*.
- A third process which is a clone of the LMPSpec, called by us *LMPClone*.

In our model we restrict ourselves to *trees*, i.e., we represent all the above three processes in the form of *trees*.This has been stated briefly in the preceding chapter. Without the tree-like representation we would loose the Markov property of the MDP that will be constructed from LMPSpec,LMPImpl and LMPClone. However,there are very simple unfolding theories that allow to represent any countable LMP as a countable tree-like LMP. Since we are working with a discount factor $\gamma < 1$, RL methods can be extended to the countable setting. As discussed earlier we have only the complete model of LMPSpec but we have absolutely no prior knowledge about LMPImpl. As a matter of fact , as seen in chapter 3, RL approach does not demand the model of the system only the complete observation of its current state is needed in that case the RL agent learns simply by interacting with the system by trial and error method.

States of the LMP will be represented as the nodes of the tree, interaction with its environment will be represented as labels given to the edges of the tree, and transitions will be the arrows directing from one node to the next according to the actions taken by its environment. To make things more clear we shall consider the following figure, in which a process is represented in the form of a tree. This process has four states as there are four nodes in the tree. There are three actions possible which we represented with letters a, b, c. The value next to the action in the Figure **??** represents the probability with which an action is accepted from a state.

RL is generally used to solve the so-called Markov decision problems(MDPs). In other words, the problem that we are attempting to solve with RL must be an MDP or its variant. Before proceeding any further we will model our task of finding the divergence between the LMPs as the value of the optimal policy of some particular MDP, which is a essential element in RL approach.

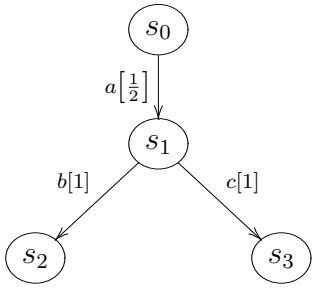


Figure 4.2: Tree representation of the LMP

This chapter is organized in the following manner, in section 4.1 we give two different approaches we had tried to solve our problem. Each approach gives rise to a specific construction of a MDP when given LMPSpec, LMPImpl and LMPClone. The first approach is the one that we had tried in this master research in the beginning , unfortunately we will show that this approach do not give rise to a notion of divergence between LMPs. The second approach will give the desired notion of divergence. In section 4.2 we explain the implementation of our model and finally in the last section we provide the empirical results.

4.1 Different versions of MDPs

4.1.1 The first approach : MDP's construction of type 1

The framework of the MDP has the following elements: set of states(S), set of actions(Act), state-transition probability distribution ($Pr^a_{ss'}$), and the reward function($R^a_{ss'}$).

Suppose, let's say we want to find the divergence between LMPImpl and LMPSpec in Figure 4.3

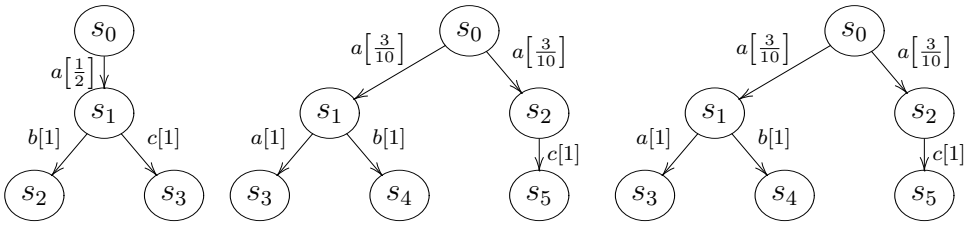


Figure 4.3: LMPImpl, LMPSpec and LMPClone

Then the MDP's construction of type 1 constructed from LMPSpec in Figure 4.3 can be represented as follows:

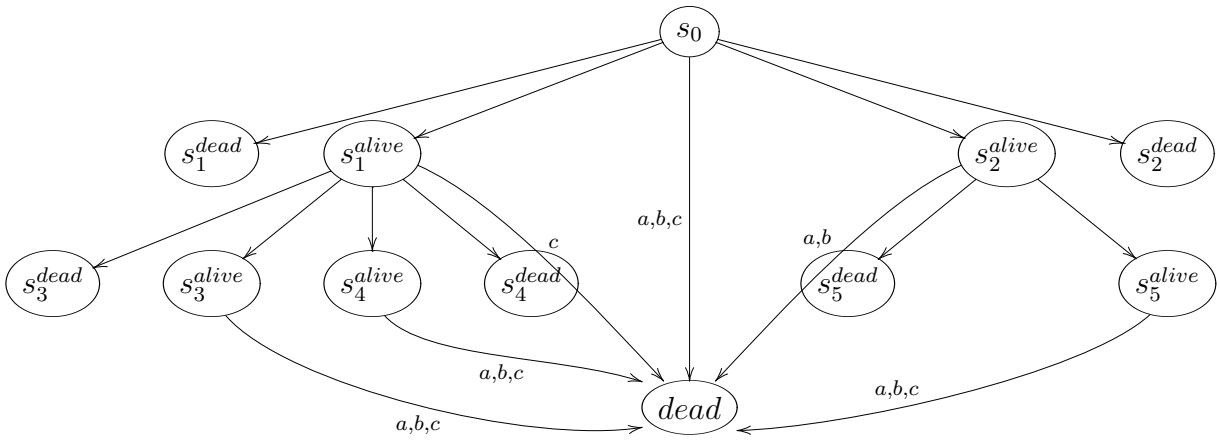


Figure 4.4: First approach: MDP construction of type 1

The four elements of MDP have been defined in the following manner:

- **States(S)**: Except the root node every state of LMPSpec has been categorized into *alive* and *dead*. There is also an additional state called *dead*.
- **Actions(Act)**: LMPSpec actions.
- **Transition probability distribution**:

$$Pr^a_{ss_1^{alive}} = Pr^{Spec}(s, a, s_1).P^{Impl}(a^{\surd}|tr(s)).P^{Clone}(a^{\surd}|tr(s)).$$

$$Pr^a_{ss_1^{dead}} = Pr^{Spec}(s, a, s_1).(1 - (P^{Impl}(a^{\surd}|tr(s)).P^{Clone}(a^{\surd}|tr(s)))).$$

$$Pr^a_{sdead} = 1 - Pr^{Spec}(s, a, S).$$

- **Reward function:**

$$R^a_{ss_1^{alive}} = \quad 0$$

$$R^a_{ss_1^{dead}} = \frac{R^a_{s,S} - Pr^a_{sdead}.R^a_{sdead} - Pr^a_{s,s_1^{alive}}.R^a_{ss_1^{alive}}}{Pr^a_{s,s_1^{dead}}}$$

$$R^a_{s,dead} = P^{Impl}(a^{\surd}|tr(s)).P^{Clone}(a^{\times}|tr(s)) - P^{Impl}(a^{\times}|tr(s)).P^{Clone}(a^{\surd}|tr(s))$$

Where

- $tr(s)$ is the unique sequence of actions connecting s_0 to the state of LMPSpec that corresponds to s in the LMPSpec. If s is the root node then this sequence will be zero.*
- $Pr^{Spec}(s, a, s_1)$ is the probability distribution of going from state s to the next state s_1 with action a in LMPSpec.
- $P^{Impl}(a^{\surd}|tr(s))$ means the probability of observing a success in LMPImpl when asking for action a if the sequence of actions $tr(s)$ has been successfully performed in LMPImpl.
- $P^{clone}(a^{\surd}|tr(s))$ is similarly defined.

Everytime we reach either of the dead states the execution of the test terminates and the reward is calculated according to Table 4.1:

LMPImpl	LMPSpec	LMPClone	Reward
S	S	S	0
F	S	S	+1
S	F	S	0
S	S	F	-1
F	F	S	-1
F	S	F	0
S	F	F	+1
F	F	F	0

Table 4.1: Reward function look-up table

where S indicates that an action has been successful in a process and on the other hand F indicates the failure. For example the first row(S S S) of the table indicates that an action was successful in all the three LMPs and this is the only scenario which allows us to move forward in a test. At the end of a test, closer the total reward is to zero, more similar the LMPSpec and LMPImpl are.

Let's have a look on how probability distributions of the MDP in Figure 4.4 can be

*Recall that we assumed the LMPspec to be in tree-like representation

calculated using the above given formulas

$$\begin{aligned} Pr^{a}_{s_0 s_1^{alive}} &= Pr^{Spec}(s_0, a, s_1).(P^{Impl}(a^{\surd}|tr(s_0)).P^{Clone}(a^{\surd}|tr(s_0)) \\ &= \frac{3}{10} \times \frac{5}{10} \times \frac{6}{10} \\ &= 0.09 \end{aligned}$$

When we are at the level different from the root level in the MDP we take the normalized probability in the case of LMPImpl and LMPClone. We shall explain what we mean by that by showing the calculation for probability distribution at the first level.

$$\begin{aligned} Pr^{a}_{s_1^{alive} s_3^{alive}} &= Pr^{Spec}(s_1, a, s_3).(P^{Impl}(a^{\surd}|tr(s_1^{alive})).P^{Clone}(a^{\surd}|s_1^{alive})) \\ &= 1 \times 0 \times \frac{1}{2} \\ &= 0 \end{aligned}$$

Here we need the observation probability of test aa from state s_1 in case of LMPImpl and LMPClone, which is shown in the table below:

	$a^{\times}$	$a^{\surd}a^{\times}$	$a^{\surd}a^{\surd}$
Impl	0. 5	0. 5	0
Clone	0. 4	0. 3	0. 3

Remaining probability distributions can be calculated in the similar manner. Whose values are given below.

$$\begin{aligned} Pr^{a}_{s_0 s_1^{dead}} &= Pr^{Spec}(s_0, a, s_1).(1 - P^{Impl}(a^{\surd}|tr(s_0)).P^{Clone}(a^{\surd}|tr(s_0))) \\ &= 0.3 \times (1 - (0.5 \times 0.6)) \\ &= 0.21 = Pr^{a}_{s_0, s_2^d} \end{aligned}$$

$Pr^{a}_{s_0 s_2^{alive}} = 0.09$
$Pr^{a}_{s_1^{alive} s_3^{dead}} = 0.75$
$Pr^{a}_{s_1^{alive} s_4^{alive}} = Pr^{a}_{s_2^{alive} s_5^{alive}} = 0$
$Pr^{b}_{s_1^{alive} s_4^{dead}} = Pr^{c}_{s_2^{alive} s_5^{dead}} = 0.5$
$Pr^{a}_{s_1^{alive} s_4^{dead}} = Pr^{a}_{s_2^{alive} s_5^{dead}} = Pr^{b}_{s_0 s_1^{dead}} = Pr^{c}_{s_0 s_1^{dead}} = Pr^{b}_{s_0 s_2^{dead}} = Pr^{c}_{s_0 s_2^{dead}} = Pr^{b}_{s_1^{alive} s_3^{d}} = 0$
$Pr^{c}_{s_1^{alive} s_3^{dead}} = Pr^{c}_{s_1^{alive} s_4^{dead}} = Pr^{b}_{s_2^{alive} s_5^{dead}} = 0$
$Pr^{a}_{s_0 dead} = 1 - Pr^{Spec}(s_0, a, S) = 0.4$

$$
\begin{array}{l}
Pr^{b}_{s_0 dead} = Pr^{c}_{s_0 dead} = 1 \\
Pr^{a}_{s_1^{alive} dead} = Pr^{b}_{s_1^{alive} dead} = 0 \qquad Pr^{c}_{s_1^{alive} dead} = 1 \\
Pr^{a}_{s_2^{alive} dead} = Pr^{b}_{s_2^{alive} dead} = 1 \qquad Pr^{c}_{s_2^{alive} dead} = 0 \\
Pr^{a}_{s_3^{alive} dead} = Pr^{b}_{s_3^{alive} dead} = Pr^{c}_{s_3^{alive} dead} = 1 \\
Pr^{a}_{s_4^{alive} dead} = Pr^{b}_{s_4^{alive} dead} = Pr^{c}_{s_4^{alive} dead} = 1 \\
Pr^{a}_{s_5^{alive} dead} = Pr^{b}_{s_5^{alive} dead} = Pr^{c}_{s_5^{alive} dead} = 1
\end{array}
$$

Reward function calulations are given below:

$$
\begin{array}{l}
R^{a}_{s_0 s_1^{alive}} = R^{b}_{s_0 s_1^{alive}} = R^{c}_{s_0 s_1^{alive}} = 0 \\
R^{a}_{s_0 s_2^{alive}} = R^{b}_{s_0 s_2^{alive}} = R^{c}_{s_0 s_2^{alive}} = 0 \\
R^{a}_{s_1^{alive} s_3^{alive}} = R^{b}_{s_1^{alive} s_3^{alive}} = R^{c}_{s_1^{alive} s_3^{alive}} = 0 \\
R^{a}_{s_1^{alive} s_4^{alive}} = R^{b}_{s_1^{alive} s_4^{alive}} = R^{c}_{s_1^{alive} s_4^{alive}} = 0 \\
R^{a}_{s_2^{alive} s_5^{alive}} = R^{b}_{s_2^{alive} s_5^{alive}} = R^{c}_{s_2^{alive} s_5^{alive}} = 0
\end{array}
$$

$$
\begin{array}{rcl}
R^{a}_{s_0 dead} & = & P^{Impl}(a^{\surd}|tr(s_0)).P^{Clone}(a^{\times}|tr(s_0) - P^{Impl}(a^{\times}|tr(s_0).P^{Clone}(a^{\surd}|tr(s_0) \\
& = & \frac{1}{2} \times \frac{4}{10} - \frac{1}{2} \times \frac{3}{5} \\
& = & -0.1
\end{array}
$$

$$
\begin{array}{l}
R^{b}_{s_0 dead} = R^{c}_{s_0 dead} = 0 \\
R^{a}_{s_1^{alive} dead} = 0 \\
R^{b}_{s_1^{alive} dead} = R^{c}_{s_1^{alive} dead} = \frac{1}{2} \\
R^{a}_{s_2^{alive} dead} = -\frac{1}{2} \\
R^{b}_{s_2^{alive} dead} = R^{c}_{s_2^{alive} dead} = \frac{1}{2} \\
R^{a}_{s_3^{alive} dead} = R^{b}_{s_3^{alive} dead} = R^{c}_{s_3^{alive} dead} = 0 \\
R^{a}_{s_4^{alive} dead} = R^{b}_{s_4^{alive} dead} = R^{c}_{s_4^{alive} dead} = 0 \\
R^{a}_{s_5^{alive} dead} = 0 \\
R^{b}_{s_5^{alive} dead} = R^{c}_{s_5^{alive} dead} = 0
\end{array}
$$

$$\begin{aligned} R^a_{s_0 s_1^{dead}} &= \frac{R^a_{s_0 S} - Pr^a_{s_0,s_1^a}.R^a_{s_0 s_1^a} - Pr^a_{s_0,s_1^a}.R^a_{s_0 dead}}{Pr^a_{s_0,s_1^d}} \\ &= \frac{0.02 - 0.09 \times 0 - 0.4 \times (-0.1)}{0.21} \\ &= 0.28 \end{aligned}$$

$R^b_{s_0 s_1^{dead}} = R^c_{s_0 s_1^{dead}} = 0.04$
$R^a_{s_0 s_2^{dead}} = 0.28$
$R^a_{s_1^{alive} s_3^{dead}} = \frac{1}{2}, R^b_{s_1^{alive} s_3^{dead}} = -\frac{1}{2}, R^c_{s_1^{alive} s_3^{dead}} = \frac{1}{2}$
$R^a_{s_1^{alive} s_4^{dead}} = 1, R^b_{s_1^{alive} s_4^{dead}} = \frac{1}{2}, R^c_{s_1^{alive} s_4^{dead}} = 0$
$R^a_{s_2^{alive} s_5^{dead}} = 0, R^b_{s_2^{alive} s_5^{dead}} = \frac{1}{2}, R^c_{s_2^{alive} s_5^{dead}} = -\frac{1}{2}$

4.1.2 The first appraoch: MDP construction of type 2

MDP construction of type 1(first approach) defined in subsection 4.1.1 has double the number of states when compared to the corresponding LMP and there are two different dead states. Since doubling the states at first sight is not a good idea from the point of view of computational efficiency, in the implementation that follows, we will not consider that the states are doubled. Thus,we made a few modifications to our previous MDP construction by keeping the same number of states as LMPspec plus one extra state, the Dead state. Also from any state all the arrows directing to the different dead states now will be leading to one big *Dead* state. Thus the modified MDP will look like the one in Figure 4.5:

Below we define the four elements of the MDP: set of states, set of actions, state-transition probability distribution and reward function.

- **States**: same as LMPSpec states + an additional state called *Dead*.
- **Actions**: same as LMPSpec actions.
- **State-transition probability distribution**:

$$P^a_{ss'} = \begin{cases} Pr^{Spec}(s,a,s').P^{Impl}(a^{\surd}|tr(s)).P^{clone}(a^{\surd}|tr(s)) & if \;\; s' \neq Dead \\ 1 - (Pr^{Spec}(s,a^{\surd}).P^{Impl}(a^{\surd}|tr(s)).P^{clone}(a^{\surd}|tr(s))) & if \;\; s' = Dead \end{cases}$$

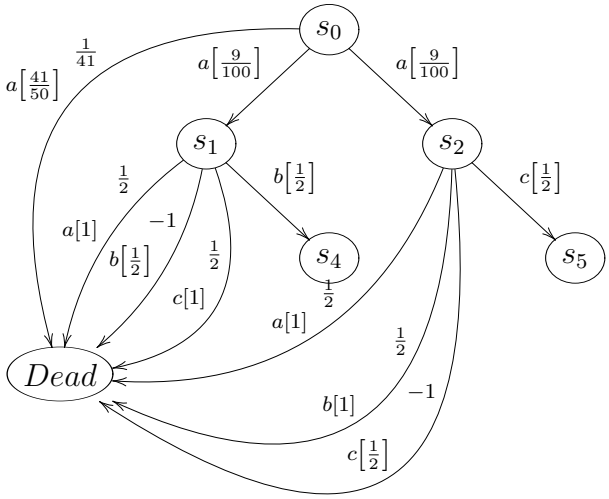


Figure 4.5: First approach: MDP construction of type 2

where

$$Pr^{Spec}(s, a^{\surd}) = \sum_{s \in LMPSpecstates} Pr^{Spec} s, a, s^{'}$$

- **Reward function:**

$$R^{a}_{ss'} = \begin{cases} 0 & if \;\; s^{'} \neq dead \\ \dfrac{R^{a}_{sS}}{Pr^{a}_{sdead}} & if \;\; s^{'} = dead \end{cases}$$

where R^{a}_{sS} represents the reward from state s with action a to all possible next states which can be represented in the following manner:

$$\begin{aligned} R^{a}_{ss'} &= Pr^{a}_{sDead}, R^{a}_{sDead} + \underbrace{\sum_{s' \in \{S \setminus dead\}} Pr^{a}_{ss'} . \underbrace{R^{a}_{ss'}}_{0}}_{0} \\ &= Pr^{a}_{sDead}, R^{a}_{sDead} \end{aligned}$$

R^{a}_{sS} can be expressed in a different way as follows:

$$R^{a}_{sS} = Pr^{Spec}(s, a^{\surd}) \cdot r(s, a^{\surd}) + Pr^{Spec}(s, a^{\times}) \cdot r(s, a^{\times})$$

where $r(s, a^{\surd})$ and $r(s, a^{\times})$ have the following formulas:

- $r(s, a^{\surd}) = P^{Impl}(a^{\times}|tr(s)) \cdot P^{Clone}(a^{\surd}|tr(s)) - P^{Impl}(a^{\surd}|tr(s)) \cdot P^{Clone}(a^{\times}|tr(s))$

- $r(s, a^{\times}) = P^{Impl}(a^{\surd}|tr(s)) \cdot P^{Clone}(a^{\times}|tr(s)) - P^{Impl}(a^{\times}|tr(s)) \cdot P^{Clone}(a^{\surd}|tr(s))$

For the understanding of readers below we give the probability distribution and reward function calculations in this new setting:

$$\begin{aligned} Pr^{a}_{s_0,s_1} &= Pr^{Spec}(s_0, a, s^{'}).P^{Impl}(a^{\surd}|tr(s_0)).P^{clone}(a^{\surd}|tr(s_0) \\ &= 0.3 \times 0.5 \times 0.6 \\ &= 0.09 \end{aligned}$$

$$\begin{aligned} R^{a}_{s_0 S} &= FSS + SFF - (SSF + FFS) \\ &= \frac{1}{2}.\frac{3}{5}.\frac{3}{5} + \frac{1}{2}.\frac{2}{5}.\frac{2}{5} - \left(\frac{1}{2}.\frac{3}{5}.\frac{2}{5} + \frac{1}{2}.\frac{2}{5}.\frac{3}{5}\right) \\ &= \frac{9}{50} + \frac{2}{25} - \left(\frac{3}{25} + \frac{3}{25}\right) \\ &= \frac{1}{50} \\ Pr^{a}_{s_0 Dead} &= \frac{41}{50} \end{aligned}$$

$$\begin{aligned} R^{a}_{s_0 Dead} &= \frac{R^{a}_{s_0 S}}{Pr^{a}_{s_0 Dead}} \\ &= \frac{1}{50}.\frac{50}{41} \\ &= \frac{1}{41} \end{aligned}$$

Let's recall that the value of the optimal policy determines the equivalence between the LMPs. The positive value means that the LMPs are not trace equivalent and on the other hand value zero or negative indicates that they are equivalent. There are some counter examples which give contrary values and hence indicate the disadvantage

of this new MDP setting. In the following counter example even though the two LMPs are not trace equivalent the value of the optimal policy is negative.

Let three LMPs be as follows:

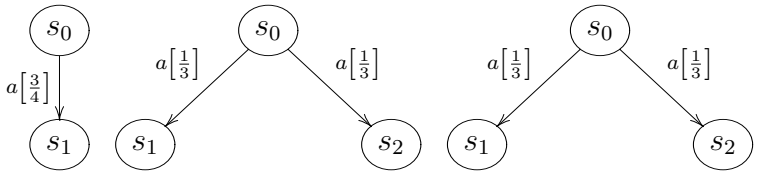


Figure 4.6: counter example 1

$$\begin{aligned}
R^a_{s_0S} &= FSS + SFF - (SSF + FFS) \\
&= \frac{1}{4}.\frac{2}{3}.\frac{2}{3} + \frac{3}{4}.\frac{1}{3}.\frac{1}{3} - \left(\frac{3}{4}.\frac{2}{3}.\frac{1}{3} + \frac{1}{4}.\frac{1}{3}.\frac{2}{3}\right) \\
&= \frac{1}{9} + \frac{1}{12} - \left(\frac{1}{6} + \frac{1}{18}\right) \\
&= -\frac{1}{50} \\
Pr^a_{s_0Dead} &= 1 - (Pr^{Spec}(s_0, a^{\surd}).P^{Impl}(a^{\surd}|tr(s_0)).P^{clone}(a^{\surd}|tr(s_0) \\
&= 1 - (\frac{2}{3}.\frac{3}{4}.\frac{2}{3}) \\
&= \frac{2}{3}
\end{aligned}$$

$$\begin{aligned}
R^a_{s_0Dead} &= \frac{R^a_{s_0S}}{Pr^a_{s_0Dead})} \\
&= -\frac{1}{50}.\frac{3}{2} \\
&= -0.03
\end{aligned}$$

The negative value of the optimal policy shows that we are not dealing with a divergence notion here. However one may say that since the optimal policy is not zero this is an indication to the fact that two LMPs are trace-equivalent. In the next counter example we can see that all the policies have the value zero but the LMPs are not trace equivalent.

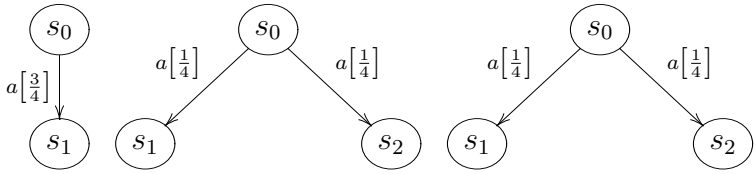


Figure 4.7: counter example 2

$$\begin{aligned} R^a_{s_0S} &= FSS + SFF - (SSF + FFS) \\ &= \frac{1}{4}.\frac{1}{2}.\frac{1}{2} + \frac{3}{4}.\frac{1}{2}.\frac{1}{2} - \left(\frac{3}{4}.\frac{1}{2}.\frac{1}{2} + \frac{1}{4}.\frac{1}{2}.\frac{1}{2}\right) \\ &= \frac{1}{16} + \frac{3}{16} - \left(\frac{3}{16} + \frac{1}{16}\right) \\ &= 0 \\ R^a_{s_0Dead} &= \frac{R^a_{s_0S}}{Pr^a_{s_0Dead})} \\ &= 0 \end{aligned}$$

4.1.3 Second approach

We want to stick with type 2 for obvious reasons but we have to find a way to give more control to the agent that has to interact with the MDP. The idea we had is to double the number of actions, replacing each action a by $a^{\surd}$ and $a^{\times}$. If the action $a^{\surd}$ is chosen by the agent, the later will obtain reward if and only if the action is successfully performed in the LMPSpec. Thus the agent has the possibility to reduce or augment the probability to receive a non zero reward if he expects this reward to be negative or positive.

Here is the MDP related to the second approach

- **States**: same as LMPSpec states + an additional state called *Dead*.
- **Actions**: each action(e.g. a) of LMPSpec has two variants($a^{\surd}$ and $a^{\times}$).
- **State-transition probability distribution**:

$$P^a_{ss'} = \begin{cases} Pr^{Spec}(s,a,s').P^{Impl}(a^{\surd}|tr(s)).P^{clone}(a^{\surd}|tr(s)) & if\ \ s' \neq Dead \\ 1 - (Pr^{Spec}(s,a^{\surd}).P^{Impl}(a^{\surd}|tr(s)).P^{clone}(a^{\surd})|tr(s)) & if\ \ s' = Dead \end{cases}$$

- **Reward function:**

$$R^a_{ss'} = \begin{cases} 0 & if\ \ s' \neq Dead \\ \dfrac{R^{a^\surd}_{sS}}{Pr^{a^\surd}_{sDead}} & if\ \ s' = Dead\ \ and\ \ a = a^\surd \\ \dfrac{R^{a^\times}_{sDead}}{Pr^{a^\times}_{sDead}} & if\ \ s' = Dead\ \ and\ \ a = a^\times \end{cases}$$

$R^{a^\surd}_{sS} = Pr^{Spec}(s, a^\surd) \cdot r(s, a^\surd)$

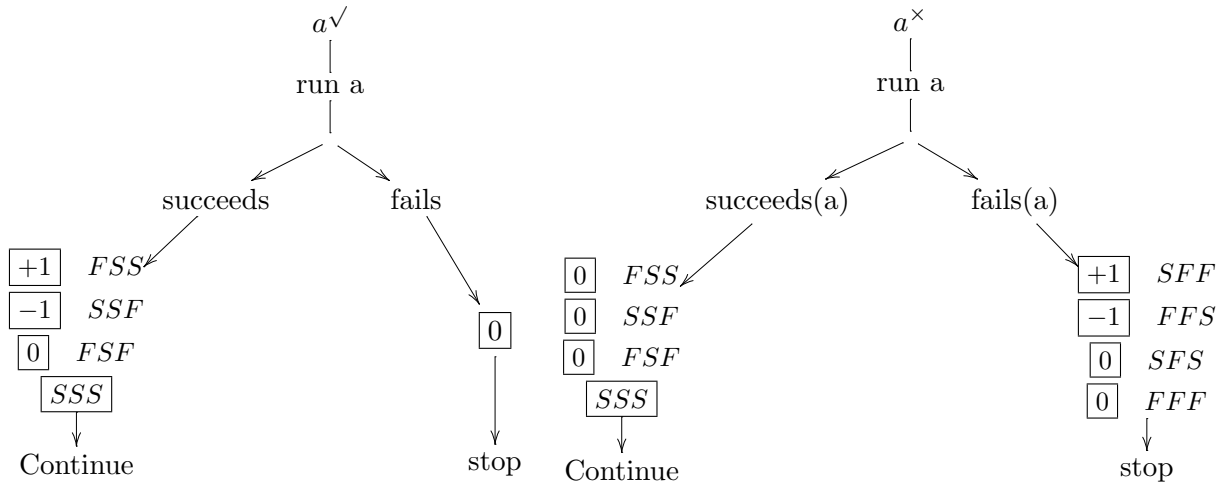
$R^{a^\times}_{sS} = Pr^{Spec}(s, a^\times) \cdot r(s, a^\times)$

where $r(s, a^\surd)$ and $r(s, a^\times)$ have the following formulas:

$$r(s, a^\surd) = P^{Impl}(a^\times | tr(s)) \cdot P^{Clone}(a^\surd | tr(s)) - P^{Impl}(a^\surd | tr(s)) \cdot P^{Clone}(, a^\times | tr(s))$$

$$r(s, a^\times) = P^{Impl}(a^\surd | tr(s)) \cdot P^{Clone}(a^\times | tr(s)) - P^{Impl}(a^\times | tr(s)) \cdot P^{Clone}(a^\surd | tr(s))$$

Reward assignment is done in a slightly different manner in the new setting, which is shown below:



The number of actions in this MDP structure is simply the Cartesian product of the LMP actions and two variants of actions i. e. for example if the LMPSpec contains a

total of three actions say a, b, cthen the number of actions in the MDP is the Cartesian product of $(a, b, c \times success(s), failure(f))$ thus we have six actions named (as, af, bs, bf, cs, cf). We shall see later in this chapter that the fact of doubling the actions does not affect the rate of convergence.

Probability distributions remain unchanged from the second version but the reward calculations are done in a slightly different manner here. Let's look at how $R(s_0, a^{\surd}.Dead)$ and $R(s_0, a^{\times}.Dead)$ can be calculated.

$$\begin{aligned} R_{sS}^{a^{\surd}} &= FSS - SSF \\ &= \frac{1}{2}.\frac{3}{5}.\frac{3}{5} - \frac{1}{2}.\frac{3}{5}.\frac{2}{5} \\ &= \frac{9}{50} - \frac{3}{25} \\ &= \frac{3}{50} \end{aligned}$$

$$\begin{aligned} R_{s_0Dead}^{a^{\surd}} &= \frac{R_{s_0S}^{a^{\surd}}}{Pr_{s_0Dead}^{a^{\surd}}} \\ &= \frac{3}{50}.\frac{50}{41} \\ &= \frac{3}{41} \end{aligned}$$

$$\begin{aligned} R_{sS}^{a^{\times}} &= SFF - FFS \\ &= \frac{1}{2}.\frac{2}{5}.\frac{2}{5} - \frac{1}{2}.\frac{2}{5}.\frac{3}{5} \\ &= \frac{2}{25} - \frac{3}{25} \\ &= -\frac{1}{25} \\ R_{s_0Dead}^{a^{\times}} &= \frac{R_{s_0Dead}^{a^{\times}}}{Pr_{s_0Dead}^{a^{\times}}} \\ &= -\frac{1}{25}.\frac{50}{41} \\ &= -\frac{2}{41} \end{aligned}$$

Other rewards can be calculated in a similar manner. The current setting of MDP looks

like the the one in Figure 4.8:

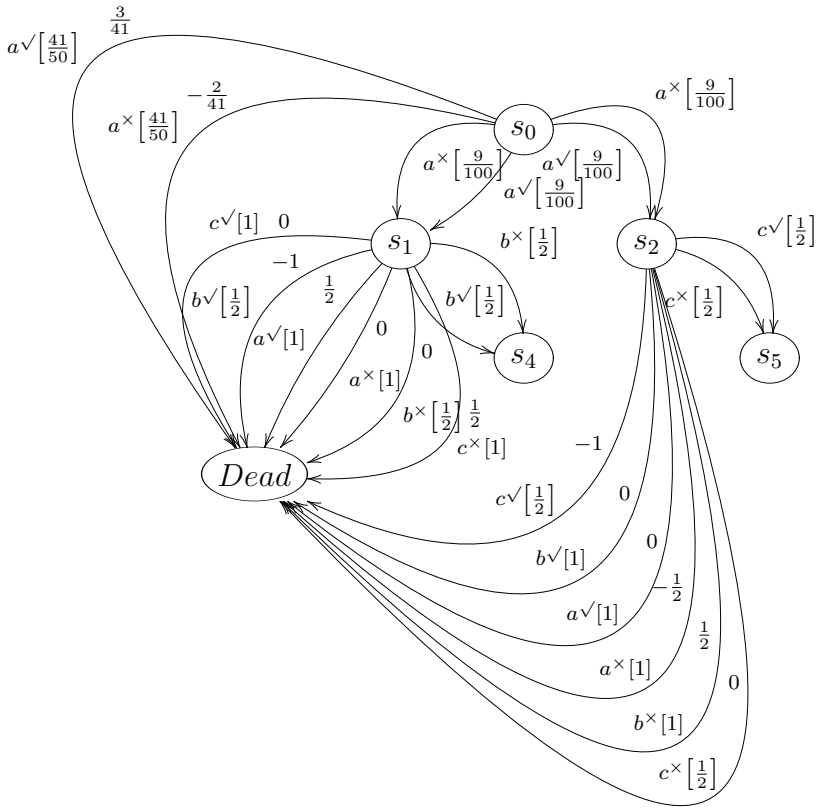


Figure 4.8: Second approach MDP construction

Now that we have the essential element(MDP) of our model we will now proceed to the implementation part of it. In the following sections we will explain our choice of RL algorithm used for implementation and reasons for restricting ourselves to the trace equivalence.

4.1.4 Second approach gives rise to a divergence notion

As shown in [8] the value of the optimal policy of the second approach MDP construction gives rise to a notion of trace equivalence divergence. That is, given any two LMPs LMPSpec and LMPImpl, the value of the optimal policy of the related MDP is always ≥ 0, and $= 0$ if and only if they are trace-equivalent.

4.2 Implementation of the model

Among all the reinforcement learning algorithms explained in chapter 3, we have chosen the one called *Q-learning*. Like many RL algorithms, Q-learning algorithm does not demand the complete prior knowledge of the model, which is exactly the case here as we have no knowledge of the LMP representing implementation(i.e LMPImpl). Thus as shown in preceding sections,we have full knowledge of the state set of the MDP but no pior information on the transition probability distributions and of the reward signal structure. For the reminder of this thesis we shall take the MDP in the subsection 4.1.3 into consideration. Q-learning algorithm for our model is given below:

Algorithm 4 Q-learning for our model.

initialize $Q(s, a)$ arbitrarily
Repeat (for each episode)(*e.g.* , 10, 000):
initialize s (s: initial state)
Repeat (for each step of episode):
Choose an action a from s using policy derived from Q (e.g., soft max)
Take action a, observe immediate reward r, and next-state $s^{'}$
update α
update Q-value: $Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a^{'}} Q(s^{'}, a^{'}) - Q(s,a) \right]$
update next-states
Until s is terminal

In the update rule of Q-learning $Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a^{'}} Q(s^{'}, a^{'}) - Q(s,a) \right]$, $Q(s^{'}, a^{'})$ represents the value for the next state-action pair and r is the immediate reward received for that transition. The discount factor γ , (a value between 0 and 1) controls the extent to which learning is concerned with long-term vs. short-term consequences of its actions (serves as an interest rate on learning). In our model the value of the γ is constant at 0.8. the α is the learning rate, which sets a trade off between exploration vs. exploitation. α is decreased slowly to ensure convergence. In our case the value of α is decreased at the rate of $\frac{1}{x}$ where x is the number of times a state-action pair has been visited.

we have adopted the technique called "sampling ". More precisely, during each episode from every state there is a possibility of performing only one action. This explains the reason why we restricted ourselves to trace equivalence.

The Q-learning algorithm works as follows: we execute Q-learning for, say 10,000

episodes on our MDP. For each step of an episode an action is chosen from a policy derived from an action selection method(eg: soft-max). Let us recall that we have three LMPs named LMPImpl(representation of implementation), LMPSpec(representation of specifications)and LMPClone(clone of LMPSpec) at our disposal. Action resulted from action selection method will be executed in the MDP. Then we observe the immediate reward and the next state of the MDP. If any one of the three LMPs reach the state *Dead* the episode ends and then the whole procedure will be repeated until we finish all the episodes. At the end of all the episodes Q-learning algorithm converges to an optimal policy which will be evaluated to see if the specification behaves exactly like the implementation. In other words if LMPSpec and LMPImpl are trace equivalent. If they are trace equivalent the value of the policy will be zero otherwise it will be a value strictly greater than zero.

Let's consider one of the example on which we have done the experiments. This example has simple and small LMPs as the most calculations like finding optimal policy could be done manually which could then be easily verified against the one resulted from the algorithm.

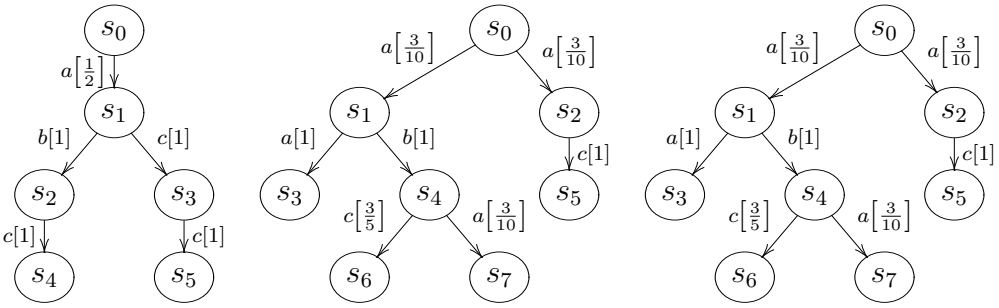


Figure 4.9: Example1 LMPs: LMPImpl, LMPSpec and LMPClone

For the above example the optimal Q-values are as shown in the following Q-value matrix: For each state in a learned Q-table, there is an action or a set of actions that has the highest Q-value, which constitutes the optimal policy. From the table 4.2 we can deduce that there are two optimal policies that can differentiate the LMPs in figure 4.9. First one is $\Pi^* = as\ as\ bf$ and the second one is $\Pi^* = as\ cf\ bf$ because from the state S1 both the actions *as* and *cf* have the same highest optimal values.

At the beginning we had chosen $\epsilon - greedy$ as action selection method which with probability $1 - \epsilon$, exploits its current knowledge by selecting a greedy action, and with probability ϵ , selects an action at random. It is important to mention here that the immediate reward when an action succeeds in both LMPSpec and LMPClone and fails in LMPImpl will be +1 and on the other hand when it succeeds both in LMPImpl and LMPSpec and fails in LMPClone will be -1 and in all the other situations will

	as	af	bs	bf	cs	cf
S0	0. 132	0. 032	0. 0	0. 0	0. 0	0. 0
S1	0. 50	0. 0	-0. 32	0. 20	0. 0	0. 50
S2	0. 0	-0. 052	0. 0	0. 43	-0. 1	0. 70
S3	0. 0	0. 0	0. 0	0. 0	0. 0	0. 0
S4	0. 0	0. 0	0. 0	0. 0	0. 0	0. 0
S5	0. 0	0. 0	0. 0	0. 0	0. 0	0. 0
S6	0. 0	0. 0	0. 0	0. 0	0. 0	0. 0
S7	0. 0	0. 0	0. 0	0. 0	0. 0	0. 0
dead	0. 0	0. 0	0. 0	0. 0	0. 0	0. 0

Table 4.2: optimal Q-value table

be 0. Positive reward indicates the possible equivalence between the LMPs and the negative reward indicates the contrary. Since from any state the sum of the transition probabilities of any action may be less than 1, when an action results with a negative reward $\epsilon - greedy$ will consider that particular action as a bad choice the next time it visits that state. Hence it takes way long to get back to the normal situation and consider equally all the actions.

4.3 Empirical results

First we have run the algorithm by keeping the ϵ value constant at 0.1 and then at 0.8 for 10000, 20000, 50000, and 100000 episodes respectively. For the resulted graphs please refer the appendix. From these plots we can observe that even though the Q-values have converged in a steady manner sometimes resulted with sub-optimal policies. Though it has learned the optimal policy in the beginning, towards the end because of the exploration in the ϵ-greedy method it will settle to a sub-optimal policy. Then we decreased the ϵ from 0. 8 to 0. 1 and then from 0. 8 to 0. 01. To understand the drawback of ϵ-greedy lets compare the results from 10000 and 20000 episodes when ϵ value has been decreased from 0. 8 to 0. 01.

As you can see in the beginning though there is a much variation in the Q-values towards the end the values have been stabilized. In the case of 10000 episodes ϵ-greedy converged to the value Q(0, as)=0. 125 which is only a test that differentiates the LMPs in fig 4.9 but it is not the optimal value. But in the second case it found the optimal policy with optimal values Q(0, as)=0. 132.

Theoretically both the ϵ-greedy and softmax action selection methods are said

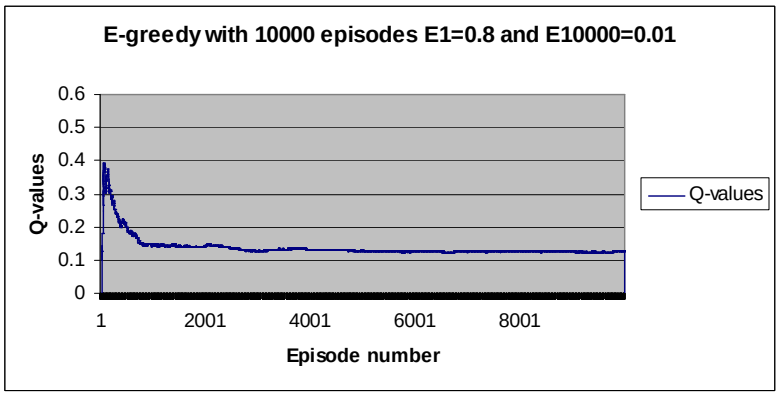


Figure 4.10: ϵ-greedy with 10000 episodes and Q(0, as)=0. 125

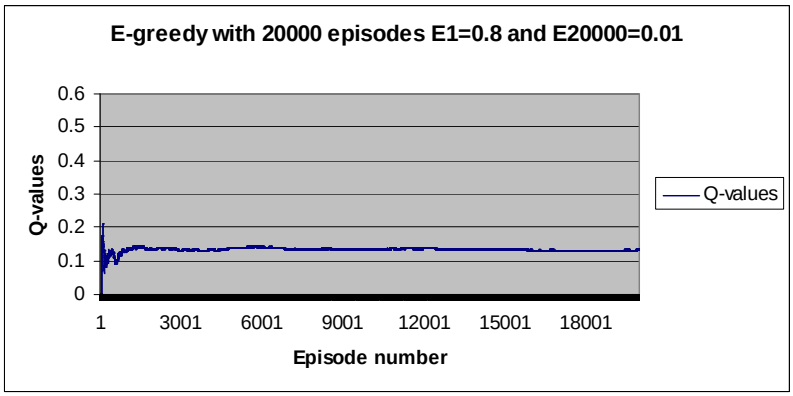


Figure 4.11: ϵ-greedy with 20000 episodes and Q(0, as)=0. 132

to converge to optimal qvalues, but for our model softmax converges faster than ϵ-greedy. Soft-max works fine with our model, but the main challenge here is to tune the *temperature(τ)* parameter. At the beginning the τ value was been kept constant at 0. 8 but the Q-values were not converging to the optimal values. Then we tried with different τ values for example 0. 9, 0. 7 etc but finally found that when the τ value is decreased from 0. 8 to 0. 01 is working fine with our model. Which can be seen from the graphs in figure 4.12 and 4.13: Figure 4.12 shows the resulting graph from the

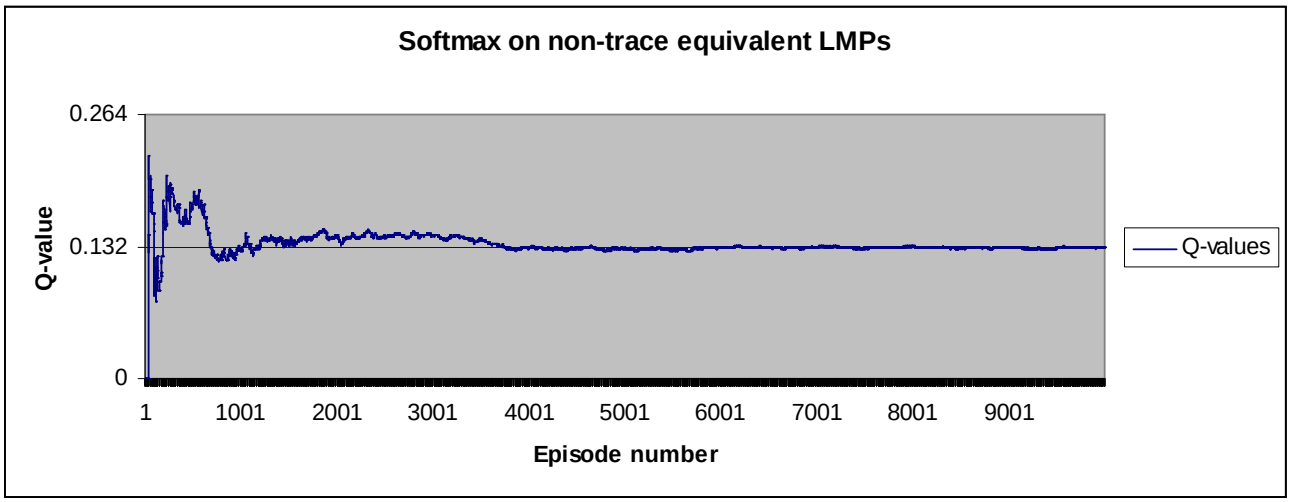


Figure 4.12: Softmax on non-trace equivalent LMPs

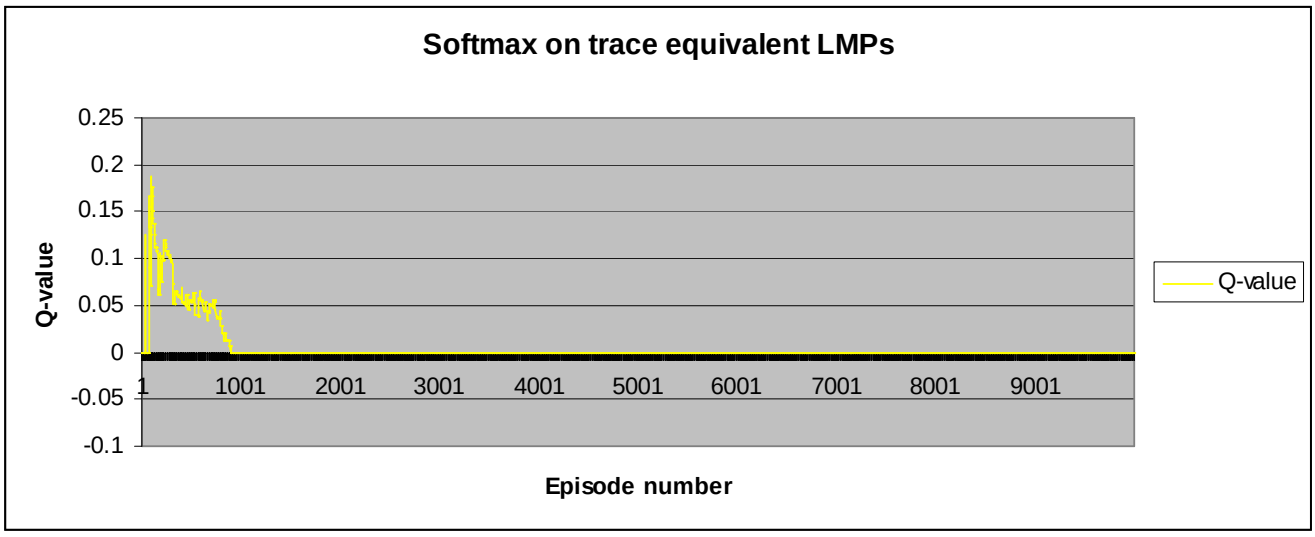


Figure 4.13: Softmax on trace equivalent LMPs

execution of Q-learning algorithm along with softmax on the LMPs of our first example

which are non-trace equivalent. (see figure 4.9). This is the result of one run execution of 10000 episodes and we can see that Q-values converged to the optimal value which is 0. 132 in this case. Figure 4.13 shows the resulting graph from the execution of Q-learning algorithm along with softmax on the trace equivalent LMPs. We can see that optimal Q-values converge to zero. which in turn proves the conformance between the LMPs.

After those preliminary experiments, we tried to find out if there was a way to improve the efficiency of our algorithm. The "traditional "Q-learning algorithm will be called Experiment1 and the new one experiment2. In experiment1 every time an action *as* is chosen from a state *s*, only the Q-value of that state-action pair will be updated i. e Q(as, s). On the other hand in experiment2 chosen action's counterpart will also be updated , in this case Q(af, s). By doing this double updation we hope that experiment2 will be better than experiment1 from the point of view of convergence rate. Because with experiment2 we have twice the information at each state every time the Q-learning back propagates.

Now let's compare the performance of the experiment1 and experiment2 by looking at two sets of results :

- Experiment1 VS Experiment2 with non-trace equivalent LMPs.
- Experiment1 VS Experiment2 with trace equivalent LMPs

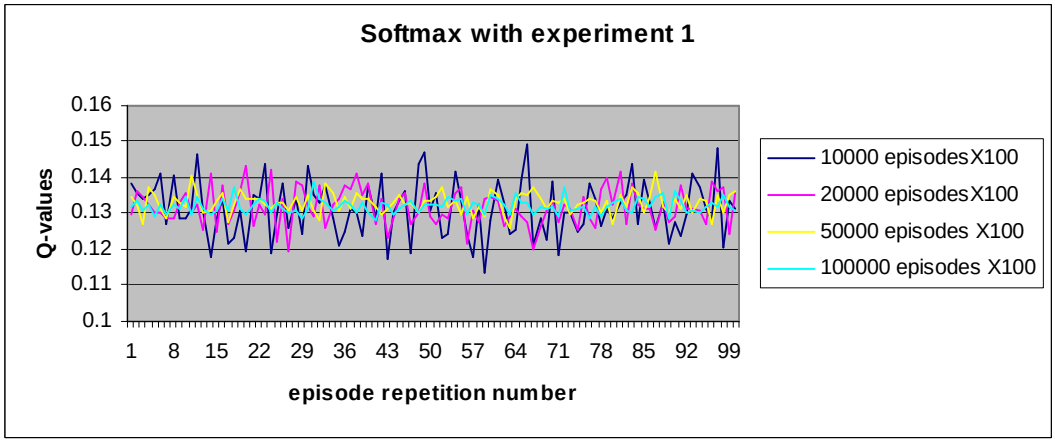


Figure 4.14: Softmax with 10000, 20000, 50000 and 100000 episodes repeated for 100 times on non-trace equivalent LMPs

Episode number	Average	Empirical Standard deviation
10000X100	0. 13212978	0. 00204975
20000X100	0. 13179057	0. 00506431
50000X100	0. 13301062	0. 00281912
100000X100	0. 13212978	0. 00204975

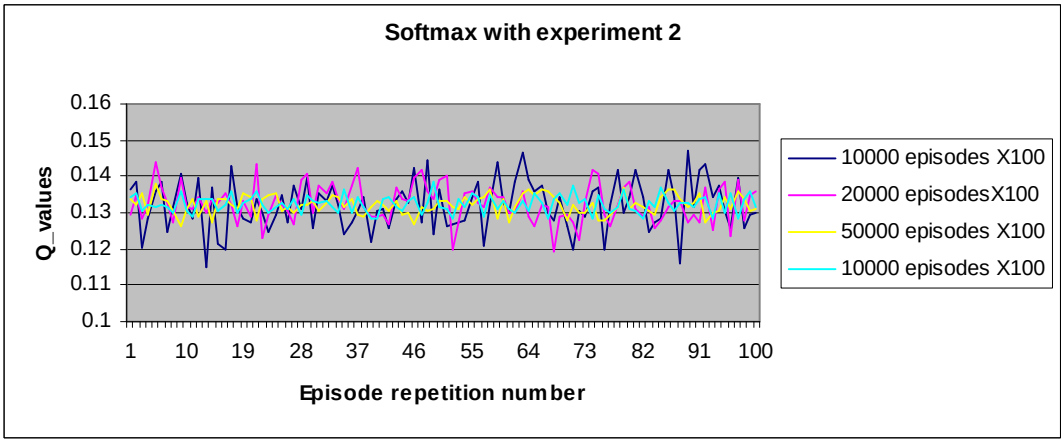


Figure 4.15: Softmax with 10000, 20000, 50000 and 100000 episodes repeated for 100 times on non-trace equivalent LMPs

Episode number	Average	Empirical Standard deviation
10000X100	0. 13207714	0. 0067708
20000X100	0. 13179057	0. 00509432
50000X100	0. 13301062	0. 00258727
100000X100	0. 13212978	0. 00231638

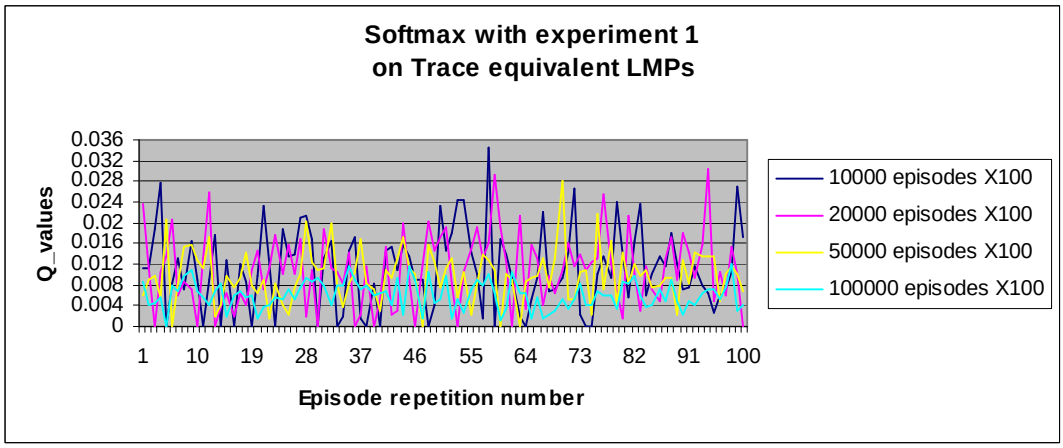


Figure 4.16: Softmax with 10000, 20000, 50000 and 100000 episodes repeated for 100 times on trace equivalent LMPs

Episode number	Average	Empirical Standard deviation
10000X100	0. 01108514	0. 00775528
20000X100	0. 01088717	0. 00693948
50000X100	0. 00972866	0. 00498668
100000X100	0. 006088	0. 00264953

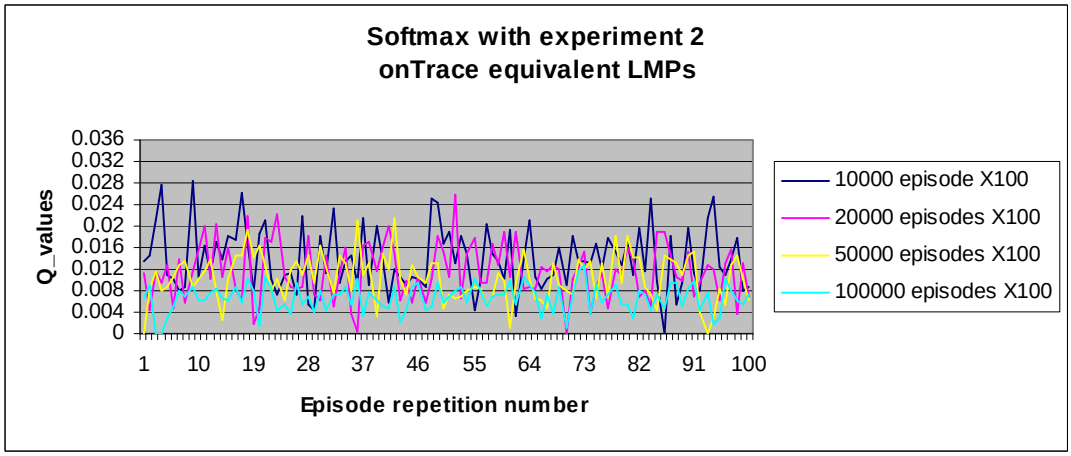


Figure 4.17: Softmax with 10000, 20000, 50000 and 100000 episodes repeated for 100 times on trace equivalent LMPs

Episode number	Average	Empirical Standard deviation
10000X100	0. 01392334	0. 00575999
20000X100	0. 01148204	0. 00513899
50000X100	0. 01041614	0. 00426676
100000X100	0. 00665485	0. 00256428

The figure 4.14 compares the results of experiment1 for the episodes 10000, 20000, 50000, and 100000 repeated for 100 times each. Experiment1 was performed on LMPs of the Figure 4.9 which are not trace equivalent and used softmax action selection method. As we can see from the Average and Empirical standard deviation listed below the Figure 4.14 experiment1 eventually converged to the optimal Q-values which in this case is 0. 132, as we have mentioned earlier if the two LMPs are not trace equivalent Q-learning gives a test whose value will be strictly greater than zero , by this positive value we can deduce that the two LMPs are not trace equivalent and the test which differentiates them is "as as bf". Similarly, Figure 4.15 compares the results of same number of episodes but of the experiment2. We hoped the results of experiment2 would be better than the ones of experiment1, but surprisingly, this is not the case as there is no significant difference between the average and empirical standard deviations of experiment1 and experiment2. The same result was observed in the case of trace equivalent LMPs which is depicted in Figures 4.16and 4.17.

To understand why the experiment2 is not performing as we expected , we made slight modifications to our previous example (figure 4.9) so that from the root node choosing either of the actions *as* or *af* has the equal importance. So the new LMPs now look as in the figure: 4.18

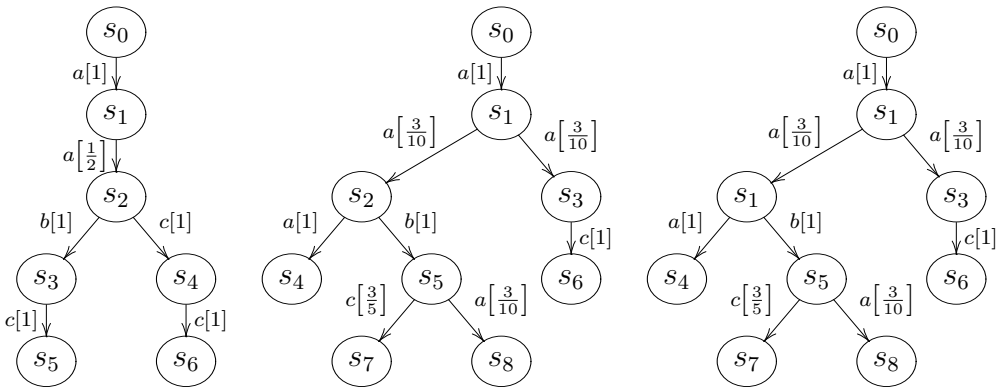


Figure 4.18: Example2 LMPs: LMPImpl, LMPSpec and LMPClone

We observed the results of 10 runs for each experiment1 and experiment2. Please refer the appendix for the results. The difference between the Example1(Figure 4.9) and Example 2(Figure 4.18) is , in Example2 , there is a extra node with transition labelled with action a and probability 1, and this node is the new root node and the remaining LMP is exactly same as the example2. We repeated both the experiments on Example2 once by initializing the Q-values to 0, and the second time initializing the Q-values to the correct values(lets recall that we know the optimal Q-values of the Example1 from the table 4.2). Once again from the results of these experiments in appendix we can observe that empirically Experiment 2 is not efficient than Experiment 2.

This has been a real surprise for us. However,we could think of the few reason why experiment1 is still better than experiment2 and they are:

- Because of the bias of the Q-learning algorithm.
- Experiment1 may be good for exploration and experiment2 may be good for exploitation.

The main reason is that in our final version of the MDP(Figure 4.8), from every state (except the Dead state) there are two variants for every action are possible (for example for an action a there are two variants of that action *as* and *af*). Though we consider two variants of actions from every state we obtain the reward only after reaching the *Dead* state i. e after the episode ends and that means algorithm gives us a signal that something has gone wrong in one of the LMPs. Thus as long as the episode

continues it really does not matter if we choose the basic action a or its variants *as* or *af*. So we can modify this MDP to the one follows, where there is an intermediate state(represented as I) between every state that eventually reaches to Dead state and the Dead state. Once being in the intermediate state we choose either action with *success* or *failure* and finally upon reaching the state Dead we receive the reward. Thus the MDP in Figure 4.8 is equivalent to the one follows: By equivalent we mean that the value of the optimal policy found in both the MDPs is same.

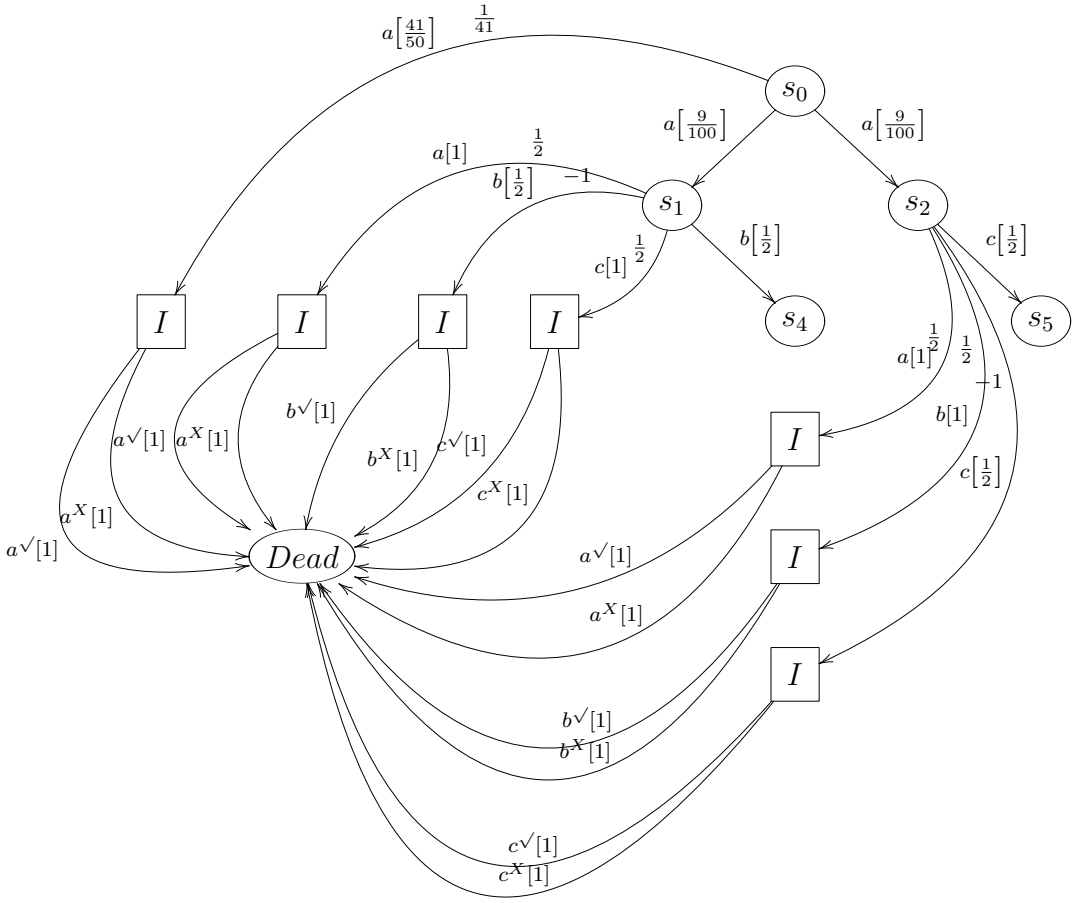


Figure 4.19: Equivalent MDP to second approach

Table 4.3 demonstrates the similarity between experiment1 and experiment2. The average and the empirical standard deviations of experiment1 and experiment2 for 100, 1000, 5000, 10000, 100000 episodes each repeated for 10000 times have been listed. These experiments were conducted on LMPs of the figure 4.18 and softmax as the action selection method. The pre calculated actual value for these LMPs is 0. 105. The experiment1's average increased as the number of episodes increased except for the 100000 episodes but it averaged to the value below the actual value. Experiment2's average also increased like in experiment1 but in the case of 5000 episodes it averaged to the value above the actual value. The empirical standard deviation for both the

experiments is almost same for the same number of episodes. Since we have observed the results starting from 100 episodes up to 100000 episodes this table is an empirical proof for the similarity between the the both experiments.

Episode number	Experiment1		Experiment2	
	average	Emp Std. deviation	Average	Emp Std. deviation
100	0. 05183493	0. 060297	0. 06503227	0. 06276288
1000	0. 05664378	0. 04946941	0. 07999226	0. 04437378
5000	0. 08502152	0. 03605331	0. 10165943	0. 0147544
10000	0. 0974141466	0. 0207266	0. 12258166	0. 01407892
100000	0. 06508017	0. 06340649	0. 06401023	0. 06251184

Table 4.3: Empirical proof showing the similarity between experiment1 and experiment2

Chapter 5

Conclusions and future work

In this chapter, we present a summary of contributions and suggestions for future works.

5.1 Contributions

In this thesis we proposed completely new approach to estimate how far two LMPs are from being trace-equivalent. We introduced trace equivalence divergence notion. The main contribution of this thesis is that this divergence can be computed via RL-algorithms and therefore can still be tractable when dealing with very huge verification problem, even with infinite ones.

We have also tried a way to double the number of updations per episode but there was no significant improvement.

5.2 Future work

In this section, we provide suggestions for future work:

- we would want to modify the construction of MDP from which we calculated divergence notion in order to speed up the calculation. In fact, there is a possibility to modify MDP in such a way that in observation FFF we no longer have to stop the episode. Since observation FFF does not indicate the possible similarity

between the LMPs instead of stoping that episode we could reset the program to the same state where this observation has resulted in order to continue until we observe success unless the probability of observing failure in that particular state is 1. In this way we could increase the rate of exploration.

- Finally it will be interesting to have similar notion of divergence but for other type of equivalence mainly for bisimulation equivalence.

A paper has been submitted under the title "Trace Equivalence Characterization through Reinforcement Learning "to Canadian Artificial Intelligence Conference with contribution from Josée Desharnais,François Laviolette and Sami Zhioua.

Bibliography

[1] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-dynamic programming.* Athena Scientific, Belmont, Mass, 1996.

[2] R. Blute, J. Desharnais, A. Edalat, and P. Panangaden. *Bisimulation for labelled Markov processes.* In Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science, pages 149158, Warsaw, June/July 1997.

[3] A. R. Cassandra, L. P. Kaelbling and M. L. Littman. *Acting optimally in partially observable stochastic domains.* . In Proceedings of the Twelfth National Conference on Artificial Intelligence. Seattle, WA, 1994.

[4] R. H. Crites, and A. G. Barto. *Improving elevator performance using reinforcement learning.* In Advances in neural information processing systems. MIT Press, Cambridge, Mass. pp. 10171023. 1996.

[5] J. Desharnais. *Labelled Markov processes.* PhD thesis, McGill University, 1990.

[6] J. Desharnais, A. Edalat, and P. Panangaden. *A logical characterization of bisimulation for labelled Markov processes.* In Proceedings of 13th Annual IEEE symposium on Logic in Computer Science, pages 478-487, Indianapolis, june 1998. IEEE.

[7] J. Desharnais, V. Gupta, R. Jagadeesan, and P. Panangaden. *Approximating continuous Markov processes.* In Proceedings of 15th Annual IEEE Symposium On Logic In Computer Science, Santa Barbara, Californie, USA, 2000. pp. 95-106.

[8] J.Desharnais, F.Laviolette , K.P.D. Moturu and S.Zhioua *Trace equivalence characterization through reinforcement learning* submitted to Canadian Artificial Intelligence Conference,Qubec.2005 p.12

[9] F. van Breugel, S. Shalit, and J. Worrell *Testing Labelled Markov Processes* In Proceedingsof 29th International Colloquium on Automata, Languages and Programming, volume 2380 of LNCS, pages 537-548, Springer-Verlag, 2002.

[10] M. Hennessy and R. Milner. *Algebraic laws for nondeterminism and concurrency.* Journal of the Association for Computing Machinery, 32(1): 137-161. 1985.

[11] Kim G. Larsen and Arne Skou. *Bisimulation through probabilistic testing* in. Information and Computation 94(1): 1-28, 1991.

[12] R. Milner. *Communication and conncurrency.* Prentice Hall, 1989.

[13] K. Narendra and M. Thathachar . *Learning Automata an introduction.* Prentice-Hall, 1989.

[14] D. Park. *Concurrency and automata on infinite sequences.* Lecture notes in Computer Science, 104, pages 167-183, 1981.

[15] R. Parr and S. Russell. *Approximating optimal policies for partially observable stochastic domains.* . In Proceedings of IJCAI-95, 1088-1094, 1995.

[16] G. Plotkin. *A structural approach to operational semantics.* Technical Report Daimi FN-19, Department of Computer Science, University of Aarhus, 1981.

[17] M. L. Puterman. *Markov decision processes-discrete stochastic dynamic programming.* John Wiley and sons, Inc. , New York, NY, 1994.

[18] R. S. Sutton and A. G. Barto. *Reinforcement learning. an introduction.* Cambridge, MA: The MIT Press, 1998.

[19] R. S. Sutton. *Reinforcement learning: past, present and future* [online]. Available from http: //www-anw. cs. umass. edu/ rich/Talks/SEAL98/SEAL98. html [accessed on December 2005]. 1999.

[20] R. S. Sutton. *Learning to predict by the method of temporal differences. Machine Learning*, 1988.

[21] R. S. Sutton. *Temporal credit assignment in reinforcement learning* PhD thesis, University of Massachusetts, Department of Computer and Information Science, 1984.

[22] C. Stirling. *Local model checking games.* In Proceedings of the 6th International Conference on Concurrency Theory (CONCUR'95), volume 962 of LNCS, pages 1-11. Springer-Verlag, 1995.

[23] G. J. Tesauro *Temporal difference learning and TDGammon.* Communications of the ACM, 38: 5868, 1995.

[24] C. J. C. H. Watkins. (*Learning from delayed rewards* PhD thesis, King's College, Cambridge, 1989.

[25] V. van Glabbeek, S. Smolka, B. Steffen and C. Tofts. *Reactive, generative and stratified models for probabilistic proceses.* In Proceedings of the 5th Annual IEEE Symposium On Logic In Computer Science, 1990.

[26] W. Zhang and T. Dietterich 1996. *High-performance job-shop scheduling with a time-delay TDλ network.* Advances in neural information processing systems 8: 10241030.

Appendix A

Additional empirical results

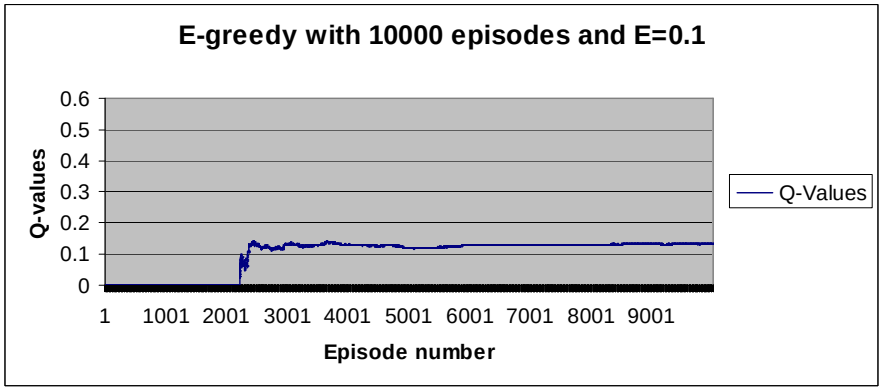


Figure A.1: ϵ-greedy with 10000 episodes and Q(0, as)=0. 132

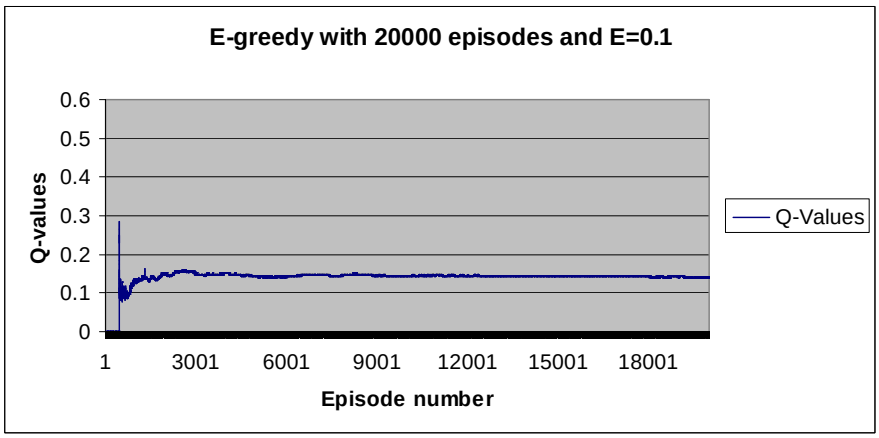


Figure A.2: ϵ-greedy with 20000 episodes and Q(0, as)=0. 140

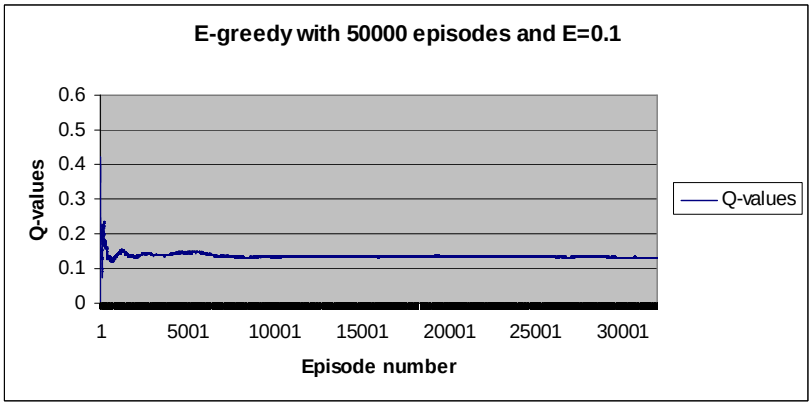


Figure A.3: ϵ-greedy with 50000 episodes and Q(0, as)=0. 131

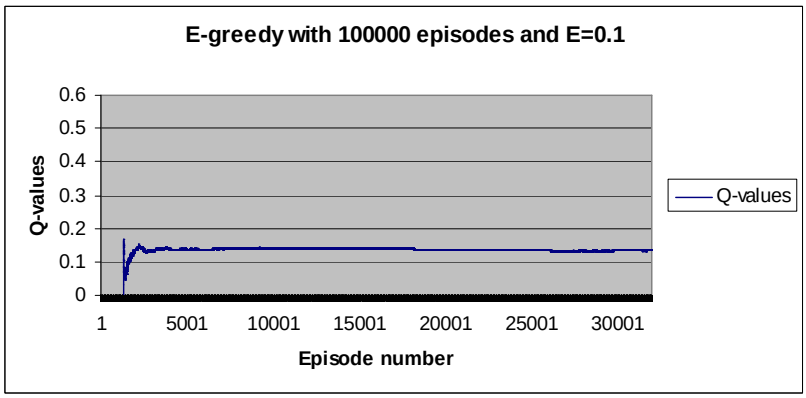


Figure A.4: ϵ-greedy with 100000 episodes and Q(0, as)=0. 134

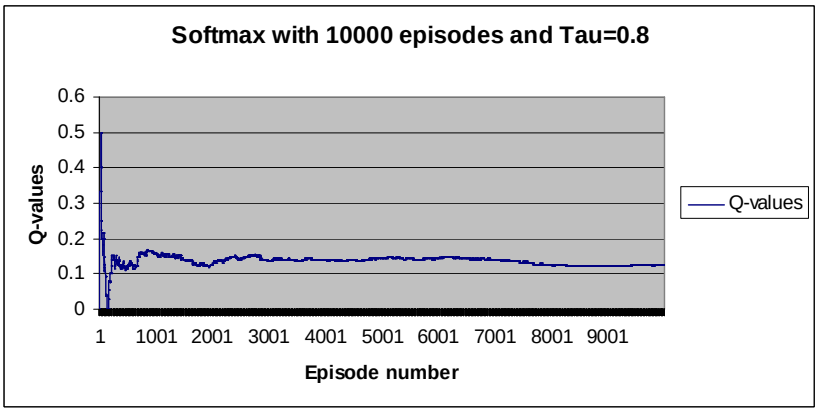


Figure A.5: Softmax with 10000 episodes and Q(0, as)=0. 127, Mean=0. 13662416 and Empirical standard deviation=0. 0195586

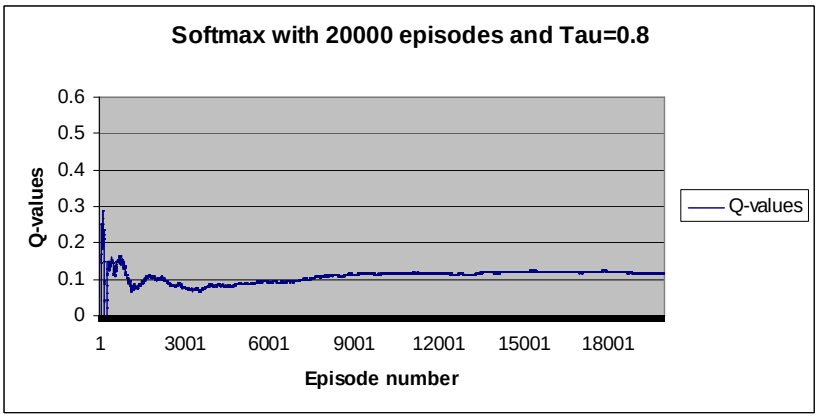


Figure A.6: Softmax with 20000 episodes and Q(0, as)=0. 114, Mean=0. 10683276 and Empirical standard deviation=0. 01988071

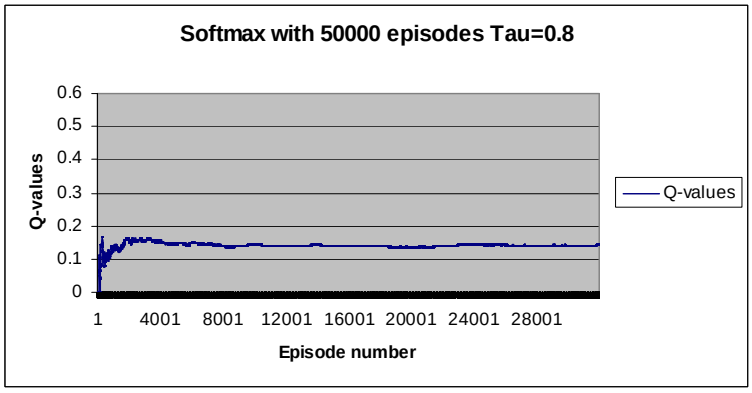


Figure A.7: Softmax with 50000 episodes and Q(0, as)=0. 114, Mean=0. 1400751 and Empirical standard deviation=0. 01493362

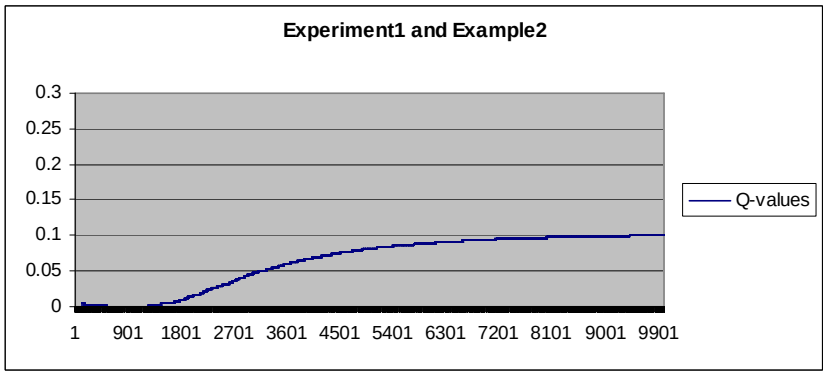


Figure A.8: Q(0, as)=0. 10025316

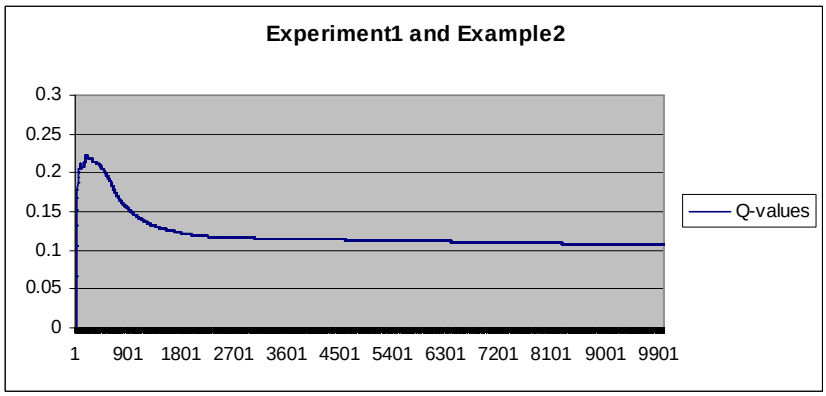


Figure A.9: Q(0, as)=0. 10866452

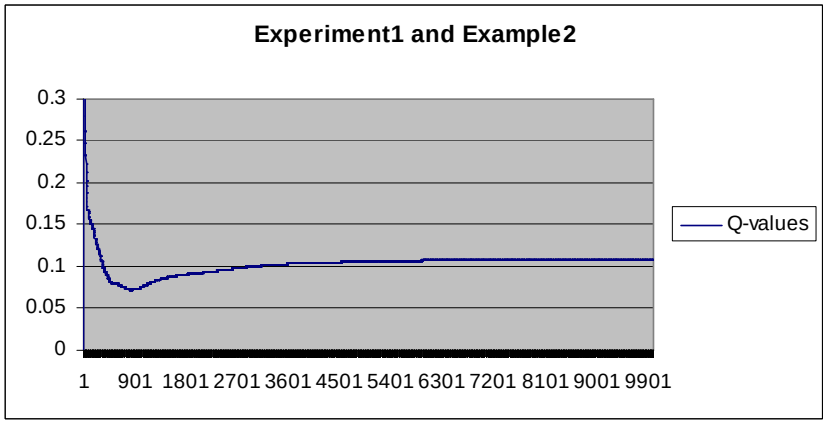


Figure A.10: Q(0, as)=0. 10385296

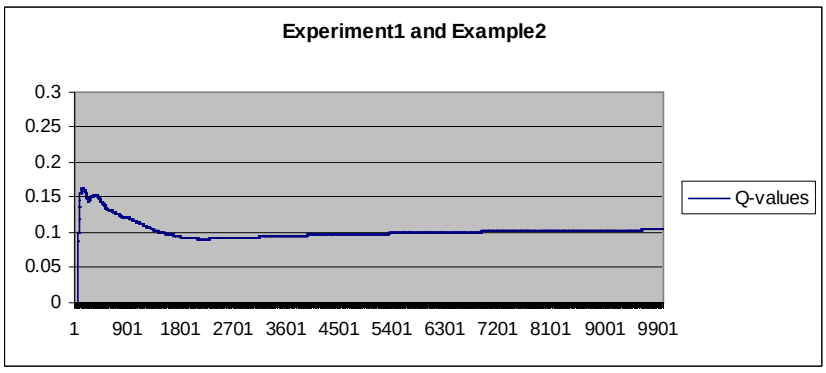


Figure A.11: Q(0, as)=0. 09783903

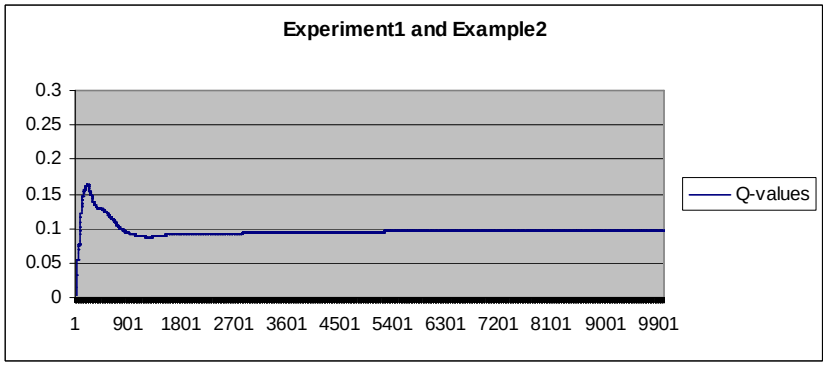


Figure A.12: Q(0, as)=0. 10807495

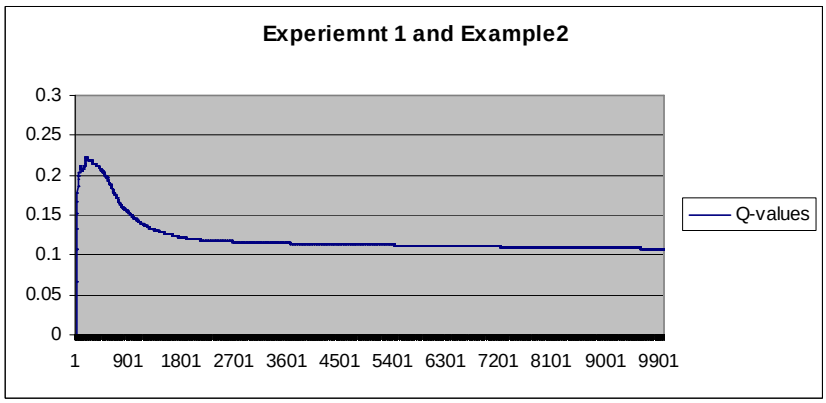


Figure A.13: Q(0, as)=0. 10673604

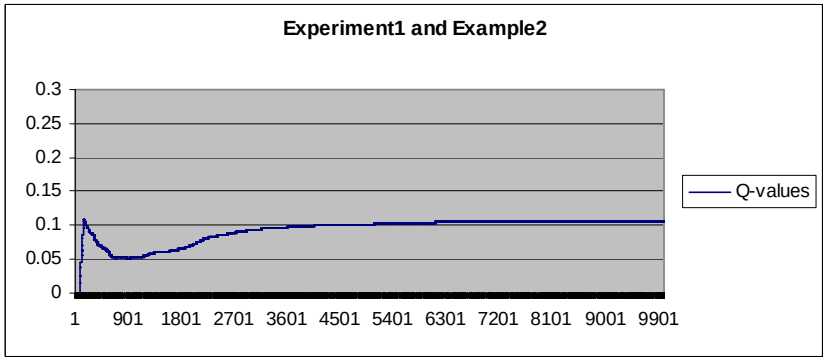


Figure A.14: Q(0, as)=0. 10414697

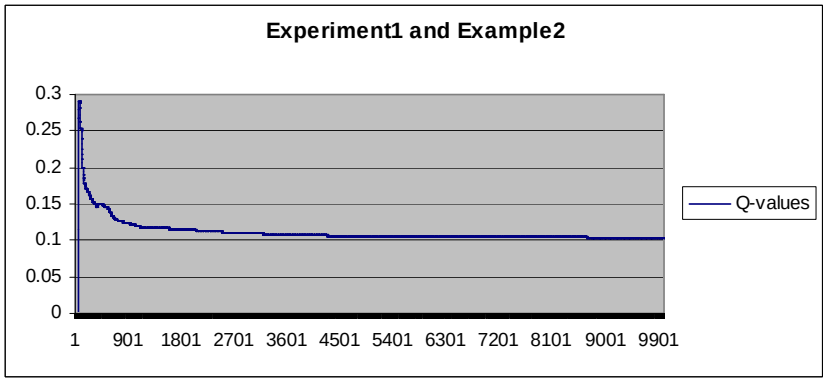


Figure A.15: Q(0, as)=0. 07871203

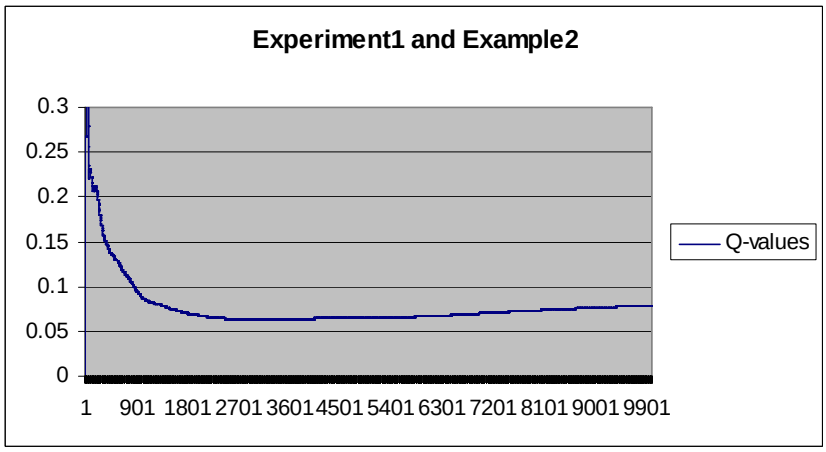


Figure A.16: Q(0, as)=0. 08872893

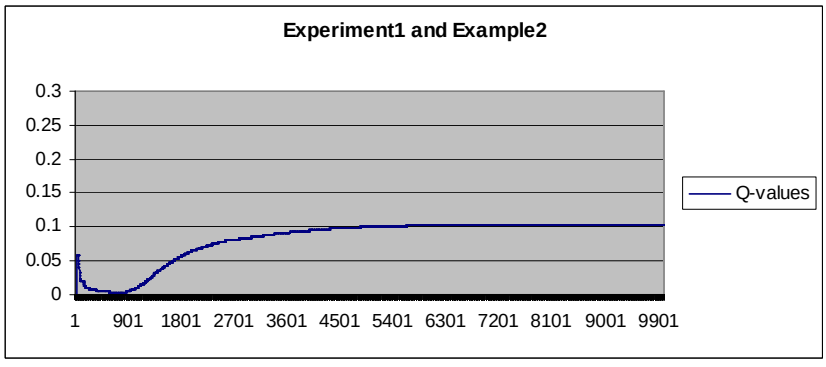


Figure A.17: Q(0, as)=0. 10386524

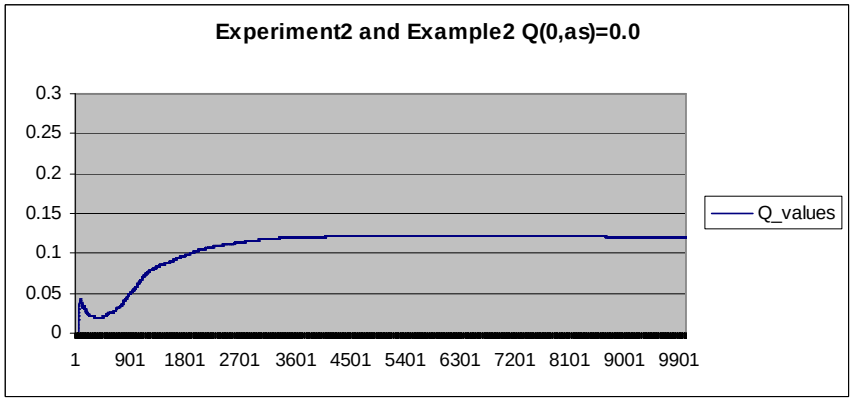


Figure A.18: Q(0, as)=0.10386524