USING REINFORCEMENT LEARNING FOR DYNAMIC LINK SHARING PROBLEMS UNDER SIGNALING CONSTRAINTS

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL AND

ELECTRONICS ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCES

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Nuri Çelik

May 2003

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Nail Akar (Supervisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Ömer Morgül

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Ezhan Karaşan

Approved for the Institute of Engineering and Sciences:

Prof. Dr. Mehmet Baray
Director of Institute of Engineering and Sciences

ABSTRACT

USING REINFORCEMENT LEARNING FOR DYNAMIC LINK SHARING PROBLEMS UNDER SIGNALING CONSTRAINTS

Nuri Çelik
M.S. in Electrical and Electronics Engineering
Supervisor: Assist. Prof. Dr. Nail Akar
May 2003

In static link sharing system, users are assigned a fixed bandwidth share of the link capacity irrespective of whether these users are active or not. On the other hand, dynamic link sharing refers to the process of dynamically allocating bandwidth to each active user based on the instantaneous utilization of the link. As an example, dynamic link sharing combined with rate adaptation capability of multimedia applications provides a novel quality of service (QoS) framework for HFC and broadband wireless networks. Frequent adjustment of the allocated bandwidth in dynamic link sharing, yields a scalability issue in the form of a significant amount of message distribution and processing power (i.e. signaling) in the shared link system. On the other hand, if the rate of applications is adjusted once for the highest loaded traffic conditions, a significant amount of bandwidth may be wasted depending on the actual traffic load. There is then a need for an optimal dynamic link sharing system that takes into account the tradeoff between signaling scalability and bandwidth efficiency. In this work, we introduce a Markov decision framework for the dynamic link sharing system, when the desired signaling rate is imposed as a constraint. Reinforcement learning

methodology is adopted for the solution of this Markov decision problem, and the results demonstrate that the proposed method provides better bandwidth efficiency without violating the signaling rate requirement compared to other heuristics.

Keywords: Link Sharing, Reinforcement Learning, Markov Decision Processes, Dynamic Link Sharing, Dynamic Programming.

ÖZET

SİNYALLEŞME KISITLAMALARI ALTINDA DİNAMİK LİNK PAYLAŞIMI PROBLEMLERİNİN GÜÇLENDİRMELİ ÖĞRENME METODUYLA ÇÖZÜLMESİ

Nuri Çelik
Elektrik ve Elektronik Mühendisliği Bölümü Yüksek Lisans
Tez Yöneticisi: Assist. Prof. Dr. Nail Akar
Mayıs 2003

Statik link paylaşım sistemlerinde, kullanıcılara, aktif olup olmadıklarına bakılmaksızın, linkin sabit bir bantgenişliği pay edilir. Öte yandan, kullanıcılara, linkin o andaki kullanım durumuna göre dinamik bir bantgenişliği verilmesine dinamik link paylaşımı denmektedir. Örneğin, dinamik link paylaşımının çokluortam uygulamalarının hız uyarlama becerisiyle bir arada kullanılması, HFC ağlarında ve genişbant kablosuz ağlarda yeni bir hizmet niteliği (QoS) yapısı sağlamaktadır. Dinamik link paylaşımında, ayrılmış bantgenişliğinin çok sık değiştirilmesi, link paylaşım sisteminde mesaj yoğunluğuna ve işlemci gücü harcanmasına neden olduğundan bir ölçeklenme sorununa yol açar. Diğer taraftan, uygulamaların hızları, bir kereye mahsus olarak en kötü şartlara göre ayarlanırsa, trafik yüküne bağlı olarak bantgenişliğinin önemli bir bölümü boşa harcanabilir. Bu yüzden, sinyalleşme oranı ve bantgenişliğinin verimli kullanılması arasındaki ödünleşimi gözönüne alan optimal bir link paylaşım sistemine ihtiyaç vardır. Bu çalışmada, dinamik link paylaşımı için, sinyalleşme oranının bir kısıtlama olarak belirtildiği bir Markov karar verme yapısı önerilmektedir. Bu Markov karar verme probleminin çözümü için güçlendirmeli öğrenme metodu seçilmiştir.

Sonuçlara göre, önerilen metod sinyalleşme oran kısıtlamalarını bozmadan, diğer buluşsallara (heuristic) göre daha yüksek bir bantgenişliği kullanım verimliliği göstermiştir.

Anahtar kelimeler: Link Paylaşımı, Güçlendirmeli Öğrenme Metodu, Markov Karar Verme Süreçleri, Dinamik Kanal Paylaşımı, Dinamik Programlama.

ACKNOWLEDGMENTS

I gratefully thank my supervisor Assist. Prof. Dr. Nail Akar for his supervision, guidance, and suggestions throughout the development of this thesis.

Contents

List of Figures

List of Tables

Table of Abbreviations

a	Action	54
AB	Average bandwidth	66
AH	Argiriou Heuristic	68
ALR	Average loss rate	48
APSR	Average policy switching rate	100
ARR	Average renegotiation rate	48
ATM	Asynchronous transfer mode	1
AUR	Average update rate	66
b	Bandwidth cost	44
BE	Bandwidth efficiency	48
bs	Buffer size	59
bs	Bucket size	73
CBR	Constant bit rate	7
CE	Cisco estimator	94
CPU	Central processing unit	4
DCM	Darken-Chang-Moody	26
DLS	Dynamical link sharing	3
DP	Dynamic programming	6
DUR	Desired update rate	78
EB	Erlang B formula	67
FA	Function approximation	61
HF	High-frequency	102
HFC	Hybrid fiber coax	2
IETF	Internet Engineering Task Force	1
IP	Internet protocol	6

ITU-T	International Telecommunication Union ……..	1
l	Loss cost ……	44
LF	Low-frequency ……..	102
LMS	Least mean squares ……...	34
MaxUR	Maximum update rate ……...	94
Mbit	Megabit ……	43
Mbps	Megabits per second …….	3
MDP	Markov decision process ……...	12
MPEG	Motion Picture Experts Group ……..	40
MSE	Mean square error ……	34
PVP	Permanent virtual path ……..	68
QoS	Quality of service ……..	1
RCBR	Renegotiated constant bit rate ……...	41
RL	Reinforcement learning ……	6
RL-CE	Cisco estimated reinforcement learning ……	94
RL-E	Estimate reinforcement learning ……...	94
RL-W	Worst case reinforcement learning ……...	94
RVI	Relative value iteration …….	47
s	Set-up cost …….	44
SMDP	Semi-Markov decision process …….	21
STC	Search then converge ……...	26
SVC	Switched virtual circuit …….	67
TD	Temporal difference ……..	23
U/H	Updates per hour ……...	78
VBR	Variable bit rate …….	39
VI	Value iteration ……...	47
VUR	Variance of update rate …….....	94

To My Family ...

Chapter 1

Introduction

User demands for multimedia applications and services are increasing at a rapid pace with unprecedented growth of the Internet. Multimedia applications have strict Quality of Service (QoS) requirements in terms of guaranteed bandwidth, delay, jitter, loss etc. New standards and QoS architectures are being developed by international standards organizations (ATM, ITU-T, IETF, etc.,) in order to provide these QoS requirements.

Efficient transport of multimedia applications requires new network capabilities such as:

- Packet scheduling mechanisms to prioritize multimedia traffic [1] and [2].
- Call admission control to limit the number of multimedia streams on a given link [3], [4], [5], and [6].
- Traffic shaping to limit the rate of multimedia streams injected towards the network [7].
- QoS routing protocols to find the best possible path satisfying the QoS requirements of the multimedia call [8] and [9].

In this thesis, we concentrate on another capability in which multimedia applications can adapt their rates to changing network conditions. Such an adaptation capability provides a promising means for using network resources efficiently while providing the required application QoS [10]. Several layers of the network protocol stack can be responsible for rate adaptation [11]. In this thesis, adaptation at the application layer is considered which means that the application is capable of adjusting its bandwidth (rate) requirement. In this work, video streaming is selected as a type of multimedia application thus rate adaptation can be achieved by various coding techniques such as layered coding [12], [13] and adaptation of compression parameters [14], [15], [16], as well as bandwidth smoothing [14], etc. Particularly, wavelet coding [16] is most suitable for continuous rate adaptation. When video streaming applications change their rates, the perceived quality of the application shows some variability. Applications such as video teleconferencing, interactive training, low-cost information distribution such as news can tolerate this variability.

Rate adaptation capability of multimedia applications is crucial in shared-link environments such as Hybrid Fiber Coax (HFC) and broadband wireless networks. The number of active users that share the links of such networks, changes randomly in time. The link sharing problem is controlling the bandwidth usage of users sharing a single link. There can be different classes of users sharing this link or there can be multiple classes of applications (real time, non-real time). This problem is explained by Jacobson [17], and Floyd and Jacobson [18] in detail. In *static link sharing*, all users are assigned a bandwidth share of the link capacity once and for the worst conditions, whereas dynamic link sharing refers to the process of allocating bandwidth to each active user based on the instantaneous utilization (i.e., instantaneous number of active users) of the given link.

The advantage of dynamic link sharing is that when the number of active users is low then they can share all the available bandwidth of the link and receive high bandwidth. The disadvantage is that as the number of users that share the same link increases, the perceived quality of multimedia applications reduce significantly when the system is left uncontrolled. However large system utilization and acceptable quality of reception can be provided when proper admission control along with application rate control is used. In this approach, when the number of active users is small, applications achieve their maximum requested rate. When the system load increases, the application transmission rate is reduced, while still remaining within acceptable levels, so that more connections can be admitted. Let's make these points clear by an example, consider a link with capacity 2 Mbps. Assume that the minimum bandwidth requirement of the users is 0.2 Mbps and the maximum is 0.5 Mbps. In static link sharing, all users are assigned 0.5 Mbps (or 0.2 Mbps) each, irrespective of the number of active users, then the system can only have 4 users maximum (or 10 users having the minimum required bandwidth). On the other hand in dynamic link sharing, the assigned bandwidth can change according to number of active users in the system therefore the instantaneous number of users can go as high as 10. Consider the case when the number of active users increases from 4 to 5; the static sharing system would not permit this user into the system, when the maximum required bandwidth is assigned to each user. On the other hand, the dynamic link sharing system would allow this user and reduce the bandwidth share of each user from 0.5 Mbps to 0.4 Mbps.

The rate control mechanism mentioned above works as follows: Controllers (headend in HFC networks [19] and base stations in wireless cellular networks) convey feedback to the already running applications through the downstream channel requesting them to change their upstream rate according to network conditions. A dynamical link sharing (DLS) system having this working mechanism is shown in Figure 1.1. Rate adaptation (bandwidth update) is a change

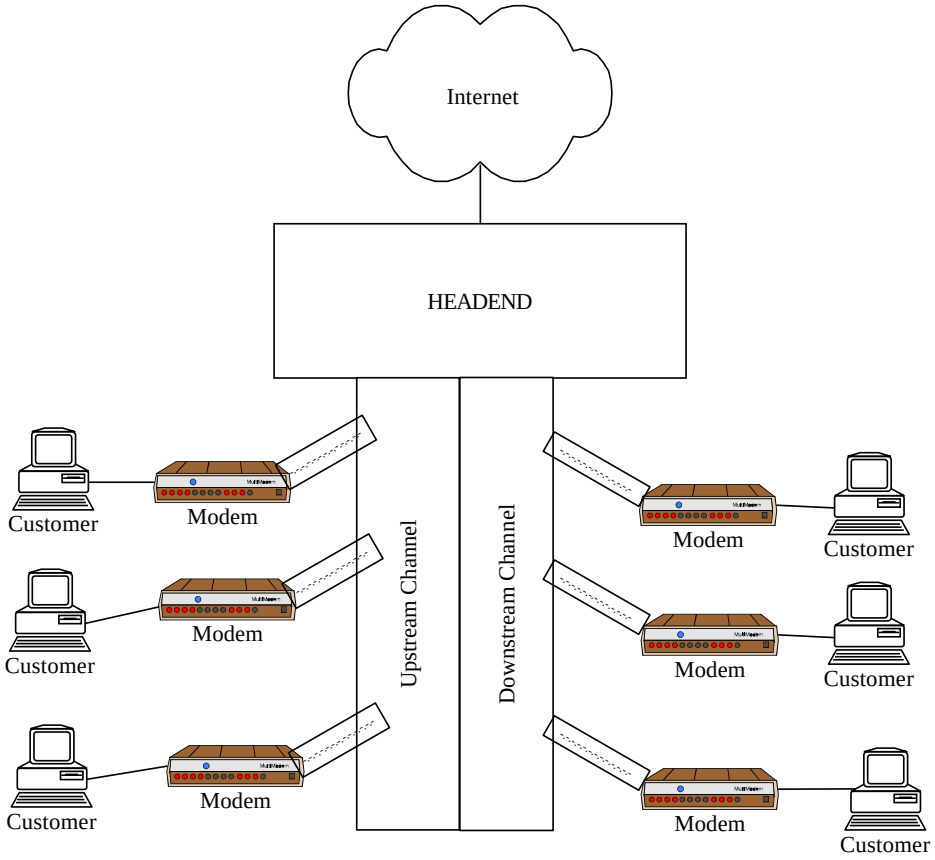

Figure 1.1: A typical link sharing system

in the encoding mechanism which can increase the CPU load on the end users. Moreover, each update means signaling between end users and the headend, thus the number of rate adaptations in unit time should be limited in order to reduce the CPU load on end users and signaling overhead on the downstream channel. Throughout this thesis, the load associated with each rate adaptation will be called *"signaling overload"*.

In this thesis, we study the optimal dynamic link sharing problem that considers the tradeoff between bandwidth efficiency and CPU load on end-users due to rate adaptation. In our proposal, the bandwidth of the given link is dynamically divided into a number of channels and each active user is then assigned a single channel. We assume that the channel bandwidths are variable but identical. At each user arrival or departure, the headend decides on the number of channels to set-up according to the number of active users.

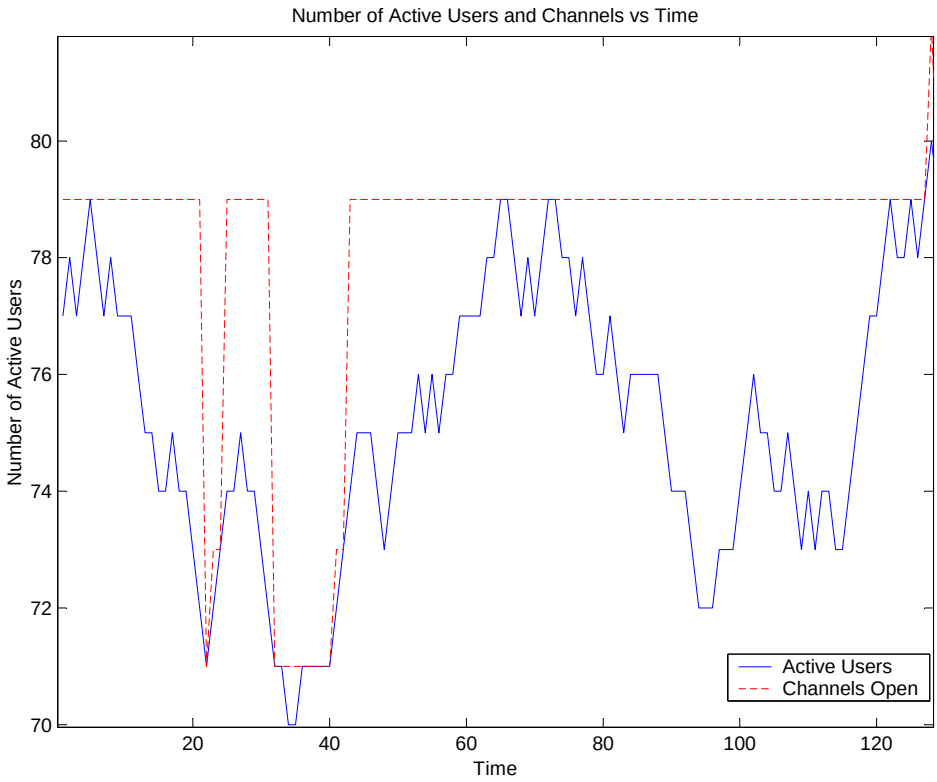

Figure 1.2: Dynamic link sharing example

In Figure 1.2, an example trace of our proposed rate adaptation algorithm is shown. The solid line shows the number of active users versus time, and the dashed line shows the number of channels allocated on the shared link. The bandwidth share of each user equals to total bandwidth divided by the number of channels. Our objective is minimizing the area between the dashed line and solid line while keeping the number of discontinuities in the dashed curve in unit time bounded by a predetermined value. We note that each discontinuity is associated with a new signaling message between the headend and the end-users thus creating signaling load. The area between the dashed line and the solid line is an indicator of how efficiently we use the shared line; if number of allocated channels on the link is much larger than the number of active users, the link is not efficiently utilized. The maximum efficiency is obtained when the number of channels is exactly equal to number of active users, however this means a larger number of discontinuities in the number of channels, resulting in a larger number of bandwidth updates in unit time. In our proposal, we attempt to maximize the

bandwidth utilization under signaling constraints, i.e., the maximum number of bandwidth updates is a-priori given as a constraint.

In dynamic link sharing problem, the headend decides on the number of channels to set-up at each arrival or departure instant according to number of active users, signaling rate, and number of channels allocated on the shared link. This problem can be formulated using average reward Markov decision process framework [20] which has been a popular paradigm for sequential decision making under uncertainty. Such problems can be solved by Dynamic Programming (DP) algorithms [20] which provides a suitable framework and algorithms to find optimal policies. Policy iteration and relative value iteration [20] are the most commonly used DP algorithms for average reward Markov decision problems. However, these algorithms become impractical when the underlying state-space of the Markov decision problem is large, leading to the so-called "curse of dimensionality". Recently, an adaptive control paradigm, the so-called "Reinforcement Learning" (RL) [21], [22] has attracted the attention of many researchers in the field of Markov decision processes. RL is based on a simulation scenario in which an agent learns by trial and error to choose actions that maximize the long-run reward it receives. RL methods are known to scale better than their DP counterparts [21].

The dynamic link sharing problem does not only arise in HFC networks for video streaming applications, but arises in other scenarios as well. This specific dynamic link sharing case is studied by [23]. This work finds the queueing solutions for certain dynamic link sharing systems. Moreover, a heuristic is proposed to reduce the signaling load on the network. We list below the other applications of dynamic link sharing problem.

- Voice over IP networks [24]. In these networks dynamical allocation of the capacity for the virtual path established between two voice over IP gateways is crucial. This capacity determines the number of voice calls that

the virtual path is capable of handling simultaneously; if one uses a fixed allocation and for the worst case conditions, then the allocated capacity would be idle in the non-busy hours. [24] proposes dynamically changing the capacity of the virtual path according to number of active users in the system which is very similar to the approach in this thesis. They have used reinforcement learning and dynamic programming algorithms in order to find the optimal capacity allocation scheme.

- Bandwidth brokers [25]. In this problem, Anjali et. al. tries to estimate the bandwidth to allocate between two differentiated services domains. A bandwidth broker acts as the resource manager for each network provider. Neighboring bandwidth brokers communicate with each other to establish inter-domain resource reservation agreements. If allocation follows the traffic demand very tightly, the resource usage is efficient but leads to frequent modifications of the reservations. This would lead to increased inter-bandwidth-broker signaling in order to propagate the changes to all the concerned networks. Contrarily, if large cushions are allowed in the reservations, the modifications are far spaced in time but the resource usage becomes highly inefficient. In [25], a Kalman filtering based scheme for estimating the traffic on an inter-domain link and forecasting its capacity requirement, based on a measurement of the current usage, is proposed.

- Renegotiated CBR [10]. This work studies the effect of adding renegotiation capability to Constant Bit Rate (CBR) links for the case of video transfer. The video sources show highly burstiness in terms of bit rate in slow time scales. If the capacity of the CBR link is adjusted once and for the highest bit rate, resource usage becomes highly inefficient. On the other hand, large buffering requirements occur when the capacity is adjusted to the mean traffic rate. In [10], the capacity of the CBR link is renegotiated based on incoming traffic rate and the buffer size. In the off-line case optimal

renegotiation schedule is found by a Viterbi-like algorithm. A heuristic is provided for the online case.

The contribution of this thesis is as follows. To the best of our knowledge, [24] is the only work to apply RL to the dynamic link sharing problem discussed above. In [24], RL and DP algorithms are used to find an optimal dynamic link sharing methodology but they only considered homogeneous traffic with small problem sizes. We extend the work [24] so that we apply RL to large sized problems and with nonhomogeneous traffic. To cope up with large state space dimensionality, we propose a function approximation algorithm based on delta-learning rule [26]. For tackling the nonhomogeneous traffic we introduce a new concept called "policy switching".

The organization of this thesis is as follows. Chapter 2 addresses Markov decision problems, reinforcement learning basics, and the related algorithms. The third chapter will be about formulation and RL implementation for the problem of resource allocation with buffering in order to demonstrate the convergence of RL algorithms. Moreover the results of the reinforcement learning approach to this introductory problem is given in the third chapter. Chapter 4 will include the formulation details and results of the dynamic link sharing problem with extensive numerical examples with large problem sizes and also covering the nonhomogeneous traffic case. We conclude in Chapter 5.

Chapter 2

Reinforcement Learning (RL)

2.1 Introduction

When we think about the nature of learning, the first thing that comes to mind is that we learn by interacting with our environment. An infant has no explicit teacher but learns many things by interacting with the environment with its sensori-motor. Using this interaction reveals information about cause and effect, and consequences of actions. This interaction with the environment is our major source of information throughout our lives. Learning from interaction is the basic idea for almost all theories of learning and intelligence.

Reinforcement learning is the process of learning how to map situations to actions in order to maximize the reward. The learner is not told which actions to take unlike most forms of machine learning, instead it must discover which actions to take by trial and error and by the help of rewards. In some extreme cases of learning, the actions may not only affect the immediate reward but also the subsequent rewards which is called *delayed reward*. The important distinguishing characteristics of reinforcement learning is *trial and error search* and *delayed reward*.

Figure 2.1: Learning agent environment interface

The learning agent environment interface is shown in figure 2.1. Here the learning agent takes an action a_t in state S_t, the environment responds the agent by taking it to another state S_{t+1} and also by giving an immediate reward r_t. The immediate reward is a reinforcement to learning-agent, that is why this method is called reinforcement learning. Because reinforcement learning methods combine neural network methods with dynamic programming algorithms, some authors call these methods "Neuro-Dynamic Programming" methods.

Let's consider an example of reinforcement learning task in order to have a full understanding of the concept. Tetris game is an example task in which one decides on the horizontal position and rotation of a falling object according to the status of the grid. The grid is empty initially and filled up with falling bricks of different shape. The falling bricks are of different shapes and at each instant, a random shape from the set is chosen. When one makes a full row of bricks, the row is destroyed and points are awarded. The objective is having the maximum points before the brick level reaches to top, therefore one should avoid the actions leaving an empty point in a row. In this example, the state of the system is the shape of the falling object and the state of the grid (fullness or emptiness of grid elements) and our action is the horizontal positioning and rotation of the falling object. The environment responds to us by giving rewards when we destroy a row. The actions one would choose would be the ones that avoids leaving empty points in a row and destroys as much rows as possible.

Tetris is a simple example that involves interaction between a decision maker and its environment in which the decision maker tries to achieve a goal despite uncertainty about its environment. At the same time, the effects of actions can not be fully predicted. The goal is explicit in the sense that decision maker can progress toward its goal using its senses. The decision maker can use its experience to improve its performance over time, the player plays better with experience.

There are six main sub-elements of a reinforcement learning system:

Learning agent This is the decision maker. The learning agent takes actions, explores the environment and finds a proper way of acting in order to maximize the reward it receives from environment.

Environment This is the source of feedback to the learning agent. The environment gives feedback to the agent in terms of a reward according to actions of the learning agent, the grid for tetris game is an example environment.

Policy Policy defines the learning agent's action at a given time and situation. A policy is a mapping from perceived states of the environment to actions to be taken in those states.

Reward function Reward function maps the state-action pairs to a single number denoting the reward that would be gained by taking that action. A learning agent's objective is to maximize the total reward received in the long run. As a result the reward function defines what is good and what is bad in the immediate sense, pleasure and pain is the biological system analogy of reward function.

Value function A value function specifies what is good in the long run. The value of a state is the total amount of expected reward that can accumulate over the future starting from that state. Values indicate the long term desirability of environmental states after taking into account the states

that are likely to follow and rewards available in those states. For example a state may yield a high reward but the following states can be very low reward yielding states.

Model of the environment Given a state and action, the model might predict the resultant next state and next reward. Models are used for planning, deciding on a course of actions by considering possible future situations before they are experienced.

If the decisions and actions of a task are dependent only on the current state and not on previous states, than that task satisfies *Markov property*. Reinforcement learning tasks that satisfy Markov property are called *Markov decision processes*, now it is time we should have a look at Markov decision processes and their solution methods.

2.2 Markov Decision Processes (MDP)

A Markov Decision Process consists of a set of states denoted by S and a set of actions denoted by A for moving between the states. Associated with each action a, there is a state (probability) transition matrix *P(a)*, where $P_{xy}(a)$ represents the probability of moving from state x to y under action a. There is also a reward or payoff function $r{:}S{\times}A \longrightarrow R$ where *r(x,a)* is the expected reward for doing action a in state x.

A policy is a mapping $\pi{:}S{\rightarrow}\ A$ from states to actions. This policy is both stationary and deterministic. Any policy induces a state transition matrix $\mathrm{P}(\pi)$, where $P_{xy}(\pi) = P_{xy}(\pi(x))$. Thus any policy yields a Markov chain $(S, P(\pi))$. Before going into details of MDPs, it is appropriate to describe some important terms about MDPs.

Two states x and y ***communicate*** under a policy π if there is a non-zero probability of reaching each state from the other. A state is ***recurrent*** if starting from the state, the probability of eventually reentering it is 1. A non-recurrent state is called ***transient***, since at some finite point in time the state will never be visited again.

An ***ergodic*** or ***recurrent*** class of states is a set of recurrent states that all communicate with each other, and do not communicate with any state outside this class. If the set of all states forms an ergodic class, the Markov chain is said to be ***irreducible***.

An ergodic or recurrent MDP is one where the transition matrix corresponding to every policy has a single recurrent class. An MDP is termed ***unichain*** if the transition matrix corresponding to every policy contains a single recurrent class, and a set of transient states. In this work, we will focus on algorithms dealing with unichain MDPs.

2.2.1 Gain and bias optimality

In average reward MDPs, the purpose is to compute policies that yield the highest expected reward per step. The policy that maximizes the average reward over all states is called *gain optimal policy*. The average reward $\rho^{\pi}(x)$ associated with a particular policy π at a state x is defined as

$$\rho^{\pi}(x) = \lim_{N \to \infty} \frac{E\left(\sum_{t=0}^{N-1} R_t^{\pi}(x)\right)}{N} \tag{2.1}$$

where $R_t^{\pi}(x)$ is the received reward at time t starting from x and actions are chosen using policy π. $E(\cdot)$ denotes the expected value. If π^* denotes gain optimal policy, then $\rho^{\pi^*}(x) \geq \rho^{\pi}(x)$ over all policies π and states x.

A key point simplifying the design of average reward algorithms is that the average reward for any policy is independent from starting state for unichain MDPs. In mathematical terms:

$\rho^{\pi}(x) = \rho^{\pi}(y) = \rho^{\pi}$.

The reason for this lies in the unichain assumption. Unichain policies create a single recurrent class of states, and a set of transient states. Transient states will be visited for once and recurrent states will be visited forever, as a result the average reward can not differ across recurrent and transient states. The effect of transient states will vanish in the limit, because a finite reward will be accumulated because of them until entering a recurrent state and its effect will diminish in the long run.

In the problems with absorbing states (in ones our purpose is reaching some specific state like treasure hunt), all of the policies reaching the goal will have the same average reward and be gain optimal. The policy reaching the goal in least number of steps is called the *bias optimal policy*[27].

2.2.2 Bellman Equation

Bellman optimality equation is one of the fundamental results showing the existence of an optimal policy for an MDP when some requirements are met.

$$V^*(x) = \max_a \left[r(x,a) + \gamma \sum_y \left(P_{xy}(a) \times V^*(y) \right) \right] \tag{2.2}$$

$$V^*(x) + \rho^* = \max_a \left[r(x,a) + \sum_y (P_{xy}(a) \times V^*(y)) \right] \tag{2.3}$$

Equation 2.2 is the discounted Bellman optimality equation. $V^*(x)$ represents the value of state x. The value of a state is the total maximum average reward one

can get beginning from that state. Intuitively, the value of a state is immediate reward of action a plus the expected value of the next state. The summation on the right hand side is an expectation calculation, estimating the value of possible next state. γ is the discount factor which is chosen as less than 1 and shows the importance of next state relative to current state.

Equation 2.3 is average reward version of Bellman optimality equation. ρ^* is the average one step reward. The value of current state plus the reward for one step should be equal to immediate reward plus the expected value of next state. Now let's turn our attention to Bellman Theorem; for a proof see [28]:

Theorem 1: *For any MDP that is either unichain or communicating, there exists a value function V^* and a scalar ρ^* satisfying Equation 2.3 such that the greedy policy π^* resulting from V^* achieves the optimal average reward $\rho^* = \rho^{\pi^*}$ where $\rho^{\pi^*} \geq \rho^{\pi}$ over all policies π.*

Greedy policy mentioned in Theorem 1 is the policy constructed by choosing the actions maximizing the right hand side of Equation2.3.

There are various methods for computing the optimal reward policies, now let's discuss some of them.

2.3 Dynamic Programming Algorithms

2.3.1 Policy Iteration

Policy iteration is introduced by Howard [29]. Policy iteration has two phases, policy evaluation and policy improvement.

1. Initialize $k = 0$ and set π^0 to some arbitrary policy

2. *Policy Evaluation*: Given a policy π^k, solve the following set of $|S|$ linear equations for the average reward ρ^{π^k} and relative values $V^{\pi^k}(x)$, by setting the value of a reference state $V(s) = 0$

$$V^{\pi^k}(x) + \rho^{\pi^k} = r(s, \pi^k(x)) + \sum_y \left(P_{xy} \pi^k(x) V^{\pi^k}(y) \right) \tag{2.4}$$

3. *Policy Improvement* Given a value function $V^{\pi^k}(x)$, compute an improved policy π^{k+1} by selecting an action maximizing the following quantity at each state,

$$max_a (r(x, a) + \sum_y \left(P_{xy}(a) V^{\pi^k}(y) \right)) \tag{2.5}$$

setting if possible, $\pi^{k+1}(x) = \pi^k(x)$

4. If $\pi^k(x) \neq \pi^{k+1}(x)$ for some state x, increment k and return to step 2.

$V(s)$ is set to 0 because there are $|S| + 1$ unknown variables but only $|S|$ equations. In [29], it is shown that this algorithm would converge in finitely many steps to give a gain-optimal policy.

2.3.2 Value Iteration

The policy iteration algorithm requires solution of $|S|$ linear equations at every iteration, this becomes computationally complex when $|S|$ is large. An alternative solution methodology is to iteratively solve for the relative values and average reward. These algorithms are called *value iteration methods* in dynamic programming literature.

The right hand side of Bellman equation is as follows:

$$max_a (r(x, a) + \sum_y \left(P_{xy}(a) V^{\pi^k}(y) \right)) \tag{2.6}$$

Let us denote the mapping in Equation 2.6 by $T(V)(x)$, this mapping is *monotone* which is an important property used in the proof of this algorithm. The value iteration algorithm is as follows:

1. Initialize $V^0(t) = 0$ for all states t, and select an $\epsilon > 0$. Set $k = 0$.
2. Set $V^{k+1}(x) = T(V^k)(x)\ \forall x \in S$
3. If $sp(V^{k+1} - V^k) > \epsilon$, increment k and go to step 2
4. For each $x \in S$, choose $\pi(x) = a$ to maximize $\Big(r(x,a) + \sum_y P_{xy}(a)V^k(y)\Big)$

In step 3, the sp denotes *span semi-norm* function which is $sp(f(x)) = max_x(f(x)) - min_x(f(x))$.

In value iteration algorithm, the values $V(x)$ can grow very large and cause numerical instabilities. A more stable version proposed in [30] is called *relative value iteration* algorithm. This algorithm chooses one reference state and value of that reference state is subtracted from value of all other states in each step as shown below:

$V^{k+1}(x) = T(V^k)(x) - T(V^k)(s)\ \forall x \in S$

2.3.3 Asynchronous Value Iteration

Both value iteration and policy iteration are synchronous algorithms because at each iteration whole state space is swept. Only the old values of the other states are used when updating the relative value of a state. It is shown that asynchronous version of value iteration given in previous section may diverge according to some conditions. However, there are some other asynchronous control methods, below an asynchronous control method by Jalali and Ferguson [31] is given.

1. At time step $t = 0$, initialize the current state to some state x, cumulative reward to $K^0 = 0$, relative values $V^0(x) = 0$ for all states x, and average reward $\rho^0 = 0$. Fix $V^t(s) = 0$ for some reference state s for all t. The expected rewards $r(x, a)$ are assumed to be known.

2. Choose an action a maximizing $\left(r(x, a) + \sum_y P^t_{xy}(a)V^t(y)\right)$

3. Update relative value $V^{t+1}(x) = T(V^t)(x) - \rho^t$, if $x \neq s$.

4. Update the average reward $\rho^{t+1} = \frac{K^{t+1}}{t+1}$, where $K^{t+1} = K^t + r(x, a)$

5. Carry out action a, and let the resulting state be z. Update the probability transition matrix entry $P'_{xz}(a)$ using a maximum-likelihood estimator. Increment t, set the current state x to z, and go to step 2.

The relative values are updated only when they are visited, as a result this algorithm is asynchronous.

2.4 Reinforcement Learning Algorithms

Note that in all of the algorithms below, average reward framework is adopted, as a result the objective is maximizing the average reward. If we use average cost framework, in which the objective is minimizing the average cost, we should replace all of the max terms with min terms.

2.4.1 Discounted Reinforcement Learning

In RL literature, most of the existing work is on maximizing the discounted cumulative sum of rewards. The discounted return of a policy π starting from a state x is defined as

$$V^\pi_\gamma(s) = \lim_{N\to\infty} E\left(\sum_{t=0}^{N-1} \gamma^t R^\pi_t(s)\right) \tag{2.7}$$

where $\gamma \leq 1$ is the discount factor, and $R_t^\pi(s)$ is the reward received at time step t starting from state s and choosing actions using policy π. An optimal discounted policy π^* maximizes the above value function over all states x and policies π.

The action value $Q_\gamma^\pi(x,a)$ denotes the discounted return obtained by performing action a in state x, and thereafter following policy π

$$Q_\gamma^\pi(x,a) = r(x,a) + \gamma \sum_y \left(P_{xy}(a) \max_{a \in A} Q_\gamma^\pi(y,a) \right) \tag{2.8}$$

Here $r(x,a)$ is the expected reward for doing action a in state x.

2.4.2 Q-Learning

In the aforementioned methods we had to have or estimate the probability transition matrix to calculate the value functions. Most reinforcement learning methods do not need any probability transition matrix to find the optimal policies. Q-Learning is proposed by Watkins [32] as an iterative method for learning Q values. All the $Q(x,a)$ values are randomly initialized to some value. At time step t, the learner either chooses the action a with the maximum $Q_t(x,a)$ value or selects a random "exploratory" action. If the agent moves from state x to state y and receives an immediate reward $r_{imm}(x,a)$, the current $Q_t(x,a)$ values are updated using the following rule:

$$Q_{t+1}(x,a) \leftarrow Q_t(x,a)(1-\alpha) + \alpha \left(r_{imm}(x,a) + \gamma \max_{a \in A} Q_t(y,a) \right) \tag{2.9}$$

where $0 \leq \alpha \leq 1$ is the learning rate controlling how quickly errors in action values are corrected. Q-learning asymptotically converges to the optimal discounted policy for a finite MDP. The convergence conditions are given in [33], shortly all state action pairs must be visited infinitely often, and the learning rate α must slowly decay to zero.

2.4.3 R-Learning

R-learning is an average reward RL technique proposed by Schwartz [34]. R-learning uses the action value representation like Q-learning. The action value $R^{\pi}(x, a)$ represents the average adjusted value of doing an action a in state x and then following policy π. That is,

$$R^{\pi}(x, a) = r(x, a) - \rho^{\pi} + \sum_{y} \left(P_{xy}(a) \max_{a \in A} R^{\pi}(y, a) \right) \tag{2.10}$$

Here ρ^{π} is the average reward of policy π. R-learning can be described by the following steps,

1. Set $t = 0$. Initialize all the $R_t(x, a)$ to 0. Let the current state be x

2. Choose an action a that has the highest $R_t(x, a)$, or with some probability choose a random exploratory action.

3. Carry out action a. Let the next state be y, and the reward be $r_{imm}(x, y)$. Update the R values and the average reward ρ using the following update rules:

$$R_{t+1}(x, a) \leftarrow R_t(x, a)(1 - \alpha) + \alpha \left(r_{imm}(x, y) - \rho_t + \max_{a \in A} R_t(y, a) \right) \tag{2.11}$$

$$\rho_{t+1} \leftarrow \rho_t(1 - \beta) + \beta \left(r_{imm}(x, a) + \max_{a \in A} R_t(y, a) - \max_{a \in A} R_t(x, a) \right) \tag{2.12}$$

4. Set the current state to y and go to step 2.

Note that ρ is updated when a non-exploratory action is chosen in step 3. Here α is the learning rate controlling how quickly the errors in the estimated action values are corrected and β is the learning rate for updating ρ. Both parameters should be less than 1 and should be properly decayed to zero throughout the learning process.

2.4.4 Gosavi's RL Algorithm

Unlike RL algorithms given above, this algorithm can be applied to Semi-Markov Decision Processes (SMDPs) in which the transition times between states are not constant. There are some proposed algorithms for solving semi-Markov average reward problems but their convergence proofs are not complete [35] or they are shown to diverge in some cases [36]. Gosavi's RL algorithm has a proof in [37] using method of ordinary differential equations. This algorithm is the first Average Reward RL algorithm that has a convergence proof.

1. Set number of iterations $m = 0$. Initialize action values $Q(s, a) = 0 \; \forall i \in S$ and $a \in A_s$. Set the cumulative cost $C = 0$, the total time be $T = 0$, and the cost rate $\rho = \rho^s$ (can be zero). Start system simulation.

2. While $m < MAX\_STEPS$ do

   If the system state at iteration m is $i \in S$,

   (a) Calculate p,α, and β using the iteration count m and the number of times state i has been visited.

   (b) With probability of $(1 - p)$, choose an action $u \in U_i$ that maximizes $Q(i, u)$, otherwise choose a random (exploratory) action from the set U_i

   (c) Simulate the chosen action. Let the system state at the next decision epoch be j. Also let $t(i, u, j)$ be the transition time, and $g(i, u, j)$ be the immediate cost incurred in the transition resulting from taking action u in state i.

   (d) Change $Q(i, u)$ using:

$$Q(i,u) \leftarrow (1-\alpha)Q(i,u) + \alpha\left(g(i,u,j) - \rho t(i,u,j) + \max_v Q(j,v)\right) \tag{2.13}$$

(e) In case an exploratory action was chosen in step 2(b) go to step 2(f), else

- Update total cost $C \leftarrow (1-\beta)C + \beta g(i,u,j)$
- Update total time $T \leftarrow (1-\beta)T + \beta t(i,u,j)$
- Calculate cost rate $\rho \leftarrow \frac{C}{T}$
- Go to step 2(f)

(f) Set current state i to new state j, and $m \leftarrow m+1$. Go to step 2(b)

For MDPs set all $t(i,u,j) = 1$. Calculation of p, α and β will be described in the next sections. In setting initial value of ρ, estimate of average reward will result in faster convergence.

2.5 Learning Rate Schedules

In the above sections, it is mentioned that learning rates α, β should be decreased to 0, in order for the algorithm to converge. The decreasing mechanism has a large effect on the rate of convergence of the algorithm. In this section, we will present some widely used methods for decreasing the learning parameters but first we will describe the need for decreasing these parameters. Consider the update rule for Q-learning:

$$Q_{t+1}(x,a) \leftarrow Q_t(x,a)(1-\alpha) + \alpha\left(r_{imm}(x,a) + \gamma \max_{a \in A} Q_t(y,a)\right) \tag{2.14}$$

By some modification Equation 2.14 becomes:

$$Q_{t+1}(x,a) \leftarrow Q_t(x,a) + \alpha\left(r_{imm}(x,a) + \gamma \max_{a \in A} Q_t(y,a) - Q_t(x,a)\right) \tag{2.15}$$

According to action-value representation definition, $Q(x,a)$ is the reward gained when action a is chosen in current step and optimal policy is applied afterwards. As a result $Q(x,a)$ should be equal to immediate cost (r_{imm}) plus

the maximum discounted reward of the next state ($\max_{a \in A} Q_t(y, a)$). As our objective is maximizing the average reward gained, we choose the actions with highest Q values, that is why the maximum of the Q values of next state is chosen. In Equation 2.15, it is seen that the difference of $Q(x, a)$ from its expected value is added to $Q(x, a)$ in order to decrease this difference. This is called a *"temporal difference (td) method"* in learning literature. Here the optimal value of $Q(x, a)$ is our target and we want to reach there by adding the differences to its value. In order to prevent oscillations around the target point, there is a learning rate α applied, which is chosen between 0 and 1. For better approximation to target point this α should be small, which increases the number of iterations required to reach the target value. The appropriate method is starting with a large α (close to 1) and reducing it to zero [38]. There are various methods of reduction, now let's survey most common ones.

2.5.1 Constant Learning Rate

The simple solution is taking learning rate to be constant which results in persistent residual fluctuations. The magnitude of such fluctuations and the resulting degradation of system performance are difficult to anticipate. Choosing a small value of learning rate (η) reduces the magnitude of fluctuations, but seriously slows convergence, whereas a large value can result in instability. An example will help our understanding of the problem. In this example the coefficients of a 4th degree polynomial will be found using delta-learning rule [26]. Delta-learning is a method used for fitting a function to samples coming from an unknown function. In this method, the coefficients of approximating function are updated according to difference between real and estimated values in order to minimize the mean square error.

In Figure 2.2 performance of different learning rates approximating the polynomial is shown, the learning lasted for 100000000 iterations. In this figure it

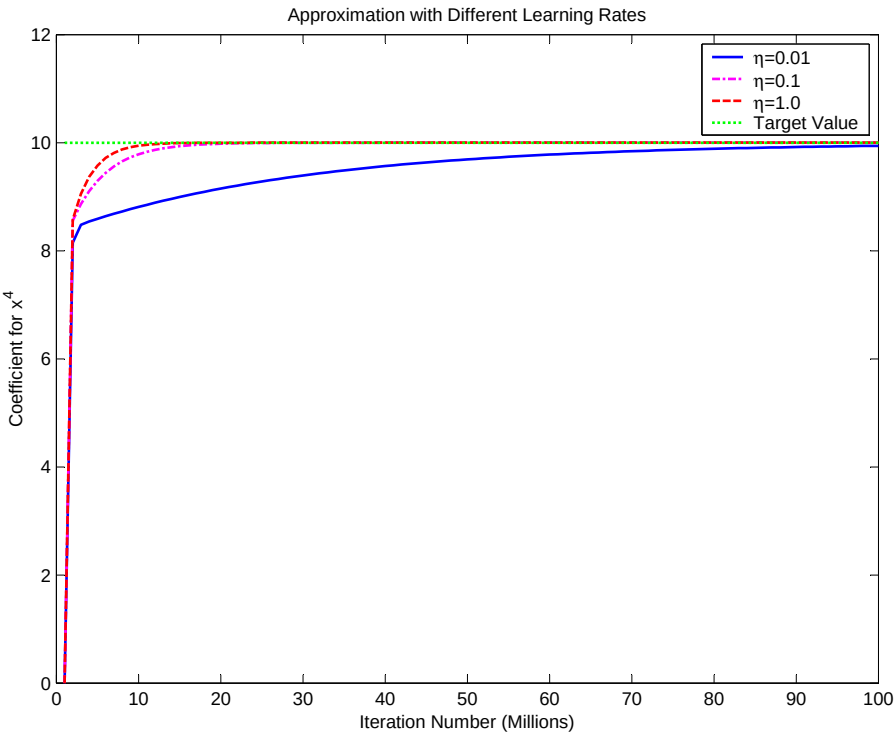

Figure 2.2: Approximation with different constant learning rates

Learning Rate (η)	Variance
0.01	$3.02x10^{-11}$
0.1	$4.94x10^{-26}$
1.0	$4.93x10^{-14}$

Table 2.1: Learning performance with different rates

is seen that all the algorithms have converged to target value $\eta = 1.0$ being the fastest. A closer examination of these plots will reveal more details about the effects of learning rates.

In Figure 2.3 the last 1000 values of approximated coefficients are drawn, we observe that the approximation with $\eta = 0.01$ haven't converged to target value yet. The approximation with $\eta = 0.1$ has converged to desired value and stayed there whereas $\eta = 1.0$ version is oscillating around the desired value. When one makes the number of iterations higher, the learning with $\eta = 0.01$ will eventually converge to the desired rate with a smaller error than bigger learning rate ones. The variance of last 1000 coefficients will be more meaningful in describing the convergence, which is shown in Table 2.1

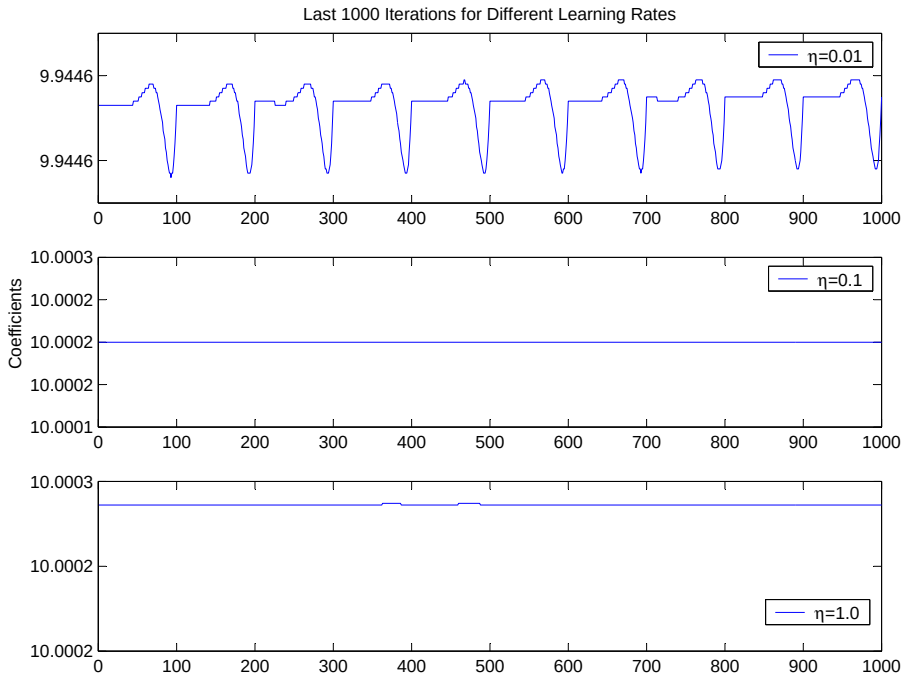

Figure 2.3: Approximation with different learning rates last 1000 iterations

From these results we can infer that $\eta = 0.1$ constant had better convergence result, but this may not be true when we consider convergence speed. A mechanism that will start with larger learning rate and later make that rate small will reduce the variance of coefficients while decreasing the number of iterations required for convergence.

2.5.2 Time Reciprocal Learning Rate

In this scheme, learning rate is taken to be a function that is inversely proportional with time. In this approach, $\eta = \frac{c}{t}$ is employed in which c is a constant and t is the time. This is the most widely used choice in stochastic approximation literature which typically results in slow convergence to bad solutions for small c and coefficients will blow-up for big c. For polynomial approximation example this scheme is not able to find the correct coefficient for $c = 1.0$ which is shown in Figure 2.4 because of high speed decreasing of the learning rate with number of iterations.

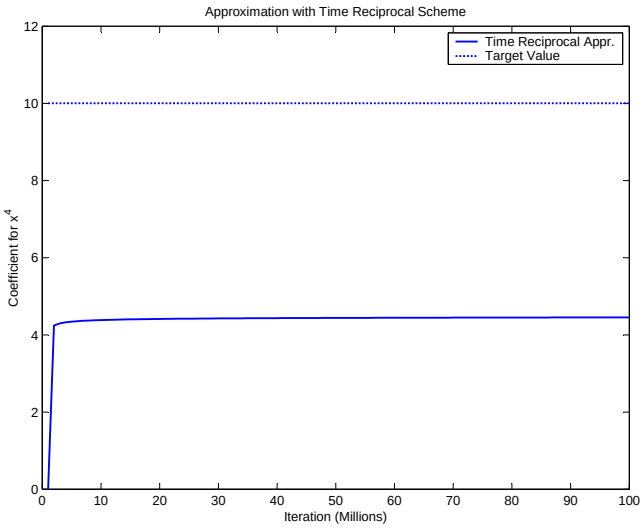

Figure 2.4: Approximation with time reciprocal learning rate

2.5.3 Darken Chang Moody (DCM) Scheme

Search Then Converge (STC) learning schedules are proposed for solution to the problems of escaping from metastable local minima, finding a "good" local minimum and achieving asymptotically optimal rate of convergence [39]. With STC schedules, η is chosen to be a fixed function of time, such as the following:

$$\eta(t) = \eta_0 \frac{1 + \frac{c}{\eta_0}\frac{t}{\tau}}{1 + \frac{c}{\eta_0}\frac{t}{\tau} + \tau\frac{t^2}{\tau^2}} \tag{2.16}$$

This function is approximately constant with value η_0 at times small compared to τ (the search phase). At times large compared with τ (the converge phase), the function decreases as $\frac{c}{t}$. This schedule has demonstrated a dramatic improvement in convergence speed and quality of solution compared to traditional learning schedules. This method combines the speed of high learning rates with accuracy of lower learning rates. Consider Figure 2.5 which shows the variance of learning parameter with number of iterations. Here $\eta_0 = c = 1.0$ and $\tau = 10^{15}$.

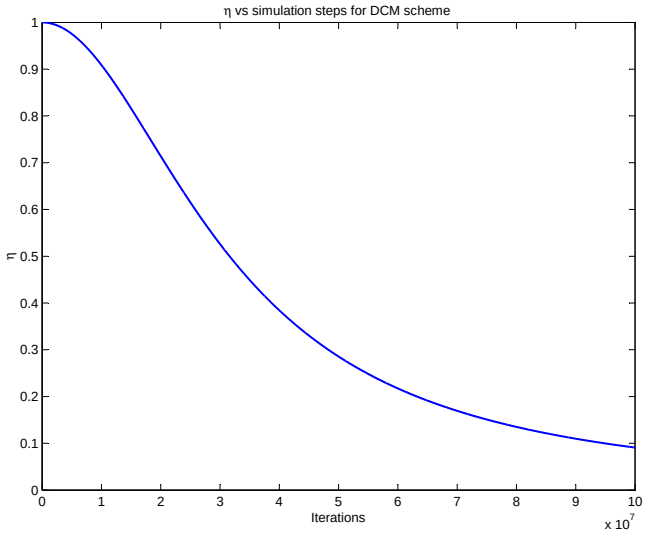

Figure 2.5: η variation with DCM scheme

Learning Rate (η)	Variance
DCM	$4.94x10^{-26}$
0.1	$4.94x10^{-26}$
1.0	$4.93x10^{-14}$

Table 2.2: Learning performance with different learning rate schedules

When we examine Figure 2.5, we infer that the learning result is as accurate as $\eta = 0.1$ constant case with a faster convergence. The learning process is drawn along with $\eta = 0.1$ constant case in Figure 2.6

It is seen that DCM scheme has the speed of $\eta = 1.0$ constant case, but the accuracy is the same with $\eta = 0.1$ constant case. The variance of last 1000 terms is the same for DCM and constant $\eta = 0.1$ case. This proves our initial assumption that DCM will combine the speed and accuracy of constant learning rate schemes.

As a conclusion, the variation in the learning rate brings the flexibility of choosing speed and accuracy during the learning process. If we have a constant learning rate, we have to sacrifice either speed of convergence or accuracy of learning, but with adaptive learning rate schedules we can have both of them and get more stable results.

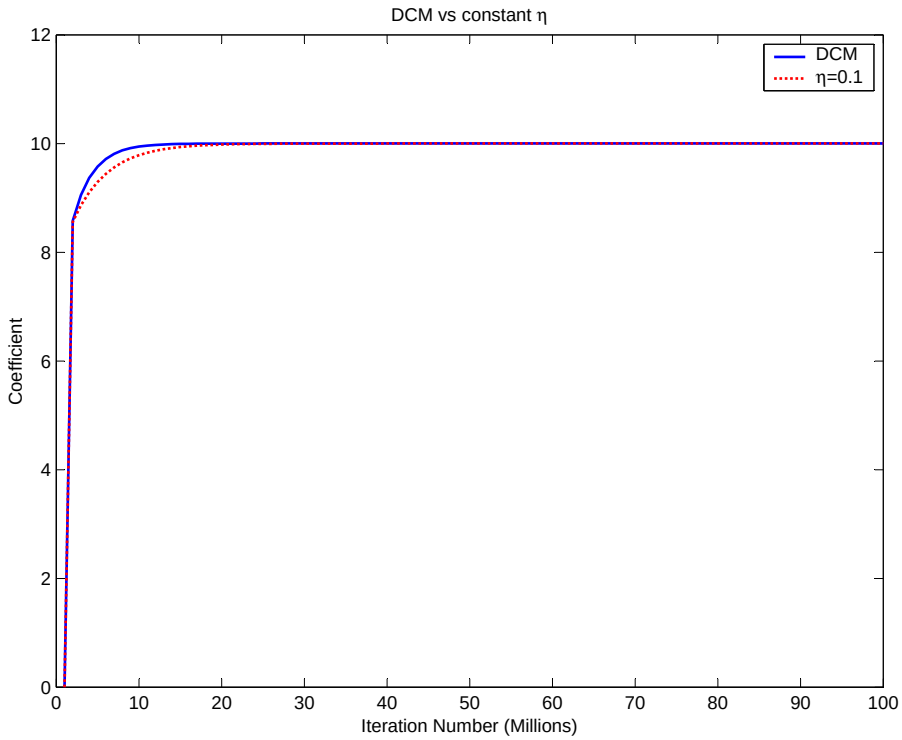

Figure 2.6: Performance of DCM learning scheme

2.6 Exploration

Exploration is essential for RL algorithms to converge. The convergence theorem of all reinforcement algorithms require that all state action pairs (x, a) are visited infinitely often [40]. An example would be helpful in understanding the importance of exploration in RL. Consider a 3-armed bandit problem, a bandit is a gambling machine in which one puts money and pull an arm, and with a probability earns some money. Figure 2.7 gives the Markov model for the bandit system, the numbers in parenthesis give the action number and associated reward, for example $(1, 2)$ means action 1 has a reward of 2.

Assume that Q-learning algorithm is used, and we have initialized all $Q(x, a)$ pairs to zero. In the first iteration, we are in state 0, we choose action 1 and have a reward of 2. This means $Q(0, 1)$ becomes a positive number according to update rule 2.9. In the next step, we end up in state 0, the maximum Q value is associated with action 1 because it is a positive number whereas the

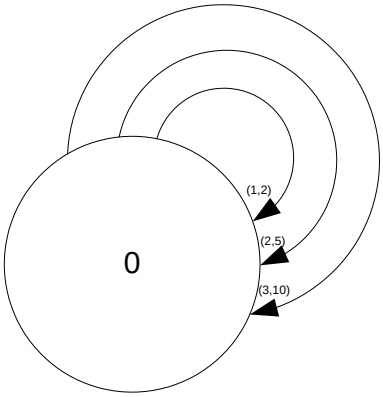

Figure 2.7: Markov model for 3-armed bandit problem

others are 0, so we again select the same action. The result is we will never have the chance of trying other actions because of our greedy manner of choosing the action with maximum Q value. We would have a reward of 10 if we have once tried action 3 and we will stick with it instead of trying action 1 forever. The method of selecting the action with maximum Q value is called *greedy action selection* and prevents the algorithm from reaching the optimal solution. In order to reach the optimal solution, we should not always be greedy and sometimes try another action with some probability which is called *exploration*. There are various methods for exploration in reinforcement learning, and this is another area of research. Exploration is not good in the short run because we may select the worst actions, but in the long run we will find the action with highest reward and maximize our reward.

In some RL problems, we should find the optimal policy while the system is running therefore too much exploration will lead the system to very bad states and performance degradations will occur. A better way will be making a trace of the system, then running the RL algorithm off-line using the trace with a high exploration rate. Later we can use the policy found and the Q values in the system with a small exploration rate in order to catch up with changing conditions in the system therefore we will reduce the amount of performance degradation due to exploration.

2.6.1 ϵ-Greedy Exploration

This is the simplest method of exploration, instead of selecting the action with highest Q value, with ϵ probability we select a random action. All actions have equal probability in the exploration step. This is not a powerful method since we never take into account the number of visits to a state-action pair.

2.6.2 Boltzmann Exploration

This is a method of assigning probabilities to each action according to Q values. Initially, all the actions have equal probabilities independent of their Q values, during the simulation the importance of Q values is gradually increased. In the limit the action with highest Q value is chosen, and exploration ends. The probabilities are assigned according to formula:

$$p_k(x,a) = \frac{e^{\frac{Q_k(x,a)}{T_k}}}{\sum_{a \in A} e^{\frac{Q_k(x,a)}{T_k}}} \tag{2.17}$$

Here $p_k(x,a)$ stands for probability of choosing action a in state x at simulation step k. T is the temperature parameter controlling the degree of randomness as in Boltzmann distribution [41], [42]. The temperature parameter T_k starts from a high value and decreased through the simulation. When T_k is high, all actions have almost equal probability which means a high degree of exploration, when T_k becomes closer to zero, the action with the highest value will be assigned a probability of 1 which means no exploration. Any of the learning rate schedules given in the previous section can be chosen as the decreasing scheme.

2.6.3 Recency-based Exploration

In recency based exploration [42], the action selected is one that maximizes the quantity $Q(x,a) + \epsilon\sqrt{N(x,a)}$ where $N(x,a)$ is a recency counter and represents the last time step when action a is tried in state x, ϵ is a small constant < 1.

2.6.4 Uncertainty Estimation Exploration

In this strategy, with a fixed probability p, the agent picks the action a that maximizes $Q(x,a) + \frac{c}{N_f(x,a)}$ where c is a constant and $N_f(x,a)$ represent the number of times that the action a has been tried in state x. With probability $1-p$, the agent picks a random action.

2.6.5 ϵ-Directed Exploration

In this exploration scheme, with ϵ probability the least visited action will be selected as the exploratory action. This way the number of visits to each action will be equal for high exploration rates. This type of exploration scheme is found to be useful for call admission problems [43].

2.6.6 Visit Probability Exploration

This exploration methodology is proposed in this thesis. In this exploration strategy, with $(1-\epsilon)$ probability, we choose the action with maximum $Q(x,a)$ value. With ϵ probability we enter exploration phase. In the exploration phase every action is assigned a probability that is inversely proportional to number of visits to that state action pair. Let's denote visits to state action pair (x,a) by $Vis(x,a)$, then probability of choosing action a' in state x in exploration phase

is given by:

$$P(a = a') = \frac{\sum_{k \in A} Vis(x,k) - Vis(x,a')}{(N-1)\sum_{k \in A} Vis(x,k)} \tag{2.18}$$

Here N denotes the size of action space for state x. By this exploration strategy, least visited actions will be visited in the exploration phase.

2.7 Generalization and Function Approximation

In the sections so far, we have assumed that the value functions or state-action values are represented as a table with one entry for each state or each state-action pair. Except in very small environments, this means impractical memory requirements. The problem is not just the memory needed for large tables, but the time and data needed to accurately fill them. We can generalize our experience with a limited subset of state space to a good approximation over a larger subset.

Generalization is a problem and there is still research on these techniques. In many tasks, most of the states will never be visited enough to make the algorithm converge. This will always be the case when the state or action spaces include continuous variables or large number of sensors, such as a visual image. The only way to learn anything at all on these tasks is to generalize from previously experienced states to ones that have never been seen.

The mostly used type of generalization is function approximation, which takes samples from a desired function (value function, state-action value) and attempts to approximate them by a function in terms of state variables [44].

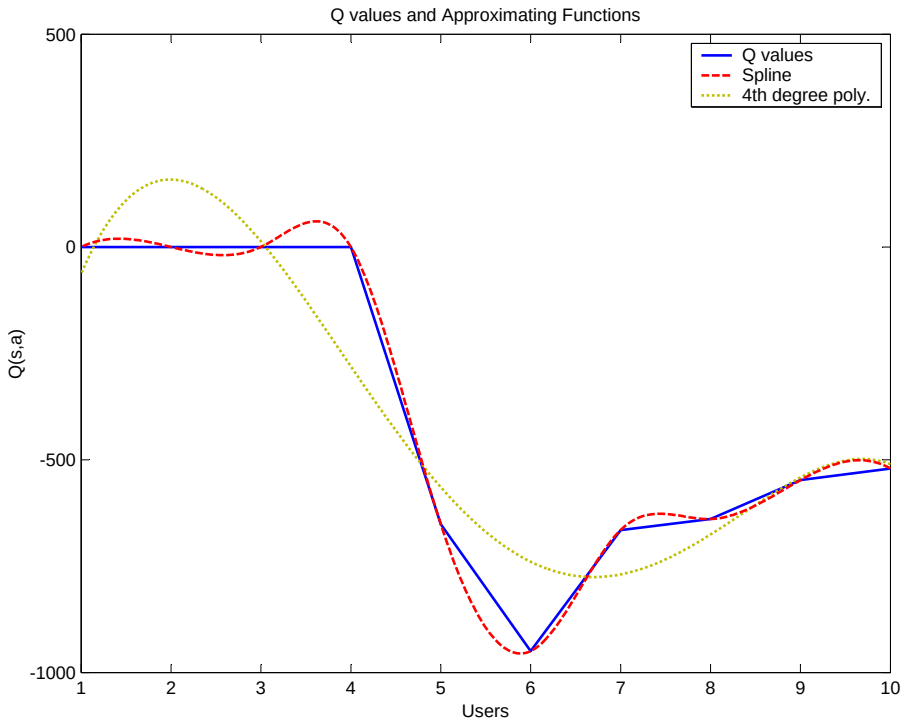

Figure 2.8: Different Approximating Functions

2.7.1 Function Approximation

A method of allowing reinforcement learning techniques to be applied in large state spaces is function approximation. In this method, a function approximator is used to represent the value function by mapping a state description to a value. In this methodology, value function is written as a function of features that describes the state of the system. Assume that our state space consists of two variables θ_1 and θ_2, the problem is that can we describe Q values of state-action pairs by the help of a $Q(x, a) = f(\theta_1, \theta_2)$ type of function? The type of $f(\theta_1, \theta_2)$ is problem dependent, one can choose polynomial type of functions with varying degree or one can use some other type of functions like cubic splines. The best methodology is solving the problem using look-up table method for a small state space and examine the Q values, this way one can see what type of function would be useful for approximating the Q values in a large state space problem. If our initial type of function is not suitable to problem, we are in a bad situation.

In Figure 2.8, Q values for a link sharing problem is shown as a solid blue line. We can fit this Q value distribution, a function in terms of $Users$, in the same figure the approximations are also drawn. As we see a four degree polynomial is not enough to extract all the information in the Q values, on the other hand spline type of approximation has shown very good results. In finding the approximating functions, a rule called *Delta Learning Rule* by Widrow and Hoff is employed [26].

2.7.2 Delta Learning Rule

In this rule, one has some sample values of an unknown function and one wants to fit some function to these values. This rule has the objective of minimizing the mean square error (the sum of squared differences between samples and approximated values), sometimes this rule is called **LMS** standing for Least Mean Square [26]. Suppose you want to approximate y_k values by a function $y'_k = f(\theta_1, \theta_2, ...\theta_n)$. The mean square error is given by:

$MSE = \sum_k (y - y')^2$

In order to minimize MSE, we should increase or decrease y', i.e. we have to change $(\theta_1, \theta_2, ...\theta_n) = \vec{\theta_t}$. The best way is to increase these parameters according to their effect on y', if we change them in the direction of the gradient of y' this objective is satisfied. Call $(y - y') = \varepsilon$

$\nabla \varepsilon_{\vec{\theta_t}} = -2 \times \varepsilon \times \nabla y'_{\vec{\theta_t}}$

thus the update rule for delta learning rule is:

$$\vec{\theta}_{t+1} = \vec{\theta_t} + \alpha \times \varepsilon \times \nabla y'_{\vec{\theta_t}} \tag{2.19}$$

Here α is a small learning rate. An example would clarify the mathematical notation used, we try to find the coefficients of a 4th degree polynomial. In this example, 100 samples from a 4th degree polynomial is taken and these samples are approximated again by a 4th degree polynomial using the delta learning rule. In here, $y' = r[0] \times x^4 + r[1] \times x^3 + r[2] \times x^2 + r[3] \times x + r[4]$ will be used as the approximating function. Here $\vec{\theta_t} = (r[0], r[1], r[2], r[3], r[4])$, and the learning lasted for 100 million iterations. The algorithm for this example is as follows:

1. Initialize r parameters to random values
2. Pick a sample (y, x) point, evaluate the approximator at x to find y' and calculate $\varepsilon = (y - y')$
3. Update r parameters as follows:

   $r[0] \leftarrow r[0] + \alpha \times \varepsilon \times x^4$

   $r[1] \leftarrow r[1] + \alpha \times \varepsilon \times x^3$

   $r[2] \leftarrow r[2] + \alpha \times \varepsilon \times x^2$

   $r[3] \leftarrow r[3] + \alpha \times \varepsilon \times x$

   $r[4] \leftarrow r[4] + \alpha \times \varepsilon$
4. Go to step 2 until a target error or maximum number of iterations is reached.

The original polynomial coefficients and the coefficients found by approximation is given in the Table 2.3. As we observe, there is some error for 10 million iterations, but the exact polynomial is found by learning that lasted for 100 million iterations. The polynomial learned after 10 million iterations and the original one is shown in Figure 2.9

As a result the delta-learning rule can be used to describe samples of an unknown function by an approximating function.

Term	Orig. Coef.	Appr. Coeff. (10 m. iter.)	Appr. Coeff. (100 m. iter.)
Constant	-953.5	-941.1	-953.5
x	13290	13083	13290
x^2	-50095	-49288	-50095
x^3	62848	61734	62848
x^4	-25607	-25105	-25607

Table 2.3: Delta Learning Results

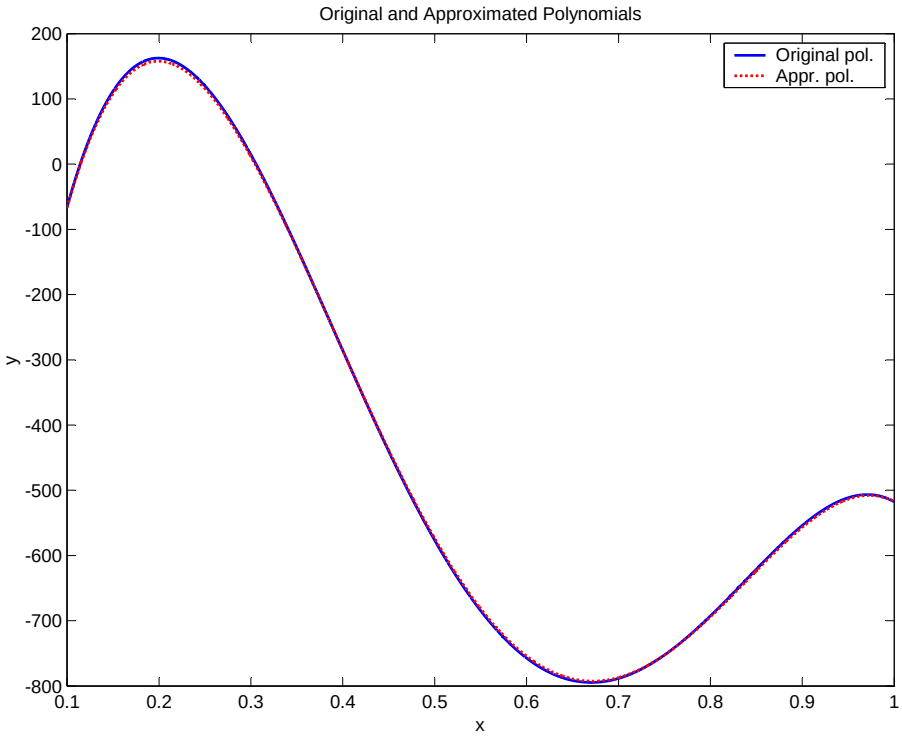

Figure 2.9: Approximation by Delta Learning Rule

2.7.3 Combining Reinforcement Learning with Delta Learning Rule

Reinforcement learning can be combined with the delta learning rule and can be applied to very large or continuous state space problems. Consider the update rule for Q-learning:

$$Q_{t+1}(x,a) \leftarrow Q_t(x,a) + \alpha\left(r_{imm} + \gamma \max_{a \in A} Q_t(y,a) - Q_t(x,a)\right)$$

From the form of update rule we can see a similarity to delta learning rule, in each update the $Q(x,a)$ value is brought closer to its real value, the term that is

multiplied by α is the difference of what Q must be and current Q and a fraction of this difference is added to Q value. Let $(r_{imm}+\gamma \max_{a\in A} Q_t(y,a)-Q_t(x,a)) = td$, then this td is made smaller at each iteration. Note that td is similar to ε in the delta learning rule. Assume that $Q(s,a)$ can be described by a vector function $\vec{\theta_t}$ t. Then at each iteration we can update them as follows and have a function approximated version of Q-learning:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha \times td \times \nabla Q_t(s,a)_{\vec{\theta}_t} \tag{2.20}$$

Thus instead of updating the element in a memory cell, we update a set of coefficients after calculating the td at each iteration.

2.8 RL Methods vs DP Methods

Some requirements and characteristics of DP methods such as model requirement and size of state space encourage people to use RL methods for solving their problems.

RL methods do not require the state transition probability matrix $P_{xy}(a)$ as an input. Thus RL methods save us from the burden of deriving the probability transition matrix. This property is important because in some problems it is almost impossible to derive the probability transition matrix. Moreover some problems may have continuous state spaces, in which the state transition matrix do not exist, where the DP methods become obsolete.

In addition to these, DP methods are synchronous meaning that at each update all the state space is swept. On the other hand, in RL methods, one updates only the visited states. This has the advantage that the states that are not likely

to be experienced will not be learned and time will be spent on more important states that will be visited much often. Another problem occurs in problems with large state spaces, the complexity of DP algorithms increase with the size of state space and after some point these algorithms become intractable, this is called *curse of dimensionality* in DP literature. When we turn our attention to RL methods, we see that there are methods of generalization such as function approximation, which makes RL algorithms attractive for applications to problems with large state spaces, because by function approximation one can update a large part of state space in a single sweep and get rid of the computational complexity of sweeping all the state space. Another problem with large state spaces is the memory requirement to store the value of each state action pair, function approximation can reduce the memory requirement of a large state space to a few number of features and coefficients as shown in the section of function approximation for an example of fourth degree polynomial.

Chapter 3

Introductory Problem: Resource Allocation with Buffering

This problem is chosen as an introductory problem for the reinforcement learning (RL) methods. This problem seemed as a good choice, because having an optimal solution it gave us the opportunity of comparing performance of RL methods with DP methods. The state space associated with this problem is intentionally chosen small, and the probability transition matrix is easy to find which makes this problem relatively easier to solve.

3.1 Resource Allocation with Buffering

By the developments in multimedia and the Internet technologies, the transfer of compressed video over the Internet has become widespread. The inherent variable bit rate (VBR) nature of compressed video makes the transfer painful because of the constant capacity links.

A key characteristic of compressed video source is its burstinesss, that is, the source exhibits peak rates which can be significantly larger than the long term

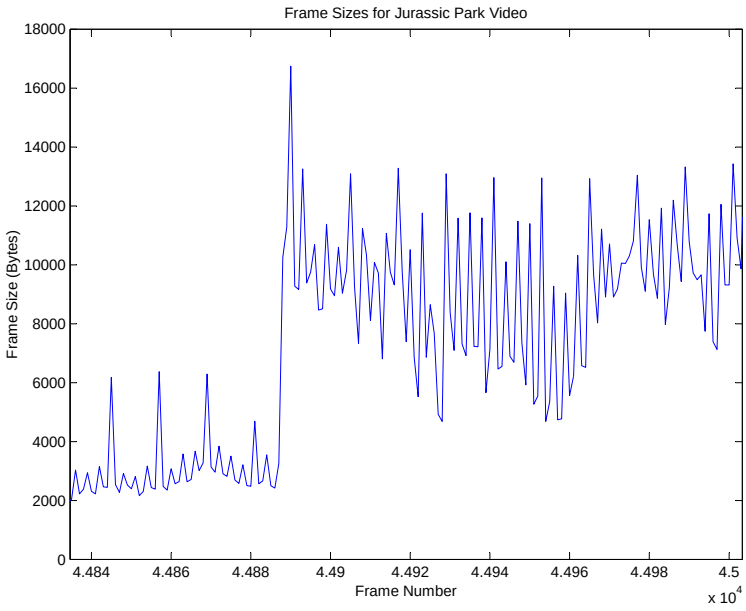

Figure 3.1: Sample bit rate vs time plot for a compressed video

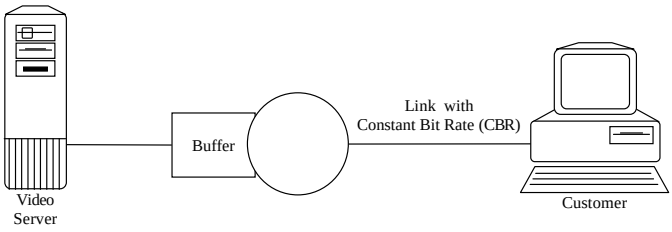

Figure 3.2: Transfer of compressed video over the Internet

average rate. In Figure 3.1 frame sizes for a section of MPEG encoded Jurassic Park movie is shown, there is a variation in the source rate not only over a period of milliseconds to seconds but also over a period of tens of seconds to minutes, corresponding to scenes with differing information content. Peak bit rates may occur in scenes of high motion or flashing lights.

In the broadband integrated services network paradigm [45], variable bitrate traffic from a source is queued at a buffer at the end system, and the network drains the buffer at a given drain rate as seen in Figure 3.2. The drain rate is chosen based on a traffic descriptor supplied by the source. If sources exhibiting sustained bursts are allowed only a single (static) traffic descriptor to describe their behavior (i.e. CBR links), they are faced with a number of choices each

having a drawback of its own. Assume that the drain rate is chosen close to the long term average rate in order to efficiently use the bandwidth. Then during sustained peaks, either the data buffer at the end-system has to be very large, or there will be many losses. If low loss rates are desired , then one has to make data buffers very large in size which leads to expensive buffering at the end systems and long delays for the sources. For an MPEG 1 encoded version of Star Wars movie, the required buffer size when the link capacity is 5% over the mean bit rate is 100MBit [10]. Setting the link capacity to the maximum bit rate of the video file can be an option but most of the time your bandwidth will not be fully utilized, you will pay for the capacity you don't use. As a result, there is a tradeoff between bandwidth efficiency and delay; when you increase your bandwidth efficiency by setting link capacity close to mean rate, you will end up with longer delays and vice versa. Other solution proposed for this problem is renegotiating the link capacity according to the buffer size and incoming rate to buffer which is called *renegotiated constant bit rate (RCBR)* [10]. Keshav et. al. proposed an architecture that estimates the incoming bandwidth using an AR(1) type of estimator, and renegotiates the bandwidth according to buffer size and buffer thresholds.

RCBR scheme exploits the slow time scale variation of compressed video files. Users of RCBR service are given the option to renegotiate their service rate at any time. Renegotiation consists of sending a signalling message along the path, requesting an increase or decrease of the current service rate. If the request is feasible, the network allows the renegotiation, and upon the completion of the request, the source is free to send data at the new CBR rate. The renegotiation brings some overhead to system in terms of signaling and processing but this can be kept small by limiting the renegotiation rate to a desired value.

In Figure 3.3, the renegotiation scheme is shown. The dashed line shows the renegotiated CBR and the blue line is the video bitrate. When the renegotiated

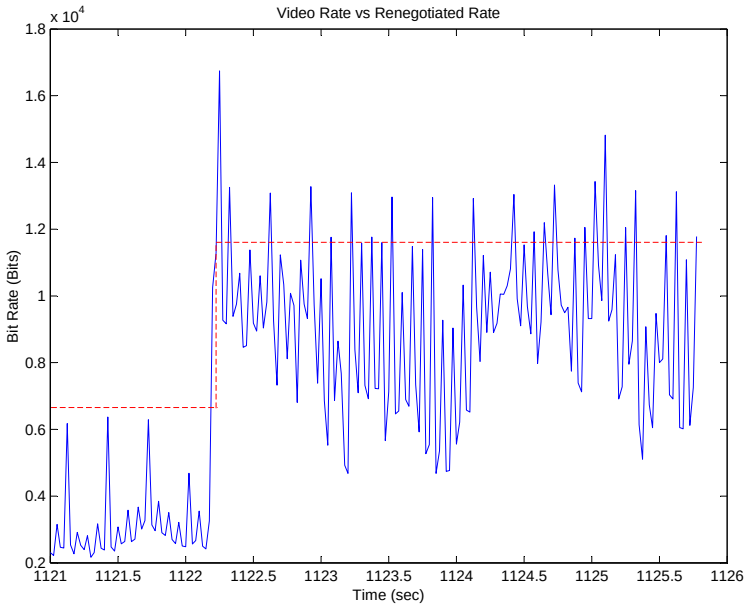

Figure 3.3: Video bitrate vs renegotiated bitrate

CBR is below the video bitrate, buffering occurs. When buffer size reaches a value, a new CBR should be renegotiated requesting an increase in the bandwidth. When the renegotiated CBR is above the video bitrate, the buffer is emptied and when the buffer size goes below some value, there is a request to reduce the bandwidth. By means of RCBR, slow time scale variations in the source bitrate are compensated, fast time scale variations in the source bitrate are compensated by the buffer. A small buffer is enough for fast time scale variations which means lower buffering delays.

The calculation of optimal renegotiation schedule is the subject of this study. For stored (offline) applications, the solution can be found by using a Viterbi-like algorithm [10]. For interactive (online) applications there is a heuristic, making use of the estimate of the incoming rate and the buffer size [10]. Our solution will use reinforcement learning to find the optimal renegotiation schedule for offline applications, also a cost mechanism that will limit the number of bandwidth changes will be employed in the formulation to meet the CPU load requirements of the network elements associated with bandwidth updates.

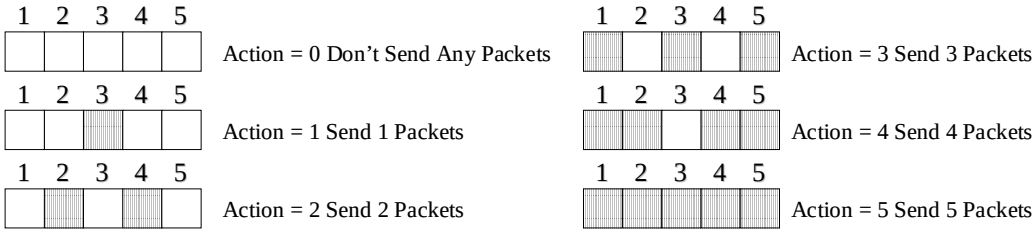

Figure 3.4: Packet sending scheme for $k = 5$

3.2 Formulation

3.2.1 General Case

The system will be modeled as a discrete time system. A time slot is the amount of time required to transmit a fixed size packet. will be equal to time a constant packet is completely sent by the system. Assume 1000 byte packets and a 30Mbit link, then a time slot is equal to 254.31 μsec. A real world video trace will be used as the traffic source, this trace will be segmented into fixed size packets and packet arrival times will be calculated. Then the number of packets arriving at each time slot will constitute the input traffic.

Bandwidth Control

The system's decision instants are the start of each k_{th} time slot, k being any positive integer. The action is defined to be the number of packets to send in the next k time slots. For example, let $k = 5$, then the decided number of packets to send can be any integer between 0 and 5. This way we can renegotiate 6 different bandwidths for the CBR link, assuming a 30 Mbit/sec line, we can renegotiate bandwidths of 0, 7.5, 15, 22.5, 30 Mbit/sec. The packets should be sent in a way that avoids bursts, and is suitable for the renegotiated bandwidth scheme, for example when 15Mbit/sec is renegotiated which is sending 3 packets in the next 5 slots then a packet should be sent at each 2nd time slot, i.e we should evenly distribute the packets to send between the k slots. The packet sending

mechanism is very similar to a time division multiplexing scheme, and is shown in Figure 3.4 for $k = 5$. In this figure, the black painted cells denote that a packet is sent in that slot whereas the white ones show the slots that no packets are sent.

Cost Structure

There are three type of costs involved in this problem.

Bandwidth Cost (b) For every unit of reserved bandwidth, we assume an associated cost b

Bandwidth Change Cost (s) This is the cost associated with each bandwidth change. Each change in reserved bandwidth requires signaling in the network elements which is costly in terms of CPU time and overhead. A cost of s is incurred for each renegotiation.

Loss Cost (l) Each packet lost in the system due to buffer overflow results in glitches in the video and audio, which causes degradation in customer satisfaction. As a result, the system should refrain from losing packets, thus there should be a cost associated with each lost packet.

As a result, our total cost is $r(x, a) = bandwidthcost \times bandwidth + bandwidthchangecost \times (1 - \delta(|sign(previousaction - action|)) + losscost \times numberofpacketslost$.

Markov Chain Formulation

We have included the number of packets in the buffer, and the previously chosen action in the state definition. The system should be aware of the buffer size in order to avoid loss of packets, and the previous action is important because

it indicates whether a renegotiation is required or not. As a result our state is $S = \{(k,l) \quad | \quad 0 \leq k \leq |A|, \quad 0 \leq l \leq |B|\}$, k stands for the previous bandwidth, l is the number of packets in the buffer. Here $|A|$ and $|B|$ denote the size of action space, and the maximum buffer size respectively.

$p_{i,j}(a), i, j \in S$ is the probability that next state will be j, when action a is chosen in state j.

$c_i(a), i \in S$ is the expected cost incurred until the next state when action a is chosen in the present state i.

As the system is sampled at discrete epochs, this problem fits into the Markov decision framework and we can use RL methods associated with it.

3.2.2 Simple Case: $k = 1$

In this thesis, we do not study the general k case but instead we focus only on the $k = 1$ case. The reason for this decision is that the $k = 1$ case is not only simple but it also is sufficient to compare and contrast the DP and RL methods. For the particular $k = 1$ case, decisions are made at each time slot.

Traffic Source

For packet arrivals, we assume a binomial arrival process:

$$P(X = x) = (1 - p) \times p^x$$

where the discrete random variable X denotes the number of packets arriving at each time slot and $E(X) = \frac{p}{(1-p)}$. At each slot, we can send only 1 packet so expected number of packets arriving at each slot should be less than 1. For $E(X) = 0.75$, p is calculated to be 0.429.

Buffer

For this problem, a maximum buffer size of 20 is chosen. When the maximum buffer size is exceeded, incoming packets will be dropped and this is reflected in the cost. Since the size of the buffer directly affects the size of the state space, we should keep size of the state space small for dynamic programming algorithms to be numerically tractable. Furthermore, RL algorithms require infinitely many visits to each state in order to converge, thus a small state space makes the algorithm faster to converge.

Actions

We have two different actions for this simple problem, our decision is whether sending or not sending a packet in the next time slot. We have encoded the action space so 0 corresponds to sending no packets in the next time slot whereas action 1 corresponds to sending a packet in the next time slot.

State Space Definition

State space is the same that is defined for the general case which is $S = (k, l)$. Size of state space equals to 42, which is equal to 2×21. 21 is the number of buffer states, and 2 stands for the size of action space, as previous action is also in action space. The size of state space is small therefore function approximation for RL is not needed and also DP methods will perform reasonably well for this small sized problem.

Probability Transition Matrix

The probability transition matrix $P_{xy}(a)$ $(x = (k_1, l_1)\ y = (k_2, l_2))$ is as follows:

- $P_{xy}(a) = 0$ when $l_2 < (l_1 - a)$
- $P_{xy}(a) = (1-p)p^{l_2-l_1+a}$ when $l_2 < 20$
- $P_{xy}(a) = 1 - \sum_{k=l_1-a}^{19-l_1+a}(1-p)p^{k-l_1+a}$ when $l_2 = 20$

where a denotes the current action.

DP Algorithms

Value iteration (VI) and relative value iteration (RVI) will be used as dynamic programming algorithms. As you know the update rule for dynamic programming algorithms require the term $\max_a(r(x,a) + \sum_y P_{xy}(a)V^{\pi^k}(y))$ to be calculated. Our $r(x,a)$ depends on number of packets lost, which is hard to find without simulation. We have substituted it with expected number of packets lost for a state action pair (x,a) that is $E(LostPackets) = p^{21-x+a} \times \frac{1}{1-p}$

RL Algorithms

R-learning and Gosavi's RL algorithm is used for calculation of RL policies for this problem.

Exploration Methodology

Visit probability exploration is used for exploration purposes. As a result, least visited actions had the highest probability of being chosen in the exploration mode. Also the exploration rate ϵ is gradually reduced to zero from a high value, which is found to be useful for learning. Initial value of ϵ is 0.1.

Learning Rates

Learning rates α is reduced to zero using DCM scheme. Initial value of α is 0.2. β is chosen to be constant and its value is 0.001.

Performance Measures

We have three performance measures for this problem. These are:

- Average cost per time slot
- Bandwidth efficiency which is $BE = \frac{TotalPacketsCarried}{MaxPacketsCouldBeCarried} \times 100$

  We want to maximize the bandwidth efficiency. This indicates the amount of bandwidth that is reserved and used.
- Average loss rate which is $ALR = \frac{TotalBytesLost}{TotalIncomingBytes} \times 100$

  We want to minimize the number of bytes lost, because this is an indicator of customer satisfaction.
- Average renegotiation rate which is $ARR = \frac{TotalNumberofRenegotiations}{TotalSimulationTime}$

  Average renegotiation rate should be kept small in order to reduce the CPU load on the core network elements and signaling overhead in the network.

3.3 Implementation and Results

A discrete time simulator is written in C programming language. Also a simulator to evaluate the policies found by RL algorithms is written. VI and RVI algorithms are implemented using C language.

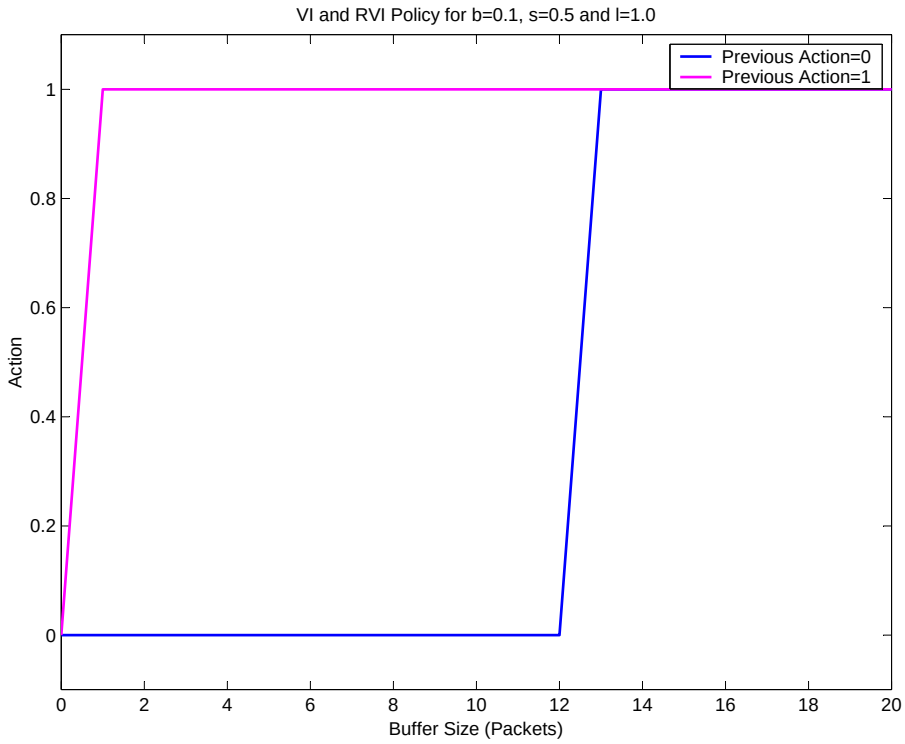

Figure 3.5: Policy found using DP

3.3.1 RVI and VI Results

RVI and VI methods are proven to converge to optimal policies. For this small problem their convergence time is very small but it grows exponentially by the size of state space. We expect the optimal policy to be a hysteresis type of curve for small bandwidth change costs, the system will buffer the packets, and then send them until the buffer becomes empty, then the system will buffer incoming packets again.

In Figure 3.5, policy found for $b = 0.1$, $s = 0.5$ and $l = 1.0$ is shown. As we observe the found policy shows a hysteresis type of behavior. Assume that buffer size is 0 and your previous action is zero, than you are on the lower curve. The corresponding action is 0 meaning "not send any packets" so the buffer size gradually increases. When buffer size equals to 13 packets, the system decides to send packets (action=1) because increasing buffer size beyond this value can cause packet losses. As action equals 1 we move on to the upper curve, as the

expected number of incoming packets (0.75) is less than sending rate, the buffer size decreases slowly. When the buffer size equals 0, there are no more packets to send, thus the system chooses the action 0 therefore setting the reserved bandwidth to zero.

The average cost associated with this policy is 0.094. Bandwidth efficiency is 100 percent, and the average renegotiation rate is one renegotiation in every 30 time slot, and the loss ratio is 0.7 %. Assume that the policy always chooses the action 1, then the average cost would be 0.1 which is higher than 0.094. When the policy always chooses the action 0, then the average cost would be again 1 which is higher than 0.094.

Let's observe the learned policy for an extreme case, the first that comes to mind is setting the setup cost s to zero. The expected policy is sending the packets whenever there exists a packet in the buffer, and closing down the link when the buffer is empty, i.e the space between the upper line and lower line on the policy plot would become smaller.

In Figure 3.6, the policy associated with setup cost equals to zero case is shown. As you see, whenever a packet comes to system, the link is made ready to send the packet. When the buffer empties, the link is closed down. The average cost for this policy is 0.0817.

The second extreme case is the one in which the setup cost is very high. This time the policy should be always sending the packets and never renegotiating the bandwidth. An example is shown in Figure 3.7, where the setup cost is 0.85. The system waits until the buffer reaches a threshold and establishes the link, the link stays on forever. When you trace the upper curve, you see that it stays as 1 when the buffer size is 0. Here the average cost is 0.1067 and the bandwidth efficiency is 75 %. The average cost is slightly higher than 0.1 because of losses.

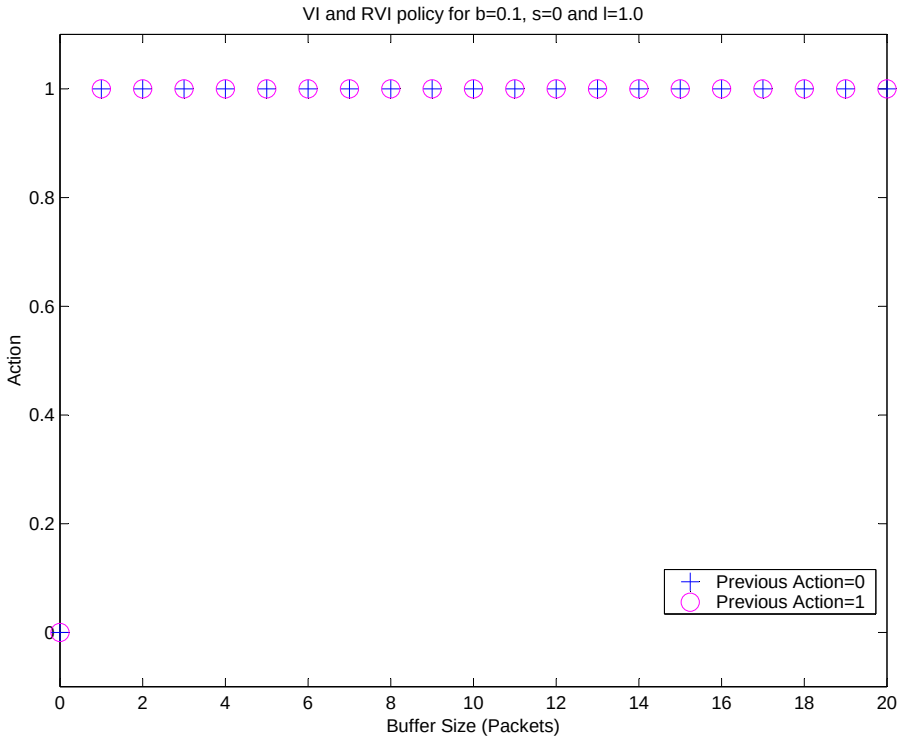

Figure 3.6: Policy found using DP for $s = 0$, $b = 0.1$, and $l = 1.0$

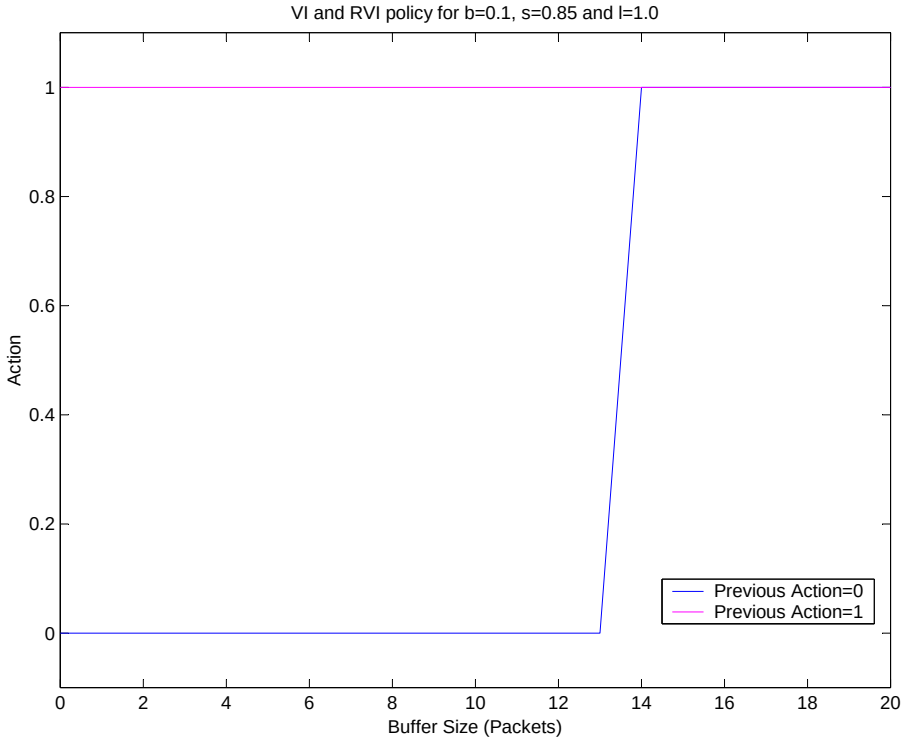

Figure 3.7: Policy found using DP $s = 0.85$, $b = 0.1$, and $l = 1.0$

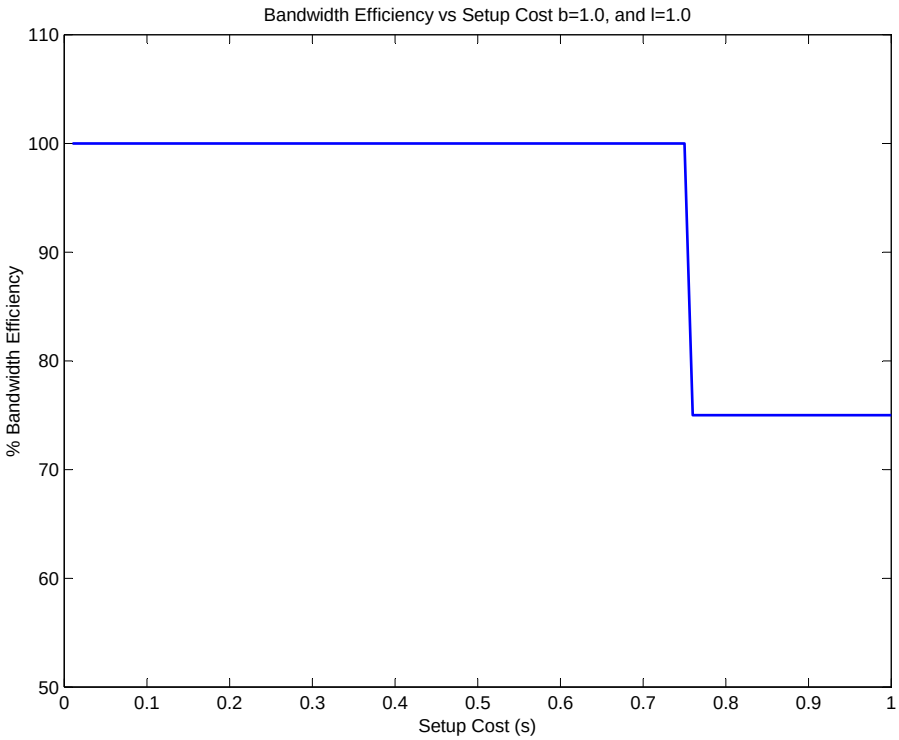

Figure 3.8: Bandwidth efficiency vs setup cost

As the expected number of incoming packets in a time slot is 0.75, the bandwidth efficiency when the link is always active turned out to be 75 %.

Bandwidth Efficiency Vs Setup Cost

One of our objectives was providing a bandwidth efficiency close to 100 percent. We provide this by means of bandwidth cost, but if the setup cost becomes reasonably expensive compared to bandwidth cost, the system can choose always sending the incoming packets, i.e the highest bandwidth therefore the efficiency decreases because the system is sending packets even when there are no packets to send. Figure 3.8 shows the bandwidth efficiency vs setup cost curve. When setup cost is 0.75, no renegotiation scheme is selected.

This graph shows that after some value of setup cost, the system finds out that renegotiations are more expensive than bandwidth and decides not to tear

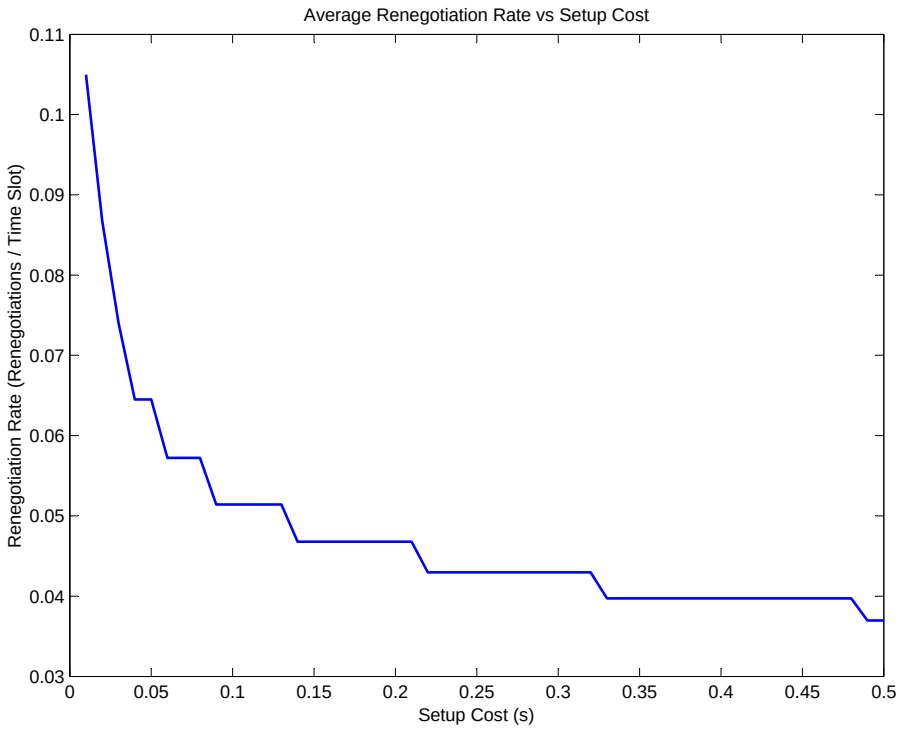

Figure 3.9: Renegotiation rate vs setup cost

the link down. As a result, the system stays on when there are no packets are in the buffer, so bandwidth is reserved even when there is no need.

Renegotiation Rate vs Setup Cost

One will expect that when the setup cost becomes higher, the system will reduce the number of renegotiations; this is shown in Figure 3.9. Number of renegotiations decrease with the setup cost, when the setup cost increases, renegotiation rate will eventually decrease to zero. In this system, we control the number of renegotiations by adjusting the setup cost. A cost scheme can be found that is suitable for the requirements of the system by trial and error. If the resulting renegotiation rate is higher than what the system can handle, you can increase the setup cost and rerun the algorithm and calculate a policy.

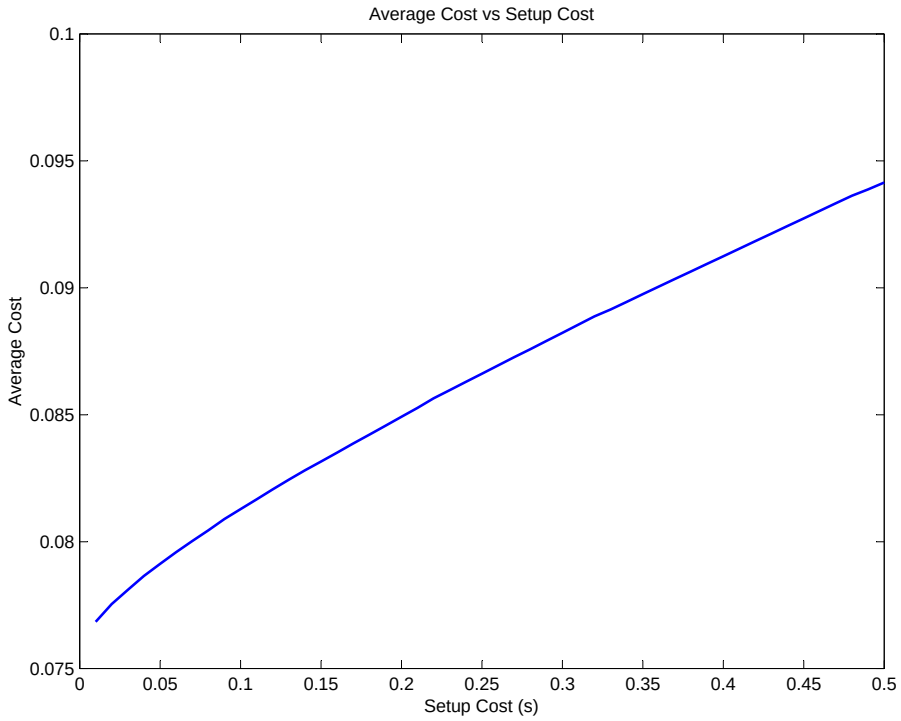

Figure 3.10: Average cost vs setup cost

Average Cost vs Setup Cost

We expect the average cost to increase with the setup cost, but after some value the average cost will not increase because the system decides not to renegotiate therefore the effect of setup cost on average cost diminishes.

In Figure 3.10 we see average cost vs setup cost curve, the average cost increases with increasing setup cost. Here $b = 0.1$ and $l = 1.0$. The average cost is less than b for all s, so the renegotiation system is better in average cost than the case of choosing $a = 1$ for all the time.

Loss Rate vs Setup Cost

Loss rate is expected to increase by setup cost, because when setup cost increases the system tries to decrease the number of renegotiations, thus the point when the system starts to send packets will increase, so the average buffer size will

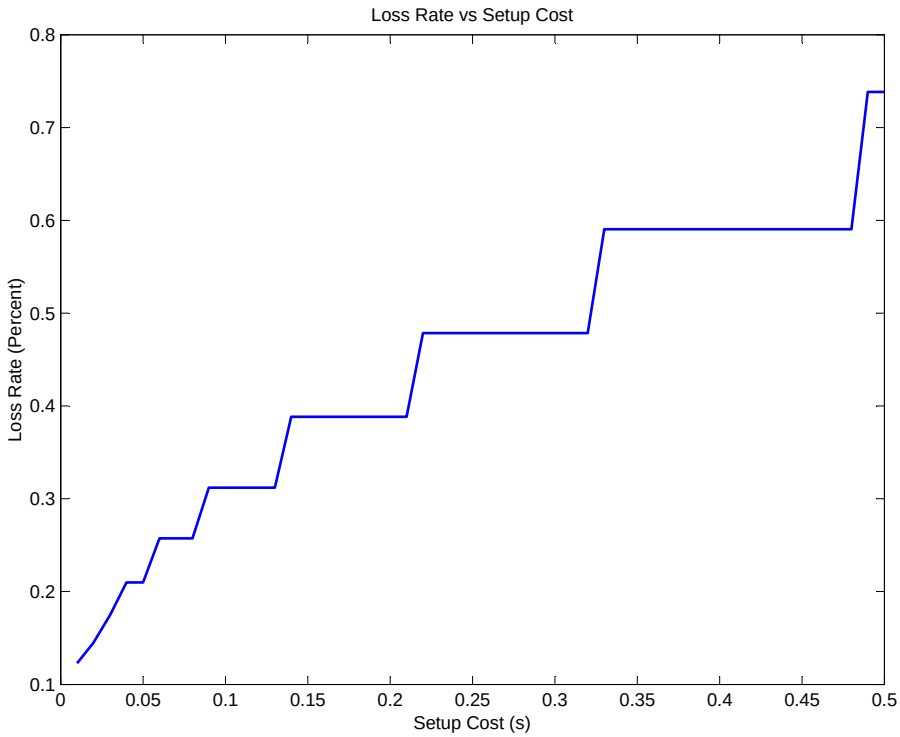

Figure 3.11: Loss rate vs setup cost

be larger which means there is a higher probability of packet loss. This effect is observed in Figure 3.11.

As we observe, the loss rate is very small for all cost combinations, this is because we have chosen the loss cost to be 1.0 which is much higher than the bandwidth cost. You can adjust this loss cost in order to have less loss in the system.

3.3.2 RL Results

R-learning and Gousavi's RL algorithms were implemented. There are two different programs for each algorithm, one program finds the algorithm by simulation and the other program uses this algorithm and runs the system and evaluates the performance of the policy. The results of the RL algorithms happened to be same with the policies of DP so the RL algorithms converged to optimal policies. This problem is important in the sense that, it constituted an example for the

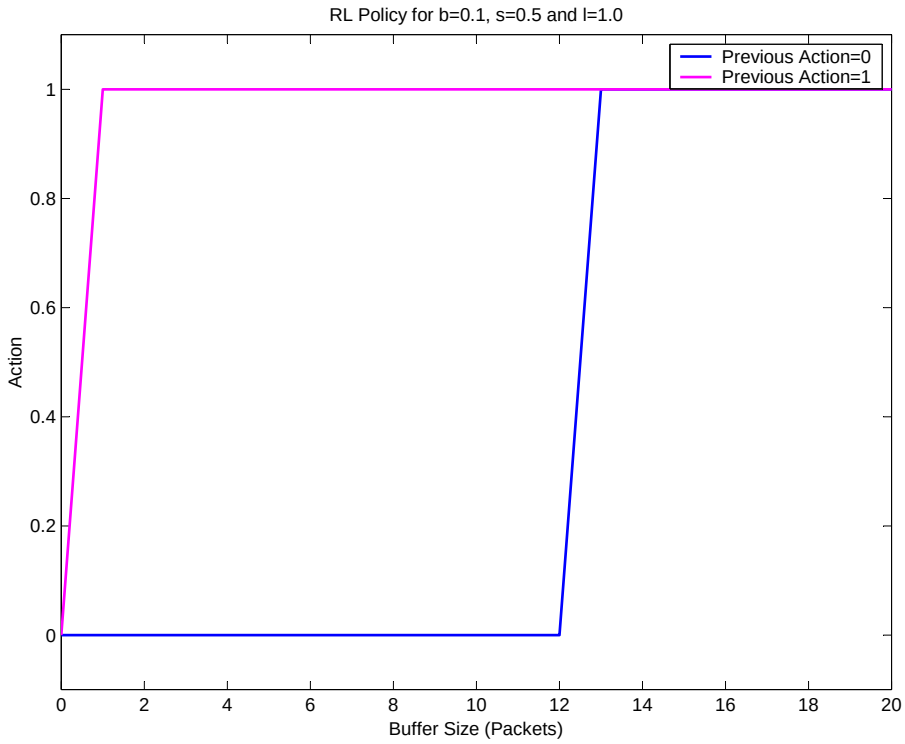

Figure 3.12: Policy found by RL for b=0.1, s=0.5 and l=1.0

convergence proof of RL algorithms to optimal policies. In Figure 3.12, it is seen that the policy found by RL algorithms are the same found by DP algorithms for the same cost structure of $b = 0.1$, $s = 0.5$, and $l = 1.0$.

Now let's check whether the policies found for the extreme cases are the same with DP policies. The first extreme case is the one with setup cost equal to zero, the system allocates the maximum bandwidth and sends the packet whenever there exists a packet in the system, this behavior is depicted in Figure 3.13, this policy is found by RL. As you see this policy is the same as optimal policy found by DP algorithms.

The second extreme case is when the setup cost is too high. This time the policy found will never select action 0, or never renegotiate. This policy is depicted in fig 3.14, which is again the same as DP policy.

As a result, RL algorithms found the same policies with DP algorithms for these three different cost examples. In order to have a convincing convergence

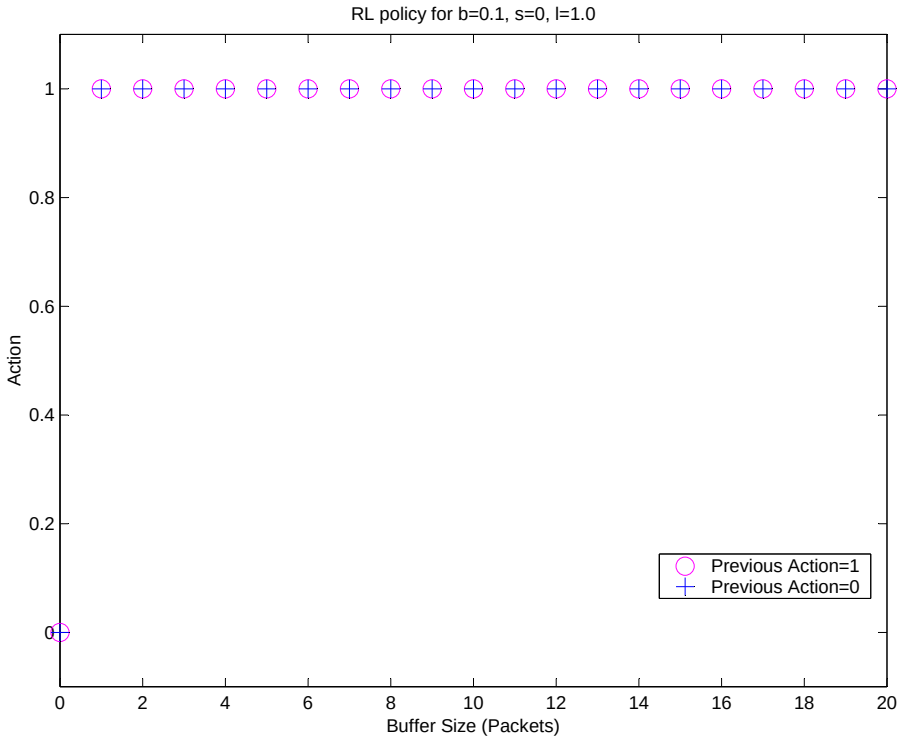

Figure 3.13: Policy found by RL for b=0.1, s=0 and l=1.0

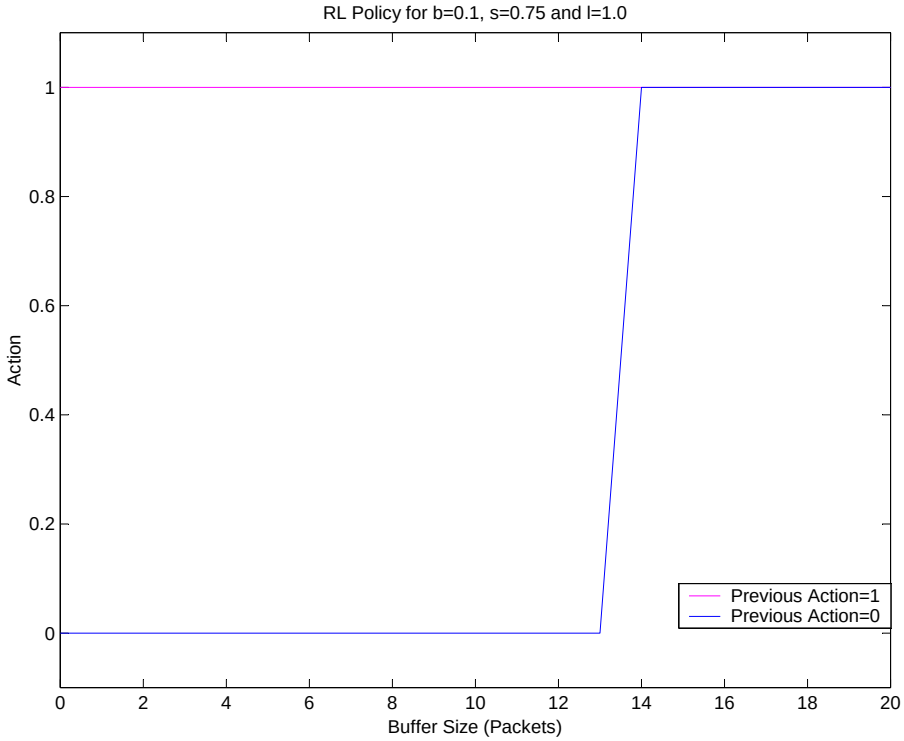

Figure 3.14: Policy found by RL for b=0.1, s=0.85 and l=1.0

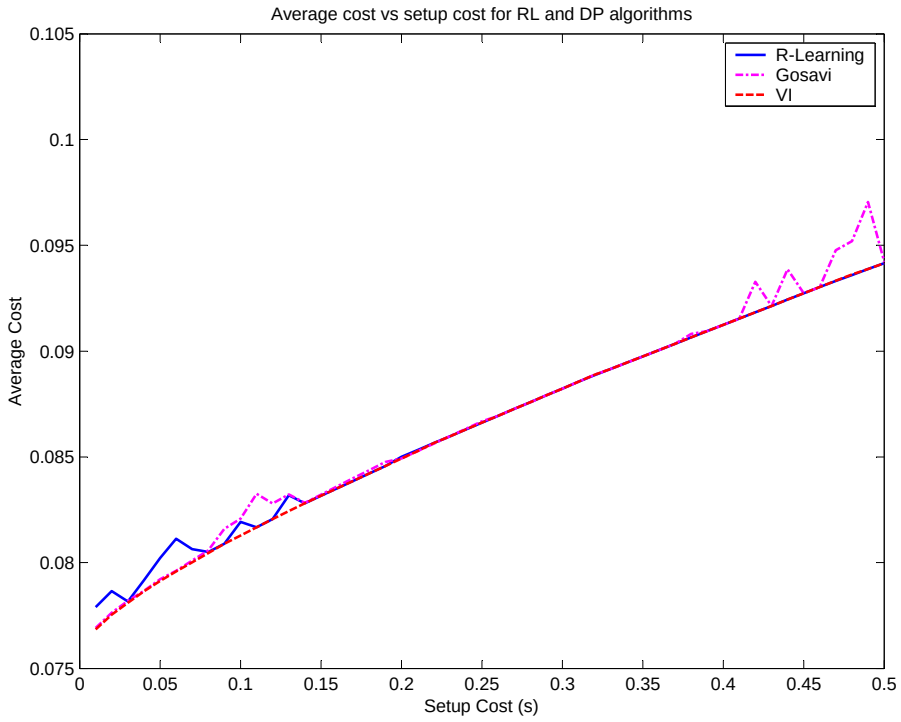

Figure 3.15: Comparison of DP and RL algorithms, $b = 0.1$ and $l = 1.0$

s	DP	R-learn	Gosavi
0	0.077	0.078	0.077
0.1	0.081	0.082	0.082
0.2	0.085	0.085	0.085
0.3	0.088	0.088	0.088
0.4	0.091	0.091	0.091
0.5	0.094	0.094	0.094

Table 3.1: Average costs for different algorithms, $b = 0.1$ and $l = 1.0$

proof, we should have more example points. For this purpose, RL algorithms are run for 50 different cost combinations and the results are compared with DP algorithms in terms of average cost. This is shown in Figure 3.15.

As you may have noticed RL algorithms are different from DP algorithms at certain points. At these points, RL algorithms were unable to find the optimal policy but they found some sub-optimal ones. This is due to setting of all learning parameters to the same value for all simulations. The parameters should be adjusted according to cost values, the adjustment is done by trial and error and there is no robust mechanism to adjust them. Also these results are obtained

by running the simulation for just one time, for more accurate results, the simulations are done for many times with different random number generator seeds and the average of them is drawn. This is not applied in this simulation because of time constraints because there are 50 points in the plot, if you simulate 40 times for each simulation, it makes 2000 runs which is very time consuming. At most of the points, the algorithms have found the optimal policy, which is an encouraging result. Table 3.1 shows that the difference in the performances of algorithms is minor, and the different points are confined to end points of the simulations which proves that the difference is due to the constant choice of the learning parameters.

3.3.3 Function Approximated RL Results

This problem's main purpose is to show convergence results for RL methods. As a result we should show that RL algorithm with function approximation can converge to some reasonable policy. It is known that, function approximated RL algorithms have no guarantee to converge to optimal policies, they converge to some suboptimal policy. For this example, we have calculated the $Q(s,a)$ values for a small example, it is found that the Q values can be represented by a fourth degree polynomial. In Figure 3.16, the $Q[0][BufferSize][0]$ values are plotted along with its 4th degree polynomial approximation.

One may see that the error with a 4th degree polynomial is very small, therefore we can extract many of the features of Q values with a 4th degree polynomial. As a result the following type of function is used to approximate the Q values where bs is the buffersize:

$$Q[bs] = r[0] + r[1] \times \frac{bs}{20} + r[2] \times \frac{bs}{20}^2 + r[3] \times \frac{bs}{20}^3 + r[4] \times \frac{bs}{20}^4 \qquad (3.1)$$

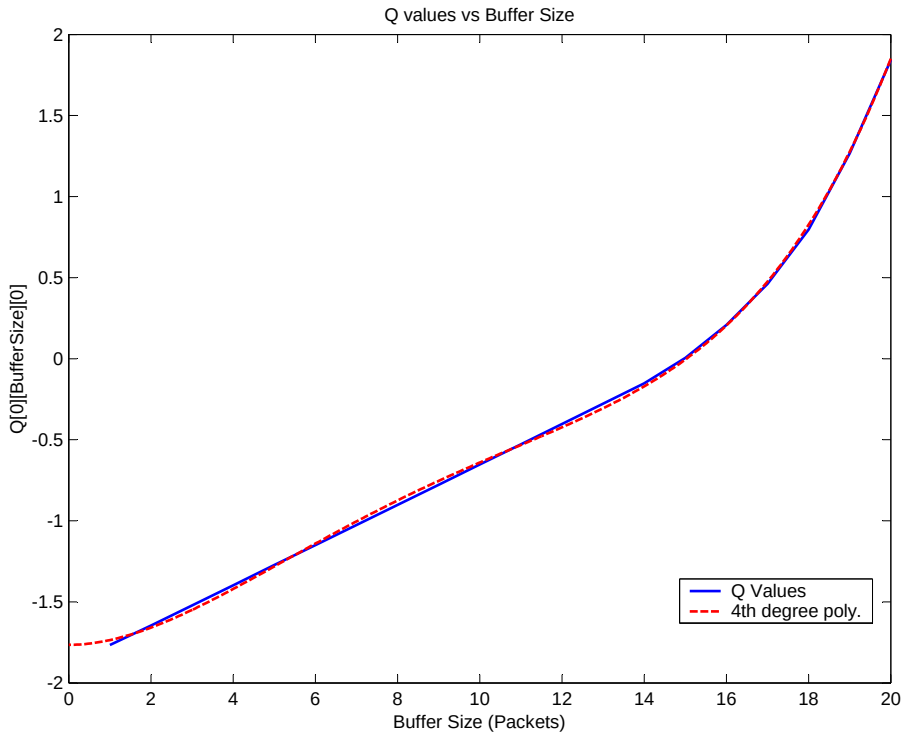

Figure 3.16: Q values for tabular R-learning

Here the r values are different for each previous action and action value, this makes a parameter set of 20 different r values. This way you can store the 84 element action-state space in terms of 20 values. Figure 3.17 shows the original Q values with approximated ones. The functional form of Q values is very similar for both representations, but the values are different. The simulations are run for 500 million steps, and the parameters are obtained through the delta-learning rule.

Although the Q values are different from tabular representation method, the resultant policies are very close to optimal policies. We infer that the form of Q value function is important in function approximation, because the minimum of Q values in a state is chosen as an action, if all the Q values shift by some amount, the policy will not change and the average cost will be the same. For this purpose, we have run 50 simulations with function approximation scheme and compared the average costs of function approximation with that of the tabular representation method. The graph is given in Figure 3.18.

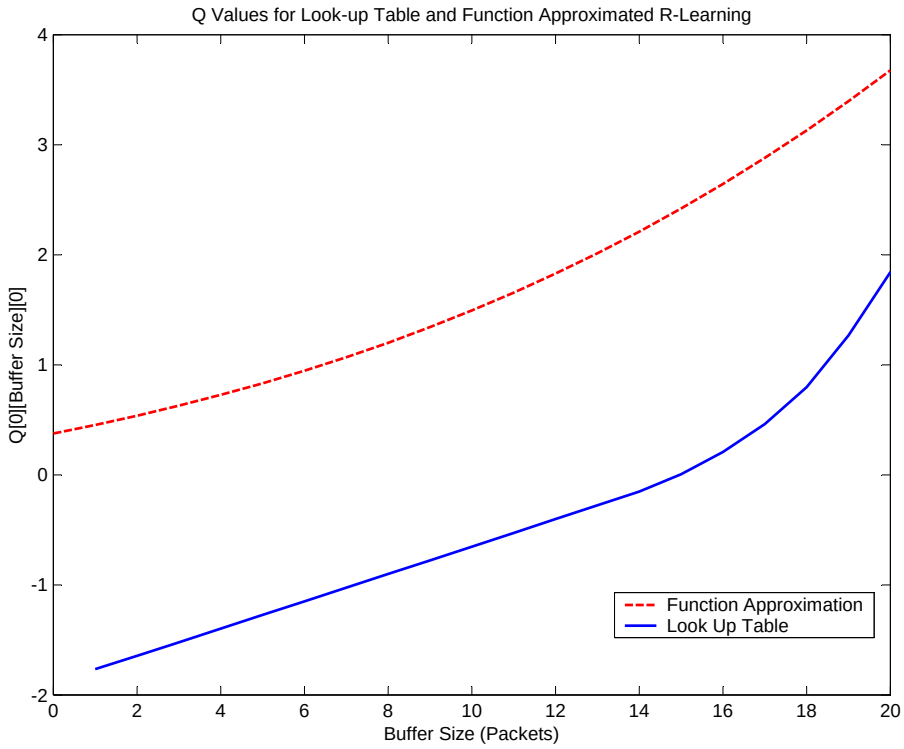

Figure 3.17: Q values for table representation and function approximated R-learning

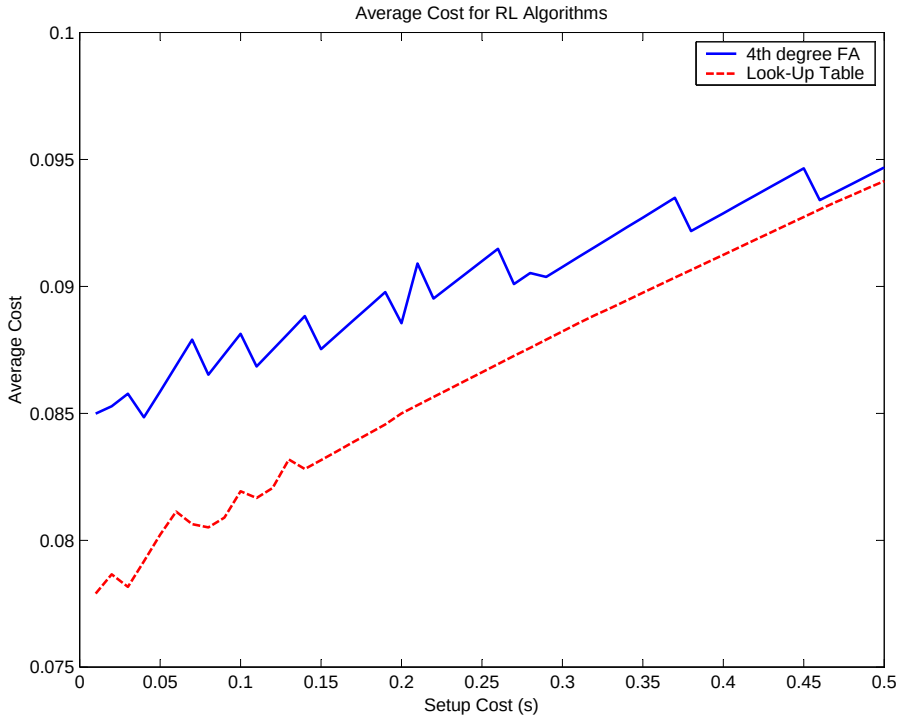

Figure 3.18: Average costs for function approximation RL and lookup table RL, $b = 0.1$ and $l = 1.0$

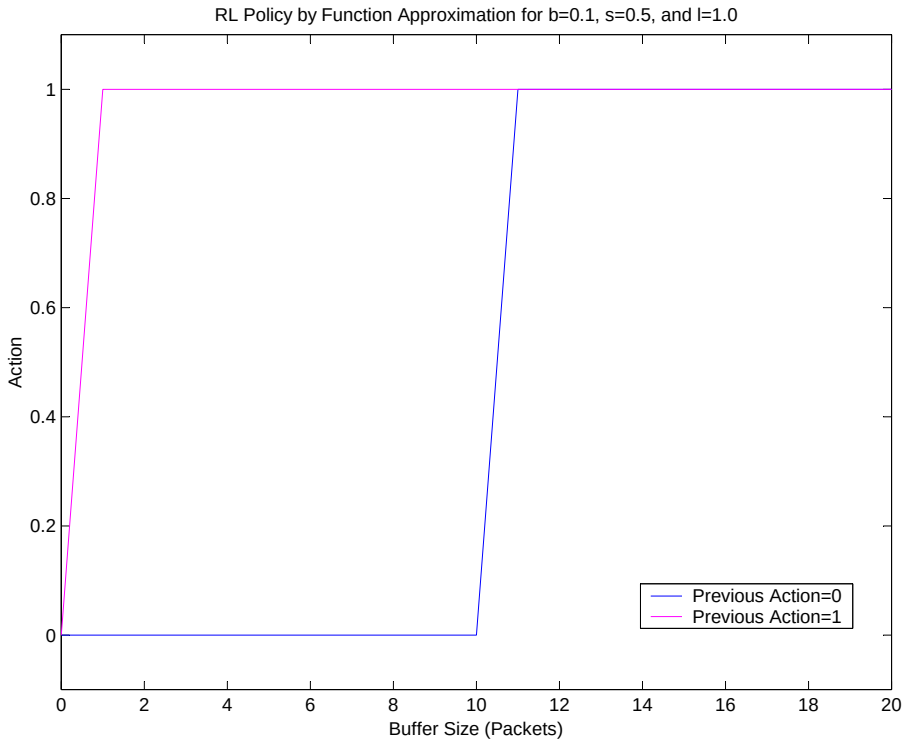

Figure 3.19: An example policy found by function approximated RL

An example policy found by function approximated R-learning is shown in Figure 3.19, as you may notice, this policy is not very different from lookup table policies. Only the link establishing and tearing down points are slightly different from the optimal policy.

3.4 Discussion

On a simple example problem, it is shown that RL policies can find optimal policies as DP algorithms do without having to find the probability transition matrix for the underlying problem. This model-free nature of reinforcement learning algorithms opens up a new space for researchers for problems with hard-to-derive state transition matrices. Moreover, some problems may not have a model and these can not be solved by DP algorithms. RL algorithms have the possibility of being applied to real world problems with ease because one needs to obtain just the traces of the problem data and apply the algorithm

there. DP algorithms become inefficient when the state space of the problem is very large, their computational complexity increase, the look-up table method becomes obsolete. On the other hand RL algorithms can work, when you can implement a simulator for the system. In addition to these, RL algorithms can be applied to problems with continuous state spaces where DP algorithms are not applicable.

Function approximation is a good tool for state generalization. You can find suboptimal policies for problems with large state spaces. Also you can reduce the computational complexity of a problem by sacrificing the optimal policy and obtaining a suboptimal one. In the previous sections, it is shown that the suboptimal policies found by function approximated RL are indeed very close to optimal policies.

By solving this simple problem, the experience and knowledge for applying RL to more complex problems is obtained. For some problems, optimal solution may not ever be known, thus we should learn ways of obtaining better results by RL by first applying it to a simpler problem with known optimal solutions.

Chapter 4

Dynamic Link Sharing Problem

In the dynamic link sharing problem, a link is shared by a number of users. The link is divided into a number of channels, which gets an equal share of the link capacity. Determining the number of channels according to the number of active users is a Markov decision problem (MDP). In this problem, our objective is maximizing the average bandwidth per user whilst limiting the number of bandwidth updates (update rate) to a given number. The number of bandwidth updates is limited in order to reduce the amount of signaling between the network elements. Moreover, in this work end users are assumed to stream video over the shared link, as a result there is a limit on the rate adaptation rate of the encoders, they can not change their encoding bitrate arbitrarily, thus an upper limit should be put on the bandwidth update rate of the system.

In this chapter, first the general formulation for the system would be given, then the proposed heuristics and their drawbacks will be discussed. MDP and RL formulation for the problem will be shown. In real world, the user arrival rate process is not homogeneous, statistics show that telephone traffic makes two peaks throughout the day, so an RL framework for nonhomogeneous arrivals will be proposed and its working conditions will be discussed. Later the results for

homogeneous traffic will be presented, and the results for nonhomogeneous traffic will follow.

4.1 General Formulation

Before going into details of performance measures, it is appropriate to give the conventions adopted for this problem.

Assume that there are k channels set-up at time t, then the allocated rate for each channel is b_k. Assuming link capacity C, b_k can be defined as:

$b_k = \frac{C}{k}$

This b_k can be known beforehand as a function of k by the end users, or can be broadcasted by headend using the downlink. The end users have maximum and minimum bandwidth requirements denoted by b_{max} and b_{min}, respectively. k_{max} defines the maximum number of channels in the system and k_{min} is the minimum number of channels and they are defined as follows:

$k_{max} \leq \frac{C}{b_{min}}$, $k_{min} \geq \frac{C}{b_{max}}$

The number of channels on the link is selected by headend between these numbers and the bandwidth per user is determined. The system should always make number of channels larger than or equal to number of active users as long as the maximum channel limit is reached due to QoS requirements. When the maximum channel limit is reached, the incoming users are denied access to shared link system. Assume a link capacity of 2Mbps, $b_{max} = 0.5Mbps$ and $b_{min} = 0.2Mbps$, then $k_{max} = 10$ and $k_{min} = 4$. The number of channels can not exceed 10, therefore at most 10 concurrent users can exist in these system. Assume that the number of active users is 5 ,and number of channels established on the link is 8. Therefore each user will have 0.25 Mbps assigned bandwidth.

4.1.1 Performance Measures

Performance measures for such a shared link system are average allocated rate for a connection, and update rate. Average allocated rate should be kept high while keeping the update rate as low as possible.

Average Bandwidth (AB)

Average bandwidth is defined as the average bandwidth a user gets during its lifetime. Assume that the holding time for a connection is H. Let $b(t)$ be the rate allocated to the connection at time t. The mean rate allocated to the connection is:

$$\hat{AB}(H) = \frac{\int_0^H B(t)dt}{H}$$

Average Update Rate (AUR)

Average update rate is the average number of bandwidth updates that take place in unit time. For calculation of AUR, the bandwidth updates are counted during a time window, and this number is divided to this window for finding average. In our work, the duration of time window is chosen as an hour.

4.1.2 Traffic Characteristics

In this thesis, we have assumed that bandwidth requests are identical and they arrive at the headend according to a homogeneous Poisson process with rate λ. We also assume exponential holding times for each connection with mean $\frac{1}{\mu}$. The arrival rate (λ) is determined using the formula $p = EB(k_{max}, \frac{\lambda}{\mu})$ where p is the rejection probability which is typically chosen as 0.01. EB is the Erlang's B formula. This formula gives the maximum arrival rate (λ) possible in order

to have p rejection probability when the maximum number of channels is k_{max}. Erlang's B formula is given below:

$$EB(k_{max}, \rho) = \frac{\frac{\rho^{k_{max}}}{k_{max}!}}{\sum_{i=0}^{k_{max}} \frac{\rho^i}{i!}}$$

where $\rho = \frac{\lambda}{\mu}$

4.2 Stationary Case

In the stationary case, λ and μ are chosen as fixed. The solution is finding a policy which is a mapping from number of users to number of channels to set-up.

4.2.1 Solutions Proposed

Switched Virtual Circuit (SVC) Heuristic

In this approach, the number of established channels is exactly equal to the number of active users. This is equivalent to establishing a virtual circuit for each incoming user which is called switched virtual circuit in ATM. This means we update the rate of connections at each user arrival and departure. For example for a 100 channel system, the bandwidth update rate is about 3500 updates/hour on the average which can not be tolerated by the end users and the core network elements. No control is imposed on the bandwidth update rate, which is the main drawback of this heuristic. On the other hand, this heuristic is the best in bandwidth efficiency. As the number of channels is equal to number of users, the bandwidth of the link is fully utilized.

Permanent Virtual Path (PVP) Heuristic

This approach works on the principle of setting-up maximum number of channels that is allowed on the link and never changing the number of channels. Then the users would always have their minimum required bandwidth b_{min} all the time. This approach has the minimum network utilization because the link will be fully utilized only when all the users are active at the same time. On the other hand this approach has the minimum bandwidth update rate which is 0.

Argiriou Heuristic

This heuristic finds a static policy taking into account the average bandwidth function according to number of users [23]. The bandwidth assigned to a user is equal to total link capacity over number of active users in SVC heuristic, which shows a behavior as shown in Figure 4.1 with a solid line, here total link capacity is 2Mbps, maximum number of channels is 10 and minimum number of channels is 1. Argiriou Heuristic (AH) approximates this curve by a piecewise constant line, as shown by dashed line in the figure. The dash-dot line is the bandwidth assigned by PVP heuristic.

As you may have noticed, when the number of active users is high, number of allocated channels does not make a significant change in the bandwidth of the users, as a result this heuristics sets up more channels when the number of users is high and takes advantage of this in order to reduce the frequency of bandwidth updates. This heuristic has the drawback of not having an explicit bandwidth update rate control mechanism, also it is defined only for a maximum of 10 channels case, no general form is given.

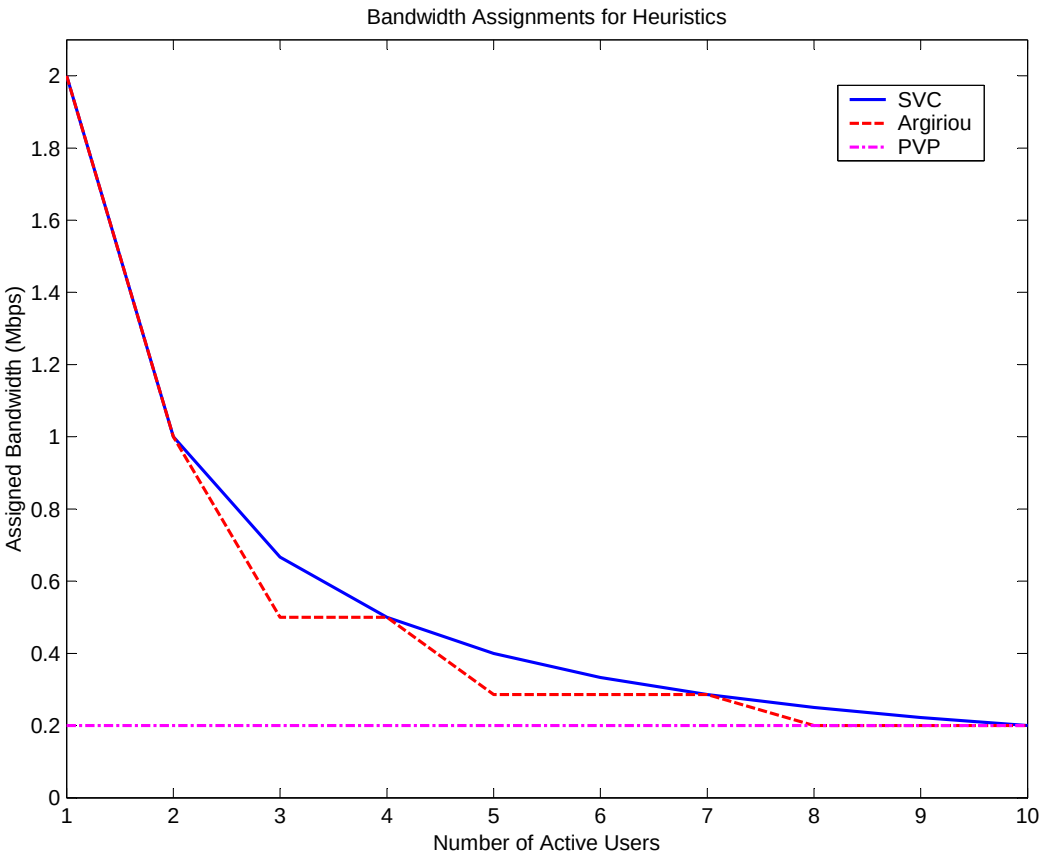

Figure 4.1: Bandwidth assignments for various heuristics

RL Approach

We propose a method that learns the user-channel mapping by simulation. Also a rate control scheme is adopted. The end-users specify the maximum number of bandwidth updates they can tolerate and you can find a mapping policy according to these specifications. The advantage of this approach is that it can be applied to systems of any size with a few modifications as it learns the policy only by simulation.

4.3 Non-Stationary Case

The analysis of communication systems revealed that the arrival process of customers changes according to time of day. A sinusoidal change model is proposed

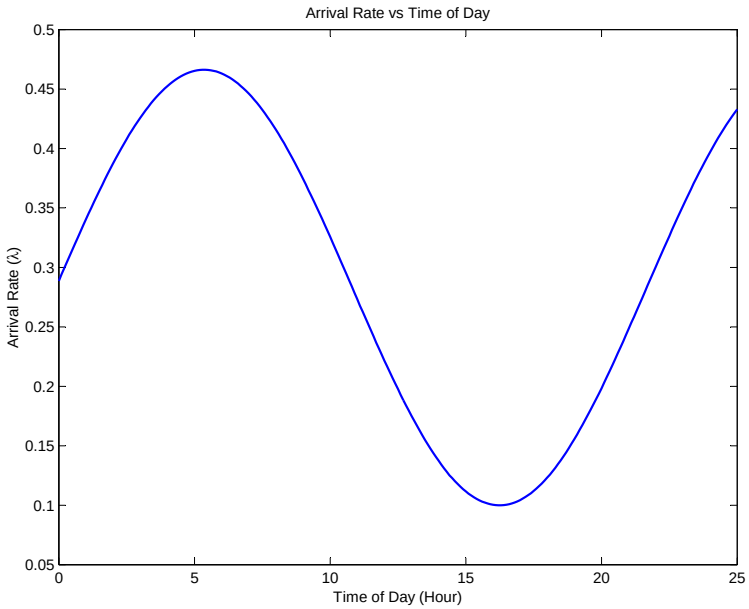

Figure 4.2: Arrival rate change model

for the daily changes which makes two peaks during the day [46]. The traffic model chosen is shown in Figure 4.2.

4.3.1 Solution Proposed

The arrival rate space can be quantized to a finite number of levels. Then RL algorithms can be used to find policies off-line for each of these quantized levels using stationary traffic. If the frequency of arrival rate change is small enough then a mechanism at the headend can detect the rate of arrival and load the policy trained for that traffic rate and use it for assigning bandwidth to users. We call this approach *quasi-stationary approximation* when the frequency of arrival rate change is small enough to be detected.

Traffic Detection

We have assumed an exponential model of traffic for the arrivals, as a result if we can estimate the interarrival time of customers we can have an estimate of arrival

rate λ. The detection mechanism uses Robbins-Monro stochastic approximation scheme to have an estimate of the arrival rate [47]. The detection mechanism can be described as follows:

- Let τ_k be an interarrival time at time k
- $\tilde{\tau} \leftarrow 0.99 \times \tilde{\tau} + 0.01 \times \tau_k$
- $\tilde{\lambda} = \frac{1}{\tilde{\tau}}$

Then the headend can select the policy trained with a rate of traffic that is closest to $\tilde{\lambda}$.

4.4 Semi Markov Decision Process (SMDP) Formulation

Decision instants for this system are chosen as arrival or departure points of a connection. As the time between decision instants are continuous, this is an SMDP.

4.4.1 State Definition

Assume a connection request arrives at the system or some user tears down its connection, the number of active users at this instant is denoted by $Users$, the number of channels on the link by $Channels$, and the number of tokens by $Tokens$. The meaning of token will be defined later in the section for bandwidth update rate control. The state of the system at this instant is defined as:

$S = \{(Users, Channels, Tokens) \quad | \quad 0 \leq Users \leq k_{max} + 1, \quad k_{min} \leq Channels \leq k_{max}, \quad 0 \leq Tokens \leq t_{max}\}$

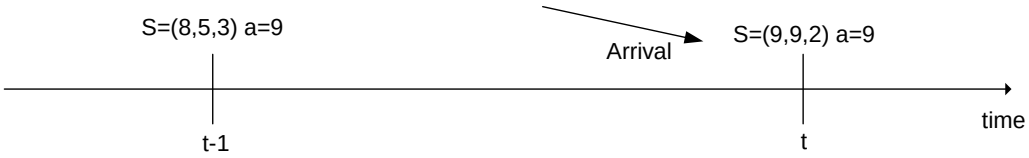

Figure 4.3: Example run of the DLS system

4.4.2 Action Definition

It is explained that our action is selecting the number of channels to establish on the link. We say that the number of channels should be larger than the number of active users in order not to reject any user. Therefore the action space for state S is defined to be $A = \{max(Users, k_{min}), max(Users, k_{min}) + 1, ..., k_{max}\}$ meaning that the number of channels can be any number between the maximum of number of active users and minimum number of channels and maximum number of channels.

An example run of the system is depicted in Figure 4.3.

Consider the system at the instant $(t-1)$, the state of the system is $S = (8, 5, 3)$, and the system chooses the action of establishing 9 channels for the next time interval. In the next decision moment t, an arrival event occurs, as a result the number of users becomes 9, as there is a bandwidth update in the previous decision instant the number of tokens becomes 2, and the number of channels becomes 9, so the state of the system is $S = (9, 9, 2)$ for instant t.

4.4.3 Cost Definition

Our one step cost is defined as follows:

$g(S, a, j) = c \times a \times t(S, a, j)$

Here the system is initially at state S, action a is chosen and the next state is j, $t(S, a, j)$ is the sojourn time from state S to state j. c is the cost of one

channel per unit time, this should be chosen greater than 0 in order to prevent the waste of bandwidth. Assume that c equals zero, then the system sets up the maximum number of channels in order to minimize the bandwidth update rate, therefore the policy learned would be PVP policy, in which the number of channels is set to maximum.

4.5 Bandwidth Update Rate Control Mechanism

In this thesis, we propose a mechanism that sets a bound on the number of bandwidth updates in unit time. This mechanism is very similar to the leaky bucket algorithm used in ATM [7].

Suppose you have a bucket which stores tokens. The tokens arrive at the bucket with a rate t_a tokens per hour, and a token is taken out of the bucket at each bandwidth update. A bandwidth update is not allowed when there is no token left in the bucket. The arrival rate of the tokens should be set to the desired update rate. If one wants the system to make 50 bandwidth updates in an hour on the average, the token arrival rate should be set to 50 tokens/hour.

The size of the bucket (bs) is also effective in the rate control mechanism. Consider that the bucket is full of tokens at the beginning of an hour. The number of tokens arriving in that hour equals to t_a, as a result the maximum number of bandwidth updates that could be made in an hour equals to:

$$Updates_{max} = bs + t_a$$

As you will have $Updates_{max}$ number of tokens to use during an hour, the bucket size should be set to a small number. A smart system could save the tokens in non-busy hours and tries to use them in the busy hours which can

not be tolerated because of end-user requirements. Thus the size of bucket is finite, when the bucket is full, the incoming tokens are lost without being used. Therefore the bandwidth update rate could be less then token arrival rate but not more than token arrival rate plus the bucket size.

It was explained before that number of channels set-up should always be larger than or equal to the number of active users. In token bucket methodology, we have said that no bandwidth update is allowed when there are no tokens left in the bucket. Then the last token left in the bucket should be used wisely. One should set the number of channels to the maximum available when the number of active users is higher than the number of established channels. One should not make any changes when the number of active users is less than the number of established channels.

Examples can help us in understanding this token mechanism. Suppose $k_{max} = 10$ for a shared link system.

- At a decision epoch the headend examines the system and observes that there are 9 active users, 8 channels established on the link and 3 tokens in the bucket. The state of the system is $S = (9, 8, 3)$, as the number of tokens in the bucket is 3, there is no danger of having no token in the next decision instant. As the number of users is more than number of channels, the system should increase the number of channels, it can set-up 9 or 10 channels according to its policy.

- At another decision time, the system is found to be $S = (6, 8, 3)$. The system could reduce the number of channels. The options for number of channels is 6,7,8,9,10. Here it may seem irrational to allocate 10 channels on the link when there are 6 users, because there are already 8 channels allocated on the link. We leave the RL algorithm free to discover this type

of actions on its own. Increasing the channels when there is a need for reducing is not rational and RL will eventually find it out.

- At another decision epoch, the system state is $S = (9, 8, 1)$. This time the system should increase the number of channels but as there is only 1 token left, it sets the number of channels to maximum of 10.
- The system state is $S = (6, 8, 1)$. The system can reduce the number of channels, but as there is 1 token left, the system decides not to change the number of channels on the system.
- State is $S = (8, 10, 0)$, therefore the channel number stays at 8.

4.6 Implementation

4.6.1 Stationary Case

Two event-driven simulators are written using C programming language. The first simulator learns the policy by RL algorithms and the second simulator evaluates the policy found.

Gosavi's RL algorithm was used for learning purposes. First a look-up table version of RL algorithm will be implemented, then a function approximation scheme will be applied.

For all implementations, size of the token bucket is set to 10 tokens.

Implementation Details

For the first case, a small size version of the problem will be studied. For this case $b_{max} = 0.5Mbps$ and $b_{min} = 0.2Mbps$ is chosen, this makes $k_{max} = 10$ and

$k_{min} = 4$. Customer service rate μ is chosen to be 0.00555 1/sec and is constant. Given μ and number of channels, ρ is calculated to be 4.4 using Erlang's B formula for 0.01 rejection probability which makes a λ of 0.02442 1/sec. Capacity of the link is chosen to be 2Mbps.

For the second case, 100 channels maximum will be studied. For this case $b_{max} = 0.2Mbps$ and $b_{min} = 0.02Mbps$ is chosen, this makes $k_{max} = 100$ and $k_{min} = 10$. Customer service rate μ is chosen to be 0.00555 1/sec and is constant. Given μ and number of channels, ρ is calculated to be 84.01 using Erlang's B formula for 0.01 rejection probability. Capacity of the link is chosen to be 2Mbps.

For lookup table version, ϵ-directed exploration scheme will be adopted, whereas for function approximation one ϵ-greedy exploration scheme will be used. The reason for choosing ϵ-greedy for RL with function approximation is that due to state aggregation, we don't have the number of visits to a specific state. For both versions the exploration rate is gradually decreased from 0.95 to 0, by halving the rate at each 1.000.000th step.

For the lookup table version, the learning rate α is gradually decreased from 1.0 to 0 according to number of visits to a specific state, the reducing scheme of α is similar to that of ϵ. For RL with function approximation, the learning rate α is gradually decreased from 1.0 to 0 according to simulation steps. For both versions, β is reduced using the DCM scheme by number of simulation steps, the parameters for DCM scheme are $\eta_0 = 0.5$ and $\tau = 1.000.000$.

Simulators are run for 100 billion connection request arrivals. And the learned policy is written to a file. Evaluating simulator runs the system using the policy learned and evaluates its performance.

The other implementation details will be given in their corresponding sections because their effects on system performance will be discussed along with the results.

4.6.2 Non-Stationary Case

For non-stationary case, simulators are written using the C programming language. The policy learning simulators are the same with stationary case ones, only their training traffic generators are different.

Also there is a simulator that loads previously learned RL policies and selects them according to traffic rate it has detected. This simulator also generates nonhomogeneous Poisson traffic of rate $\lambda(t)$ which is a continuous function of time, for generation of this type of traffic a method called the *Thinning Algorithm* is used [48].

4.7 Results For Stationary Case

4.7.1 10 Channel Case

Initial Results

In Table 4.1, the results for reinforcement learning algorithms are shown. As we observe, the average bandwidth becomes closer to that of the SVC approach when the desired update rate is increased. When one decreases this rate, the average bandwidth becomes closer to that of the PVP approach which is shown in Table 4.2.

The function approximated version of RL performs better than the look-up table version. According to theory look-up table will perform better because it is guaranteed to converge to optimal policy but this is not proven for this example. Size of the state space for this problem is large (7 x 7 x 11 =539 states), as a result every state is not visited infinitely often, then the convergence condition is not satisfied. On the other hand, the function approximated version of RL

Tabular Method			Function Approximation		
DUR(U/h)	AUR(U/h)	AB(Mbps)	DUR(U/h)	AUR(U/h)	AB(Mbps)
10	10.000	0.2383	10	10.000	0.2375
20	20.000	0.2762	20	20.000	0.2909
30	30.000	0.3052	30	30.000	0.3201
40	39.996	0.3286	40	38.800	0.3351
50	50.000	0.3400	50	46.715	0.3509
60	59.950	0.3448	60	54.159	0.3618
70	69.396	0.3518	70	60.986	0.3660
80	77.198	0.3608	80	79.717	0.3633
90	81.697	0.3673	90	89.699	0.3638
100	85.775	0.3721	100	71.273	0.3802
110	85.560	0.3804	110	108.613	0.3648

Table 4.1: Results for 10 Channel Case DUR=Desired Update Rate, AUR= Average Update Rate, AB=Average Bandwidth, U/h=Updates/Hour

Heuristic	AUR(U/h)	AB(Mbps)
SVC	110.651	0.3988
PVP	0.000	0.2000

Table 4.2: SVC and PVP Heuristics Results for 10 Channel Case, AUR= Average Update Rate, AB=Average Bandwidth, U/h=Updates/Hour

makes generalization over the states, as a result it can update values of many states with a visit to a specific state.

For this case, the delta learning rule is applied to approximate the Q values. Function approximation is done for the $Users$ dimension of the Q function, the other dimensions remained as look-up table type. For approximation along the $Users$ axis, a second degree polynomial is used.

When we compare average bandwidths for the AH and RL results for the same average update rate, we observe that RL performs almost as better as the

AUR(U/h)	AB(Mbps)
55.763	0.3610
71.293	0.3723
77.496	0.3807
93.161	0.3908

Table 4.3: Argiriou Heuristic Results for 10 Channel Case, AUR= Average Update Rate, AB=Average Bandwidth, U/h=Updates/Hour

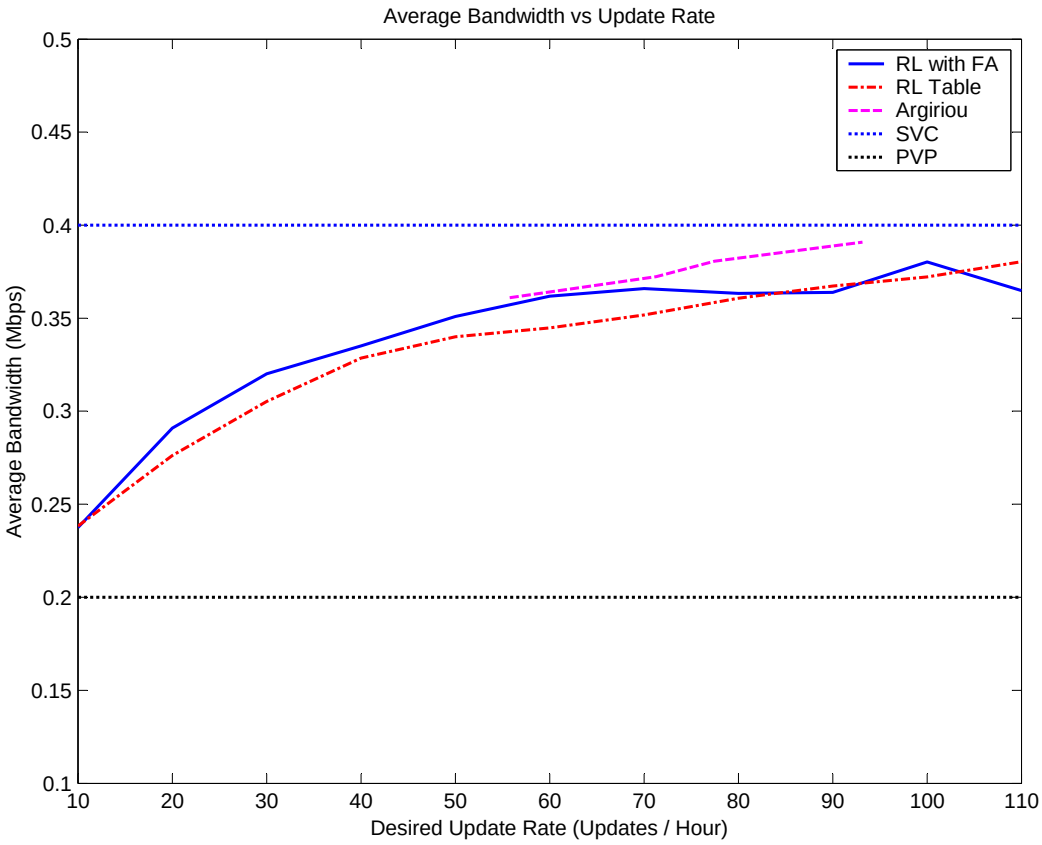

Figure 4.4: Results for 10 Channel Case

AH which is specifically designed for this example. Another important point to note is AH [23] has no specific update rate control whereas our algorithm has a strict upper limit on the number of updates in unit time. The difference is due to this rate control mechanism.

For AH, the update rate control is done by means of a parameter, when you increase this parameter, the system behaves conservatively in terms of update rate, there are only 4 possible values of this parameter and the corresponding update rates and bandwidth are tabulated in Table 4.3.

As we notice, the performance of RL is not very good for update rates close to that of SVC, this is due to small size of the token bucket, the tokens are saved for later usage but some of the incoming tokens are lost due to bucket overflow. This is seen in the difference between DUR and AUR for rates close to SVC. Another point to mention is that, for all the simulations the same parameter set

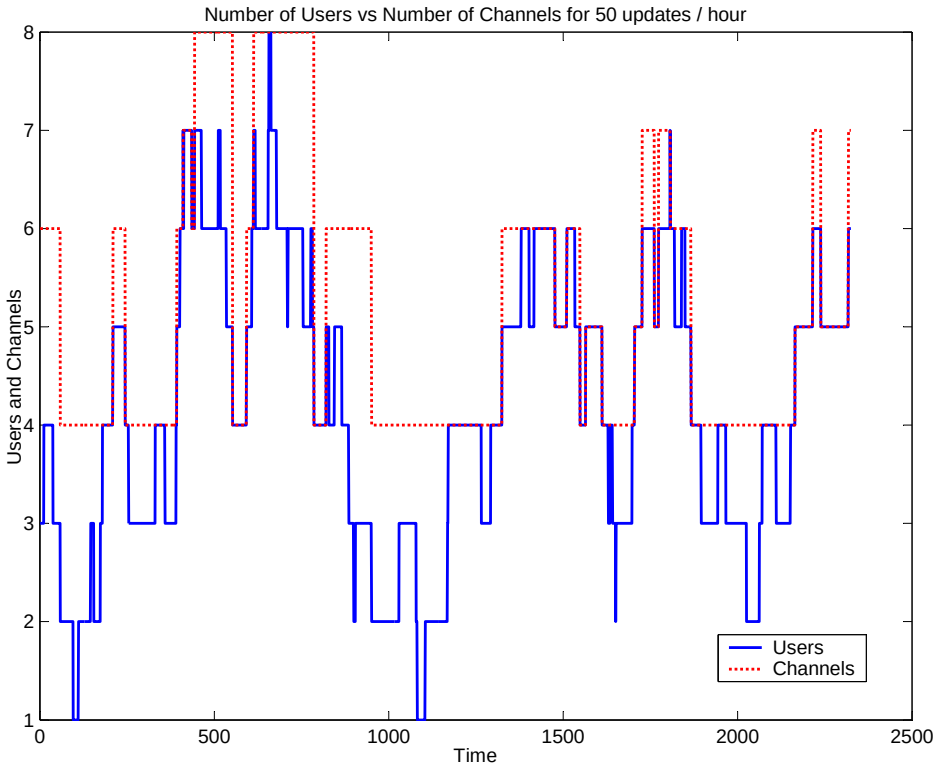

Figure 4.5: Trace for 10 Channel Case with RL algorithm for 50 updates/hour

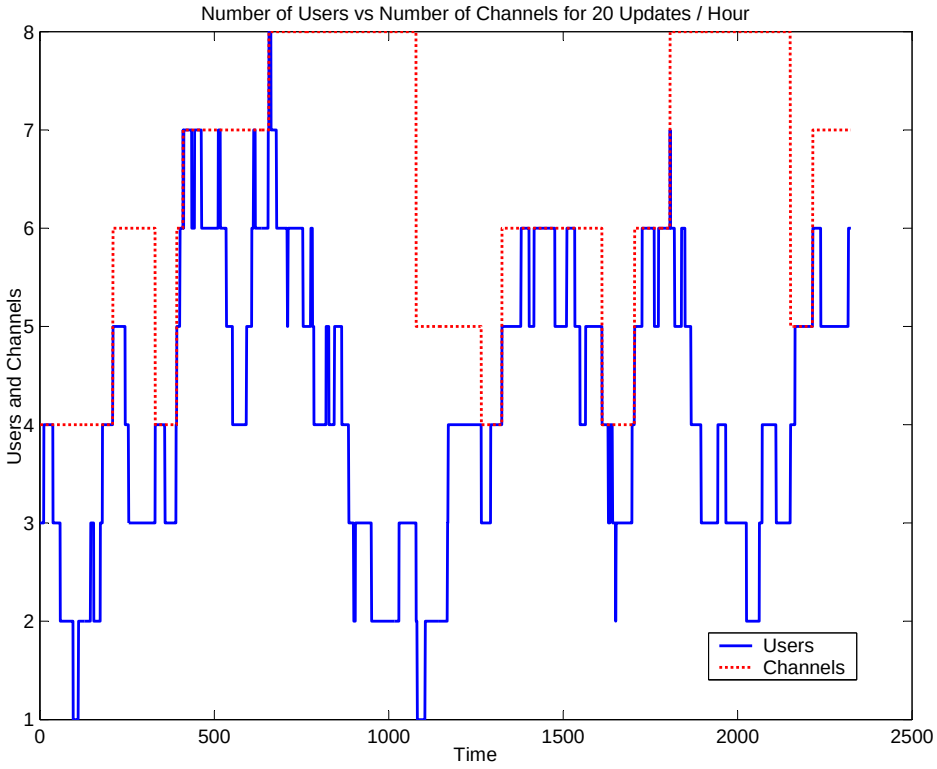

Figure 4.6: Trace for 10 Channel Case with RL algorithm for 20 updates/hour

is used, this parameter set works better for DUR of 50 updates / hour but at higher rates it does not work that well.

In Figures 4.5 and 4.6, the traces for RL policies with 50 updates/hour and 20 updates/hour bandwidth update rate limit are shown. In these plots 50 call arrivals are shown, as you see the algorithms avoid the setting up of maximum channels case when you run out of tokens. The system spends its tokens taking into account the number of tokens left. The RL algorithms perform very well under the update rate constraint and the learned policies change as you see in the plots. For a lower update rate limit, the system sets up more channels than users and for a higher update rate the system tries to track the number of users.

Token Coding

The state space for this problem is still large because we have applied function approximation only on the $Users$ axis, as a result the state space remained large. A method for decreasing the size of the state space is coding the number of tokens. In the initial simulations, number of tokens is taken as a state which makes 11 states, this method is called linear token state. We propose a non-linear coding of the token state, in this coding mechanism $TokenStateMax$ defines the maximum number of states used for coding number of tokens. When $TokenStateMax = 2$, there are 3 states, they are:

- $TokenState = 0$: There are no tokens left in the bucket.
- $TokenState = 1$: There are one token left in the bucket.
- $TokenState = 2$: There are two or more tokens left in the bucket.

This way the size of state space is decreased from 11 to 3. The performance of RL policies with non-linear coding of token states is shown in Figure 4.7.

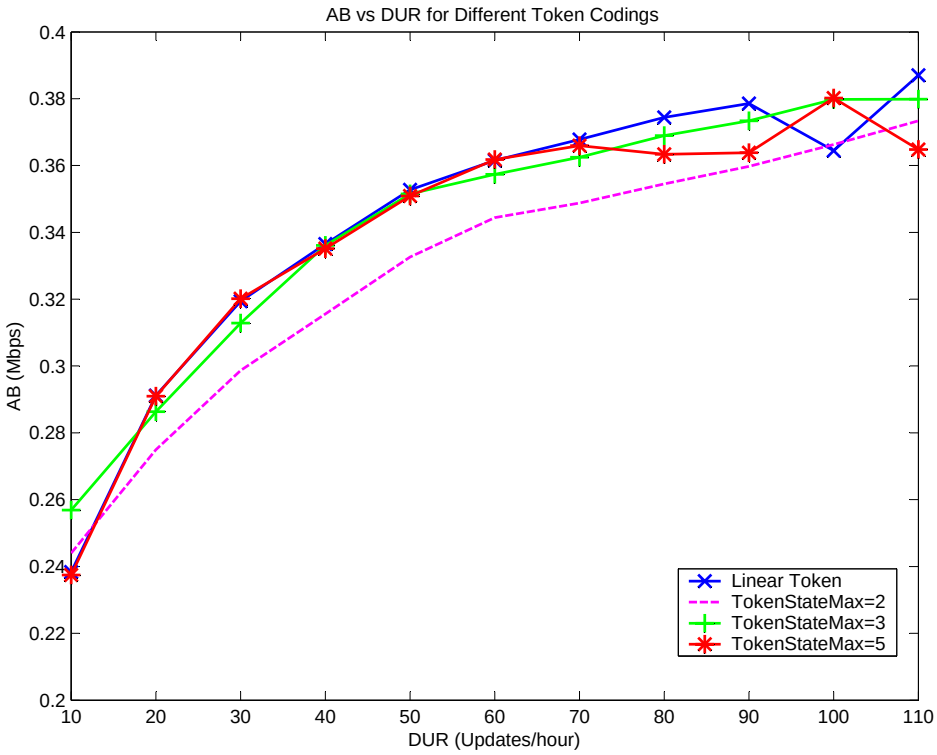

Figure 4.7: Performance comparison for different token coding mechanisms

In general non-linear token coding mechanism does not make any difference from linear token coding scheme. For a $TokenStateMax$ of 5, the difference is negligible, but for smaller $TokenStateMax$, there is some difference because the system can not know the exact number of tokens so behaves conservatively. For $TokenStateMax = 2$, the system can either have 2 tokens or 10 tokens at $TokenState = 2$, as a result the conservative option is chosen therefore we get a lower AB for this case. For lower DUR, small token state performs better because the number of tokens is always very close to 0 and small number of states decodes this number of tokens better than higher number of states in which the number of visits is a problem. For larger number of states, some states are not visited often; this makes the RL algorithm unable to learn the best actions for these states. When we decrease the number of states, we end up with better results.

Improved Function Approximation Scheme

We have applied function approximation on the $Users$ axis, done nonlinear coding of token state but the state space of the problem still remains very large. We are trying to apply RL to shared link systems where 1000 channels may exist, as a result the state space should further be decreased.

For this purpose, we tried to apply function approximation along the $Channels$ dimension of the state space. This part is problematic because the form of Q function on this axis is very complex. For $Users$ part a second degree polynomial is enough and performed very well, however a second degree polynomial resulted in very poor results for $Channels$ axis. Examination of the Q values revealed very important results for this axis. Consider the Q values for setting up 6 channels when there are 5 users and 4 tokens which is drawn as a solid line in Figure 4.8.

As one may notice, the minimum of this function is when the number of channels is 6, because the system does not want to change the number of channels most of the time. The reason for a second degree polynomial approximation not working is clear when this plot is shown. One can not extract all the features of this plot using a two degree polynomial, namely a parabola. As a result we used a fourth degree polynomial approximation structure, the results for this approximation is shown as dashed line on the same plot. It approximates the Q values better than second degree but is not complete. The reason for this lies in the coefficients of polynomial approximating this function, when we give Matlab the complete Q function, calculated coefficients are on the order of thousands, which is very hard to find using delta learning. Long simulations with careful observation and adjustment of learning rates are required for better convergence, because a wrong adjustment brings the coefficients to infinity.

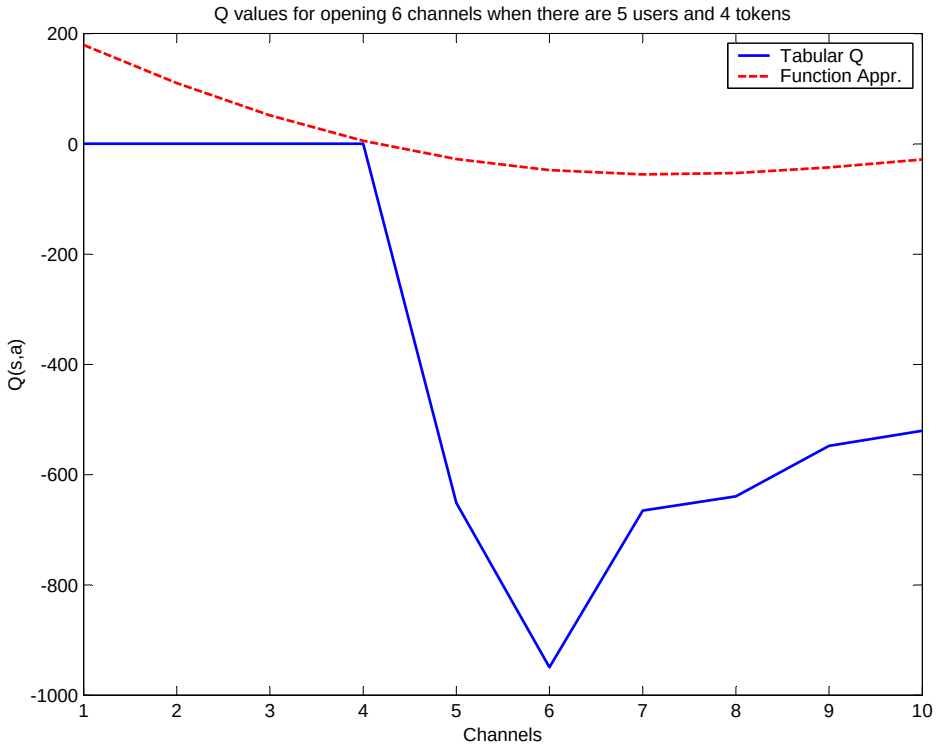

Figure 4.8: Function approximation along *Channels* axis

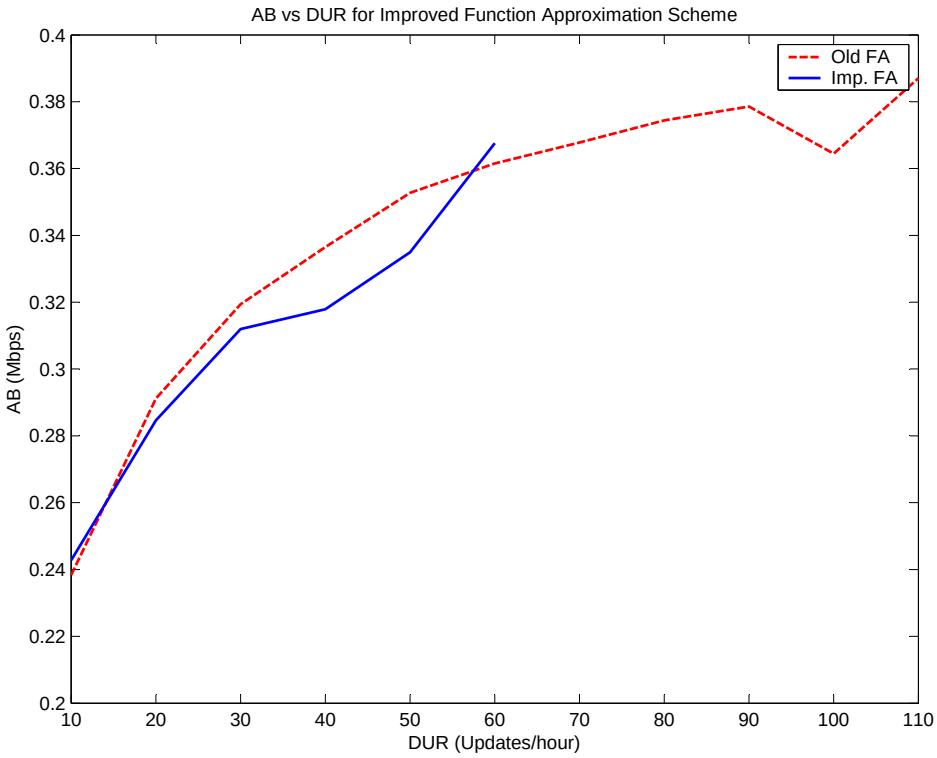

Figure 4.9: Improved function approximation results

In Figure 4.9 the results for improved function approximation structure is given. The results are obtained only up to a DUR of 60 updates/hour because after this rate, the coefficients approach infinity very quickly. Due to this reason, the improved function approximation scheme is left for future work. The problem with function approximation is that, no convergence guarantee exists for these methods, even there are cases showing divergence when function approximation is used with RL algorithms [49].

4.7.2 100 Channel Case

Dynamic link sharing problem with a maximum of 100 channels is a problem with a very large state space, there are $91 \times 91 \times 11 = 91091$ states. This number equals to $|Users| \times |Action| \times |Tokens|$. For this problem, the token state is nonlinear with a $TokenStateMax$ of 3. As a result the state space size is $3 \times 91 \times 4 = 1092$, by function approximation and token coding, the state space is reduced approximately 90 folds in size. In this size calculation, size of user space is taken as 3 due to second degree approximating polynomial, when $TokenStateMax = 3$, the token state is encoded to 4 states. Therefore the size of state space is reduced to 1092.

For 100 channels, $\mu = 0.00555$ is chosen, $\rho = 84.01$ is found using Erlang's B formula for a rejection probability of 0.01. Bandwidth of the shared link is 2Mbps, $k_{max} = 100$ and $k_{min} = 10$.

Figure 4.4 shows the performance of the RL approach in the 100 channel case. The system performance approaches PVP for small DUR and to the SVC for large DUR. As the DUR increases, one observes a saturation type of behaviour for AB. This is mainly due to nonlinear token space encoding mechanism. When you increase the DUR, you increase the token arrival rate. As token arrival rate increases, the number of tokens reaches levels that is not well encoded with

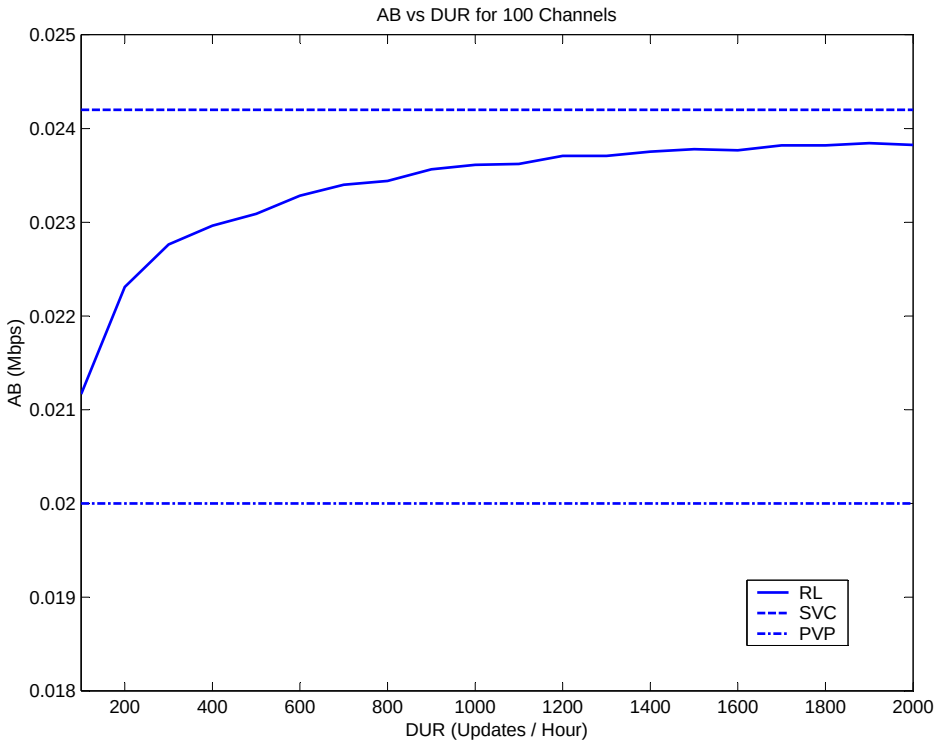

Figure 4.10: Results for 100 Channels

nonlinear mechanism. In this case the token space is encoded by a single state ($TokenState = 3$) when the number of tokens is bigger than or equal to 3. Thus the system would not differentiate between the cases when the number of tokens in the bucket is 3 and 10 as they will be encoded to the same state. This does not constitute a problem when DUR is small because number of tokens can not be as high as 10 but this can happen when DUR is close to AUR of SVC approach. Therefore the AB does not increase much when DUR increases after some point due to nonlinear token encoding mechanism.

Table 4.4 tabulates the ABs for different DUR values. It is observed that DUR and AUR have substantial differences, this is mainly due to the small size of the token bucket and nonlinear encoding mechanism of the token state. One has a DUR of 2000 but only stores 10 of these tokens, most of them will be lost due to bucket overflow when they are not used.

DUR(Updates/hour)	AUR(Updates/hour)	AB(Mbps)
100	94.866	0.0212
200	174.428	0.0223
300	260.840	0.0228
400	362.040	0.0230
500	459.106	0.0231
600	564.176	0.0233
700	659.446	0.0234
800	754.682	0.0234
900	860.695	0.0236
1000	909.904	0.0236
1100	1046.892	0.0236
1200	1117.131	0.0237
1300	1203.401	0.0237
1400	1309.357	0.0238
1500	1411.126	0.0238
1600	1454.961	0.0238
1700	1593.930	0.0238
1800	1664.658	0.0238
1900	1667.318	0.0238
2000	1873.361	0.0238

Table 4.4: Results for 100 Channel Case DUR=Desired Update Rate, AUR= Average Update Rate, AB=Average Bandwidth

Heuristic	AUR(Updates/hour)	AB(Mbps)
SVC	3325.727	0.0242
PVP	0	0.0200

Table 4.5: SVC and PVP results for 100 Channel Case , AUR= Average Update Rate, AB=Average Bandwidth

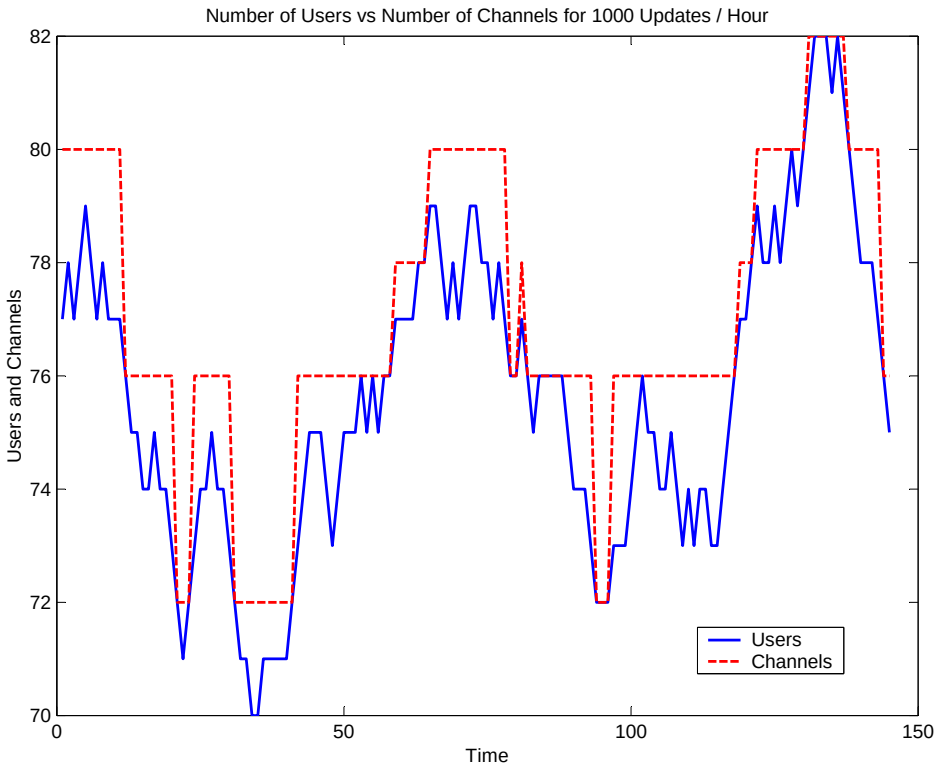

Figure 4.11: A system trace for 100 Channels for 1000 updates / hour

Figure 4.11 shows a sample run of the system with the policy found by the RL algorithm for 1000 updates / hour. As you see the system runs well with an update rate that is a third of the SVC's update rate. The difference in the resulting bandwidth is very small, RL gives 97.5 % of SVC AB with a much smaller AUR.

The only heuristics given for this case is SVC and PVP. As no general form is given Argiriou heuristic is not applicable to problem of this size. As you may notice, we propose a versatile methodology that can be used to find policies that lie in between the SVC and PVP policies. By adjusting the DUR, the user can bring the system closer to a PVP system or vice versa.

4.8 Results for Non-Stationary Case

The arrival rate of customers to the system changes during the day. We have assumed a sinusoidal model for the change of traffic arrival rate [46]. Here we use the learning methodology developed in the previous section for stationary traffic. We will divide the system arrival rate to a number of levels and train a policy for each of these rates by RL. Then these policies will be loaded to headend and headend will choose which policy to apply using instantaneous traffic characteristics.

This section is formed as follows. First we will present non-stationary case esults for the 10 channel case, and there will be discussion. Secondly we will give the results for 100 channels case.

4.8.1 10 Channels Case

For 10 channels case, the maximum ρ is calculated to be 4.4, we assume that this arrival rate changes from 1.2 to 4.4 during the day, as a result we will quantize this space to a number of levels and train a policy for each of them. The arrival rate space is divided into 33 pieces, each being apart by 0.2 units from the other.

Policy Training

In Figure 4.12 the ABs for different heuristics are drawn vs the system load ρ. As the system load increases, the number of customers in the system increase therefore the average bandwidth decreases. Argiriou and SVC have policies that do not change according to incoming traffic. On the other hand we have trained 33 different policies for different traffic using RL for a constant DUR of 56 updates/hour. 56 is chosen because this is the average update rate of Argiriou heuristic when $\rho = 4.4$.

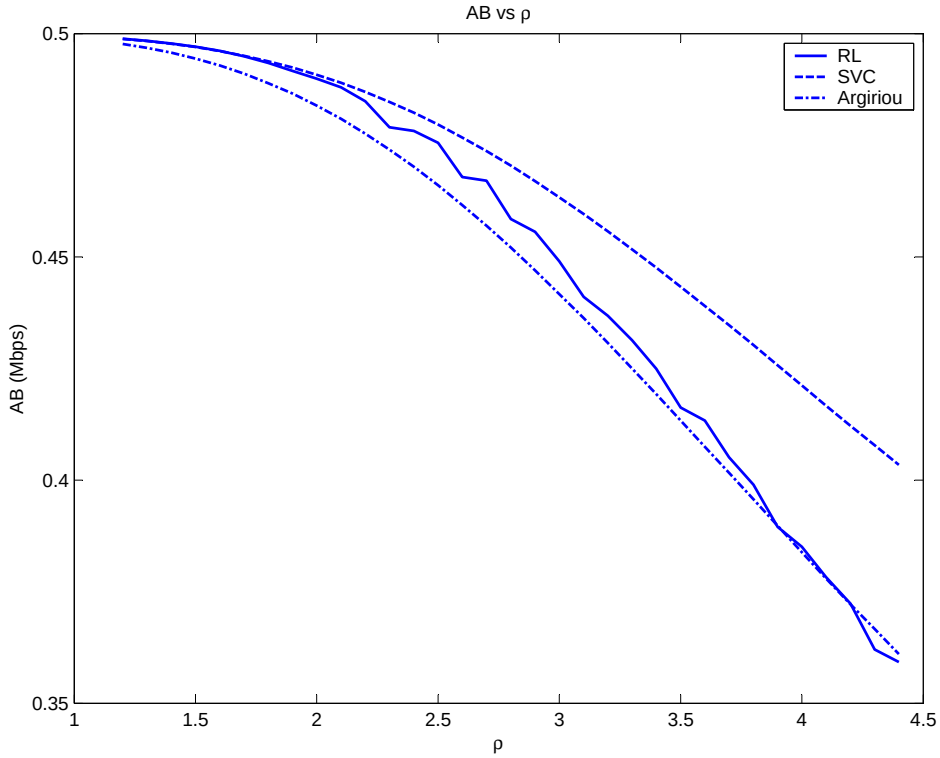

Figure 4.12: Performance of heuristics and RL vs ρ

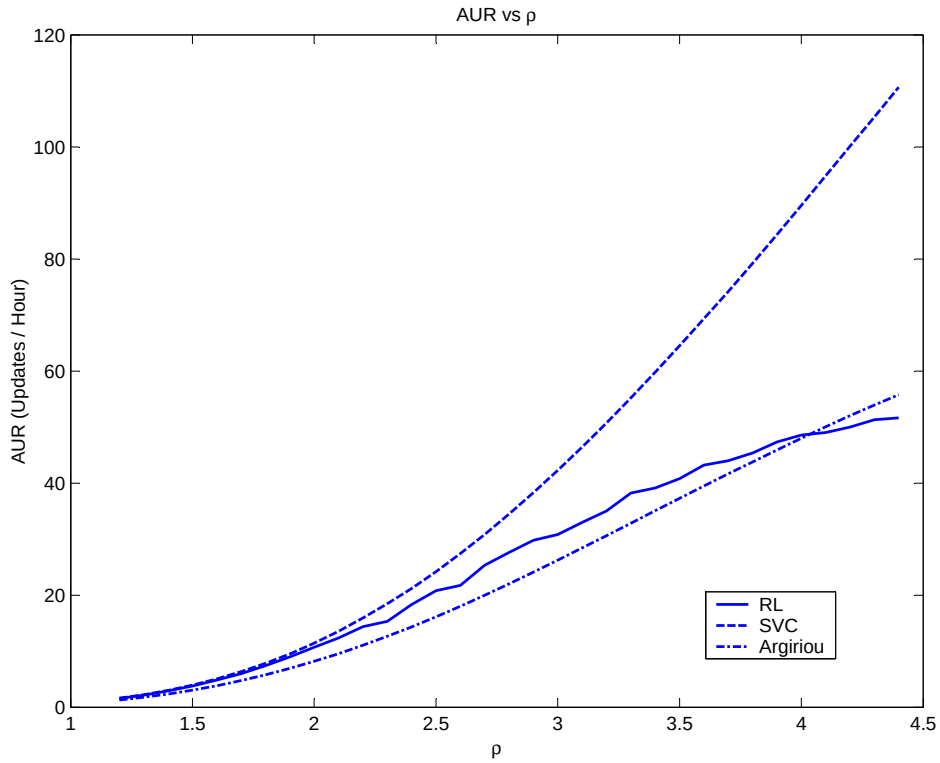

Figure 4.13: AUR of heuristics and RL vs ρ

As you see our RL policies perform as SVC when the load is mild and they become close to Argiriou when the load is high. In the end RL AB is less than that of Argiriou because that heuristic does not impose an upper limit on the number of updates. The disadvantage of Argiriou is that, it always tries to stay below the SVC update rate (almost half of it), but when the traffic is low, even the SVC update rate can be tolerable. Figure 4.13 shows the average update rates for different methods. The update rate of SVC heuristic grows exponentially with traffic rate, Argiriou also shows an exponential growth, whereas RL policy saturates at a value as a result of token bucket mechanism. Average bandwidth of RL stays below that of Argiriou for $\rho = 4.4$ case, because the average update rate of RL is below Argiriou.

Policy Equivalence

A question that comes to mind is how the 33 policies found by RL are different from each other? In order to find out the difference, all the policies can be tested using the same traffic arrival rate, for this purpose all 33 policies are evaluated using loads of $\rho = 1.2$ and $\rho = 4.4$, and the corresponding average bandwidths were drawn in Figure 4.14.

In Figure 4.14, it is observed that average bandwidth does not change much from one policy to other, as a result there is no need for headend to change policies for small arrival rate changes. In order to show this, we have evaluated policies trained for $\rho = 4.4$ and $\rho = 1.2$ using all the traffic rates in the range. The resulting plots are shown in Figure 4.15, as you see the two policies trained at the two ends of the arrival rate space perform almost equally well. "RL 4.4" means RL policy trained with $\rho = 4.4$ load.

Figure 4.15 shows that the RL policies have ABs very close to SVC when the system load is low. Considering the state space for the problem, we observe

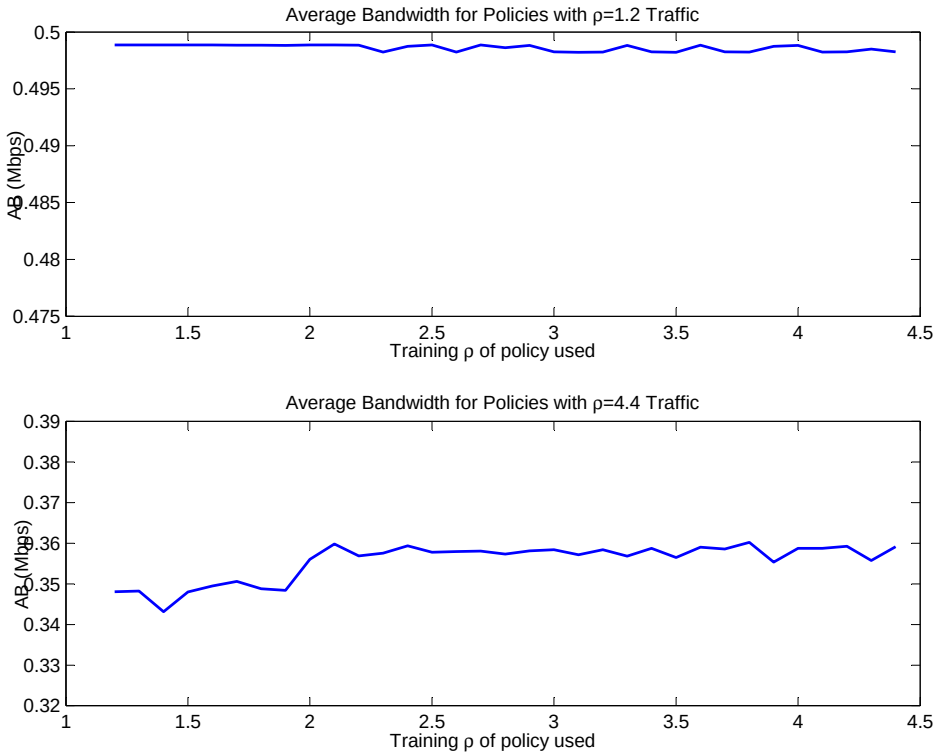

Figure 4.14: AB for 33 policies with $\rho = 4.4$ and $\rho = 1.2$ traffic

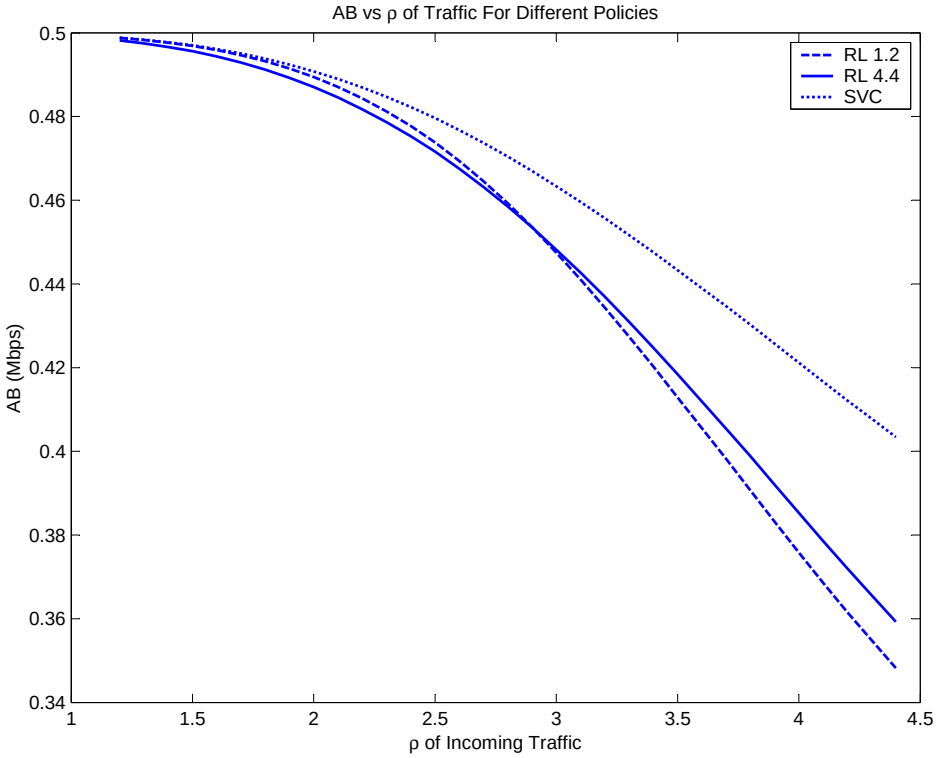

Figure 4.15: AB for 2 policies with varying traffic rate

that the mapping from number of users to number of channels changes by the number of tokens. One might expect that the policy when the number of tokens is set to maximum would be very similar to SVC policy. As the number of tokens decreases in the bucket, the system understands that it has done many bandwidth updates and tries to be more conservative on the number of channels. For lightly loaded traffic conditions the resulting AUR is much less than our imposed DUR of 56 updates/hour as shown in Figure 4.13. AUR being less than imposed DUR means that the token bucket would be full most of the time, therefore the performance of the system would be close to SVC.

Varying Traffic Results

In this part, the policies trained are used by the headend. The arrival rate of the system changes at the start of the hour, and stays the same throughout the hour, this is a piecewise constant traffic generation. Two methods are employed for detection of traffic rate, in the first one number of arrivals are counted for 15 minutes and the next fifteen minutes the arrival rate calculated by this number of arrivals will be used as the traffic estimate, this is called 15 min estimator. In the second one, the arrival rates are calculated again in 15 minute intervals, but this time the highest of these arrival rates for the last hour is used for the next hour, this one is called Cisco Estimator as we have the inspiration for this from Cisco Auto Bandwidth Estimator [50]. The nearest policy to the estimated arrival rate is used.

The traffic rate sinusoidally changes from $\rho = 1.2$ to $\rho = 4.4$. As you see in Table 4.6, SVC has the highest AB. RL and Argiriou have almost the same AB but the RL algorithm has less AUR and MaxUR than Argiriou heuristic. As we have discussed before, Argiriou has no rate control mechanism, therefore for busy times the update rate goes as high as 90 updates/hour. We claimed that RL update rate will never exceed number of tokens plus the token rate, as you

	AB(Mbps)	AUR(U/H)	MaxUR(U/H)	VUR(U^2/H^2)
SVC	0.4646	42.03	188.94	1892.0
Argiriou	0.4491	23.91	90.02	503.02
RL-E	0.4470	23.71	58.52	378.17
RL-W	0.4462	23.28	58.93	378.95
RL-CE	0.4469	23.60	58.93	377.60

Table 4.6: Results for changing arrival rate VUR means variance of update rate, MaxUR means maximum update rate, AUR means average update rate, and AB means average bandwidth

see the MaxUR for RL is about 59 which is less than token rate of 50 plus token bucket size of 10. RL-E uses the 15 min estimator mechanism, RL-CE uses Cisco estimator and RL-W uses only one policy which is trained at the highest arrival rate. As a result, we see that for small arrival rate changes, policy switching does not have a significant advantage as seen from the outputs. Also we have shown above that the difference between these 33 policies are minor, as a result the policy switching method is not beneficial for this example.

4.8.2 100 Channels Case

We have discussed above that we should minimize the state space of the problem using reinforcement learning in order to tackle large problem sizes. The benefit of a larger problem size is that the range of traffic arrival rate change is bigger for a large size problem. In the previous sections, we have seen that policies trained by different traffics with close arrival rates do not differ much. Our expectation is that the policies should differ from each other when their training rates are very distant. From Erlang's B formula the ρ parameter for 100 channels is 84.01 which is much bigger than 4.4 for the 10 channel case.

Policy Training

For this problem, the traffic arrival rate is divided into 34 equal levels starting from 18 and ending at 84 with 2 unit spacing. 34 different RL policies are trained for these 34 quantization levels. The DUR for RL policies is chosen to be 500 which is about a seventh of the SVC policy. The performance of RL policies with respect to the SVC policy are shown in Figures 4.16 and 4.17.

SVC heuristic's update rate linearly increases with the arrival rate, at the end point it is about 3500 updates/hour which is very high. A user streaming video can not tolerate that many updates in an hour which makes almost an update in a second. On the other hand, RL update rate stayed at about 500 all the time, while keeping the bandwidth at about 95 % of the SVC heuristic. This is a substantial improvement because we keep our update rate constant while keeping our bandwidth in close to that of a policy whose update rate increases linearly. In Figure 4.18, the ratio of RL AB to that of SVC is drawn in terms of percentage. As you see the ratio is about 95 % for all traffic rates.

Policy Equivalence

In the previous sections with 10 channels case, we have compared the policies trained with different arrival rates and found them almost identical. This time we have a larger arrival rate space, and the policies are different from each other. In Figure 4.19, the average bandwidth for the 34 policies is drawn. The traffic for the upper curve is constant $\rho = 18.0$ whereas for the lower curve it is $\rho = 84.0$. We have evaluated all the policies with the end point traffics. As you see for the two end points, the policy having the highest AB is the policy trained with that traffic. Also the performance degradation in using a policy different than the trained one is substantial, it is over 30 % for $\rho = 18.0$ case. These results

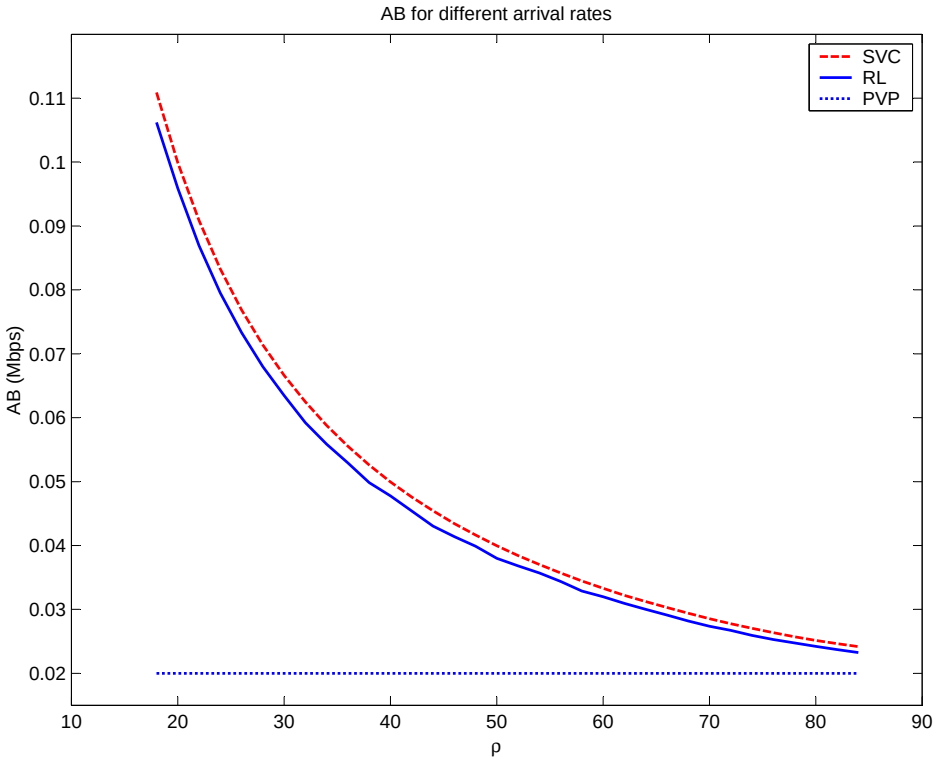

Figure 4.16: Average bandwidth for different arrival rates

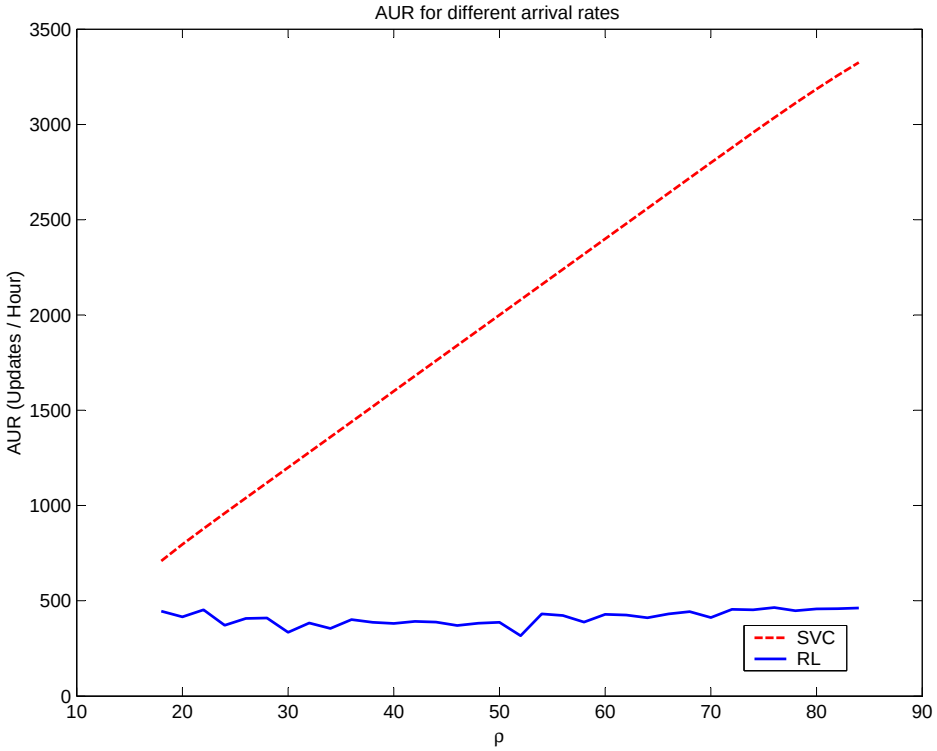

Figure 4.17: Update rate for different arrival rates

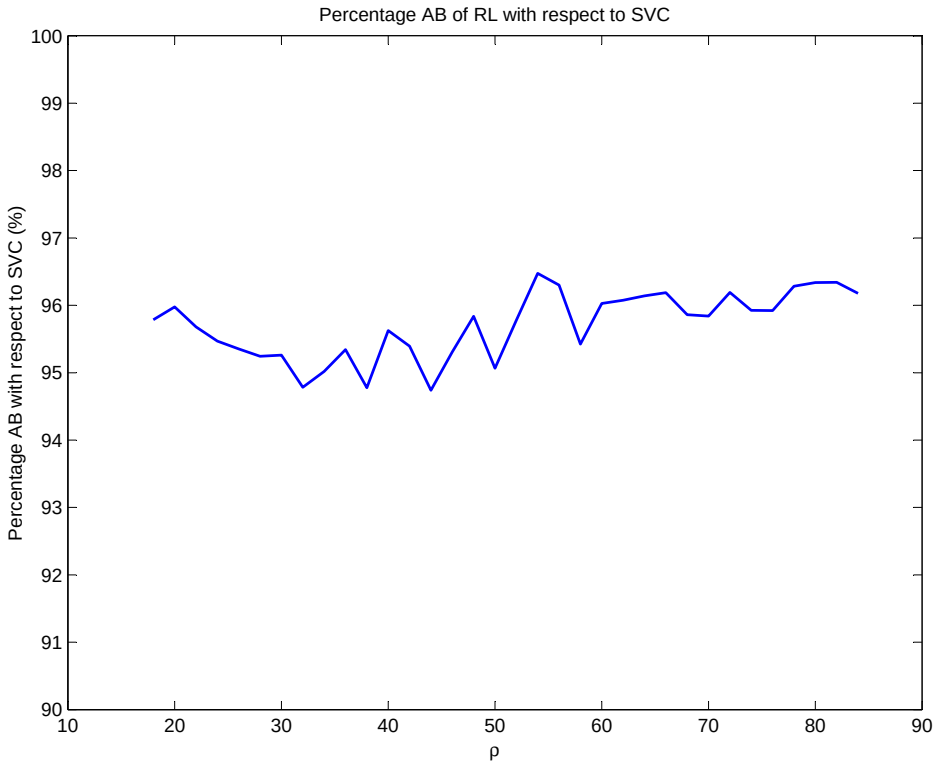

Figure 4.18: Percentage of RL AB with respect to SVC AB for different arrival rates

demonstrate that the policies are different from each other and policy switching by headend will be very useful for this problem.

Varying Traffic Results

The varying traffic for this example is generated using thinning algorithm which is useful for generating non-homogeneous Poisson processes [48]. The interarrival times are exponentially distributed with parameter λ. The rate λ changes continuously with respect to the time of day. The arrival rate is estimated by the headend using a Robbins-Monro type estimator using the observed interarrival times. In Figure 4.20, the generated traffic rate and the estimated traffic rate are depicted. $\mu = 0.00555$ as a result, ρ varies between 18.0 and 84.0 in a sinusoidal fashion.

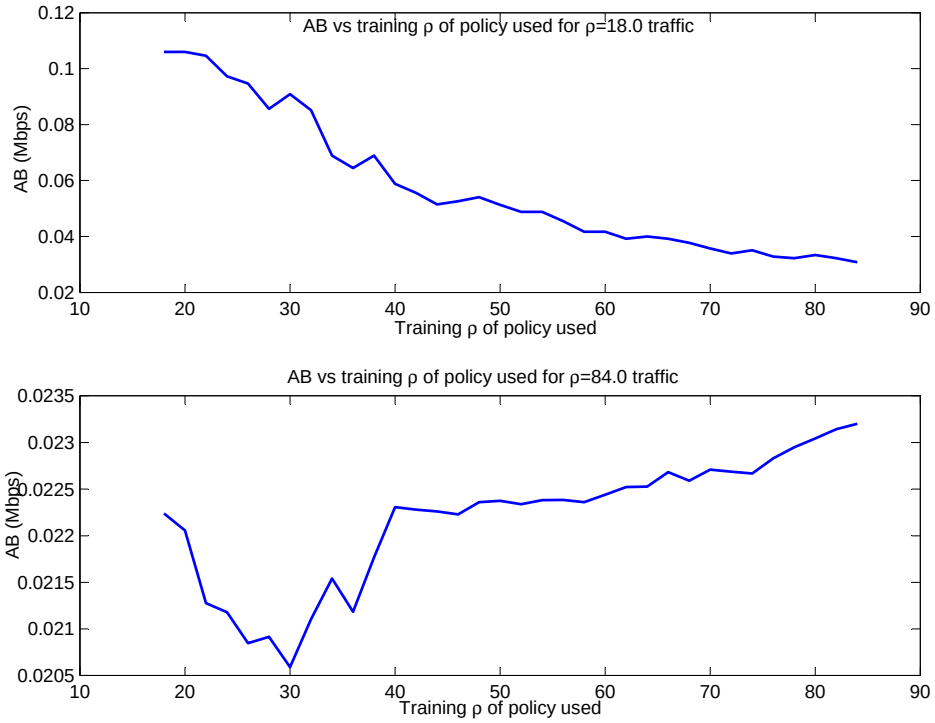

Figure 4.19: Performances of RL policies with two stationary traffic rates

In Figure 4.20, the estimated arrival rate varies rapidly when the arrival rate is high. This is mainly due to the high variance of interarrival times because the variance of exponential distribution is its rate parameter. As we estimate the arrival rate from interarrival times, high interarrival time variance yields high estimator variance. When we apply filtering to estimated values, this problem is eliminated, but then delay between the actual traffic rate and the estimated traffic rate increases. Then the detection mechanism has no use because the selected policy will not be used when it is needed and performance degradations will occur.

The estimated arrival rate will be used to select the appropriate policy. A hysteresis type of function will be used for selecting policies. Consider the 34 level quantization, the distance between levels is 2, assume that estimated ρ is 19, and the policy used is for 44, then the policy trained for 20 would be chosen, the convention is in the way that always the policy with higher training rate is chosen, this is why 18 is not chosen for this case. Later on, the estimated traffic becomes

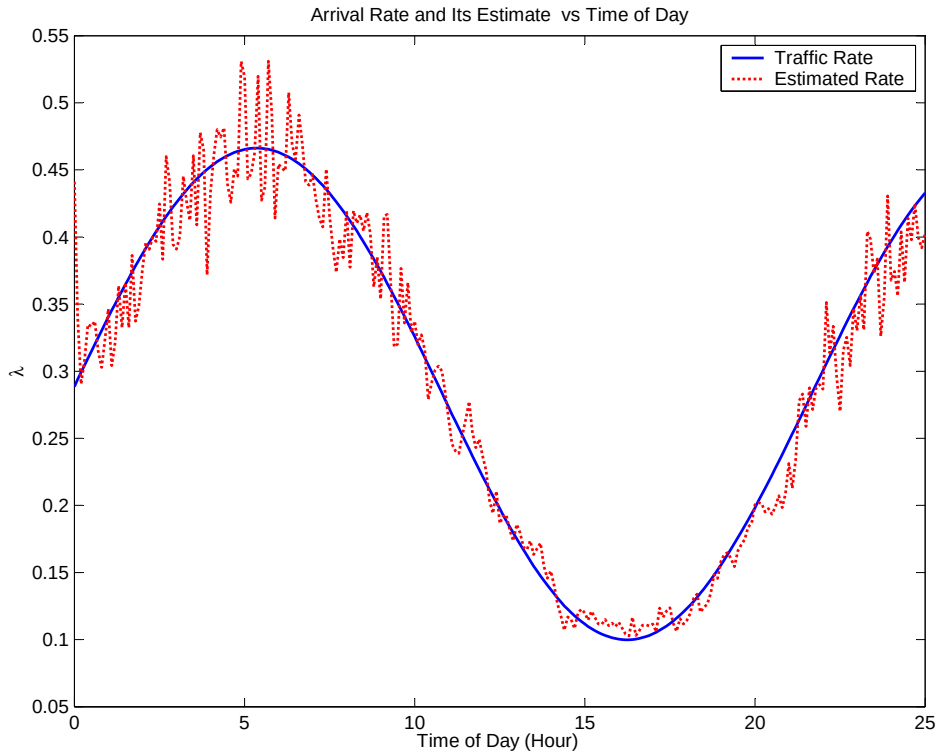

Figure 4.20: Arrival Rate Change and Its Estimate

21.5, the policy will not be changed and 20 will be run because of the hysteresis type of behavior. The policy will not be changed till the estimated traffic is ± 2 of the quantization level. This hysteresis levels are different for different levels of quantization but chosen to be plus minus a quantization distance. Hysteresis is used in order to prevent very rapid policy changes due to oscillations in the estimated traffic rate.

In Table 4.7, the results for RL algorithms with arrival rate estimation mechanism is shown. RL1 denotes the reinforcement learning policy with 1 quantization level whereas RL34 denotes reinforcement learning policies with 34 levels. Number of policy switchings increase with number of levels, this is due to oscillations in the estimated arrival rate, these oscillations are so high that the hysteresis mechanism becomes ineffective after some quantization level is exceeded. The effect of policy changing is obviously seen when RL1 and RL2 is considered, the average bandwidth is almost doubled by the introduction of an additional policy to the system.

Method	AB(Mbps)	AUR(U/H)	APSR(C/H)
RL1	0.0285	179.87	0
RL2	0.0454	299.77	0.738
RL3	0.0463	348.85	1.381
RL5	0.0482	376.13	6.023
RL9	0.0493	405.72	52.498
RL17	0.0498	415.40	159.774
RL34	0.0500	430.51	272.441
SVC	0.0527	2103.0	0
PVP	0.0200	0	0

Table 4.7: Results for changing arrival rate AUR means average update rate, AB means average bandwidth, and APSR means average policy switching rate, U/H=updates per hour, C/H=changes per hour

Figure 4.21 shows the dependency of the average bandwidth on number of quantization levels. The average bandwidth saturates after a number of quantization levels is reached, this is due to similarity of policies with close training traffic rates. When the distance between quantization levels drops under a level, the policies become similar so no gain is obtained. According to policy change rates and bandwidths, 5 quantization levels are enough for this problem. After this value the policy change rate dramatically increases as a result of oscillations in the estimated arrival rate. There is no study done for the number of policy changes in a headend, so no maximum is known for this number. Also a policy change is changing the pointer of the policy to another memory location in the headend which is not very costly in terms of CPU load.

In Figure 4.22, the policy change schedule for 5 and 34 levels of quantization is depicted. The unnecessary policy changes due to oscillations are clearly observed for 34 quantization levels. There are some oscillations even in the 5 level case, because the variance of interarrival times are so big in the upper parts of traffic arrival rate. The delay between actual rate and chosen policy is very small, as a result this mechanism performs well. When the frequency of arrival rate change is very high, the delay of detection and policy choosing becomes important and the system may not work or perform very bad.

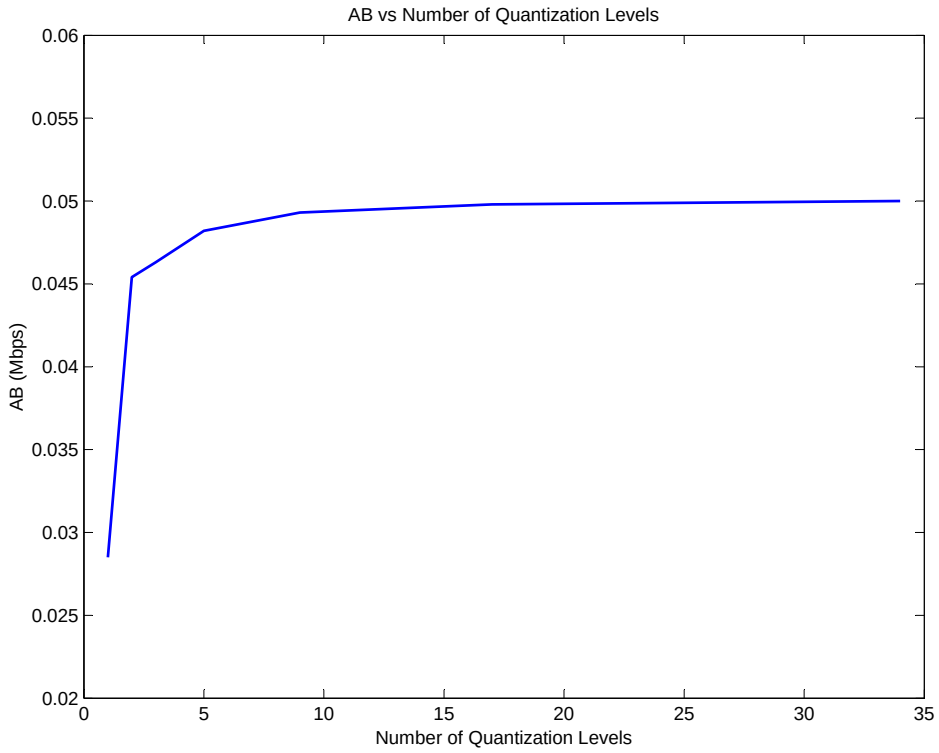

Figure 4.21: Average bandwidth vs number of quantization levels

The difference between SVC bandwidth and RL34 bandwidth is very small, and the difference is mainly due to delay in detecting the policy. Moreover, the oscillations make the system work with a policy that is not suitable to conditions of that time. The average update rate is about 400 for RL policies, we have given the upper limit as 500 while training the policies, average update rate is less than upper limit because the token bucket size is 10 which is not enough for saving tokens in mild load conditions. If the token bucket is made larger, the system will save the tokens when the arrival rate is very low, and use them later when the conditions get worse.

Figures 4.23 and 4.24 shows the performance of RL34 along with the SVC heuristic. The difference in the average bandwidths is indistinguishable whereas the update rates show a great difference. RL update rate is always lower than the minimum update rate of SVC and the performance in terms of bandwidth is 95 %. As a result we can say that switching between RL learned policies is a useful method in environments with changing arrival rates. The weakest point

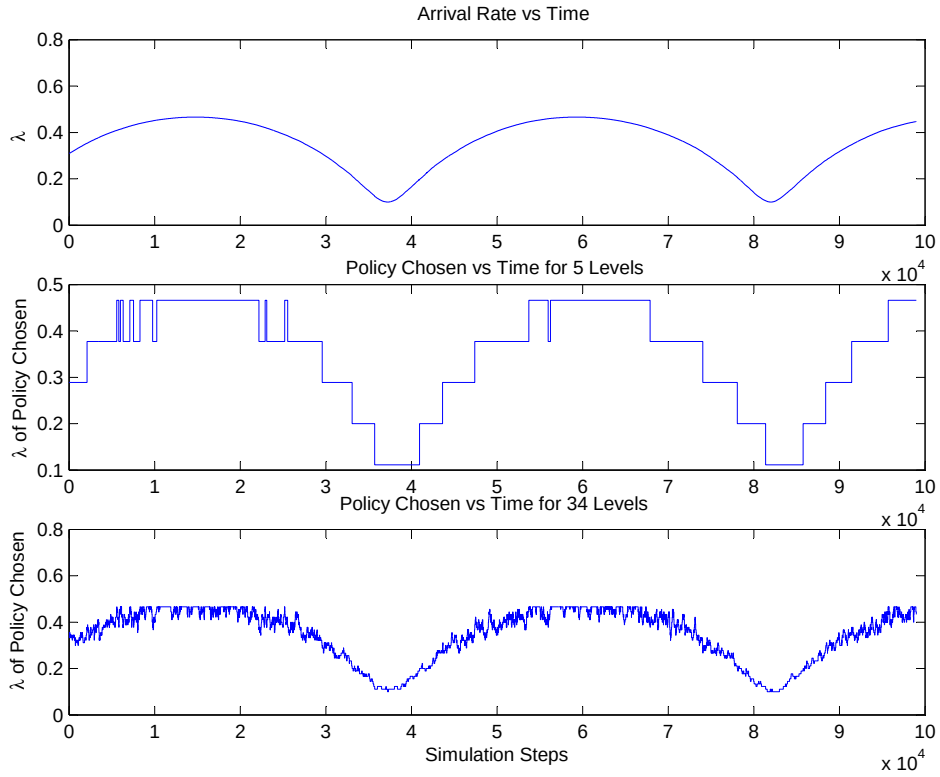

Figure 4.22: Policy change mechanism with 5 and 34 levels

of our proposed methodology is that when the frequency of arrival rate change is very high, the detection delay makes this algorithm ineffective. In order to prove this hypothesis, the frequency is increased fifty times, and the arrival rate now makes 100 peaks in a day instead of 2 peaks as shown in Figure 4.25. The one with 2 peaks is called the Low Frequency (LF), whereas with 100 peaks is called High Frequency (HF) scheme.

In Figure 4.26, the relative performances of high frequency and low frequency arrival schemes to that of SVC are depicted. As the results indicate, when the arrival rate changes rapidly, the performance of our system degrades. This can be noted by the following example, you detect the arrival rate to be 50 in terms of ρ and selected the policy associated with that, but when you begin to run the policy 50, the arrival rate becomes 84 and you are in a worse situation because the policy 50 will not perform well at an arrival rate of 84 therefore the system performance will degrade. This system works in a condition when the arrival

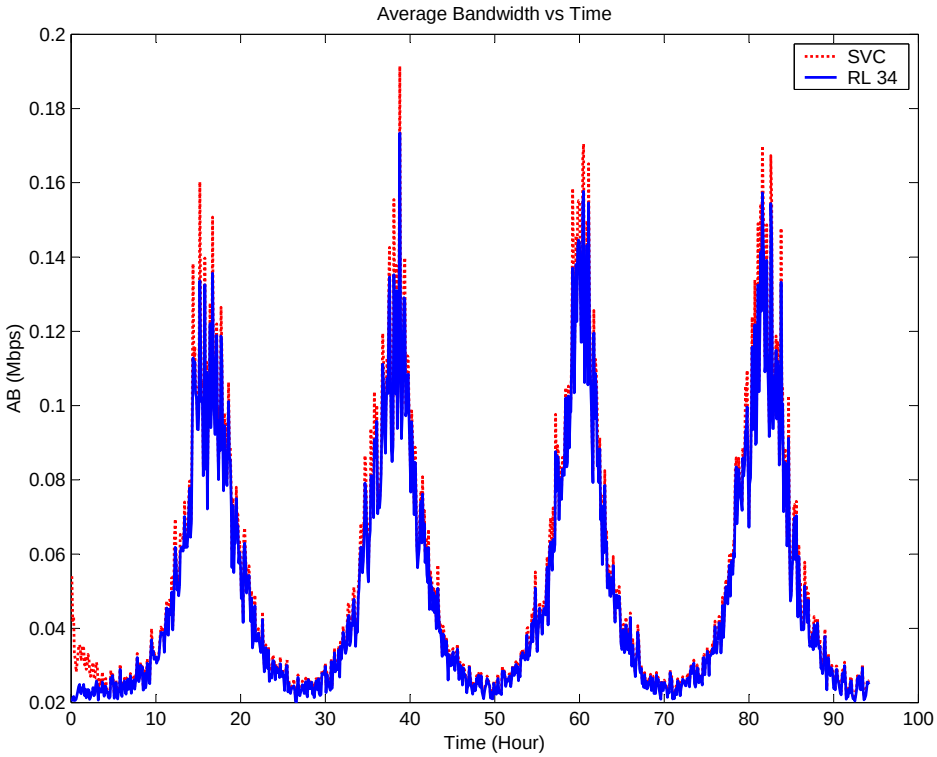

Figure 4.23: Comparison of SVC and RL34

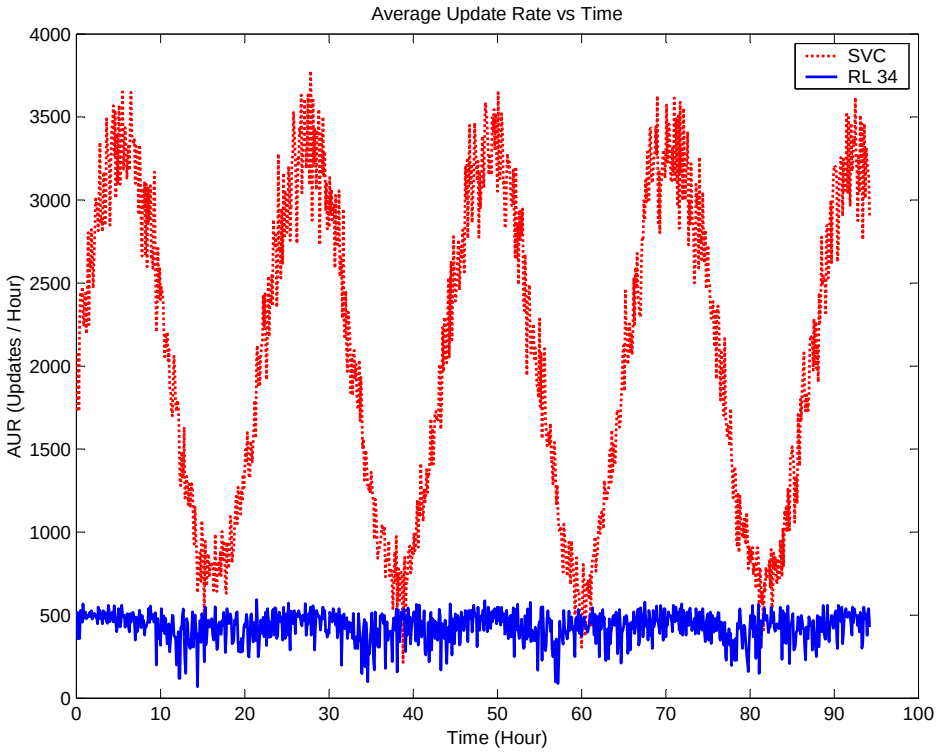

Figure 4.24: Comparison of SVC and RL34

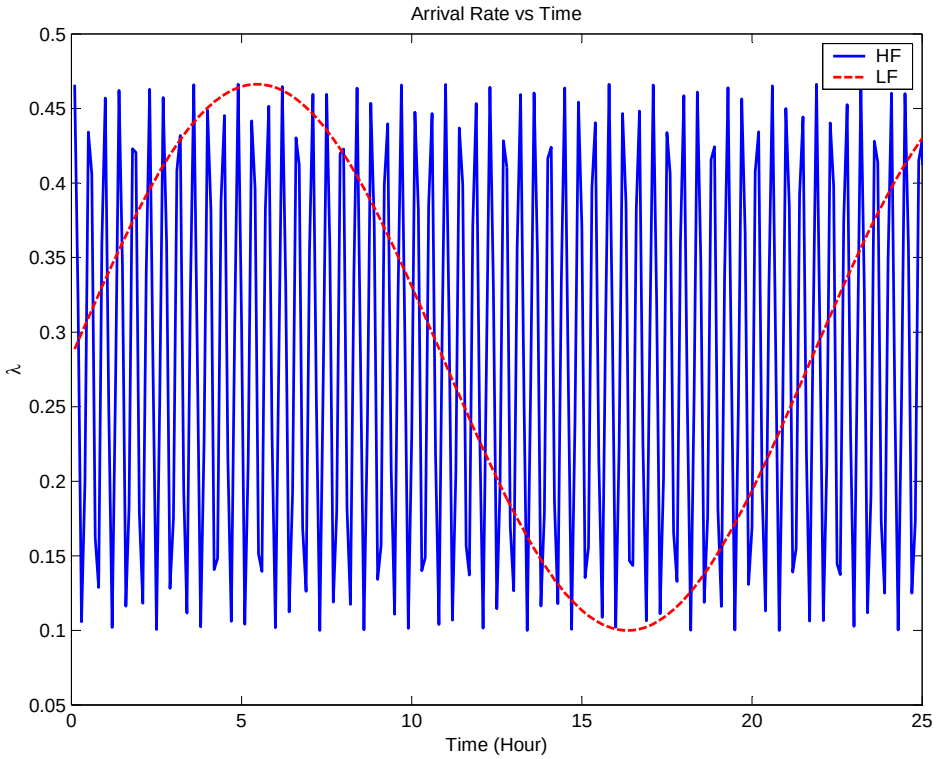

Figure 4.25: High frequency and low frequency arrival rate change

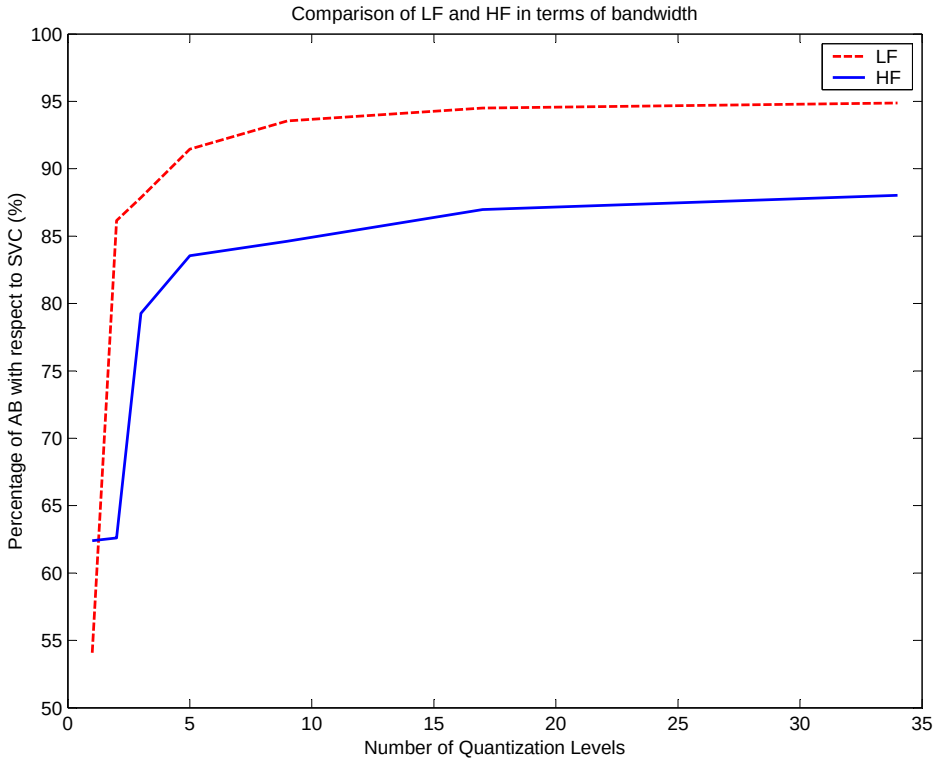

Figure 4.26: Low frequency vs high frequency

rate becomes for example 55 when you select the policy of 50. On the other hand, according to statistics, the telecommunication traffic has not an arrival rate change mechanism such as HF, it is found to be making two peaks in a day. This is the point where our proposal gives very good results.

As a conclusion, RL approach to link sharing problem with a framework of traffic detection and appropritate policy switching can perform as better as SVC with a much less update rate.

Chapter 5

Conclusions

Developments in the Internet technology fostered the use of multimedia applications over the Internet. Multimedia applications have stringent delay and jitter requirements, and intelligent network elements are needed for providing these QoS requirements. Reinforcement learning (RL) methods are proven to be beneficial in applications where some intelligence and adaptivity to changing conditions are required.

Dynamic programming (DP) methods are widely adopted for finding the minimal average cost solutions to a wide range of problems. The major disadvantage of the dynamic programming algorithms is that one has to have a-priori Markovian model for the underlying system. The model requirement reduces the size of the set of problems that could be solved by dynamic programming methods. In many problems, either the model is very hard to derive, or the properties of the system change making it very hard to find a robust system model. On the other hand, reinforcement learning methods do not require any model of the system, and these methods can learn the system either on-line or off-line. The learning mechanism can change its learned policy when a change in the system model is

detected, meaning that the learning continues as the system is working (on-line) which brings adaptivity to reinforcement learning methods.

In this thesis, a reinforcement learning approach to the dynamic link sharing problem is proposed. Unlike proposed heuristics (Argiriou [23], SVC, and PVP), our proposed RL framework includes a bandwidth update rate control mechanism which limits the number of bandwidth updates in unit time to a number that is determined according to user preferences and system properties. A leaky-bucket like algorithm used in the simulation limits the number of updates successfully.

The headend decides on the number of channels on the shared link which determines the bandwidth assigned to each user depending on the previous number of channels and number of active users. We believe that, this thesis is one of the first applications of RL to dynamic link sharing. The results show that RL algorithms can learn policies that are very close to optimal. The system proposed in this work is very complex to model therefore DP algorithms are ineffective. The RL framework proposed in here has a performance very close (about 95%) to the maximum achievable bandwidth while keeping the update rate under a limit in unit time.

In previous works on dynamic link sharing problem, the changing conditions of system in time are not taken into account while developing policies. These proposals have just one policy that is designed for worst case conditions. In this thesis it is shown that some of these proposals perform poorly in mild network conditions as they are designed for the worst case conditions. An adaptive system that can choose the policy that is appropriate for current network conditions can outperform the single policy heuristic. We propose a traffic detection method in this work; the detected traffic rate will be used to select an appropriate policy. These policies are found using the RL approach for a constant rate of traffic and are stored in the headend.

Our proposal has shown robustness by keeping the update rate almost constant, and below a given limit while keeping the performance constant with respect to maximum bandwidth. The update rate of SVC increased linearly with increasing rate of traffic where the update rate of our system is kept constant. The performance in terms of average bandwidth assigned to each user is kept at about 95% of the SVC heuristic despite the limited update rate.

To conclude, in this thesis an RL approach to dynamic link sharing problem is given which is able to track network condition changes and take necessary measures in terms of policy changes where the policies are found by RL algorithms for different network conditions.

The directions that would be taken for future work are as follows:

- Function approximation scheme mentioned in the thesis would be improved in order to have a more robust approximation to original Q values so that one can apply the framework given in here to large problem sizes.
- Different distributions of call holding time for voice traffic would be studied other than exponential distribution in order to show the performance of RL algorithms for various types of distributions.
- Internet traffic can be studied using RL framework. Internet traffic shows burstiness in the traffic rate which does not occur in exponential distribution. This burstiness will enable us to test the performance of RL algorithms in different scenarios.
- The RL algorithms and solutions can be applied to networks (such as voice over IP network) in order to show the gain by the dynamic link sharing. The effects of dynamic scheme is not obvious when a single node is considered, a network environment will help us understand the gain introduced by the dynamic link sharing.

Bibliography

[1] K. Nichols, S. Blake, F. Baker, and D. Black, "Definition of the differentiated services field (DS field) in the IPv4 and IPv6 headers," RFC 2474, 1998.

[2] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated services," RFC 2475, 1998.

[3] H. Perros and K. Elsayed, "Call admission control schemes: A review," *IEEE Magazine on Communications special issue on congestion control*, pp. 82–91, 1996.

[4] J. Wroclawski, "Specification of the controlled-load network element service," RFC 2211, 1997.

[5] J.Wroclawski, "The use of RSVP with IETF integrated services," RFC 2210, 1997.

[6] S. Shenker, C. Partridge, and R. Guerin, "Specification of guaranteed quality of service," RFC 2212, 1997.

[7] The ATM Forum, "ATM Inter-Network Interface (AINI) specification."

[8] Z. Wang and J. Crowcroft, "Quality of service routing for supporting multimedia applications," *IEEE JSAC*, vol. 14, no. 7, pp. 1228–1234, 1996.

[9] R. Guerin, et al, "QoS routing mechanisms and OSPF extensions," RFC 2676, 1999.

[10] M. Grossglauser, S. Keshav, and D. N. C. Tse, "RCBR: a simple and efficient service for multiple time-scale traffic," *IEEE/ACM Transactions on Networking*, vol. 5, no. 6, pp. 741–755, 1997.

[11] B.Vandalore, W. Feng, R. Jain, and S. Fahmy, "A survey of application layer technologies for adaptive streaming of multimedia," *Journal of Real Time Imaging*, vol. 7, no. 3, pp. 221–235, 2001.

[12] A. Eleftheriadis and D. Anastasiou, "Optimal data partitioning of MPEG-2 coded video," in *First IEEE International Conf. on Image Processing*, Texas, USA, 1994.

[13] P. Pancha and M. Zarki, "Prioritized transmission of variable bit rate MPEG video," in *IEEE Globecom'92*, Orlando, USA, 1992, pp. 1135–1138.

[14] N. G. Duffield, K. K. Ramakrishnan, and A. R. Reibman, "SAVE: an algorithm for smoothed adaptive video over explicit rate networks," *IEEE/ACM Transactions on Networking*, vol. 6, no. 6, pp. 717–728, 1998.

[15] J. Bolot and T. Turletti, "A rate control mechanism for packet video in the Internet," in *IEEE Infocom'94*, Toronto, Canada, 1994, pp. 1216–1223.

[16] D. Taubman and A. Zakhor, "A common framework for rate and distortion based scaling of highly scalable compressed video," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, no. 4, pp. 329–354, 1996.

[17] D. Clark and V. Jacobson, "Flexible and efficient resource management for datagram networks," unpublished manuscript.

[18] S. Floyd and V. Jacobson, "Link-sharing and resource management models for packet networks," *IEEE/ACM Transactions on Networking*, vol. 3, no. 4, pp. 365–386, 1995.

[19] C. Adjih, N. Argiriou, M. Chaudier, E. Debert, F. Dumontet, L. Georgiadis, and P. Jacquet, "An architecture for ip quality of service provisioning over catv networks," in *EMMSEC 1999*, Stockholm, Sweden, 1999.

[20] H. Tijms, *Stochastic Models - An Algorithmic Approach*. Chichester, UK: John Wiley & Sons, 1994.

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

[22] D. Bertsekas and J. Tsitsiklis, *Neuro-Dynamic Programming*. Belmont, MA: Athena Scientific, 1996.

[23] N. Argiriou and L. Georgiadis, "Channel sharing by rate-adaptive streaming applications," in *Proc. IEEE Infocom'02 - The Conference on Computer Communications*, New York, USA, 2002, pp. 753–762.

[24] N. Akar and C. Sahin, "Reinforcement learning as a means of dynamic aggregate QoS provisioning," *Lecture Notes in Computer Science*, vol. 2698, pp. 101–115, 2003.

[25] T. Anjali, C. Scoglio, and G. Uhl, "A new scheme for traffic estimation and resource allocation for bandwidth brokers," *Computer Networks*, vol. 41, pp. 761–777, 2003.

[26] B. Widrow and M. E. Hoff, "Adaptive switching circuits," in *IRE WESCON*, New York, 1960, pp. 96–104.

[27] S. Mahadevan, "Average reward reinforcement learning: Foundations, algorithms, and empirical results," *Machine Learning*, vol. 22, no. 1-3, pp. 159–195, 1996.

[28] D. Bertsekas, *Dynamic Programming: Deterministic and Stochastic Models*. Engelwood Cliffs, New Jersey: Prentice Hall, 1987.

[29] R. Howard, *Dynamic Programming and Markov Processes*. Cambridge, MA: MIT Press, 1960.

[30] D. White, "Dynamic programming, Markov chains, and the method of successive approximations," *Journal of Mathematical Analysis and Applications*, vol. 6, pp. 373–376, 1963.

[31] A. Jalali and M. Ferguson, "A distributed asynchronous algorithm for expected average cost dynamic programming," in *In Proceedings of the 29th IEEE Conference on Decision and Control*, Honolulu, HI, 1990, pp. 1394–1395.

[32] C. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Cambridge, England, 1989.

[33] J. Tsitsiklis, "Asynchronous stochastic approximation and Q-learning," *Machine Learning*, vol. 16, pp. 185–202, 1994.

[34] A. Schwartz, "A reinforcement learning method for maximizing undiscounted rewards," in *Prooceedings of the Tenth International Conference on Machine Learning*, MA, USA, 1993, pp. 298–305.

[35] S. J. Bradtke and M. O. Duff, "Reinforcement learning methods for continuous-time Markov decision problems," in *Advances in Neural Information Processing Systems*, G. Tesauro, D. Touretzky, and T. Leen, Eds., vol. 7. The MIT Press, 1995, pp. 393–400.

[36] T. Das, A. Gosavi, S. Mahadevan, and N. Marchalleck, "Solving semi-Markov decision problems using average reward reinforcement learning," *Management Science*, vol. 45, no. 4, pp. 560–574, 1999.

[37] A. Gosavi, "A convergent reinforcement learning algorithm for solving Markov and semi-Markov decision problems under long-run average cost," *Accepted in the European Journal of Operational Research*, 2001.

[38] R. Jacobs, "Increased rates of convergence through learning rate adaptation," *Neural Networks*, vol. 1, pp. 295–307, 1988.

[39] C. Darken, J. Chang, and J. Moody, "Learning rate schedules for faster stochastic gradient search," in *Proc. Neural Networks for Signal Processing 2*, Piscataway, NJ, 1992, pp. 3–13.

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

[41] L. Lin, "Reinforcement learning for robots using neural networks," Ph.D. dissertation, Carnegie-Mellon Univ, Pittsburgh, PA, 1993.

[42] R. Sutton, "Integrated architectures for learning, planning and reacting based on approximating dynamic programming," in *Proceedings of the Seventh International Conference on Machine Learning*, Austin, TX, 1990, pp. 216–224.

[43] H. Tong and T. X. Brown, "Reinforcement learning for call admission control and routing under quality of service constraints in multimedia networks," *Machine Learning*, vol. 49, pp. 111–139, 2002.

[44] J. N. Tsitsiklis and B. V. Roy, "An analysis of temporal-difference learning with function approximation, Tech. Rep. LIDS-P-2322, 1996.

[45] E. Ayanoglu and N. Akar, "Broadband integrated services network," in *Wiley Encyclopedia on Communications*, J. G. Proakis, Ed. Hoboken, NJ: John Wiley & Sons, 2003.

[46] B. Groskinsky, D. Medhi, and D. Tipper, "An investigation of capacity control schemes in a dynamic traffic environment," *IEICE Trans. Commun.*, vol. E84-B, pp. 263–274, 2001.

[47] H. Robbins and S. Monro, "A stochastic approximation method," *Annals of Mathematical Statistics*, vol. 22, pp. 400–407, 1951.

[48] P. Lewis and G. Shedler, "Simulation of nonhomogeneous poisson processes by thinning," *Nav. Res. Logistics Quart.*, vol. 26, pp. 403–414, 1979.

[49] L. Baird, "Reinforcement learning through gradient descent," Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, 1999.

[50] "Cisco MPLS autobandwidth allocator," White Paper, Cisco, 2001.