NOTE TO USERS

Page(s) not included in the original manuscript and are unavailable from the author or university. The manuscript was scanned as received.

v

This reproduction is the best copy available.

UMI®

Adaptive Algorithms for Routing and Traffic Engineering in Stochastic Networks

By

Sudip Misra, B.Sc. (Honours) (IIT Kharagpur), M.C.S. (New Brunswick)

A thesis submitted
to the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Ottawa-Carleton Institute of Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario
November, 2005

Your file Votre référence
ISBN: 0-494-13394-5
Our file Notre référence
ISBN: 0-494-13394-5

Dedicated to My Family

Acknowledgements

I would like to extend my sincere gratitude to those people who helped me in different ways during the course of my Ph.D. program. First and foremost, I am undoubtedly most indebted to my Thesis supervisor, Professor John Oommen, for the tremendous amount of encouragement, guidance, and support he has provided to me during the different phases of the program. Professor Oommen has not only demonstrated to me the qualities of an excellent supervisor, but also those of a nice human being.

The motivation for working on the problems discussed in this Thesis arose partially during the course of my full-time R&D work on routers and gateways at Nortel Networks and Atreus Systems Corporation, Ottawa. I discussed some of the problems facing the telecommunication industries with Professor Oommen, and we, together, agreed to try to use Learning Automata as a tool to solve them. I am grateful to him for introducing to me the extremely interesting area of Learning Automata.

I thank Professor Jean-Pierre Corriveau for all his advice and direction during my Doctoral program. All the administrative staff at the School also deserve my sincere gratitude.

Special thanks are also due to Alonso Perez, Sandy McBride of ChartWELL Inc., Kamel Toubache, and Edith Lam of the Government of Ontario for granting me the flexibility to work on my Ph.D. while I was employed. I am extremely grateful to Dr. Isaac Woungang for the several days of hard work he put along with me for converting the Thesis in LaTex format.

Finally, and most importantly, I would like to extend my heartfelt gratitude to my family members for the continuous help, support, and guidance they have provided to me so far in life. I thank my grandparents for instilling the values of life in me, and my parents for working extremely hard to provide me with good education. My brother and sister have been understanding and supportive all along, and for this I am particularly grateful.

Abstract

In this Thesis, we document the results of our investigation of four important problems in the domain of telecommunication network routing and traffic engineering. The techniques we use involve stochastic learning and Learning Automata (LA).

Our first contribution consists of two efficient solutions for maintaining single-source shortest path routing trees in networks, where the weights of the links connecting the nodes of the network change continuously in a random manner following an unknown stochastic distribution.

In the second problem, we were interested in maintaining shortest paths between *all* pairs of nodes in a dynamically changing network. Again, for this problem, we have proposed two LA-based efficient solutions.

Our third contribution was in the area of QoS routing and traffic engineering in networks. More specifically, we have proposed an adaptive online routing algorithm that computes bandwidth-guaranteed paths in MPLS-based networks, using a Random-Race-based learning scheme that computes an optimal *ordering* of routes.

The last problem that we studied was that of designing routing schemes that would successfully operate in the presence of adversarial environments in Mobile *Ad Hoc* Networks (MANETs). The need for fault tolerant routing protocols for MANETs was identified recently by Xue and Nahrstedt, and in our research, we have proposed a new fault-tolerant routing scheme that uses a stochastic learning-based weak estimation procedure.

All our proposed solutions have been demonstrated to be superior to the state-of-the-art.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this Thesis[1], we report the results of a research endeavor involving challenging problems in the areas of *network routing* and *traffic engineering* (TE) in telecommunication networks. Since the scope of these areas is extensive, we have restricted our studies primarily to four problems (described in Sections 1.1.1 through 1.1.4) that are quite fundamental in the area of telecommunication networks. We have proposed adaptive algorithms that are capable of operating in dynamic network environments, for example, in those where the individual status of the links constantly change in a random manner with an underlying (unknown) probabilistic distribution. The particular details of the behavior of the dynamic environment that we have considered for each of the reported problems are listed in the corresponding Chapters. In this attempt, we had used results reported in the area of artificial learning theory (particularly using *Learning Automata* (LA) [66, 71], and *Random Races* (RR) [64]) to address the problems investigated. We have also experimentally established the performance of our algorithms. The solutions proposed by us in this Thesis, for *all* the addressed problems, have demonstrated a superior performance when compared to the solutions

[1]The paper containing some of the works reported in this Thesis won the **ACM SIGART/AAAI Doctoral Consortium Award** sponsored by the United States National Science Foundation (NSF) and Microsoft Corporation.

characterizing the state-of-the-art.

Although research was primarily targeted to problems relevant in telecommunication networks, given the generic nature of the problems we studied, we believe that most of these results will be equally applicable for solving analogous problems in other application domains such as transportation, spatial databases, and aerospace. For example, the solutions we have reported for the dynamic shortest path problems are intuitively applicable for all application domains where the cost of the links are updated continuously in a random fashion, and where the user wants to traverse the shortest paths between the different nodes in the network. Thus, in our solutions to the problems, we have avoided making any application domain specific assumptions. For example, we believe that our solution for fault-tolerant routing in Mobile *Ad Hoc* Networks (MANETs), can also be extended to fault-tolerant routing of cargo in civil and military applications.

This Thesis is intended to have an "application" flavor – it is not meant to be theoretical. However, we made every effort, wherever possible, to provide the formal rationale for why our proposed solutions work. Theoretical considerations of the problems studied in this Thesis have their own challenges, and are far from trivial. For example, for the automata learning solutions to the dynamic shortest path problems, the system that we encounter (to be discussed in further detail in Chapters 4 and 5) is a *game* of LA, because each automaton is not only responding to the Environment (the randomly changing network topology), but also to the collective responses of the other automata. In this setting, not only do we have no analysis, but we are also unaware of any available *tools* of analysis. Indeed, most of the currently available analysis for games of automata have to do with well structured games, for example, two (or multi-person) zero-sum games.

The work reported here essentially utilizes the formal proven theoretical results available in the corresponding published pieces of literature, with an attempt to en-

hance them to solve the respective problems. The proposed algorithms were tested rigorously for numerous artificial random network topologies in various settings, so as to verify their properties, and to see if the results obtained are consistent.

1.1 Motivation and the Proposed Problems of Study

The primary goal of this research endeavor was to address some of the shortcomings present in a *selected* set of existing network routing and traffic engineering algorithms. We also proposed to use artificial learning techniques to design algorithms that would adapt to the changes in the environment in which they operate.

LA [66, 71] have been traditionally used to model biological learning systems. They have also been used for learning an *optimal* action out of several possible actions that a random Environment offers. Upon choosing an action, the learning automaton is either rewarded or penalized by the Environment with a certain probability. As time progresses, the automaton eventually attempts to choose the best possible action. The learning loop involves two entities, the *Random Environment* (RE) and a *Learning Automaton.* Learning is accomplished by actually interacting with the Environment and processing its responses to the actions that are chosen, while gradually converging toward an ultimate (hopefully, optimal) goal.

While the Learning Automaton attempts to find an optimal action out of several possible actions, it fails if the system is required to determine an optimal *ordering* of the actions (instead of merely determining a single optimal action). This is the problem we encountered while attempting to solve the *Quality-of-Service* (QoS) and the TE routing problem.

In the Thesis, we have addressed the following fundamental, yet challenging, inter-related problems:

(i) Dynamic Single-Source Shortest Path Routing

(ii) Dynamic All-Pairs Shortest Path Routing

(iii) Quality-of-Service (QoS) & Traffic Engineering (TE) Routing

(iv) Fault Tolerant Routing in MANETs

As mentioned, we have succeeded in proposing *adaptive learning algorithms* for all these problems, and have evaluated their performance with respect to some of the best known solutions. We summarize each of the problems below, and their respective solutions, later in the Thesis.

1.1.1 Dynamic Algorithms for Single-Source Shortest Path Routing

The *Single-Source Shortest Path Problem* (SSSP) consists of finding shortest paths from one node to all others in a network. This problem becomes particularly interesting in *dynamic* scenarios where the topology and structure of the network constantly change – i.e., the network links are inserted and deleted from the graph, and the link-costs are continuously randomly changing. In such an environment, we consider the problem when the system attempts to *maintain* shortest paths between a single node and all other nodes in the network. This problem is non-trivial when we seek a solution that does not re-compute everything "from scratch" following each topology update.

Although some solutions have been reported for certain settings, the relatively difficult, but more realistic network scenarios are fully dynamic, i.e., where the routing algorithms should handle multiple heterogeneous changes in the structure of the networks, besides the changes in the link-costs. This problem has received very limited attention in the literature (e.g., [31], [82]). We have proposed two solutions, which have been *published* in [54], [55], [56], and [58].

Prior to this, to our knowledge there has been no reported solution to the dynamic

SSSP problem using LA. In the past, LA solutions of various other problems have shown superior results. In this regard, we have, in our research, investigated this problem, and proposed LA-based solutions, which are superior to the currently available ones using a *linear reinforcement learning* scheme (details in Chapter 4) [54, 58]. The algorithm has been rigorously experimentally evaluated, and has been found to be significantly superior to the algorithms previously available in the literature. In particular, our solution can be used to determine the shortest path for the "statistical" average network, which converges irrespective of whether there are new changes in link-costs or not. As opposed to this, the existing algorithms will fail to exhibit such a behavior and would recalculate the affected shortest paths after each link-cost update.

In another piece of work [55, 56], we have also proposed a new set of learning algorithms for maintaining such shortest paths using the *generalized pursuit learning* technique that pursues a set of actions at any time instant, instead of a single action. This solution is also described in Chapter 4.

1.1.2 Dynamic Algorithms for All-Pairs Shortest Paths Routing

As the name indicates, the problem of determining *All-Pairs Shortest Paths* (APSP) concerns finding shortest paths between all-pairs of nodes in a network. The problem that we encounter, in such environments, is that of designing efficient algorithms that can *maintain* the shortest paths between all pairs of nodes in the network, even though there are updates on the structure of the graph by virtue of increase or decrease in edge-weights. As in the dynamic SSSP problem, this problem is also non-trivial when we do not want to re-compute everything "from scratch" following each update in network link costs. Our proposed solution has been *published* in [57].

Prior to our solution, the state-of-the-art solution to this problem (considering

general graphs, real valued edge-weights and no constraints on the maximum value that the edges can assume) was proposed by Demetrescu and Italiano [21]. Note that this problem is "hard" if the edge-weights are assumed to be real-valued, or if the maximum value that can be assumed by any edge is unconstrained - alternate solutions are available for the case when such constraints were not involved (e.g., [35], [40]). Demetrescu and Italiano [21] proposed a remarkable algorithm in 2003, that solved the same problem for general digraphs with edge-weights that can assume positive real-values but with a substantially improved running time per edge-update operation. The description and limitations of their algorithm are described in detail in Chapter 5.

As in the previous problem, we used LA to solve the dynamic version of the problem where the edge-weights change randomly following an (unknown) stochastic distribution. We proposed a *new algorithm* [57] for environments in which the edge-weights change stochastically and continuously, and where we require the algorithm to converge to the underlying "average" solution. The existing algorithms (mentioned above) would fail to converge to such a solution.

The major *challenge* that we faced was that we could not extend the LA-based solution proposed for the dynamic SSSP problem to the dynamic APSP problem, because we wanted to avoid maintaining the shortest path tree at every node in the network. The reasons for this were two-fold. First of all, maintaining a shortest path tree at every node in the network is inefficient. Secondly, and most importantly, there appears to be no straight-forward strategy by which we could use our "single-source" methodology of maintaining an action probability vector at each node in the network, because an edge that could potentially lie in the shortest path between a pair of nodes, may not necessarily lie in the shortest path between another pair of nodes. To address this challenge we introduced a novel data-structure to store and update the set of action probability values. These will be discussed in more detail in Chapter 5.

We assessed the performance of our proposed algorithm through rigorous experi-

mentation. We established that on an average our algorithm achieves a significant level of improvement over the existing algorithms.

1.1.3 Adaptive Quality of Service & Traffic Engineering Routing

The area of QoS routing is concerned with selecting routing paths while meeting strict end-to-end service requirements involving resource constraints, while achieving optimum throughput in the network. Two basic considerations in QoS routing in integrated services packet-switched networks concern: (1) routing traffic with bandwidth guarantees, and (2) routing traffic with delay guarantees. Some of the algorithms proposed traditionally for solving the former[2] class of problems are the *Widest-Shortest Path* (WSP) [34], and the *Shortest-Widest Path* (SWP) [106] algorithms. More recently, Vasilakos *et al.* [104] proposed the SELA routing algorithm for QoS routing in *Asynchronous Transfer Mode* (ATM) networks.

The traditional QoS Routing algorithms, viz., WSP, SWP, do not provide routing decisions considering the underlying traffic distribution in a network, and therefore can lead to under-utilization of network resources. One of the recent TE algorithms, which has gained widespread popularity is called the *Minimum Interference Routing Algorithm* (MIRA) [42]. The algorithm, although elegant, is complex, computationally intensive, and is based on minimum-interference theory.

In our research, we designed an efficient adaptive online routing algorithm for the computation of bandwidth-guaranteed paths in Multiprotocol Label Switching (MPLS) based networks, using a learning scheme that computes an optimal ordering of routes. This work has *two-fold* contributions. The first is that we propose a new class of solutions other than those available in the literature incorporating the family of stochastic

[2]We do not address the later class of problems in this Thesis.

Random-Races algorithms. The most popular previously proposed MPLS TE solution attempted to find a superior path to route an incoming path setup request. Our algorithm, on the other hand, tries to learn an *optimal ordering* of paths through which requests can be routed according to the *rank* of the paths in the order learnt by the algorithm. The second contribution of our work is that we have proposed a routing algorithm that has better performance than the important algorithms in the literature. Our conclusions are based on three important performance criteria: (i) the rejection ratio, (ii) the percentage of accepted bandwidth, and (iii) the average route computation time per request. While some of the previously proposed algorithms were designed to achieve low rejections and high throughput of route requests, they are unreasonably slow. Our algorithm, on the other hand, in general, attempts to reject the least number of requests, achieve the highest throughput, and compute routes in the fastest possible time, as compared to the algorithms we used as benchmarks for comparison.

This work is under consideration of publication.

1.1.4 Fault-Tolerant Routing Algorithms for Mobile *Ad Hoc* Networks

MANETs have currently received a significant amount of attention among researchers, because of their cost-effectiveness, and ease of being deployed in terrains where there are no fixed communication infrastructures. In MANETs, the mobility of the nodes introduces continuously changing infrastructures. If a node moves out of the range of operation, other nodes will not be able to communicate with it. Additionally, there are no designated routers for making routing decisions. All nodes in MANETs take part in routing by acting as routers for one another. Further, the mobile nodes operating in such environments are typically very low powered. These are some of the features of MANETs that makes them susceptible to being very fault-prone.

One of the challenging issues facing routing in MANETs is fault tolerant routing [110, 111]. The idea is to efficiently route packets in the presence of "misbehaving" nodes (for example, nodes that drop packets).

The traditionally proposed routing algorithms (e.g., [39], [52], [67], [77]), route packets assuming that the environment behaves perfectly, that there are no adversaries in the network, and that all the nodes in the network are well-behaved and cooperate with each other [111]. However, in reality, MANETs are susceptible to the introduction of a plenty of faulty nodes, because of the way in which they operate. The performance results of the existing routing algorithms in such environments will be significantly degraded. For example, it was shown that if there are 20% faulty nodes in a network, the normal performance of one of the popular algorithms used in many currently deployed MANETs, called *Dynamic Source Routing* (DSR) algorithm, degrades by approximately 30% [111]. Designing a solution for the problem of fault tolerant routing in MANETs is inherently hard [110]. While the existing multipath routing algorithms [61] are capable of achieving a high packet delivery, they introduce a high overhead on the network. On the other hand, the unipath routing algorithms (e.g., DSR [39]) are low-overhead algorithms, but they are very poor in fault-tolerance.

Xue and Nahrstedt [110], in a pioneering paper, addressed this problem. They proposed a solution which is capable of reducing the overhead introduced by the multipath routing algorithm, while guaranteeing a certain level of successful delivery of packets.

In our research, we have attempted to propose an extended solution to the problem. By incorporating a estimation scheme based on stochastic learning [72, 73], we were able to propose a solution that is capable of further reducing the overhead by about $40 - 50\%$. We assumed non-stationary environments, particularly where the nodes become increasingly likely to be faulty as they move away from the center of their region of operation. We established the superiority of our new algorithms by rigorous experimentation.

1.2 Practical/Industrial Usefulness of the Studies

Although the superiority of proposed adaptive learning solutions can be verified, it is profitable to mention a few potential applications. Although the proposed problems were studied as problems of interest in telecommunications, they can also find use in other application areas, such as transportation, aerospace, and the military. We catalogue below some of the many potential benefits of our novel solutions.

1.2.1 Usefulness of the Dynamic Shortest Path Routing Algorithms

As will be discussed in Chapters 4 and 5, there are many algorithms proposed in the literature to solve these problems. They can be broadly classified as: (i) the Static Shortest Path Algorithms, such as those of Dijkstra's [19], and Bellman-Ford's [10], and (ii) the Dynamic Shortest Path Algorithms (e.g., [82], [31]). The traditional static shortest path algorithms are undoubtledly of little use in dynamic applications because using them would involve re-computing the shortest paths "from scratch" following unit changes in the network topology. However, although the dynamic algorithms, as such, address these limitations of the static algorithms, in such scenarios, the prior dynamic algorithms are inefficient, and have inherent limitations discussed in Chapters 4, and 5. We believe that our proposed LA-based algorithms are ideal for use in such application areas. We base this claim on the results already obtained.

First of all, the application of our algorithms in telecommunication is straightforward. Furthermore, in the *transportation* domain, there are complex road networks in urban areas. The costs of routing shipments from a warehouse to (all) other retail outlets constantly change. Changes occur because of a range of reasons such as, road constructions, accidents, traffic jams, office hours, and the presence of emergency vehicles. There are often several alternative routes that can be taken by a vehicle, whenever

the predetermined route is unsuitable to deliver the shipments in time. Shipment vehicles may be equipped with suitable technology (like GPS) to guide them through alternative routes. The proposed learning algorithms can then be used to achieve such a vehicle routing. The same arguments would be true if aeroplanes have to be redirected adaptively to take care of changes.

In the *military*, the proposed learning algorithms also have several potential applications. For example, in complex military networks, ground forces may have to be rapidly rerouted based on sudden changes in enemy information and strategies along existing routes, and self-guided missiles may need to change their trajectories (within micro-seconds) to account for changes in the path along which it travels.

1.2.2 Usefulness of the QoS and TE Routing Algorithms

The usefulness of QoS routing is not new. QoS routing is quite popular in the telecommunications industry because of the increased demand for satisfying multiple customer demands, and to obtain increased utilization of network resources, while satisfying the varied user requirements.

The area of TE routing arose to prominence relatively recently to satisfy the same goals of QoS while optimizing the usage of valuable network resources by distributing the load among different paths. The commercialization of TE routing algorithms has not been as much as that of the QoS routing algorithms. There are not many commercial networks that run the existing TE algorithms (like the ones proposed in [42], and [92]). The research in TE is thus relatively less mature than in the field of QoS routing.

Our work was an extension of the existing works done in the above-mentioned two areas. We designed an efficient adaptive learning algorithm that would successfully operate in realistically occurring network environments, where the network resources

change dynamically in a stochastic manner. The problem that we encountered was to determine how to efficiently and optimally order the different competitive routes occurring between a pair of source-destination nodes.

1.2.3 Usefulness of Fault Tolerant Routing Algorithm

In Section 1.1.4, we had reviewed the usefulness of fault tolerant routing in MANETs. Most existing routing algorithms popularly deployed in MANETs suffer from the above problem. Both Xue and Nahrstedt's algorithm [110], and our proposed algorithm can be very useful in achieving a balance between the two sides of the above mentioned problem. Our solution will, indeed, prove to be very useful in non-stochastic environments, to efficiently route packets in the presence of faulty nodes in the network. Our solution, if deployed in the field, could also be very useful for significantly reducing the high overhead problem plagued by the currently popular algorithms that guarantee a high packet delivery rate.

Although we introduced our solution keeping in mind the MANET network environments, we believe that our solution can be extended for fault tolerant routing in other network environments as well. For example, *Wireless Sensor Networks* are another class of networks that have become very popular recently. Like the MANET nodes, the sensor nodes are also very low powered. So, lowering the amount of overhead is also an important issue in such networks. Although we did not attempt to extend and test our proposed fault tolerant routing algorithm on Wireless Sensor Networks, we believe that our solution could be extended to such network types as well.

Finally, we would like to reiterate that due to the generic nature of our estimation solution for fault tolerant routing, we believe that our solution could also be applicable for network domains other than MANETs. As noted earlier, we believe our solution for fault-tolerant routing in MANETs can also be extended for fault-tolerant routing

of cargo to different (mobile) stations in civil and military transportation applications.

1.3 Organization of the Thesis

This Thesis consists of eight Chapters – including this introductory Chapter.

Chapter 2 provides a comprehensive summary of the area of Learning Automata. It provides a foundational overview of the fundamental concepts of the topic, and presents the different conventional and recently proposed learning algorithms. In this Chapter we also present experimental results from the existing pieces of literature, and how they compare with the other algorithms. These results justify the choice of the different schemes we would be using in our research as tools for devising our algorithms.

Chapter 3 provides a brief comprehensive review of the interesting field of learning using Random Races. We show how the existing "Teacher"-"Student" interaction based feedback loop, where the "Student" makes a selection, and the "Teacher" rewards or penalizes the "Student", falls short in problem domains which involve the optimal ordering of actions, and require not just the choice of a single optimal action. In this Chapter, we present results that justify the choice of the different schemes (e.g., *multiple race-track learning* in the absence of handicaps) that we would use in our research as tools for devising our algorithms.

After these foundational Chapters, in Chapters 4, 5, 6, 7 we present the four classes of routing and traffic engineering problems that we studied. In these Chapters, we first present the state-of-the-art of the respective problem studied, and then provide details of our proposed solution, and the results obtained from the several experiments conducted to establish their superiority.

In Chapter 4, we discuss the problem of determining single-source shortest paths, and argue the unsuitability of the popular static solutions [10, 19] for use in dynamic stochastic networks, present a summary of the existing dynamic algorithms, and then

describe in considerable detail, two remarkable solutions by Ramalingam and Reps [82], and Frigioni *et al.* [31]. We then present the work we have done so far in this area, describe our proposed algorithms, and present the experimental results that we have obtained.

In Chapter 5, we discuss the problem of determining all-pairs shortest paths, reason the unsuitability of the popular static solutions [19, 28] for use in dynamic stochastic networks, present a summary of the existing dynamic algorithms, and then describe the pioneering effort undertaken by Demetrescu and Italiano [21], and the solution they proposed. Then we present the work we did in this area, and the results obtained.

In Chapter 6, we present the problem of QoS/TE routing, the challenges they present, the existing solutions by Wang and Crowcroft [106], and Guerin *et al.* [34], Vasilakos *et al.* [104], and Kodialam and Lakshman [42], and the inadequacies of these algorithms. Our proposed solution is then introduced.

In Chapter 7, we address the area of fault tolerant routing in MANETs. First, we provide a brief description of the MANETs, some popular routing algorithms currently deployed in the fields, how those algorithms fall short in adversarial MANET environments. We then describe, in detail, the current state-of-the-art algorithm proposed recently for fault tolerant routing in MANETs. Finally, we provide our solution, and establish its efficiency through a rigorous set of experiments.

We conclude the Thesis in Chapter 8, by providing a summary of the work that we had embarked on during the doctoral research, our contributions, and propose how these solutions can be further enhanced.

Chapter 2

Adaptive Learning Automata

2.1 Introduction

What is a *Learning Automaton*[1]? What is "Learning" all about? What are the different types of Learning Automata (LA) available? These are some of the fundamental issues that this Chapter attempts to describe, so that we can understand the potential of the mechanisms, and their capabilities in solving some of the fundamental challenging problems that occur in the area of computer and communication networks.

The linguistic meaning of *automaton* is a self-operating machine or a mechanism that responds to a sequence of instructions in a certain way, so as to achieve a certain goal. The automaton either responds to a pre-determined set of rules, or adapts to the environmental dynamics in which it operates. The latter types of automata are of interest to our research, and are termed as *adaptive automata.* The term *learning* in Psychology means the act of acquiring knowledge and modifying one's behavior based on the experience gained. Thus, in our case, the adaptive automaton we study in this Chapter, adapts to the responses from the Environment through a series of interactions with it. It then attempts to learn the best action from a set of possible actions that

[1]Singular: Automaton, Plural: Automata.

are offered to it by the random stationary or non-stationary Environment in which it operates. The Automaton, thus, acts as a decision maker to arrive at the best action.

The operations of the learning automata can be best described through the words of the pioneers Narendra and Thathachar [66]: "... a decision maker operates in the random environment and updates its strategy for choosing actions on the basis of the elicited response. The decision maker, in such a feedback configuration of decision maker (or automaton) and environment, is referred to as the learning automaton. The automaton has a finite set of actions, and corresponding to each action, the response of the environment can be either favorable or unfavorable with a certain probability" ([66], pp. 3).

LA, thus, find applications in optimization problems in which an optimal action needs to be determined from a set of actions. It should be noted that in this context, learning might be of best help only when there are high levels of *uncertainty* in the system in which the automaton operates. In systems with low levels of uncertainty, LA-based learning may not be a suitable tool of choice [66].

The first studies with LA models date back to the studies by mathematical psychologists like Bush and Mosteller [13], and Atkinson *et al.* [1]. In 1961, the Russian mathematician, Tsetlin [97] studied deterministic LA in detail. Varshavskii and Vorontsova [105] introduced the stochastic variable structure versions of the LA. Tsetlin's deterministic automata [97], and Varshavskii and Vorontsova's stochastic automata [105] were the major initial motivators of further studies in this area. Following them, several theoretical and experimental studies have been conducted by several researchers: K. Narendra, M. A. L. Thathachar, S. Lakshmivarahan, M. Obaidat, K. Najim, A.S. Poznyak, N. Baba, L. G. Mason, G. Papadimitriou, and my supervisor, B. J. Oommen, just to mention few. A comprehensive overview of research in the field of Learning Automata can be found in the classic text by Narendra and Thathachar [66], and in the recent special issue of the journal, *IEEE Transactions on Systems, Man, and Cyber-*

netics, Part B [71]. In this Chapter, we discuss some of the important results in this area.

Finally, it should be noted that none of the work described in this Chapter is original. Most of the discussions, terminologies, and all algorithms that are explained in this Chapter are taken from the corresponding existing pieces of literature. Thus, the notation and terminology can be considered to be "off the shelf", and fairly standard.

2.2 A Learning Automaton

In the field of Automata Theory, an automaton can be defined as a quintuple consisting of a set of states, a set of outputs or actions (of the automaton), an input (to the automaton), a function that maps the current state and input with the next state, and a function that maps a current state and input into the current output [66].

Definition 2.1: A Learning Automaton is defined by a quintuple $A, B, Q, F(.,.), G(.)$, where [2, 66]:

(i) $A = \{\alpha_1, \alpha_2, \ldots, \alpha_r\}$ is the set of outputs or actions, and $\alpha(t)$ is the action chosen by the automaton at any instant t.

(ii) B is the set of inputs to the automaton, $\{\beta_1, \beta_2, \ldots, \beta_r\}$. $\beta(t)$ is the input at any instant t, while the set B can be finite or infinite.

(iii) $Q = \{q_1(t), q_2(t), \ldots, q_s(t)\}$ is the set of finite states, where $q(t)$ denotes the state of the automaton at any instant t.

(iv) $F(.,.) : Q \times B \mapsto Q$ is a mapping in terms of the state and input at the instant t, such that, $q(t+1) = F[q(t), \beta(t)]$. It is called a *transition function*, i.e., a function that determines the state of the automaton at any subsequent time

instant $t+1$. This mapping can either be deterministic or stochastic, depending on the environment in which the automaton operates.

(v) $G(.)$: is a mapping $G : Q \mapsto A$, and is called the *output function.* Depending on the state at a particular instant, this function determines the output of the automaton at the same instant as: $\alpha(t) = G[q(t)]$. This mapping can, again, be considered to be either deterministic or stochastic, depending on the Environment in which the automaton operates [66]. Without loss of generality, G is deterministic.

If the sets Q, B and A are all finite, the automaton is said be *finite.* Some of the different types of automata are described in Section 2.4.

2.3 Environment

The Environment, E, typically, refers to the medium in which the automaton functions. The Environment possesses all the external factors that affect the actions of an automaton. Mathematically, an Environment can be abstracted by a triple $\{A, C, B\}$. A, C, and B are defined as follows [2, 66].

(i) $A = \{\alpha_1, \alpha_2, \ldots, \alpha_r\}$ is a set of actions

(ii) $B = \{\beta_1, \beta_2, \ldots, \beta_r\}$ is the output set of the environment, and

(iii) $C = \{c_1, c_2, \ldots, c_r\}$ is a set of penalty probabilities, where element $c_i \in C$ corresponds to an input action α_i.

The process of learning is based on a learning loop involving the two entities: the Random Environment (RE), and the LA, as described in Figure 2.1. In the process of learning, the LA continuously interacts with the Environment to process responses to

its various actions. Finally, through sufficient interactions, the LA attempts to learn the optimal action offered by the RE. The actual process of learning is represented as a set of interactions between the RE and the LA.

The RE offers the automaton with a set of possible actions $\{\alpha_1, \alpha_2, \ldots, \alpha_r\}$ to choose from. The automaton chooses one of those actions, say α_i, which serves as an input to the RE. Since the RE is aware of the underlying penalty probability distribution of the system, depending on the *penalty probability* c_i corresponding to α_i, it "prompts" the LA with a reward (typically denoted by the value "0"), or a *penalty* (typically denoted by the value "1"). The reward/penalty information (corresponding to the action) provided to the LA helps it to choose the subsequent action. By repeating the above process, through a series of Environment-Automaton interactions, the LA finally attempts to learn the *optimal* action from the Environment [74]. We now provide a

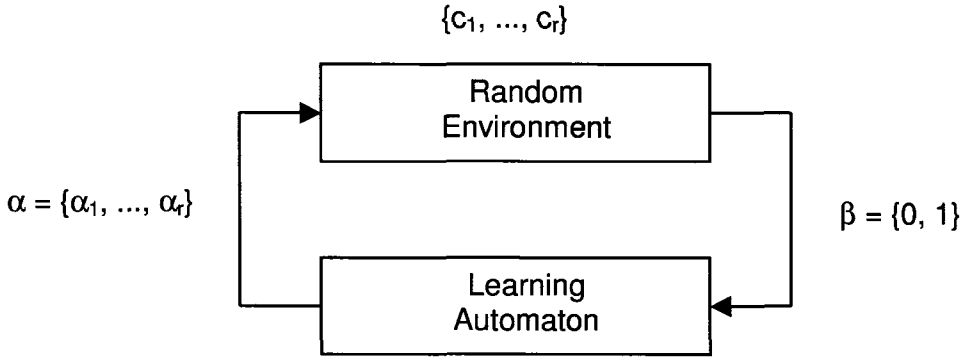


Figure 2.1: The Automaton - Environment Feedback Loop [66]

few important definitions used in the field of LA. Given an action probability vector

$P(t)$ at time "t", the *average penalty* is defined as [2, 66]:

$$\begin{aligned} M(t) &= E[\beta(t)|P(t)] = Pr[\beta(t) = 1|P(t)] \\ &= \sum_{i=1}^{r} Pr[\beta(t) = 1|\alpha(t) = \alpha_i]\ Pr[\alpha(t) = \alpha_i] \\ &= \sum_{i=1}^{r} c_i p_i(t) \end{aligned} \tag{2.1}$$

The average penalty for the "pure-chance" automaton is given by [2, 66]:

$$M_0 = \frac{1}{r}\sum_{i=1}^{r} c_i \tag{2.2}$$

As $t \mapsto \infty$, if the average penalty $M(t) < M_0$, at least asymptotically, the automaton is generally considered to be better than the pure-chance automaton. $E[M(t)]$ is given by [2, 66]:

$$E[M(t)] = E\{E[\beta(t)|P(t)]\} = E[\beta(t)] \tag{2.3}$$

Based on the comparison with the pure-chance automaton, any automaton that performs better than the pure-chance automaton is considered *expedient.* Mathematically, this definition can be expressed as follows:

Definition 2.2: A learning automaton is considered *expedient* [2, 66] if:

$$\lim_{t\mapsto\infty} E[M(t)] < M_0 \tag{2.4}$$

Definition 2.3: A learning automaton is said to be *absolutely expedient* [2, 66] if:

$$E[M(t+1)|P(t)] < M(t), \tag{2.5}$$

where $M(t)$ is the expected penalty probability at instant "t", and $P(t)$ is the action probability vector at "t".

Definition 2.4: A learning automata is considered *optimal* [2, 66] if:

$$\lim_{t\mapsto\infty} E[M(t)] = c_l \tag{2.6}$$

where $c_l = \min_i\{c_i\}$.

Definition 2.5: A learning automata is considered ϵ-*optimal* if [2, 66]:

$$\lim_{n\mapsto\infty} E[M(t)] < c_l + \epsilon \tag{2.7}$$

where $\epsilon > 0$, and can be arbitrarily small, by the suitable choice of some parameter of the LA. It should be noted that at present there are no optimal learning automata. Marginally sub-optimal performance, also termed above as ϵ-optimal performance, is what LA researchers attempt to attain.

2.4 Classification of Learning Automata

2.4.1 Deterministic Learning Automata

An automaton is termed as a *deterministic automaton*, if both the transition function $F(.,.)$ and the output function $G(.)$ defined in Section 2.2 are deterministic mappings. In other words, in a deterministic automaton, the subsequent state and action can be uniquely specified, provided the initial state and input are given [66].

2.4.2 Stochastic Learning Automata

If, however, either the transition function $F(.,.)$, or the output function $G(.)$ defined in Section 2.2 are stochastic – i.e., those functions do not provide a unique value every time, and their output is a random variable – the automaton is termed to be a *stochastic automaton.* In such an automaton, if the current state and input are specified, the subsequent states and actions cannot be specified uniquely. In such a case, $F(.,.)$ only provides the probabilities of reaching the various states from a given state. Let $F^{\beta_1}, F^{\beta_2}, \ldots, F^{\beta_m}$ denote the conditional probability matrices, where each of these conditional matrices $F(\beta)$ for $\beta \in \mathbf{B}$ is a $s \times s$ matrix, whose elements are given by [66]:

$$\begin{aligned} f_{ij}^{\beta} = Pr\{q(t+1) = q_j | q(t) = q_i, \beta(t) = \beta\}, \; i &= 1, 2, \ldots, s; \\ j &= 1, 2, \ldots, s; \\ \beta &\in \{\beta_1, \beta_2, \ldots, \beta_m\}. \end{aligned} \tag{2.8}$$

In Equation 2.8, each element f_{ij}^{β} of the matrix F^{β} represents the probability of the automaton moving from state q_i to the state q_j on receiving an input signal β from the RE. F^{β} is a Markov matrix. By virtue of the law of probabilities, we have $\forall i$:

$$\sum_{j=1}^{s} f_{ij}^{\beta} = 1, \text{ where } \beta \in \mathbf{B} \tag{2.9}$$

Similarly, in a stochastic automaton, if $G(.)$ is stochastic, we have [66]:

$$\begin{aligned} g_{ij} = Pr\{\alpha(t) = \alpha_j | q(t) = q_i\}, \; i &= 1, 2, \ldots, s; \\ j &= 1, 2, \ldots, r; \end{aligned} \tag{2.10}$$

where g_{ij} represents the elements of the conditional probability matrix of dimension $s \times r$. Intuitively, g_{ij} denotes the probability that when the automaton is in state q_i, it chooses the action α_j. As in Equation 2.9, we have [66]:

$$\sum_{j=1}^{r} g_{ij} = 1, \text{ for each row } i = 1, 2, \ldots, s \tag{2.11}$$

In a stochastic LA, if the conditional probabilities f_{ij}^{β} and g_{ij} are constant, i.e., they do not vary with the time step "t" and the input sequence, the automaton is termed to be a *Fixed Structure Stochastic Automaton* (FSSA). The popular examples of these types of automata were proposed by Tsetlin [97], Krylov [45], and Krinsky [44]. Since we do not use the automata, we will not describe them here. Their details can be found in [66].

2.4.3 Variable Structure Learning Automata

Unlike the FSSA, *Variable Structure Stochastic Automata* (VSSA) are the ones in which the state transition probabilities are not fixed. In such automata, the state transitions or the action probabilities themselves are updated at every time instant using a suitable scheme. The transition probabilities f_{ij}^{β} and the output function g_{ij} vary with time, and the action probabilities are updated on the basis of the input. These automata are discussed here in the context of linear schemes. But the concepts discussed below can be extended to non-linear updating schemes as well. The types of automata that update transition probabilities with time were introduced in 1963 by Varshavskii and Vorontsova [105]. A VSSA depends on a random number generator for its implementation. The action chosen is dependent on the action probability distribution vector, which is, in turn, updated based the reward/penalty input that the automaton receives from the RE.

Definition 2.6 The VSSA is a quintuple $q, A, B, \mathbf{T}, G$ [66], where q represents the different states of the automaton, A is the set of actions, B is the set of responses from the environment to the automaton, G is the output function, and T is the action probability updating scheme $T : [0,1]^r \times A \times B \mapsto [0,1]^r$, such that

$$P(t+1) = T(P(t), \alpha(t), \beta(t)), \tag{2.12}$$

where P is the action probability vector, $P(t) = [p_1(t), p_2(t), \ldots, p_r(t)]^T$, in which each element of the vector.

$$p_i(t) = Pr[\alpha(t) = \alpha_i],\ i = 1, \ldots, r, \text{ such that } \sum_{i=1}^{r} p_i(t) = 1 \ \ \forall t \tag{2.13}$$

Normally, VSSA involve the updating of both the state, and action probabilities. For the sake of simplicity, in practice, it is assumed that in such automata, each state corresponds to a distinct action, in which case the action transition mapping G becomes the identity mapping, and the number of states, s, is equal to the number of actions, r $(s = r < \infty)$.

VSSA can be analyzed using a discrete-time Markov Process, defined on a suitable set of states [66]. If a probability updating scheme T is time invariant, $\{P(t)\}_{t \geq 0}$ is a discrete-homogenous Markov process, and the probability at the current time instant $P(t)$, (along with $\alpha(t)$, and $\beta(t)$) completely determines the probability at the next time instant $P(t+1)$. Hence, different updating schemes T identify the different types of learning algorithms as follows [2]:

- *Absorbing algorithms* are the ones in which that the updating scheme T is chosen in such a manner that the Markov process has absorbing states.

- *Non-absorbing algorithms* are the ones in which the Markov process has no ab-

sorbing states.

- *Linear algorithms* are the ones in which $P(t+1)$ is a linear function of $P(t)$.
- *Nonlinear algorithms* are the ones in which $P(t+1)$ is a non-linear function of $P(t)$.

In a VSSA, if a chosen action α_i is rewarded, the probability for the current action is increased, and the probabilities for all the other actions are decreased. On the other hand, if the chosen action α_i is penalized, the probability of the current action is decreased, whereas the probabilities for the rest of the actions could, typically, be increased. This leads to the following different types of learning schemes for VSSA [2, 66]:

- **Reward-Penalty (RP):** In both the cases when the automaton is rewarded as well as penalized, the action probabilities are updated.
- **Inaction-Penalty (IP):** When the automaton is penalized, the action probability vector is updated, whereas when the automaton is rewarded, the action probabilities are neither increased nor decreased.
- **Reward-Inaction (RI):** When the automaton is rewarded, the action probability vector is updated, whereas when the automaton is penalized, the action probability vector is unchanged.

A learning automaton is considered as a *continuous automaton* if the probability updating scheme T is continuous, i.e., the probability of choosing an action can be any real number in the closed interval $[0, 1]$.

In a VSSA, if there are r actions operating in a stationary environment with $\beta = \{0, 1\}$, a general action probability updating scheme for a continuous automaton is

described below [66]. We assume that the action α_i is chosen, and thus, $\alpha(t) = \alpha_i$. The action probabilities can be represented as follows:

$$\begin{aligned} \text{For } \beta(t) &= 0,\ \forall j \neq i\ , \quad p_j(t+1) = p_j(t) - g_j\{P(t)\} \\ \text{For } \beta(t) &= 1,\ \forall j \neq i\ , \quad p_j(t+1) = p_j(t) + h_j\{P(t)\} \end{aligned} \tag{2.14}$$

Since $P(t)$ is a probability vector, $\sum_{j=1}^{r} p_j(t) = 1$. Therefore,

$$\begin{aligned} \text{When } \beta(t) &= 0, \quad p_i(t+1) = p_i(t) + \sum_{j=1, j\neq i}^{r} g_j\{P(t)\} \\ \text{When } \beta(t) &= 1, \quad p_i(t+1) = p_i(t) - \sum_{j=1, j\neq i}^{r} h_j\{P(t))\} \end{aligned} \tag{2.15}$$

The functions h_j and g_j satisfy the following characteristics [66]:

1. They are *continuous* in the interval $[0, 1]$,
2. They are *nonnegative*, and
3. The following are satisfied:

$$\begin{aligned} \forall i = 1, 2, \ldots, r, \quad \forall p \in (0,1),\ 0 < g_j(P) < p_j, \\ \text{and} \qquad 0 < \sum_{j=1, j\neq i}^{r} [p_j + h_j(P)] < 1 \end{aligned} \tag{2.16}$$

For *continuous linear* VSSA, the following four learning schemes are possible. They are explained for the 2-action case; their extension to the r-action case, where $r > 2$, are straightforward, and can be found in [66].

- Linear Reward-Inaction Scheme (L_{RI})
- Linear Inaction-Penalty Scheme (L_{IP})

Table 2.1: Properties of the Learning Schemes.

Learning Scheme	Learning Parameters	Usefulness (good/bad)	Optimality	Ergodic/Absorbing (when useful)
L_{RI}	$a, b = 0$	Good	ϵ-optimal, as $a \mapsto 0$	Absorbing (stationary environments)
L_{IP}	$a = 0, b$	Very Bad	Not even expedient	Ergodic (non-stationary environments)
L_{RP}	$a = b$	Bad	Never ϵ-optimal	Ergodic
$L_{R-\epsilon P}$	$a, b << a$	Good	ϵ-optimal as $a \mapsto 0$	Ergodic

- Linear Reward-Penalty Scheme (L_{RP})
- Linear Reward-ϵ-Penalty Scheme ($L_{R-\epsilon P}$)

For a 2-action LA [66], let

$$\begin{aligned} g_i\{P(t)\} &= a\, p_j(t) \\ \text{and } h_j\{P(t)\} &= b\,\{1 - p_j(t)\} \end{aligned} \tag{2.17}$$

In Equation 2.17, a and b are called the reward and penalty parameters, and they obey the following inequalities: $0 < a < 1$, $0 \leq b < 1$. Equation 2.17 will be used further to develop the action probability updating equations. The above-mentioned linear schemes are quite popular in LA because of their analytical tractability [66]. They exhibit significantly different characteristics as can be seen in Table 2.1.

2.4.3.1 Continuous Linear Reward-Inaction Scheme (L_{RI})

The L_{RI} scheme was first introduced by Norman [65], and then by Shapiro and Narendra [89]. It is based on the principle that whenever the automaton receives a favorable

response (i.e., reward) from the environment, the action probabilities are updated, whereas if the automaton receives an unfavorable response (i.e., penalty) from the environment, the action probabilities are unaltered.

The probability updating equations for this scheme can be simplified to be as below [66]:

$$\begin{aligned} p_1(t+1) &= p_1(t) + a(1-p_1(t)) \quad \text{if} \quad \alpha(t) = \alpha_1, \text{and } \beta(t) = 0 \\ p_1(t+1) &= p_1(t) \quad \text{if} \quad \alpha(t) = \alpha_1, \text{and } \beta(t) = 1 \\ p_1(t+1) &= (1-a)p_1(t) \quad \text{if} \quad \alpha(t) = \alpha_2, \text{and } \beta(t) = 0 \\ p_1(t+1) &= p_1(t) \quad \text{if} \quad \alpha(t) = \alpha_2, \text{and } \beta(t) = 1 \end{aligned} \tag{2.18}$$

We see that if action α_i is chosen, and a reward is received, the probability $p_i(t)$ is increased, and the other probability $p_j(t)$ (i.e., $j \neq i$) is decreased. If either α_1 or α_2 is chosen, and a penalty is received, $P(t)$ is unaltered.

Equation 2.18 shows that the L_{RI} scheme has the vectors $[1,0]^T$ and $[0,1]^T$ as two absorbing states. Indeed, with probability 1, it gets "stuck" (i.e., absorbed) into one of these absorbing states. Therefore, the convergence of the L_{RI} scheme is dependent on the nature of the initial conditions and probabilities. The scheme is not suitable for non-stationary environments. On the other hand, for stationary random environments, the L_{RI} scheme is both absolutely expedient, and ϵ-optimal [66].

The L_{IP} and L_{RI} schemes can be devised similarly, and are omitted from further discussion here, as we will not be using them in our discussions in the future Chapters. They can be found in [66].

2.4.4 Discretized Learning Automata

The VSSA algorithms presented in Section 2.4.3 are *continuous*, i.e., the action probabilities can take any real value in the interval $[0, 1]$. In LA, the choice of an action is determined by a Random Number Generator (RNG). In order to increase the speed of convergence of these automata, Thathachar and Oommen [94] introduced the *discretized* algorithms for VSSA, in which they suggested the discretization of the probability space. The different properties (absorbing, ergodic, and estimator) of these learning automata, and the updating schemes of action probabilities for these discretized automata (like their continuous counterpart) were later studied in detail by Oommen *et al.* [49, 68, 69, 70].

Discretized automata can be perceived to be somewhat like a hybrid combination of FSSA and VSSA. Discretization is conceptualized by restricting the probability of choosing the actions to only a fixed number of values in the closed interval $[0, 1]$. Thus, the updating of the action probabilities is achieved in steps rather than in a continuous manner as in the case of continuous VSSA. Evidently, like FSSA they possess finite sets. On the other hand, because they have action probability vectors which are random vectors, they behave like VSSA.

Discretized LA can be of two types: (i) *Linear* – in which case the action probability values are uniformly spaced in the closed interval $[0, 1]$, and (ii) *Non-Linear* – in which case the probability values are unequally spaced in the interval [0,1] [49, 94].

Perhaps the biggest motivation behind discretization is overcoming the persistent limitation of continuous learning automata, i.e., the slow rate of convergence. This is achieved by narrowing down the underlying assumptions of the automata on the environment. Originally, the assumption was that the RNGs could generate real values with arbitrary precision. In the case of discretized LA, if an action probability is reasonably close to unity, the probability of choosing that action increases to unity

(when the conditions are appropriate) directly, rather than asymptotically [49, 94].

The second important advantage of discretization is that it is more practical in the sense that the RNGs used by continuous VSSA can only *theoretically* be assumed to be *any* value in the interval $[0, 1]$. But almost all machine implementations of RNGs use pseudo-RNGs. In other words, the set of possible random values is not infinite in $[0, 1]$, but finite [49].

Last, but not the least, discretization is also important in terms of implementation and representation. Discretized implementations of automata use integers for tracking the number of multiples of $\frac{1}{N}$ of the action probabilities. This not only increases the rate of convergence of the algorithm, but also reduces the time, in terms of the clock cycles it takes for the processor to do each iteration of the task, and the memory needed [49]. Discretized algorithms have been proven to be both more time and space efficient than the continuous algorithms.

Similar to the continuous LA paradigm, the discretized versions, the DL_{RI}, DL_{IP}, and DL_{RP} automata have also been reported. Their design, analysis, and properties are given in [68, 69, 70], and are omitted here, as their use in this Thesis will be marginal.

2.5 Estimator Algorithms

2.5.1 Rationale and Motivation

As we have seen so far, the rate of convergence of the learning algorithms is one of the most important considerations, which was the primary reason for designing the family of "discretized" algorithms. With the same goal Thathachar and Sastry designed a new-class of algorithms, called the *Estimator Algorithms* [85, 98, 99, 100], which have faster rate of convergence than all the previous families. These algorithms,

like the previous ones, maintain, and update an action probability vector. However, unlike the previous ones, these algorithms also keep running estimates for each action that is rewarded, using a *reward-estimate vector*, and then use those estimates in the probability updating equations [49, 98, 100]. The reward estimates vector is, typically, denoted in the literature by $\hat{D}(t) = [\hat{d}_1(t), \ldots, \hat{d}_r(t)]^T$. The corresponding state vector [2, 100] is denoted by $Q(t) =< P(t), \hat{D}(t) >$

In a random environment, these algorithms help in choosing an action by increasing the confidence in the reward capabilities of the different actions. For example, these algorithms process each action a number of times, and then (in one version) could update the probability of the action with the highest reward estimate [2]. This leads to a scheme with better accuracy in choosing the correct action. The previous non-estimator VSSA algorithms update the probability vector directly on the basis of the response of the environment to the automaton, where, depending on the type of vector updating scheme being used, the probability of choosing a rewarded action in the subsequent time instant is increased, and the probability of choosing the other actions is decreased. However, estimator algorithms update the probability vector based on both the estimate vector, and the current feedback provided by the environment to the automaton. The environment influences the probability vector both directly and indirectly, the latter by being involved in the estimation of the reward estimates of the different actions. This may, thus, lead to increases in action probabilities different from the currently rewarded action [49, 98, 100].

Even though there is an added computational cost involved in maintaining the reward estimates, these estimator algorithms have an order of magnitude superior performance than the non-estimator algorithms previously introduced [98, 100]. Lanctôt and Oommen [49] further introduced the discretized versions of these estimator algorithms, which were proven to have an even faster rate of convergence.

2.5.2 Continuous Estimator Algorithms

Thathachar and Sastry introduced the class of continuous estimator algorithms [85, 98, 99, 100] in which the probability updating scheme T is continuous, i.e., the probability of choosing an action can be any real number in the closed interval $[0, 1]$. As mentioned subsequently, the discretized versions of these algorithms were introduced by Ommen and his co-authors, Lanctôt and Agache [5, 49]. These algorithms are only briefly explained in Section 2.5.3.

2.5.2.1 Pursuit Algorithm

The family of Pursuit algorithms is a class of estimator algorithms that *pursues* an action that the automaton "currently" perceives to be the optimal one. The first pursuit algorithm, called the CP_{RP} algorithm, and introduced by Thathachar and Sastry [95, 98], pursues the optimal action by changing the probability of the current optimal action on receiving a reward, or penalty by the environment. In this case, the *currently perceived* "best action" is rewarded, and *its* action probability value increased with a value directly proportional to its distance to unity, namely $1-p_m(t)$, whereas the "less optimal actions" are penalized, and their probabilities decreased proportionally [2].

To start with, based on the probability distribution $P(t)$, the algorithm chooses an action $\alpha(t)$. Whether the response was a reward or a penalty, it increases that component of $P(t)$ which has the maximal current reward estimate, and it decreases the probability corresponding to the rest of the actions. Finally, the algorithm updates the running estimates of the reward probability of the actions chosen, this being the principal idea behind keeping, and using the running estimates [95]. The estimate vector $\hat{D}(t)$ can be computed using the following formula which yields the maximum

likelihood estimate:

$$\hat{d}_i(t) = \frac{W_i(t)}{Z_i(t)}\ , \quad \forall i = 1, 2, \ldots, r, \tag{2.19}$$

where $W_i(t)$ is the number of times the action α_i has been rewarded until the current time t, and $Z_i(t)$ is the number of times α_i has been chosen until the current time t. Based on the above concepts, the CP_{RP} algorithm (adapted with little changes from the version of the algorithm presented in [2]) is formally given below without any further explanation.

Algorithm: CP_{RP}

Parameters

- λ: the speed of learning parameter , where $0 < \lambda < 1$
- m: index of the maximal component of $\hat{D}(t)$, $\hat{d}_m(t) = \max_{i=1,2,\ldots,r}\{\hat{d}_i(t)\}$
- $W_i(t)$: the number of times the i^{th} action has been rewarded up to the time t, with $1 \leq i \leq r$
- $Z_i(t)$: the number of times the i^{th} action has been chosen up to the time t, with $1 \leq i \leq r$

... (Continued to next page)

Algorithm: CP_{RP} (Continued from previous page)

BEGIN

Initialization

$p_i(t) = 1/r$, for $1 \leq i \leq r$

Initialize $\hat{D}(t)$ by picking each action a small number of times

Repeat

Step 1

At time t pick $\alpha(t)$ according to probability distribution $P(t)$

Let $\alpha(t) = \alpha_i$

Step 2

If α_m is the current optimal action, update $P(t)$ according to the following equations

$$\begin{aligned} p_m(t+1) &= p_m(t) + \lambda(1 - p_m(t)) \\ p_j(t+1) &= p_j(t) - \lambda p_j(t) \end{aligned} \tag{2.20}$$

Step 3

Update $\hat{D}(t)$ according to the following equations

if $\alpha(t) = \alpha_j$ **then**

$$\begin{aligned} W_i(t+1) &= W_i(t) + (1 - \beta(t)) \\ Z_i(t+1) &= Z_i(t) + 1 \\ \hat{d}_i(t+1) &= \frac{W_i(t+1)}{Z_i(t+1)} \end{aligned} \tag{2.21}$$

end

$\forall j \neq i$

$$\begin{aligned} W_j(t+1) &= W_j(t) \\ Z_j(t+1) &= Z_j(t) \\ \hat{d}_j(t+1) &= \hat{d}_j(t) \end{aligned} \tag{2.22}$$

End-Repeat

END

The reward-inaction version of this pursuit algorithm is also similar in design, and is described in [5, 49]. As we have seen from the above pseudo-code of the CP_{RP} algorithm, it is similar in principle to the L_{RP} algorithm, because both the CP_{RP}, and the L_{RP} algorithms increase/decrease the action probabilities of the vector independent of whether the environment responds to the automaton with a reward or a penalty. The major difference lies in the way the reward estimates are maintained, used, and are updated on both reward/penalty. It should be emphasized that whereas the non-pursuit algorithm moves the probability vector in the direction of the most recently rewarded action, the pursuit algorithm moves the probability vector in the direction of the action with the highest reward estimate. Thathachar and Sastry [95] have theoretically proven their ϵ-optimality, and experimentally proven that these pursuit algorithms are more accurate, and several orders of magnitude faster than the non-pursuit algorithms.

We have used various versions of the Pursuit algorithms in our research.

2.5.2.2 TSE Algorithm

A more advanced estimator algorithm, which we refer to as the TSE algorithm to maintain consistency with the existing literature [2, 5, 49], was designed by Thathachar and Sastry [99, 100].

Like the other estimator algorithms, the TSE algorithm maintains the running reward estimates vector $\hat{D}(t)$, and uses it to calculate the action probability vector $P(t)$. When an action $\alpha_i(t)$ is rewarded, according to the TSE algorithm, the probability components with a reward estimate greater than $\hat{d}_i(t)$ are treated differently from those components with an a value lower than $\hat{d}_i(t)$. The algorithm does so by increasing the probabilities for all the actions that have a higher estimate than the estimate of the chosen action, and decreasing the probabilities of all the actions with a lower estimate. This is done with the help of an indicator function $S_{ij}(t)$ which assumes the value 1 if $\hat{d}_i(t) > \hat{d}_j(t)$, and the value 0 if $\hat{d}_i(t) \leq \hat{d}_j(t)$. Thus, the TSE

algorithm uses both the probability vector $P(t)$ and the reward estimates vector $\hat{D}(t)$ to update the action probabilities. The algorithm is described below (adapted with little changes from the version of the algorithm presented in [2]). On careful inspection of the algorithm, it can be observed that $P(t+1)$ depends indirectly on the response of the environment to the automaton. The feedback from the environment changes the values of the components of $\hat{D}(t)$, which, in turn, affects the values of the functions $f(.)$ and $S_{ij}(t)$ [2, 49, 99, 100].

Algorithm: TSE

Parameters

- λ: the speed of learning parameter , where $0 < \lambda < 1$
- m: index of the maximal component of $\hat{D}(t)$,
- $\hat{d}_m(t) = \max_{i=1,2,\ldots,r}\{\hat{d}_i(t)\}$
- $W_i(t)$: the number of times the i^{th} action has been rewarded up to the time t, with $1 \leq i \leq r$
- $Z_i(t)$: the number of times the i^{th} action has been chosen up to the time t, with $1 \leq i \leq r$
- $S_{ij}(t)$: an indicator function

$$S_{ij}(t) = \begin{cases} 0 & \text{if } \hat{d}_i(t) \leq \hat{d}_j(t) \text{ and} \\ 1 & \hat{d}_i(t) > \hat{d}_j(t) \end{cases}$$

- f : $f : [-1, 1] \mapsto [-1, 1]$ a monotonic, increasing function satisfying $f(0) = 0$, and $f(1) = 1$

BEGIN

Initialization

$p_i(t) = 1/r$, for $1 \leq i \leq r$

Initialize $\hat{D}(t)$ by picking each action a small number of times

Repeat

Step 1

At time t pick $\alpha(t)$ according to probability distribution $P(t)$; Let $\alpha(t) = \alpha_i$;

Step 2

Update $P(t)$ according to the following equations:

$$\begin{aligned} p_j(t+1) &= p_j(t) - \lambda[f(\hat{d}_i(t) - \hat{d}_j(t))] \times \mathcal{S}, j \neq i \\ p_i(t+1) &= p_i(t) + \lambda \sum_{j \neq i} [f(\hat{d}_i(t) - \hat{d}_j(t))] \times \mathcal{S} \end{aligned} \tag{2.23}$$

where $\mathcal{S} = \left[S_{ij}(t)p_j(t) + S_{ji}(t)]\frac{p_i(t)}{r-1}(1 - p_j(t))\right]$

Step 3

Update $\hat{D}(t)$ according as in the algorithm CP_{RP}

End-Repeat

END

Analyzing the algorithm carefully, we obtain three cases. If the i^{th} action is rewarded, the probability values of the actions with reward estimates higher than the reward estimate of the currently selected action are updated using the following equation [99]:

$$p_i(t+1) = p_j(t) - \lambda \left[f(\hat{d}_i(t) - \hat{d}_j(t)) \frac{\{p_i(t) - p_j(t)p_i(t)\}}{r-1} \right] \tag{2.24}$$

When $\hat{d}_i(t) < \hat{d}_j(t)$, the function $f(\hat{d}_i(t) - \hat{d}_j(t))$ is monotonic and increasing, and $f(\hat{d}_i(t) - \hat{d}_j(t))$, is thus, negative. This leads to a higher value of $p_j(t+1)$ than that of $p_j(t)$, which indicates that the probability of choosing actions, that have estimates greater than that of the estimates of the currently chosen action, will increase.

For all the actions with reward estimates smaller than the estimate of the currently selected action, the probabilities are updated based on the following equation:

$$p_j(t+1) = p_j(t) - \lambda f\left(\hat{d}_i(t) - \hat{d}_j(t)\right) p_j(t) \tag{2.25}$$

The sign of the function $f\left(\hat{d}_i(t) - \hat{d}_j(t)\right)$ is negative, which indicates that the probability of choosing actions, that have estimates less than that of the estimate of the currently chosen action, will increase.

Thathachar and Sastry have proven that the TSE algorithm is ϵ-optimal [99]. They have also experimentally shown that the TSE algorithm often converges several orders of magnitude faster than the L_{RI} scheme [99].

2.5.2.3 Generalized Pursuit Algorithm

Agache and Oommen [5] proposed a generalized version of the Pursuit algorithm (CP_{RP}) proposed by Thathachar and Sastry [95, 98]. Their algorithm, called the *Generalized Pursuit Algorithm* (GPA), generalizes Thathachar and Sastry's Pursuit algorithm by pursuing all those actions that possess higher reward estimates than the

chosen action. In this way the probability of choosing a wrong action is minimized. Agache and Oommen experimentally compared their pursuit algorithm with the existing algorithm, and found that their algorithm is the best in terms of the rate of convergence [5].

In the CP_{RP} algorithm, the probability of the best estimated action is maximized by first decreasing the probability of all the actions in the following manner [5]:

$$p_j(t+1) = (1-\lambda)p_j(t), \quad j = 1, 2, \ldots, r \tag{2.26}$$

The sum of the action probabilities is made unity, by the help of the probability mass Δ, which is given by [5]:

$$\Delta = 1 - \sum_{j=1}^{r} p_j(t+1) = 1 - \sum_{j=1}^{r}(1-\lambda)p_j(t) = 1 - \sum_{j=1}^{r} p_j(t) + \lambda \sum_{j=1}^{r} p_j(t) = \lambda \tag{2.27}$$

Thereafter, the probability mass Δ is added to the probability of the best-estimated action. The GPA algorithm, thus, equi-distributes the probability mass Δ to the action estimated to be superior to the chosen action. This gives us [5]:

$$p_m(t+1) = (1-\lambda)p_m(t) + \Delta = (1-\lambda)p_m(t) + \lambda \tag{2.28}$$

where $\hat{d}_m = \max_{j=1,2,\ldots,r}(\hat{d}_j(t))$. Thus, the updating scheme is given by [5]:

$$\begin{aligned} p_j(t+1) &= (1-\lambda)p_j(t) + \frac{\lambda}{K(t)}, \quad \text{if} \quad \hat{d}_j(t) > \hat{d}_i(t) \\ p_j(t+1) &= (1-\lambda)p_j(t), \quad \text{if} \quad \hat{d}_j(t) \leq \hat{d}_i(t) \\ p_i(t+1) &= 1 - \sum_{j \neq m} p_j(t) \end{aligned} \tag{2.29}$$

where $K(t)$ denotes the number of actions that have estimates greater than the estimate

of the probability reward of the action chosen at the given time instant.

The GPA algorithm can be formally presented by modifying the CP_{RP} algorithm with Equation 2.29. The formal algorithm is omitted, but can be found in [5].

2.5.3 Discrete Estimator Algorithms

As we have seen so far, discretized LA are superior to their continuous counterparts, and the estimator algorithms are superior to the non-estimator algorithms in terms of the rate of convergence of the learning algorithms. Utilizing the previously proven capabilities of discretization in improving the speed of convergence of the learning algorithms, Lanctôt and Oommen [49] enhanced the Pursuit and the TSE algorithms. This led to the designing of classes of learning algorithms, referred to in the literature as the *Discrete Estimator Algorithms* (DEA) [49]. To this end, as done in the previous discrete algorithms, the components of the action probability vector are allowed to assume a finite set of discrete values in the closed interval $[0, 1]$, which is, in turn, divided into the number of sub-intervals that is proportional to the resolution parameter N. Along with this, a reward estimate vector is maintained to keep an estimate of the reward probability of each action [49].

Lanctôt and Oommen showed that for each member algorithm belonging to the class of DEAs to be ϵ-optimal, it must possess a set of properties known as the *Property of Moderation* and the *Monotone Property*. Together these properties help prove the ϵ-optimality of the DEA algorithms [49].

Property 2.1: A DEA with r actions and a resolution parameter N is said to posses the *property of moderation*, if the maximum magnitude by which an action probability can decrease per iteration is bounded by $1/rN$ [49].

Property 2.2: Suppose there exists an index m and a time instant $t_0 < \infty$, such that $\hat{d}_m(t) > \hat{d}_j(t)$, $\forall j$ s.t. $j \neq m$ and $\forall t$ s.t. $t \geq t_0$, where $\hat{d}_m(t)$ is the maximal component of $\hat{D}(t)$. A DEA is said to posses the *Monotone Property*, if there exists an integer N_0 such that for all resolution parameters $N > N_0$, $p_m(t) \mapsto 1$ with probability one as $t \mapsto \infty$, where $p_m(t)$ is the maximal component of $P(t)$ [49].

The discretized versions of the Pursuit Algorithm, and the TSE Algorithm possessing the moderation, and the monotone properties are presented below.

2.5.3.1 Discrete Pursuit Algorithm

The Discrete Pursuit Algorithm, referred to as the DPA in the literature [49], is similar to a great extent to its continuous pursuit counterpart, i.e., the CP_{RP} algorithm, except that the updates to the action probabilities for the DPA algorithm are made in discrete steps. On the other hand, the continuous Pursuit algorithm uses a continuous function to update the action probabilities. Therefore, the equations in the CP_{RP} algorithm that involve multiplication by the learning parameter λ are substituted by the addition or subtraction by quantities proportional to the smallest step size [49].

As in the CP_{RP} algorithm, the DPA algorithm operates in three steps as described in the outline of the algorithm provided below. If $\Delta = \frac{1}{rN}$ (where N denotes the resolution, and r the number of actions) denotes the smallest step size, the integral multiples of Δ denote the step sizes in which the action probabilities are updated. Like the continuous reward-inaction algorithm, when the chosen action $\alpha(t) = \alpha_i$, is penalized, the action probabilities remain unchanged. However, when the chosen action $\alpha(t) = \alpha_i$ is rewarded, and the algorithm has not converged, the algorithm decreases, by the integral multiples of Δ, the action probabilities which do not correspond to the highest reward estimate.

The DPA algorithm is formally described below (adapted from [49]).

Algorithm: DPA

Parameters

m: index of the maximal component of $\hat{D}(t)$, $\hat{d}_m(t) = \max_{i=1,2,\ldots,r}\{\hat{d}_i(t)\}$
$W_i(t)$: the number of times the i^{th} action has been rewarded up to the time t, with $1 \leq i \leq r$
$Z_i(t)$: the number of times the i^{th} action has been chosen up to the time t, with $1 \leq i \leq r$
N: resolution parameter
Δ: $\Delta = \frac{1}{rN}$ is the smallest step size

BEGIN

Initialization

$p_i(0) = 1/r$, for $1 \leq i \leq r$
Initialize $\hat{D}(t)$ by picking each action a small number of times

Repeat

Step 1

At time t pick $\alpha(t)$ according to probability distribution $P(t)$.

... (Continued to next page)

Algorithm: DPA (Continued from previous page)

Step 2
Let $\alpha(t) = \alpha_i$. Update $P(t)$ according to the following equations
if $\beta(t) = 0$ *and* $p_m(t) \neq 1$ **then**

$$\begin{aligned} p_j(t+1) &= \max_{\forall j \neq m}\{p_j(t) - \Delta, 0\} \\ p_m(t+1) &= 1 - \sum_{\forall j \neq m} p_j(t+1) \end{aligned} \tag{2.30}$$

end
else

$$p_j(t+1) = p_j(t) \text{ such that } 1 \leq j \leq r$$

end

Step 3
Update $\hat{D}(t)$ in the same way as done in CP_{RP}

End-Repeat
END

Lanctôt and Oommen have shown that the DPA algorithm possesses the properties of moderation and monotonicity, and that is thus ϵ-optimal [49]. They have also experimentally proved that in different ranges of environments from simple to complex, the DPA algorithm is at least 60% faster than the CP_{RP} algorithm [49].

2.5.3.2 Discrete TSE Algorithm

Lanctôt and Oommen also discretized the TSE algorithm, and have referred to it as the *Discrete TSE algorithm* (DTSE) [49]. Since the algorithm is based on the continuous version of the TSE algorithm, it obviously has the same level of intricacies, if not more. Lanctôt and Oommen theoretically proved that like the DPA estimator algorithm, this algorithm also possesses the moderation and the monotone properties,

while maintaining many of the qualities of the continuous TSE algorithm. They also provided the proof of convergence of this algorithm.

There are two notable parameters in the DTSE algorithm [49]:

1. $\Delta = \frac{1}{rN\theta}$, where N is the resolution parameter as before, and
2. θ is an integer representing the largest value any of the action probabilities can change by in a single iteration.

A formal description of the DTSE algorithm is omitted here, because we will not be explicitly requiring it in the subsequent Chapters in the Thesis. It can be found in [49].

2.5.3.3 Discretized Generalized Pursuit Algorithm

Agache and Oommen [5] provided a discretized version of their GPA algorithm presented earlier. Their algorithm, called the *Discretized Generalized Pursuit Algorithm* (DGPA) also essentially generalizes Thathachar and Sastry's Pursuit algorithm [95, 98] but unlike the TSE, it pursues all those actions that possess higher reward estimates than the chosen action.

In essence, in any single iteration, the algorithm computes the number of actions that have higher reward estimates than the current chosen action, denoted by $K(t)$, whence the probability of all the actions that have estimates higher than the chosen action is increased by an amount $\Delta/K(t)$, and the probabilities for all the other actions are decreased by an amount $\Delta/(r-K(t))$, where $\Delta = 1/rN$ denotes the resolution step, and N the resolution parameter. The DGPA algorithm has been proven to possess the moderation, monotone, and the ϵ-optimality properties [5]. The detailed steps of the DGPA algorithm are omitted here.

All the continuous and the discretized versions of the estimator algorithms presented above were experimentally evaluated by their previous authors (e.g., [49], [5]). Lanctôt and Oommen [49] compared the rates of convergence between the discretized

Table 2.2: The number of iterations until convergence in 2-action environments for the TSE algorithms (Adapted from [49])

Probability of Reward		**Mean Iterations**	
Action 1	**Action 2**	**Continuous**	**Discrete**
0.800	0.200	28.8	24.0
0.800	0.400	37.0	29.0
0.800	0.600	115.0	76.0
0.800	0.700	400.0	380.0
0.800	0.750	2200.0	1200.0
0.800	0.775	8500.0	5600.0

and the continuous versions of the Pursuit, and the TSE estimator algorithms. In their experiments, they required that their algorithm achieve a level of accuracy of not making any errors in convergence in 100 experiments. To initialize the reward estimates vector, 20 iterations were performed. The experimental results for the TSE algorithms are summarized in Table 2.2. The corresponding results of the pursuit algorithms are provided in Table 2.3. The results have shown that the discretized TSE algorithm was faster (between 50-76%) than the continuous TSE algorithm. Similar observations were obtained for the Pursuit algorithm. The discretized versions of the Pursuit algorithms were found to be at least 60% faster than their continuous counterparts. For example, with $d_1 = 0.8$ and $d_2 = 0.6$, the continuous TSE algorithm took an average of 115 iterations to converge, whereas the discretized TSE took only 76. Another set of experimental comparisons was performed between all the estimator algorithms presented so far in several ten-action environments [49]. Their results [49] are summarized below. The results show that the TSE algorithm is much faster than the Pursuit algorithm. Whereas, the continuous Pursuit Algorithm took 1140 iterations to converge, the TSE algorithm took only 310. The same observation applies to their discrete versions. Similarly, it was observed that the discrete estimator algorithms were much faster than the continuous estimator algorithms. For example, for environment E_A, while the con-

Table 2.3: The number of iterations until convergence in 2-action environments for the Pursuit algorithms (Adapted from [49])

Probability of Reward		**Mean Iterations**	
Action 1	**Action 2**	**Continuous**	**Discrete**
0.800	0.200	22	22
0.800	0.400	22	39
0.800	0.600	148	125
0.800	0.700	636	357
0.800	0.750	2980	1290
0.800	0.775	6190	3300

Table 2.4: Comparison of the discrete and continuous estimator algorithms in a benchmark with ten-action environments [49].

Environment	**Algorithm**	**Continuous**	**Discrete**
E_A	Pursuit	1140	799
E_A	TSE	310	207
E_B	Pursuit	2570	1770
E_B	TSE	583	563

tinuous algorithm took 1140 iterations to converge, the discretized algorithm took 799 iterations. The GPA and the DGPA algorithms were compared for the benchmark 10-action environments. There results are summarized in Table 2.5 (taken verbatim from [5]). The DGPA algorithm was found to converge much faster than the GPA algorithm. This once again proves the efficiency of the discretized algorithms over the continuous ones.

2.5.4 Stochastic Estimator Learning Algorithm (SELA)

The SELA algorithm belongs to the class of discretized LA, and was proposed by Vasilakos and Papadimitriou [103]. It has, since then, been used for solving problems

Table 2.5: Experimental comparison of the performance of the GPA and the DGPA algorithms in benchmark ten-action environments [5].

Environment	GPA		DGPA	
	λ	**No. of Iterat.**	N	**No. of Iterat.**
E_A	0.0127	948.03	24	633.64
E_B	0.0041	2759.02	52	1307.76

Note: The reward probabilities for the actions are:
E_A : 0.7 0.5 0.3 0.2 0.4 0.5 0.4 0.3 0.5 0.2
E_B : 0.1 0.45 0.84 0.76 0.2 0.4 0.6 0.7 0.5 0.3

in the domain of Computer Networks [6, 7, 78, 103, 104]. It is an Ergodic scheme, which has the ability to converge to the optimal action irrespective of the distribution of the initial state [103, 104]. The SELA algorithm is formalized below.

As before, let $A = \alpha_1, \alpha_2, \ldots, \alpha_r$ denote the set of actions, $B = \beta_1, \beta_2, \ldots, \beta_r$ denote the set of responses that can be provided by the Environment, where $\beta(t)$ represents the feedback provided by the Environment corresponding to a chosen action $\alpha(t)$ at time t. Let the probability of choosing the k^{th} action at the t^{th} time instant be $p_k(t)$. SELA represents the estimated Environmental characteristics as the vector $E(t)$, which can be defined as $E(t) = \{D(t), M(t), U(t)\}$ [103, 104], which is explained below.

$D(t) = \{d_1(t), d_2(t), \ldots, d_r(t)\}$ represents the vector of the reward estimates, where [103, 104]:

$$d_k(t) = \frac{\sum_{i=1}^{W} \beta_k(t)}{W} \tag{2.31}$$

In the above equation, the numerator on the right hand side represents the total rewards received by the Automaton in the window size representing the last W times a particular action α_k was selected by the algorithm. W is called the *learning window* [103, 104].

The second parameter in $E(t)$ is called the *Oldness Vector*, and is represented as $M(t) = \{m_1(t), m_2(t), \ldots, m_r(t)\}$, where $m_k(t)$ represents the time passed (counted as the number of iterations) since the last time the action $\alpha_k(t)$ was selected [103, 104].

The last parameter $U(t)$ is called the *Stochastic Estimator Vector*, and is represented as $U(t) = \{u_1(t), u_2(t), \ldots, u_r(t)\}$, where the stochastic estimate $u_i(t)$ of action α_i is calculated using the following formula [103, 104]:

$$u_i(t) = d_i(t) + N(0, {\sigma_i}^2(t)) \tag{2.32}$$

In the above formula, $N(0, {\sigma_i}^2(t))$ represents a random number selected from a normal distribution, which has a mean of 0, and a standard deviation of $\sigma_i(t) = \min\{\sigma_{max}, a\, m_i(t)\}$, where a is a parameter signifying the rate at which the stochastic estimates become independent, and σ_{max} represents the maximum possible standard deviation that the stochastic estimates can have [103, 104].

The SELA algorithm is presented below (certain sections of the algorithm are taken verbatim from [104]):

Algorithm: SELA

Parameters

N: resolution parameter
$D(t)$: true estimation vector
$M(t)$: oldness vector
$U(t)$: stochastic estimator vector
r: number of actions
Δ: the smallest step size $\Delta = \frac{1}{rN}$
W: learning window

BEGIN

Initialization

$p_i(t) = 1/r$, for $1 \leq i \leq r$
$d_i(t) = m_i(t) = u_i(t) = 0$, $\forall i \in \{1, 2, \ldots, r\}$

Repeat

Step 1

At time t pick $\alpha(t)$ according to probability distribution $P(t)$
Let $\alpha(t) = \alpha_i$.

Step 2

For the action α_k, receive the feedback $b_k(t)$.

Step 3

Compute the new true estimate $d_k(t)$ of the selected action α_k

$$d_k(t) = \frac{\sum_{i=1}^{W} \beta_k(t)}{W}$$

... (Continued to next page)

Algorithm: SELA (Continued from previous page)

Step 4
Update the oldness vector as follows: $m_k(t) = 0$ and
$m_i(t) = m_i(t-1) + 1 \quad \forall i \neq k$

Step 5
Compute the new stochastic estimate $b_k(t)$ $\forall i$:
$u_i(t) = d_i(t) + N(0, {\sigma_i}^2(t))$

Step 6
Sort the n actions in the increasing order of their stochastic estimate of mean reward so that the first element of that classification (which is the most optimal action α_m) will be the one with the highest value and the last one (and the w^{th} optimal action α_w) is the one with the lowest value. So, $u_m(t) = \max\{u_i(t)\}$ and $u_r(t) = \min\{u_i(t)\}$.

Step 7
Update the probability vector as follows: For every possible action, α_i $(i = 1, 2, \ldots, m-1, m+1, \ldots, r)$ with $p_i(t) > 0$ set:
$p_i(t+1) = p_i(t) - \frac{1}{N}$, where N is the resolution parameter, and determines the step size of Δ $(\Delta = 1/N)$ of the probability updating. For the optimal action α_m, set: $p_m(t+1) = 1 - \sum_{i \neq m} p_i(t+1)$, with $0 \leq p_i \leq 1$ and $\sum_{i=1}^{n} p_i = 1$.

End-Repeat
END

In symmetrically distributed noisy stochastic environments, SELA is shown to be ϵ-optimal. In other words, SELA converges to the optimal action in an environment that has such an option [104].

SELA has previously found applications in routing problems in ATM networks [6, 104].

2.6 Conclusions

In this Chapter we have discussed most of the important learning mechanisms reported in the LA literature. We first showed how the Markov chain formulation of the simplistic Fixed Structure LA could be extended to Variable Structure LA. We presented the fundamental concepts of LA, and then discussed the two classes of VSSA schemes, namely, the Continuous and Discretized Learning Schemes. The variable nature of the action probabilities of Variable Structure LA makes them a viable option to solve realistic network routing problems that we have studied as part of our research. In each case we have briefly summarized the theoretical and experimental results of the different learning schemes.

Our proposed solutions to the problems that we have considered in the subsequent Chapters are based on theoretical foundations of the LA that we have introduced here.

Chapter 3

Random Races-Based Learning

3.1 Principles of Random Races

The *Theory of Random Races* (RR) was pioneered by Ng, Oommen, and Hansen [64]. RR was inspired by the *Theory of Learning Automata* (LA). Consider the case of a learning machine (LM) interacting with a random environment (RE). Traditionally, in LA, a learning machine (automaton) is offered the set of actions by a random environment. The automaton chooses only one of the offered actions at a time, namely the one that it perceives to be the best action at that instant. The environment, which knows the "best action", either rewards the automaton or penalizes it with a certain penalty probability. Based on the continuous interaction between the automaton and the environment, the automaton attempts to learn the optimal action, i.e., the action that has the minimum penalty probability. This optimal action is eventually chosen more frequently than any other action.

3.1.1 Setting

The above LA-based philosophy would work fine in problem domains where one needs to fine the *best action*. However, the above solution is of limited help if the problem required *ordering* the set of actions in terms of their optimality. Ng, Oommen, and Hansen [64] identified this deficiency with the existing LA literature and proposed a solution, which is described below. Consider a LM interacting with a RE as depicted

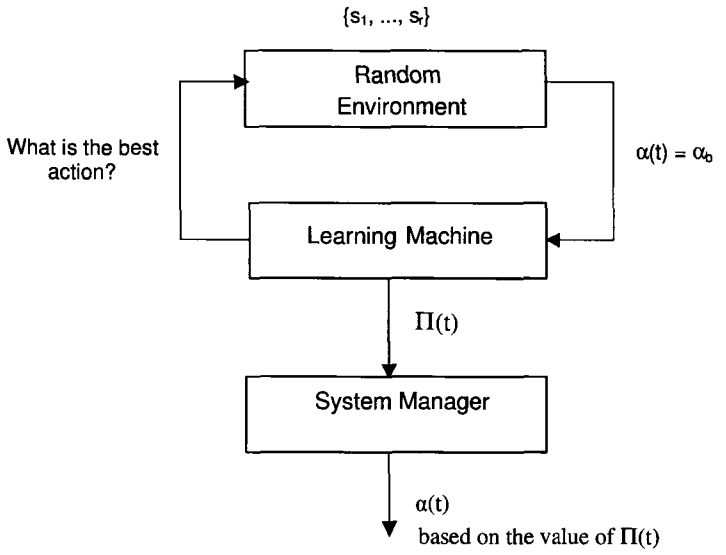


Figure 3.1: A learning machine interacting with an environment [64]

in Figure 3.1 [64]. Consider that the RE offers a set of r actions, $A = \{\alpha_1, \ldots, \alpha_r\}$. Unlike the LA-based solution where the LM chooses its best-perceived action that is in turn either rewarded or penalized by the RE, in RR, the LM (acting as a student) asks the RE (acting as a teacher) "which action do you think is the best action?". The RE suggests the action α_b to be the best action with probability s_b, which is, in fact, unknown to the LM. At any time instant t, the output of the LM is the permutation $\Pi(t)$, which is returned to the system manager interacting with the real world. The LM converges to an order of actions, say $\Pi(\infty)$, hopefully sorted in descending order

of their suggestion probabilities. As $t \rightarrow \infty$, $\Pi(t)$ should supposedly converge to $\Pi^{\star}$ with a probability arbitrarily close to unity, which will, in turn, be used by the system manager to choose the actions in the optimal ordering [64].

In terms of the notation and nomenclature, $S = [s_1, s_2, \ldots, s_r]$ is called the *suggestion probability vector*, and in this setting the "Teacher" (Environment) is said to be "suggestive", and so the probability with which the RE suggests the actions to the LM satisfies:

$$P_r[\alpha(t) = \alpha_b] = s_b(t) \tag{3.1}$$

$$\text{where } \sum_{i=1}^{r} s_b = 1 \tag{3.2}$$

We now briefly review the theoretical definitions pertinent to RR (taken from [64]), where the problem for the LM is to learn the optimal ordering of the actions offered by the suggestive RE. In the definitions below, ${}_ip_j(t)$ denotes the probability that at time "t" the automaton picks a permutation in which α_j succeeds α_i.

Definition 3.1: If $\forall i, j$, where $i \neq j$, the probability ${}_ip_j(t) \rightarrow 1$ as $t \rightarrow \infty$, with probability unity whenever $s_i > s_j$, the LM is defined to be *permutationally optimal.*

Definition 3.2: If $\forall i, j$, where $i \neq j$, the probability ${}_ip_j(t)$ can be made as close to unity as desired as $t \rightarrow \infty$, whenever $s_i > s_j$, the LM is defined to be *permutationally ϵ-optimal.*

Definition 3.3: If $\forall i, j$, where $i \neq j$, the probability ${}_ip_j(t) > 0.5 \quad t \rightarrow \infty$, whenever $s_i > s_j$, the LM is defined to be *permutationally expedient.*

3.1.2 Multiple Race-Track Learning With No Handicaps

Let us consider a setting where r racers run on multiple non-interfering tracks as shown in Figure 3.2. In other words, the racers process the information offered by the RE in a non-interfering manner. This scheme is called the *Learning Multiple-Race Track* (LMRT) strategy [64]. In this Thesis, we only consider the case where the racers do not have an *a priori* information of the ordering of the actions at the beginning of the race. In other words, all the racers start at the same origin. However, if the LM has *a priori* information about the actions, it is possible that the racers receive differential treatment with respect to the starting position in the race. Such a consideration is omitted in this paper, but the theoretical results can be found in [64]. In Figure 3.2,

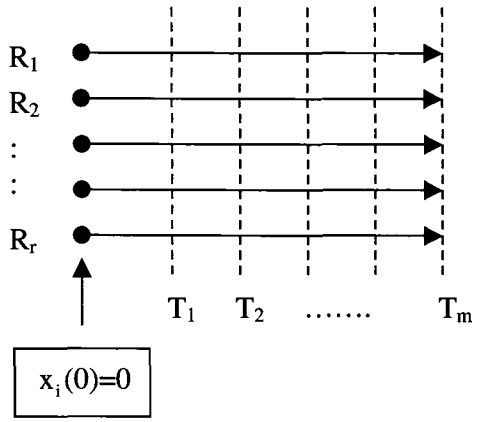


Figure 3.2: Multiple-track racers with no a priori information

we have r racers running on r non-interfering tracks. The learning parameter in this scheme is the length of the tracks, N. The output of this random race is the ordering in which the racers complete the course, i.e., the ordering in which the learning has converged, and it is represented by $\{R_i\}$, where $1 \leq i \leq r$. The positions of the racers at any time instant is represented by $x_i(t)$, where $x_i(0) = 0$, representing the fact that in this scheme, all the racers receive equal treatment at the beginning of the race.

Whenever the LMRT asks the Environment about the best action, at that instant the suggestion of the Environment, $\alpha_i(t)$, is used by the LMRT to move R_j one step towards the target (T_m) by incrementing the value of $x_j(t)$ by unity [64].

It can be shown that the above scheme is absorbing and that it is effective in learning the optimal ordering of the different actions participating in the random race with an arbitrarily large accuracy [64]. The following interesting result that we use in this Thesis for solving our problem is stated below without proof. The readers are referred to [64] for further details.

Theorem 3.1: If an LMRT utilizes no *a priori* information in all suggestive random environments, it is permutationally ϵ-*optimal* [64].

3.1.3 Multiple Race-Track Learning With Handicaps

The LMRT scenario that we considered in the previous Section was where the racers are not provided with any *a priori* information. In this Section, we will consider the case where the racers are provided with *a priori* information. In other words, the racers have handicaps at the start of the race, and are provided with *a priori* information before interacting with the environment [64], and are, in turn, initially positioned on the multiple race tracks as shown below in Figure 3.3. Notice that R_1 and R_3 do not start at the initial position $x_i(0)$. Let us assume that some of the r racers in the race are given preferential handicaps, as seen in Figure 3.3. Let us consider a racer R_i has an handicapped initial position $x_i(0)$. An LM may, during the course of the interaction, sometimes find this initial position misleading, and sometimes ignorable [64]. These scenarios be presented shortly.

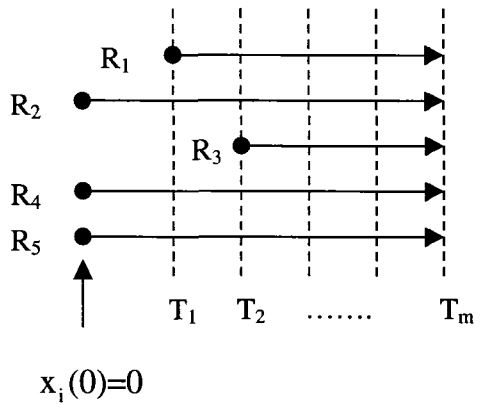


Figure 3.3: Multiple-track racers with a priori information

3.1.3.1 Multiple Race-Track Learning With Uniform Handicaps

Let us assume the case where the handicaps are uniformly distributed. We present the interesting results from [64] when the initial handicaps (*a priori* information) are useful or ignorable.

Theorem 3.2: In all suggestive random environments, if there are handicaps distributed in the interval $[0, N-1]$, and if the LM ignores any *a priori* information by providing handicaps, the LM is *permutationally expedient*, but not permutationally ϵ-*optimal* [64].

To clarify further, what Theorem 3.2 states is that, if in a race the racers are given uniformly distributed handicaps, and if the *a priori* information is ignored by the LM, the LM will have no way to unlearn the preferential treatment initially provided. This can neither be done by increasing the time in which the learning exercise takes place, nor by increasing the length of the track N [64].

However, if there is consistency between the handicaps, the *a priori* information that

was received, and the actual optimal ordering, Ng *et al.* [64] stated that the learning mechanism is capable of converging to the optimal ordering with a probability close to 1. On the other hand, the learning mechanism will not be able to assign uniformly distributed handicaps, if the *a priori* information is incorrect, and inconsistent with the optimal ordering [64].

3.1.3.2 Multiple Race-Track Learning With Non-Uniform Handicaps

In the previous Section we had seen the case where the starting position of the racers were uniformly distributed, and the *a priori* information was either used or discarded by the learning mechanism. We had seen that if the initial starting positions of the racers followed a uniform distribution, the learning mechanism is permutationally expedient [64].

We now consider the case where the starting positions of the racers do not obey a uniform distribution. In this case the following is true [64].

Theorem 3.3: If we have a suggestive environment, and if the initial starting positions (handicaps) provided to the racers in the race are geometrically distributed in the interval $[0, N-1]$, with any parameter $\beta > 0$, the learning scheme is permutationally ϵ-optimal [64].

To clarify further, if the racers in the race are given certain geometrically distributed initial positions, and if the *a priori* information is used by the learning scheme, and there is a consistency between this *a priori* information, and the actual optimal ordering, the LM will find this preferential treatment useful. Additionally, if the starting positions are distributed geometrically, the LM is capable of unlearning the initial preferential treatment if the *a priori* information is found to be incorrect and inconsistent with the actual optimal ordering [64].

3.1.4 Single Race-Track Learning

In this Section, we discuss the *Single Race-Track Learning* (LSRT) mechanism [64]. However, in the Thesis, since we do not consider single race-track learning anywhere, we mention it briefly, only for the sake of completeness.

Using notations similar to those used in the previous Sections, we have r racers $\{R_1, \ldots, R_r\}$ running on a single-race track. The LM tries to learn an optimal ordering between them in a suggestive environment, with the RE suggesting actions according to the unknown vector S. The parameter of the scheme is N, the length of the track. At time t, the position of any racer R_i is $x_i(t)$. Initially, at $t = 0$, the position is $x_i(0)$. Like before, the LM uses the suggestion of the RE to update the positions of the racers. Clearly, in single race-track scenarios, we need to take into account the issues related to how one racer would be able to overtake others. Unlike the LMRT, in the case of single race-track learning, it is possible to have a unique solution that gives the best ordering at every time instant [64].

In this case also, we may have scenarios where the racers are given handicaps or not, but we do not discuss them in further detail. The interested readers are referred to [64] for obtaining further details of these scenarios, if required.

The following theoretical results [64] are stated below without proof.

Theorem 3.4: In the case of LSRT operating in a suggestive random environment, the learning mechanism will be permutationally ϵ-optimal if the LM utilizes no *a priori* information [64].

Theorem 3.5: In the case of LSRT operating in a suggestive random environment, the learning mechanism is permutationally *ϵ-optimal* if the initial positions of the racers obey a geometric distribution in $[0, N-1]$ with any parameter $\beta > 0$ [64].

3.2 Conclusions

In this Chapter, we have considered the Random Races-based learning scheme, where the learning machine interacts with a random environment to learn the optimal *ordering* of actions. In the traditional LA-based models, the random environment offered the automaton with a set of actions, and the task of the automaton was to choose a certain action (with a certain probability). As opposed to this, in this case the random environment, on being asked the best action, suggests an action with a certain probability, and the LM has to learn the optimal ordering of the actions.

We have seen different learning scenarios, namely, where the racers do not have any handicaps, where the racers are given handicaps, where the handicap positions are uniformly distributed, and finally where the racers have geometrically distributed handicap.

The Random Races-based learning schemes find applications in situations where it is required to generate an optimal ordering of different actions, or paths in a network. The problem we encounter in Chapter 6 is tackled using the RR philosophy.

Chapter 4

Dynamic Single Source Shortest Path Routing

4.1 Introduction

In this Chapter[1], we report the results of our study of the *Dynamic Single Source Shortest Path Problem* (DSSSP).

[1]Two solution approaches are discussed in this Chapter: the linear learning approach, and the pursuit learning approach. The linear learning solution to the problem was published in the following forms:

1. Preliminary version: *Proceedings of the* 17^{th} *International Conference on Industrial & Engineering Applications of Artificial Intelligence and Expert Systems* (IEA/AIE 2004), Ottawa, Ontario, Canada (Lecture Notes in Artificial Intelligence, Vol. 3029, pp. 239-248, 2004, Springer-Verlag). This paper was nominated for the **Best Paper Award.**
2. Extended version: To appear in the *IEEE Transactions on Systems, Man, and Cybernetics*, Part B, Vol. 35, No. 6, December 2005.

The pursuit learning solution appeared in the following forms:

1. Preliminary version: *Proceedings of the* 9^{th} *IEEE Symposium on Computers and Communications* (IEEE ISCC 2004), Alexandria, Egypt, June 29 – July 1, pp. 891-896, 2004.
2. Extended version: *International Journal of Communication Systems*, Vol. 17, No. 10, pp. 963-984, 2004, John Wiley & Sons.

Multihop networks, such as the Internet and Mobile *Ad Hoc* Networks (MANETs), contain several routers and mobile hosts. The Internet typically employs routing protocols such as the Open Shortest Path Protocol (OSPF) and the Intermediate System - Intermediate System Protocol (IS-IS), and the MANETs employ protocols such as the Fisheye State Routing (FSR), the Optimized Link State Routing (OLSR), and the *Ad Hoc* On-Demand Distance Vector Routing (AODV).

In many of these protocols, each router (or a routing device) computes and stores a shortest path tree (SPT) from one router to all other routers and hosts in a routing domain [53, 63, 76, 91]. Such networks (graphs) typically contain several routers/switches (nodes) connected by links (edges) with constantly changing costs (weights), link-ups (edge-insertions) and link-downs (edge-deletions).

The problem of computing and maintaining information about the shortest paths information in a graph (with a single-source), where the edges are inserted/deleted and edge-weights constantly increase/decrease – is referred to as the *Dynamic Single Source Shortest Path Problem.* Although this problem is important, it has received little attention in the literature [26, 30, 63, 82]. The importance of the problem lies in the fact that it is representative of most practical situations in daily life, where most environments are dynamically changing. In such an environment, one needs to devise efficient solutions to maintain the shortest path even though there are updates on the structure of the graph by virtue of edge-insertion/deletion, or edge-weight increase/decrease, and hopefully, this can be achieved without re-computing everything "from scratch" following each topology update.

As will be discussed later in the Chapter, out of the four possible edge operations (insertion/deletion and increase/decrease), it has been shown that edge-insertion is equivalent to edge-weight decrease, and edge-deletion is equivalent to edge-weight increase. If all edge operations are allowed, the problem is referred to as the *fully-dynamic* problem. If only edge-insertion / weight-decrease (or edge-deletion / weight-increase)

is allowed, the problem is referred to as the *semi-dynamic* problem [30].

Abstracting the generic fully-dynamic problem to the network scenario, our intention is to devise efficient solutions to maintain SPTs while there are simultaneous stochastically-based link-updates taking place in the network.

Typically, with many present-day routing protocols[2], with a unit change in network topology (e.g., link-ups, link-downs, and link-cost changes), each router in the routing domain is intimated of the change. This change typically triggers recomputation of each router's SPT.

The well-known *static* solutions to the traditional combinatorial Single Source Shortest Path Problem [10, 19] are unacceptably inefficient in such dynamic practical scenarios, because using them would involve re-computing the shortest path tree "from scratch" each time a topology change occurs in the graph. Static algorithms are unarguably more effective in fixed-infrastructure networks because of their polynomial time-complexity. But they are extremely inefficient for time-critical rapidly changing infrastructures. For instance, if such an algorithm is used in a real-time large network routing scenario, where there are fast link-state changes, such re-computations will delay the execution of important routing functions considerably.

Two of the earliest known works on the dynamic shortest path problem date back to the papers by Spira and Pan [90] and McQuillan *et al.* [59]. While the former is theoretically proven to be inefficient, the latter has neither been proven theoretically nor through simulations.

The most recent and well-known solutions to the DSSSP on general graphs with positive real-valued edge-weights were proposed by Ramalingam and Reps [82], Fran-

[2]It should, however, be noted that some routing protocols in MANETs are based on the knowledge of a partial topology and provide the shortest path to any destination in the network. The advertised topology is a subset of the whole topology. Hence, ideally, only link-ups and link-downs concerning the advertised topology must be accounted for in the whole network. However, without getting into the intricate details of the individual routing protocols, we solve the problem for the general case described here. Our solution is also generic in the sense that it does not account for issues arising out of mobility and mobile nodes.

ciosa *et al.* [26], and Frigioni *et al.* [31]. However, the solution by Franciosa *et al.* is limited to the semi-dynamic problem only. Although these recent works are theoretical in nature, Ramalingam and Reps' and Frigioni *et al.*'s results were recently experimentally evaluated through simulations [22, 27]. While the former was found to be superior when it concerns running time, the latter was shown to be better suited when the worst-case time was the main concern, or when the number of edges to be updated had to be minimized.

The currently acclaimed fully-dynamic algorithms (mentioned above) are constrained by the following limitations:

1. The existing fully dynamic algorithms process unit changes to topology (i.e., edge-insertion/deletion or weight-increase/decrease) one change at a time, i.e., sequentially. When there are several such operations occurring in the environment simultaneously, the algorithms are quite inefficient.

2. In environments where the edge-weights change stochastically and continuously, the existing algorithms (mentioned above) would fail to converge to the actual underlying "average" solution.

The problems are worse in large topologies which have a large number of nodes and edges, and where a large number of topology changes can occur continuously at all times. In such cases the existing algorithms would fail to determine the shortest path information in a time-critical manner.

In Section 4.2 we describe, in detail, the two popular algorithms for solving the DSSSP problem previously available in the literature. In Sections 4.3, and 4.4 we provide two solution approaches (using linear learning, and pursuit learning), and some of the important results we have obtained from the experimental evaluation of our algorithms.

The idea of providing two solution approaches in this Thesis has been primarily

evolutionary. We first solved the problem by modeling the solution using a simple learning approach. This led to the design of the linear-learning approach, which had much better performance than the existing solution. We subsequently intended to see how we could possibly design better learning algorithms to solve the same problem. Although the pursuit learning solution led to more efficient algorithms, the advantage of the former algorithm is that it is simpler to implement, and easy to deploy in realistic network scenarios. We opted to report both the solution approaches. However, while reporting the results of the pursuit learning solution in Section 4.4, we have presented only those pieces of information, which were not presented while explaining the linear learning solution in Section 4.3.

4.2 The Dynamic Algorithms

This Section discusses the two most significant solutions to the DSSSP problem. In Section 4.2.1, we present Ramalingam and Reps' insertion/deletion algorithms, and in Section 4.2.2, we highlight Frigioni *et al.*'s decrease/increase algorithms. It can be seen that edge-insertions and deletions are equivalent to edge-length (weight) decrease and increase respectively. Increasing or decreasing an edge-weight can be performed by inserting a new edge (with the new weight) parallel to the edge under consideration, and then deleting the old edge [82].

4.2.1 Ramalingam and Reps' Algorithm (RR)

The first significant contribution to solving the fully dynamic DSSSP problem on general directed graphs with positive real edge-weights, was proposed by Ramalingam and Reps [79, 82]. They demonstrated their solution on a digraph with a single sink, and then calculated the shortest paths from all other nodes to the sink node. Their proposed algorithms were for cases of edge-insertions/deletions, and were based on

adaptations of the Dijkstra's solution to the static version of the problem [19].

The algorithmic descriptions below use the following notations: $G(V, E)$ denotes a digraph with a set of V nodes and E edges. The notation (u, v), an ordered pair of nodes, denotes a directed edge from u to v with a positive real-valued weight $l(u, v)$. (v, w) is the edge to be inserted in $G(V, E)$. The quantity, c denotes a positive real number indicating the length of edge (v, w), and the quantity, $dist(u)$ denotes the distance of u from the sink node, $S(G)$. The term PriorityQueue indicates a heap used to store vertices. $\forall v \in V(G)$, outdegree$_{SP}(v)$ is the outdegree of the vertex v in the shortest-path subgraph $SP(G)$, and dist(v) is the length of the shortest path from v to sink(G).

Edge-Insertion

The algorithm computes the length of the shortest path from every node $v \in V$ to the sink node. The set of all shortest path edges of the graph is denoted by $SP(G)$. The formal pseudo-code of Ramalingam and Reps edge insertion algorithm is presented below (taken from [82]).

Algorithm: InsertEdge$_{SSSP>0}(G,(v,w),c)$

Input

(i) G: a directed graph

(ii)(v,w): an edge to be inserted in G

Output

(i) $SP(G)$: the shortest-path subgraph of G after the insertion of (v,w)

(ii) dist(v): the length of the shortest path from v to sink(G), $\forall v \in V(G)$.

BEGIN

Insert edge (v,w) into $E(G)$

$l(v,w) := c$

PriorityQueue $:= \phi$

if $l(v,w) + dist(w) < dist(v)$ **then**

dist(v) := length(v,w) + dist(w)

InsertHeap(PriorityQueue, v, 0)

end

else if $l(v,w) + dist(w) == dist(v)$ **then**

Insert (v,w) into $SP(G)$ and increment outdegree$_{SP}(v)$

end

... (Continued to next page)

Algorithm: InsertEdge$_{SSSP>0}(G,(v,w),c)$ (Continued from previous page)

```
while PriorityQueue ≠ φ do
    u := FindAndDeleteMin(PriorityQueue)
    Remove all edges of SP(G) directed away from u and set outdegree_SP(u) == 0
    for every vertex x ∈ Succ(u) do
        if l(u,x) + dist(x) == dist(u) then
            Insert (u,x) into SP(G) and increment outdegree_SP(u)
        end
    end
    for every vertex x ∈ Pred(u) do
        if l(x,u) + dist(u) < dist(x) then
            dist(x) := l(x,u) + dist(u)
            AdjustHeap(PriorityQueue, x, dist(x) − dist(v))
        end
        else if l(x,u) + dist(u) == dist(x) then
            Insert (x,u) into SP(G) and increment outdegree_SP(x)
        end
    end
end
END
```

The insertion algorithm works in the following way [82]. First, an edge (v,w) is inserted into $G(V,E)$. If $\text{dist}(w)+l(v,w) < \text{dist}(v)$, it updates $\text{dist}(v) = \text{dist}(w)+l(v,w)$, and stores v into a priority queue (heap). Otherwise, if $\text{dist}(w)+l(v,w) = \text{dist}(v)$, it inserts (v,w) into $SP(G)$, and increments the outdegree of v. It then extracts from the heap a node u with the minimum priority. For each extracted node u in the above step, it traverses all outgoing edges from that node. For each node x which is a successor of u, if $\text{dist}(x)+l(u,x) = \text{dist}(u)$, it inserts (u,x) into $SP(G)$, and increments the outdegree of u. For every node x which is a predecessor of u, if $\text{dist}(u)+l(x,u) < \text{dist}(x)$ the algorithm updates the distance of x to $\text{dist}(u)+l(x,u)$, and inserts x into the heap while updating the priority to $\text{dist}(x)-\text{dist}(v)$. Otherwise, if $\text{dist}(u)+l(x,u) = \text{dist}(x)$, it inserts (x,u) into $SP(G)$, and increments the outdegree of x.

Edge-deletion

The edge-deletion algorithm works in two phases. The first phase of the algorithm determines those nodes and edges that are affected by the deletion of a particular edge, and deletes those affected edges from $SP(G)$. The second phase determines the new output value for all the affected nodes (*affectedNodesSet*), and updates SP(G).

The formal pseudo-code of Ramalingam and Reps' edge deletion algorithm is presented below (taken from [82]). In the pseudo-code given below, G is a directed graph, (v, w) is an edge to be deleted from G, WorkSet, and AffectedVertices are sets of vertices, PriorityQueue is a heap of vertices, and a, b, c, u, v, w, x, y are vertices.

Algorithm: DeleteEdge$_{SSSP>0}(G, (v, w))$

Input
(i) G: a directed graph
(ii)(v, w): an edge to be deleted in G

Output
(i) $SP(G)$: the shortest-path subgraph of G after the deletion of (v, w)
(ii) dist(v): the length of the shortest path from v to sink(G), $\forall v \in V(G)$.

BEGIN
if $(v, w) \in SP(G)$ **then**
Remove edge (v, w) from $SP(G)$ and $E(G)$.
Decrement outdegree$_{SP}(v)$
if *outdegree*$_{SP}(v) == 0$ **then**
//Phase 1: Identify the affected vertices and remove the affected edges from
//$SP(G)$
WorkSet := $\{v\}$
AffectedVertices := ϕ
end

... (Continued to next page)

Algorithm: DeleteEdge$_{SSSP>0}(G,(v,w))$ (Continued from previous page)

```
    while WorkSet ≠ φ do
        Select and remove a vertex u from WorkSet
        Insert vertex u into AffectedVertices
        for every vertex x such that (x,u) ∈ SP(G) do
            Remove edge (x,u) from SP(G) and decrement outdegree_SP(x)
            if outdegree_SP(x) = 0 then
                Insert vertex x into WorkSet
            end
        end
    end
    //Phase 2: Determine new Distances from affected vertices to sink(G)
    //and update SP(G)
    PriorityQueue := φ
    for each vertex a ∈ AffectedVertices do
        dist(a) := min({l(a,b) + dist(b)|
        (a,b) ∈ E(G) and b ∉ AffectedVertices)} ∪ {∞})
        if dist(a) ≠ ∞ then
            InsertHeap(PriorityQueue, a, dist(a))
        end
    end
    while PriorityQueue ≠ φ do
        a := FindAndDeleteMin(PriorityQueue)
        for every vertex b ∈ Succ(a) such that length(a,b) + dist(b) == dist(a) do
            Insert edge (a,b) into SP(G) and increment outdegree_SP(a)
        end
        for every vertex c ∈ Pred(a) such that l(c,a) + dist(a) < dist(c) do
            dist(c) := l(c,a) + dist(a)
            AdjustHeap(PriorityQueue, c, dist(c))
        end
    end

else
    Remove edge (v,w) from E(G)
end
END
```

From the pseudo-code above, it can be observed that in the first phase of the algorithm, it first deletes the Edge (v,w) from $SP(G)$. It then maintains a WorkSet

containing nodes which have been identified as *affected*, but which have not yet been processed. To start with, the algorithm inserts node v into the WorkSet, if (v, w) is the only shortest path edge going out of v. It then processes the nodes in the WorkSet serially. When a node u is processed, all SP edges coming into u are deleted from $SP(G)$. Finally, the algorithm inserts, into the WorkSet, all nodes that are *affected*.

During the second phase of the algorithm, for every affected node, a, the algorithm computes the new distance, dist(a), of the *affected* node, to the sink, using an approach similar to that of Dijkstra's [19], and if it is finite, it inserts a into a heap with priority, dist(a). In other words, if a is an *affected* node, and b is an unaffected node, then it determines the correct distance to the sink for each such unaffected node, and also computes the new distance for each such affected node a. Thereafter, it extracts each node, b, from the heap, which has the minimum priority. Then for each b, it determine a successor of a, such that dist(b)+$l(a, b)$ = dist(a), and inserts Edge (a, b) into $SP(G)$, and increments the outdegree of a. For each c, it determines a predecessor of a, such that dist(a) + $l(c, a)$ < dist(c), and updates dist(c) = dist(a) + $l(c, a)$, and inserts c into the heap, with priority dist(c).

4.2.2 Frigioni et al.'s Algorithm (FMN)

The second most significant solution to the fully dynamic DSSSP problem on digraphs with positive real weights was proposed by Frigioni, Marchetti-Spaccamela, and Nanni [30, 31]. We present here only a short (summarized) version of the decrease/increase algorithms [31]. As mentioned earlier, the insertion/deletion algorithms [30] are similar to the decrease/increase algorithms respectively, and are omitted from further discussion. The weight-decrease/increase algorithms on a graph G(V,E) which has source s, are based on understanding the following notations/concepts from [27], [31]:

- *weight_increase*(x, y, ϵ): Increases the weight of (x, y), $w(x, y)$, by an amount ϵ.

- *weight_decrease*(x, y, ϵ): Decreases the weight of (x, y), $w(x, y)$, by an amount ϵ.
- *backward_level* of an edge: If z is a node in G, the *backward_level* of Edge (z, q), and of node q, relative to node z, is given by $b\_\text{level}_z(q) = d(q) - w_{z,q}$.
- *forward_level* of an edge: The *forward_level* of an Edge (v, z) and of node v relative to node z, is given by $f\_\text{level}_z(v) = d(v) + w_{v,z}$. Both of these levels of an Edge (z, q) provide an indication of the shortest available path from s to q passing through z.
- *ownership*(x): For a Node x, *ownership*(x) denotes the set of edges owned by x, and *not_ownerhip*(x) denotes the set of edges not owned by x, but has an endpoint in x. owner(x, y) is the owner of (x, y).
- B_x: A max-based priority queue. Priority of (x, y) in B_x, (denoted by $b_x(y)$) is equal to $b\_\text{level}_x(y)$.
- F_x: A min-based priority queue. Priority of (x, y) in F_x, (denoted by $f_x(y)$) is equal to $f\_\text{level}_x(y)$.
- $P(x)$ is the Parent of x.

Only the major steps of the increase/decrease algorithms [31] are outlined here. The actual detailed steps are much more complex and are omitted to avoid a redundancy in the presentation, and also to maintain clarity in understanding the essence of the algorithm. Interested readers are referred to [31] to obtain further detailed pseudo-code level specifics of the algorithms.

Weight-decrease

The formal pseudo-code of the FMN algorithm, when a decrease in the weight of

edge (x, y) by a quantity ϵ, is presented below (taken from [31]).

Algorithm: Decrease(x, y : *vertex*; ϵ : **positive_real**)

Input
(i) G: a directed graph
(ii) (x, y): an edge whose weight is decreased.

Output
(i) $SP(G)$: the shortest-paths subgraph of G after the weight decrease.

BEGIN
// suppose wlog that $d(x) \leq d(y)$
Step 1
if $\epsilon \geq w_{x,y}$ **then**
Return ERROR // the weight of (x, y) becomes non-positive.
end
$w_{x,y} \leftarrow w_{x,y} - \epsilon$
Update_Local(x, y)
if $D(y) \leq D(x) + w_{x,y}$ **then**
EXIT // no distance improves
end

Step 2
$D(y) \leftarrow D(x) + w_{x,y}$
$P(y) \leftarrow x$
$C \leftarrow \phi$ // initialize an empty heap C
Enqueue$(C, < y, D(y) >)$

... (Continued to next page)

Algorithm: Decrease$(x, y : vertex; \epsilon :$ **positive_real**$)$ (Continued from previous page)

```
Step 3
    while Non_Empty(C) do
        < z, D(z) > ← Extract_Min(C)
        for each (z,v) s.t. (z,v) ∈ ownership(z) or (z,v) ∈ not_ownership(z) and
        v is high do
            if (z,v) ∈ ownership(z) then
                update b_v(z) and f_v(z)
            end
            if D(v) > D(z) + w_{z,v} then
                D(v) ← D(z) + w_{z,v}
                P(v) ← z
                Insert_or_Improve(C, < v, D(z) + w_{z,v} >)
            end
        end
    end

END
```

From the above weight-decrease algorithm, we observe that it operates in three phases. In the first phase it performs some pre-processing steps as mentioned presently. If the updated weight of the edge remains positive, then the data structures maintained at nodes x and y are updated depending on who is the owner of (x, y). If it is x, it updates the priority of x in B_y, and F_y . Otherwise, it updates the priority of y in B_x, and F_x . If the distance of y from s has decreased, the algorithm continues to Step 2; otherwise, it stops processing [31].

In the second phase, the algorithm computes $D(y)$, and $P(y)$ as follows: $D(y) = D(x) + w_{x,y}$ and $P(y) = x$. It then inserts y into an initially empty heap C, with priority $D(x) + w_{x,y}$. Finally, in the third phase, the algorithm updates the distances of the nodes from s. It computes in the following way, the new shortest path tree using a Dijkstra-like algorithm, by re-calculating only the portion of the subtree rooted at y for which the distances have changed. While the heap C is not-empty, the Node z

with the minimum priority $D(z)$, is extracted from C, and its priority is the shortest distance from s to z. It then propagates the new distance $D(z)$ along each Edge (z, v), such that either $(z, v) \in$ ownership(z) or $(z, v) \in$ not_ownership(z), and v is high for z meaning, the priority of v in B_z is greater than the new weight. For each such Edge (z, v), if $D(z)+w(z, v) < D(v)$, then $D(v) = D(z)+w(z, v)$, and $P(v) = z$. Thereafter, if v does not already exist in C, it is inserted into C, and its priority is updated [31].

Weight-increase

The weight-increase algorithm is based on the following node-coloring scheme:

- marking a node *white*, where such a Node q changes neither the distance from s nor the parent in the tree rooted in s,
- marking a node *red*, where such a Node q increases the distance from s, and
- marking a node *pink*, where such a Node q preserves its distance from s, but it replaces the old parent in the tree rooted in s.

The formal pseudo-code of the FMN algorithm, to increase the weight of edge (x, y) by a quantity ϵ, is presented below (taken from [31]).

Algorithm: Increase(x, y : *vertex*; ϵ : **positive_real**)

Input

(i) G: a directed graph

(ii) (x, y): an edge whose weight is increased.

Output

$SP(G)$: the shortest-paths subgraph of G after the weight inrease.

BEGIN

Step 1

Update_Local(x, y)

if (x, y) *is a non-tree edge* **then**

EXIT // no distance increases

end

$M \leftarrow \phi$ // initialize empty heap M

$Q \leftarrow \phi$ // initialize empty heap Q

Enqueue($M, < y, D(y) >$)

... (Continued to next page)

Algorithm: Increase(x, y : *vertex*; ϵ : **positive_real**)

Step 2
while *Non_Empty*(M) **do**
$< z, D(z) >\leftarrow$ Extract_Min(M)
if *there is a non-red neighbor* q *of* z *such that* $D(q) + w_{q,z} = D(z)$ **then**
$P(z) \leftarrow q$ // note that z is pink
end
else
color(z) $\leftarrow$ red
for *each* $v \in$ *children*(z) **do**
Enqueue($M, < v, D(v) >$)
end
end
end

Step 3a
for *each red vertex* z **do**
if z *has no non-red neighbor* **then**
$D(z) \leftarrow +\infty$
$P(z) \leftarrow$ Null
end
else
let u be the best non-red neighbor of z
$D(z) \leftarrow D(u) + w_{u,z}$
$P(z) \leftarrow u$
end
Enqueue($Q, < z, D(u) + w_{u,z} >$)
end

... (Continued to next page)

Algorithm: Increase(x, y : *vertex*; ϵ : **positive_real**) (Continued from previous page)

```
Step 3b
    while Non_Empty(Q) do
        < z, D(z) > ← Extract_Min(Q)
        for each edge (z, v) ∈ ownership(z) do
            update b_v(z) and f_v(z)
        end
        for each edge (z, h) such that h is red do
            if D(z) + w_{z,h} < D(h) then
                D(h) ← D(z) + w_{z,h}
                P(h) ← z
                Heap_Improve(Q, < h, D(z) + w_{z,h} >)
            end
        end
    end
    restore the original white color for all the red vertices

END
```

From the algorithm, it can be seen that it works in three main steps as described below. First of all, the algorithm updates the local data-structures. It then updates the priority of the inserted Edge (x, y) in the two priority queues stored at Node x or y, depending on who the owner is. If (x, y) is a tree-edge, y is inserted in a heap M with priority equal to its current distance from the source, i.e., $D(y)$.

In the second step, it colors the nodes of the graph. It extracts nodes from M in a non-decreasing order of their distances, and colors them as either *pink* or *red*.

In the third step, the algorithm computes the distances for the *red* nodes in the following manner. First, a heap Q is initialized. Each *red* node is inserted in Q, with priority equal to the sum of the distance of its best *non-red* neighbor q, and the weight of Edge (q, z). Then the values $D(z)$, and $P(z)$ for each *red* node z are initialized. Thereafter, the new distances of the *red* nodes are computed by applying a

methodology analogous to that of Dijkstra's algorithm [19].

4.3 Proposed Solution: Linear Learning Approach

4.3.1 Description

In Section 4.1, we mentioned the major limitations of the presently available algorithms to operate in dynamically changing stochastic network environments. Since such scenarios are representative of the actual environments in which the dynamic shortest path algorithms are likely to operate, the existing solutions would be limitedly useful. To the best of our knowledge, there is no known solution for finding the shortest path in a real-weighted graph where multiple edges are changing stochastically at once, and at the same time, which is more efficient than calculating everything "from scratch" for every change. The work reported here was inspired by the need for formulating an algorithm for finding the shortest path in such realistically occurring stochastic environments. Indeed, we seek to find the shortest path in the "average" graph (dictated by an "Oracle", also called the Environment). Since, on query, the edge-weights supplied by the "Environment" are assumed to follow an underlying unknown distribution, there exists a mean solution to the problem to which the algorithm would converge to after a sufficiently long time. The purpose of the work is to find the "statistical" shortest path in the average graph that will be stable – regardless of the (possibly) continuously changing weights provided by the environment. In this Section we present a new algorithm that uses Learning Automata [47, 66, 71] to generate superior results (when compared to the previous solutions).

The formal reasoning of why our algorithm works relies on the ϵ-optimal property of the L_{RI} scheme, the shortest-path property of Dijkstra's algorithm [19], and the update properties of the schemes of Frigioni *et al.*[31], and Ramalingam and Reps [82]. The

algorithm has been tested rigorously for numerous random synthetic graphs in various settings, and the results prove that the new strategy uniformly leads to superior results.

We unequivocally state that nothing we propose concerning the RR/FMN algorithms should be viewed with a "negative" connotation. We emphasize that our present contributions would not have been possible without these foundational works – *our results invoke and depend on them.* Indeed, quite modestly put, our proposed solution, the *Linear Learning-Based Shortest Path Algorithm* (LASPA) is an extension of the RR/FMN algorithms to handle dynamic stochastic graphs, where the distributions of all the edge weights are unknown. To put them in the right perspective, the RR/FMN dynamic algorithms are meant for situations where edge weights in the graph change with time, and one is interested in correctly tracking the current shortest paths. These are useful in cases where the network topology changes from time to time, and after every change we need to adopt shortest paths (to reflect current topology) without having to recalculate from scratch. Our LA-based algorithm presented here addresses a generalized (stochastic) version of this as follows – There is a network where edge weights are random (with unknown distributions), and what is desired is shortest paths with respect to averaged edge weights. Thus the desired shortest paths are unchanging, though these shortest paths cannot be obtained by standard methods, because the actual average edge weights are unknown. Thus, certain applications where dynamic single source shortest paths are useful such as those where there are sudden changes to network topology, are not the ones suited for the algorithm presented here.

The usefulness of LASPA is that all edges in a stochastic graph do not have to be probed. LASPA probes *only* those edges often that will be included in the shortest path graph. The other edges are probed minimally. We show how the edge-update schemes of the deterministic algorithms, namely, RR/FMN, can be leveraged for the generation of a high performance edge update algorithm for use in stochastic environments.

4.3.2 Motivation to the Linear Learning-Based Solution

As mentioned earlier, there is currently no efficient solution to the DSSSP problem when the edge-weights are dynamically, and stochastically changing. We believe that the reason for this is that the existing models for this problem are inadequate for this setting. We shall attempt to extend the current models by encapsulating the problem within the setting of the field of LA. To achieve this, we have to adequately model the three principal components of any LA system namely, the Automaton, the Environment, and the reward-penalty structure. In this context, we mention that in our case, the "system" would imply a *team* of LA interacting with the stochastic graph, and playing a *cooperative* game. We shall now clarify how we have achieved this.

The Automata: We propose to station an LA at every node in the graph so as to have invoked a game of automata operating in a sequential fashion. At every instance, its task is to choose a suitable edge from all the outgoing edges in that node. The intention, of course, is that it guesses that *this* edge belongs to the shortest path tree of the "average" overall graph. It accomplishes this by interacting with the Environment (described below). It first chooses an action from its prescribed set of actions. It then requests the Environment for the *current* random edge-weight for the edge it has chosen. The system computes the current shortest path by invoking either the RR or the FMN algorithms, whence the LA determines whether the choice it made should be rewarded or penalized as described below.

The Environment: The Environment consists of the overall dynamically changing graph. In the graph, there are multiple edge-weights, which change continuously and stochastically. These changes are based on a distribution that is unknown to the LA, but assumed to be known to the Environment. In a religious LA-Environment feed-

back, the Environment also supplies a Reward/Penalty signal to the LA. In our model, this feedback is *inferred* by the system, after it has invoked either the RR or FMN algorithms.

Reward/Penalty: Based on the action that the LA has chosen (namely, an outgoing-edge from a node which the LA *stochastically* "guesses" to belong to the shortest path tree), and the edge-weight that the Environment provides, the updated shortest path tree is computed. The effect of *this* choice is now determined by comparing the cost with the current "average" shortest paths, and the LA thus infers whether the choice should be rewarded or penalized. The automaton then updates the action probabilities using an appropriate scheme, and the cycle continues.

In this solution approach, we have opted to use the simple L_{RI} scheme, but in Section 4.4, we have used an efficient pursuit learning-based scheme.

4.3.3 The LASPA Algorithm

Let us consider the same abstraction of the graph topology as that considered by the FMN and RR algorithms, in which $G(V, E)$ denotes a dynamically changing directed graph with a set of V vertices, and E edges, which is to be processed by our algorithm. Suppose in the graph, randomly selected edge-weights change continuously in a dynamic fashion, depending on an (unknown) probabilistic distribution. Our aim is to determine the underlying shortest path of the average weights, which the system is unaware of. Multiple edge-weight changes occur in the graph continuously, and possibly simultaneously. As before, let (u, v) denote an ordered pair of vertices representing a directed edge from u to v with a positive real-valued weight, $w(u, v)$. The algorithm computes the length of the shortest path tree in the average graph from the source vertex s to every other vertex $v \in V$. The set of all shortest path edges of the graph

is denoted by $SP(G)$.

The problem is to find an efficient algorithm, which will help in determining the shortest paths from s to all other vertices $v \in V$ in G. Since insertions/deletions of edges are respectively equivalent to edge-weight decreases/increases, this work considers only the general case of weight-decrease/increase operations. The three efficiency considerations of the proposed algorithms investigated in this work are: (a) the average number of nodes processed per operation, (b) the average number of edges scanned per operation, and (c) the average time per operation. These are discussed in further detail below.

The proposed LA solution to DSSSP, named as LASPA, is now described. There are two variants of LASPA: (i) LASPA-RR: When LASPA uses RR, the Ramalingam and Reps' scheme, to process an edge-weight increase/decrease occurs, and (ii) LASPA-FMN: When LASPA uses FMN, the scheme due to Frigioni *et al.*, to process an edge-weight increase/decrease occurs. The general pseudo-code of LASPA follows after the following informal discussion.

Initialization:

1. To begin with, the algorithm obtains a snapshot of the directed graph with each edge having a random weight. This edge-weight is based on the random call for an edge, where each edge-cost has a (different) unknown mean and a variance. The algorithm maintains an action probability vector, $P = \{p_1(n), p_2(n), \ldots, p_r(n)\}$, for *each* node of the graph that contains the probability values for choosing different actions, $\{\alpha_1, \alpha_2, \ldots, \alpha_r\}$, offered by the random environment, which are the edges leaving the node. These are modeled as the actions, where $p_i(n)$ represents the probability of choosing action α_i at the n^{th} time instant. All elements of the action probability vector of a particular node are initialized to have a value equal to one divided by the outdegree of that node. A higher probability value

indicates a superior action. Observe that for each node, each possible outgoing edge corresponds to a probable action that can be selected for calculating the shortest path tree. Based on the chosen action, the system does shortest path computations, whence it determines whether the random environment presents the automaton with $\beta \in \{0, 1\}$, where 0 represents a reward and 1 a penalty. Let α_i denote the action corresponding to choosing the outgoing edge (x, y) from node x to node y, and let $dist(x)$ be the shortest path distance of x from the source, and $w(x, y)$ the weight of edge it (x, y). We model:

$$\beta = 0 \Rightarrow \text{dist}(x) + w(x, y) < \text{dist}(y) \tag{4.1}$$

$$\beta = 1 \Rightarrow \text{dist}(x) + w(x, y) \geq \text{dist}(y) \tag{4.2}$$

The above concept has been pictorially depicted in Figure 4.1. The values in the brackets represent the elements of the initial action probability vectors for each node. Each element of the said vector for a node corresponds to the probability of choosing the action representing the corresponding outgoing edge from that node. The vectors for nodes D and H are not shown to avoid cluttering in the figure. Also, there is no vector corresponding to Node I, as it does not have any outgoing edges.

2. Dijkstra's Algorithm is run once to determine the shortest path edges on the graph snapshot obtained in the first step. Based on this, the action probability vector of each node is updated such that the outgoing edge from a node, which is determined to belong to the shortest path edge has an increased probability than before the update. This is pictorially shown in Figure 4.2. Since the L_{RI} probability-updating scheme was chosen, only rewards are processed.
   Iterations:

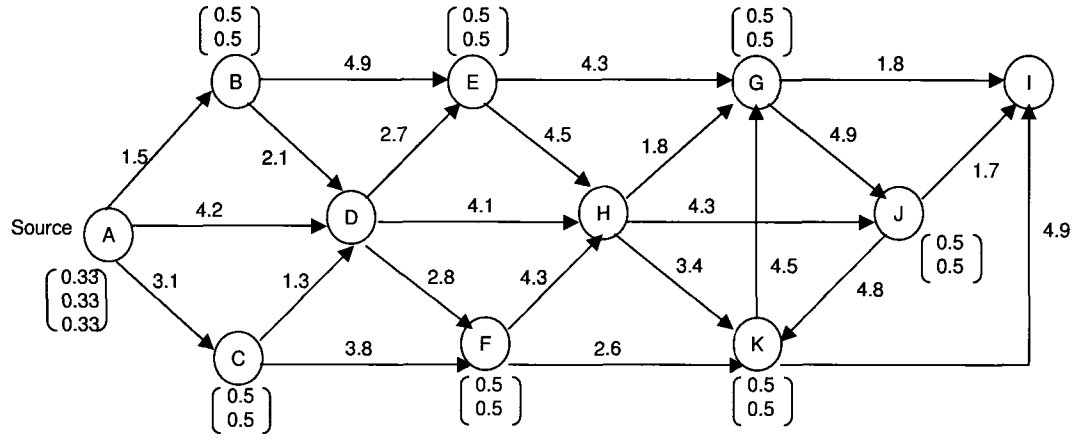


Figure 4.1: A snapshot of a hypothetical directed graph at a particular time instance (t). The randomly changing weights of the graph are real-valued. All the edge-weights have means between 0.0 and 5.0, and variances ranging between 0.0 and 0.9.

3. A node is randomly chosen from the current graph. For that node, based on the action probability vector, an edge is chosen. For example, if a node has three outgoing edges (three possible actions) and the action probability vector for that node is $\{0.3, 0.2, 0.5\}$, the edge (action) chosen is selected based on this distribution.

4. The Environment is then requested for the current weight of the edge that is randomly selected in Step 3 above. Since the edge-weights are real numbers, the edge-weight should have increased/decreased. Thus, the current shortest path tree is calculated using either of the existing edge-weight increase/decrease algorithms (i.e., either RR or FMN algorithm).

5. The action probability vectors for all the nodes are updated such that the edges that belong to the shortest path tree have a greater likelihood of being selected than before the update.

6. Steps 3-5 above are repeated a large number of times until the algorithm converge.

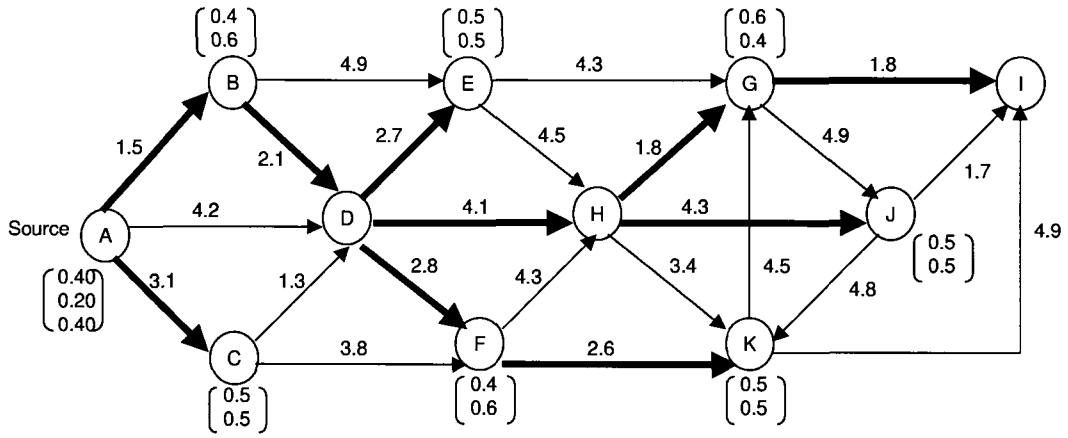


Figure 4.2: Graph of Figure 4.1 showing the shortest path edges (bold arrows) and the updated action probability vectors.

The pseudo-code of the LASPA algorithm is given below. In the interest of brevity, only the main steps of the algorithm are presented. The other functions/procedures are straightforward.

Algorithm: LASPA

Input

(i) $G(V, E) = A$ dynamically changing graph with simultaneous multiple stochastic edge updates occurring on it.
(ii) iters = total number of iterations
(iii) λ = learning parameter

Output

(i) A converged graph that has all the shortest path information
(ii) Values of all action probability vectors

BEGIN

```
G' = obtainAGraphInstance(G); // Obtain a snapshot of the graph G
for (each vertex v in G') do
    outdegree = checkNumOutgoingEdges(v)
    for (each outgoing edge e of v) do
        InitialProbability(e) = 1.0/(outdegree) // Initialize action probabilities
    end
end
updateActionProbabilityVector() // Update the action probability vectors for
                                // all vertices
// Initialize Structures and consider all vertices
for (each vertex v from source s) do
    executeDijkstraShortestPath(v) // Run Dijkstra's shortest path algorithm
                                   // once
end
updateActionProbabilityVector() // Update the action probability vectors for
                                // all vertices
```

... (Continued to next page)

Algorithm: LASPA (Continued from previous page)

```
    for (i = 0 to iters) do
        // Execute for all iterations.
        randomVertex = getRandomVertex() // Randomly choose a vertex in the
                                         // current graph.
        ap = getActionProbabilityVector(v) // Obtain the current action probability
                                           // vector for the vertex
        e = chooseAnEdge(ap) // Choose an edge based on the action probability
                             // vector for v
        currentWeight = getRandom(e) // Obtain a random value of the edge e
                                     // from the environment
        oldWeight = getExistingWeight(e) // Obtain the existing weight
        if (oldWeight > currentWeight) then
            executeDecreaseWeight(e) // Execute the decrease weight algorithm
                                     // of FMN or RR
        end
        else
            executeIncreaseWeight(e) // Execute the increase weight algorithm
                                     // of FMN or RR
        end
        updateActionProbabilityVector() // Update the action probability vectors
                                        // for all vertices
    end
END
```

The formal reasoning of why the above algorithm works probably relies on the ϵ-optimal property of the L_{RI} scheme, the shortest-path property of Dijkstra's algorithm, and the update properties of FMN/RR. We provide below an intuitive reasoning (without a formal proof) in support of our above statement.

A close inspection of the LASPA algorithm shows that the only place it calculates the shortest paths are when executing Dijkstra's algorithm on the graph's snapshot taken at the initialization step, and when executing the decrease- and increase-weight algorithms of FMN. Dijkstra's algorithm is well known to find the optimal shortest paths on a given graph topology. Since it is initially executed on the static graph snapshot, it necessarily calculates the *initial* optimal shortest paths. Furthermore, this

means that, if the increase/decrease algorithms of FMN operate correctly, the updated shortest path graph, obtained after invoking the FMN scheme on a per-node basis, will also be accurate. By virtue of the way the graph weights are presented to the algorithm, it is clear that as the number of iterations tends to infinity, the computed average weight of the edges will converge towards its true mean value. Thus, the asymptotic shortest paths obtained over the different iterations of LASPA will converge towards the list of optimal shortest paths obtained using the mean values. The proof that the shortest path is found with a probability as close to unity as desired, probably follows because of the fact that the action probability updating scheme used to select the edges for a given node, the L_{RI}, is ϵ-optimal, and thus, the probability of the scheme choosing the most pertinent edge (for every node) will be arbitrarily close to unity. Therefore, as the number of iterations tends towards infinity, the probability of choosing the optimal action (edge) for each node tends towards unity, and the overall path costs will tend to the shortest path costs of the mean edge weights.

With regard to computational superiority, the FMN algorithm recalculates all the affected shortest paths for *every change* in the edge-weights. However, in such an environment, after convergence, the LASPA-FMN algorithm will already have attained to the optimal shortest path lists for the graph, and therefore, will do nothing in terms of re-computing the shortest paths for *every change* in edge-weight. Therefore, after convergence, the performance of LASPA-FMN in terms of the *average number of processed nodes*, *the average number of scanned edges*, and the *average time per update operation* will be superior to the FMN algorithm.

Although the proposed scheme is quite powerful, our algorithm, in its current form, has some limitations as identified below. As it stands now, the algorithm has been designed for graphs where the link *structure* is fixed. The number of actions of each automaton, which is equal to the "valency" of a node, is limited by the underlying graph structure (which is provided as an input to the scheme in the initialization step). We

believe that this limitation can possibly be rectified by allowing for a large actions set, equal to *total* number of nodes, at each node automaton. This increased set of actions per node will have the effect of slowing down the convergence of the method, and seems to be a bottleneck. We are currently studying this aspect of the algorithm, and hope to resolve this in the near future.

We would also like to mention that Ramalingam and Reps have reported two versions of the edge update algorithm – one that processes unit changes to the graph topology [82], and the second that processes multiple changes simultaneously [83]. The reader will observe that in our current scheme, we used only their sequential version to generate the LASPA-RR algorithm. We also intend to study, in the near future, the performance of LASPA when it uses the latter algorithm of Ramalingam and Reps. Yet, based on the results we have currently obtained (using the sequential version of RR), we believe that the enhanced version of LASPA (that utilizes Ramalingam and Reps' algorithm for performing multiple changes in edge weights) will also perform better. This is because the LASPA algorithm can be carefully designed on top of any deterministic edge-update algorithm so that it performs well in stochastic environments[3].

A final note on the rate of convergence of LAPSA is also not irrelevant. The rate of convergence of stochastic systems is typically measured by the eigenvalues of the system, and by bounding the time curve from above and below by sub-regular and super-regular functions [66]. This has been done in the literature only for absolutely expedient schemes, but, as far as we know, no analysis has been done for such games of automata. We thus believe that the best that we (or the state-of-the-art) can offer is the experimental demonstration of the rate of convergence and its comparison with the other existing schemes. This is what we currently embark on.

[3]The problem that we have not solved (in this regard) is that of determining which edges will have to be probed so that the "unimportant" edges are visited infrequently. We are open to any helpful feedback or suggestions from readers or other researchers in the field.

4.3.4 Experimental Details

4.3.4.1 Experimental Design

Several experiments were designed to evaluate LASPA. This Thesis reports the results of the following sets of experiments:

- **Experiment Set 1:** Comparison of the performance of LASPA with FMN, and RR in a stochastic environment with simultaneous multiple edge-weight updates for a fixed unknown underlying graph structure.
- **Experiment Set 2:** Comparison of the performance results with variations in graph structures.
- **Experiment Set 3:** Sensitivity of the performance of LASPA with variations of certain parameters, while maintaining others constant.

For the above experiments, two random generators were built:

- **Random Graph Topology Generator:** This module builds a random directed graph with n nodes, e edges, and positive real edge-weights. The edge-weights are randomly generated and are normally distributed. The random graph can be built from either specifying the number of nodes and edges or by specifying the graph sparsity (described below). A sample file showing the graph topology is shown in Figure 4.3. In the Figure, "owner" refers to the node, which owns the edge (see Section 4.2.2). The rest of the elements shown in the Figure are self-explanatory.
- **Update Sequence Generator:** This module is a random generator of edge-weight-increase/decrease update sequences. It creates a mixed sequence of increase/decrease operations on edges chosen at random in the graph. During the

experiments, the sequence generator was preset such that at any instance the occurrence of an increase or decrease operation is equally likely. Figure 4.4 shows a sample output from the update sequence generator. In the Figure, "I" denotes an increase operation, and "D" denotes a decrease operation. There are 200 increase/decrease operations shown in this Figure.

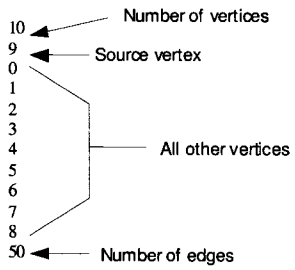
The algorithms were tested both on manually entered, and randomly generated graphs. However, the results shown were generated using random synthetic graphs only, as these permitted greater flexibility in increasing the size of the graphs, and in changing the other experimental parameters.

All the algorithms were tested with graphs of similar nature, i.e., graphs with single source, directed edges, positive real edge-weights that are normally distributed, and with stochastic edge-updates. Only the two edge operations, namely, edge-weight increase/decrease were implemented and tested. This was because, as discussed earlier, edge-deletion/insertion is equivalent to edge-weight increase/decrease respectively. All the discussions, hereafter, use *sparsity* of graphs, measured in percentage, to signify the number of edges in the graph as compared to the maximum number that can be supported by the set of nodes. The percentage density of the graphs can be obtained by subtracting the value of sparsity from 100.

4.3.4.2 Performance Metrics

Three metrics were used for evaluating the performance of the algorithms invoked in the experiments [22, 27]:

1. *Number of scanned edges*: This quantity measures the number of edges that are scanned in edge-weight increase/decrease operations. The first scanned edge is the edge whose weight is changed. Next, when an edge is scanned to check if it is in the shortest path, the counter for this index is incremented.



Origin	Dest	Current Weight	Mean Weight	Weight Variance	Owner
4	9	3.9175076563673485	4.787915413721666	0.6174698163261482	4
9	2	2.217629 1045844727	2.457289528808339	0.8163927473337957	9
8	6	2.829695633508808	2.786598042979919	0.5957601537742686	8
5	0	1.0924386721369936	1.3090224117237366	0.690630373928 1151	5
2	1	1.5533050651600857	1.7813553200505305	0.627788405 1644501	2
1	9	1.4335083819412535	1.6325221708712734	0.193275705622936 14	1
9	7	1.6398165634504638	1.5430461883933924	0.162821200258279 3	7
4	0	0.667794735551188	2.323625052578074	0.851422316079039 4	0
1	8	3.120211520567999	2.708829402891308	0.2306690209310 4888	1
8	4	3.2518432672486797	2.889704074585863 3	0.770336325389499 6	8
3	5	2.110562062378528	2.236562386908866	0.4415993088586418 5	3
7	5	2.426261980051524	2.832562618669465 7	0.707103639171067	7
4	6	1.900636640218884 1	2.417959633842292 6	0.325178839714260 8	6
1	3	4.274611110947 1325	4.689403689272289 5	0.626247093527594 8	3
0	8	3.630486125419496 4	4.053011859343687 5	0.686247791739690 2	0
2	7	4.157711372933329	3.104542346826487 4	0.776807394585858 8	2
2	9	2.150011849541 1535	1.849660968240023	0.881429260445044 8	9
9	4	4.6341902291405574	4.998846647116368	0.147114820843944 75	4
9	6	1.9729997491517726	2.141420663966457	0.127127866178073 67	6
4	5	1.097732606309691	1.426626823009035 8	0.229451295516455 38	5
3	4	2.1821736783723953	2.239422327385801 4	0.557156913556216 4	3
5	1	1.1770717714570007	1.199015249558626 8	0.792601833822146 7	5
0	6	5.232891076417296	4.842646759778381	0.592769486510203 2	0
2	3	3.9161265674871117	3.975994292357336	0.119536833442070 29	2
8	5	0.5160441133816497	1.498503428169302 6	0.866148346185209 4	8
6	9	3.536560944934987	3.049659363304783 7	0.861533174932864 2	6
8	7	4.412915195552159	3.527671341328696 6	0.839081274196376 6	7
8	3	4.428029486128453 5	4.893736263104770 5	0.721055259569872 7	8
6	3	0.17838726191552212	1.143943728507066 8	0.763870944197647 1	6
6	2	2.439430039452539 2	4.543402331452526	0.823585859413667 2	2
9	1	1.191774926743905	1.435889651 1134205	0.203510545506537 27	9
1	0	4.756349576714037	4.759487351836999	0.374360482125044 47	1
0	1	4.329613281168329	4.543350069566 25	0.765212223331097 2	0
1	2	1.875962542584718	1.294622034318057 7	0.755651045237533 8	1
0	4	2.177565865101 7995	2.756839621419867 4	0.509791291737630 8	4
1	5	3.7407070989695 72	4.152384347116263	0.747514704627920 2	5
3	1	1.659461599041 1525	1.655404648358911 3	0.270666892634626 84	3
0	2	4.726942014272581	4.418692668210433	0.525512000119885 4	0
9	8	0.7855912131901117	1.021132937 1798592	0.287745986330581 35	9
8	9	5.6327207685714455	4.537636945867101	0.708363533739143 3	8
7	0	4.057030902454548	4.787239311704459	0.830686397867717 5	7
5	2	2.152459720208025	2.229616500351606	0.301937168244257 1	5
6	8	3.944492847278641	3.679206160582917 4	0.615862286723728 4	6
0	5	2.811037705156871	2.919121155892979	0.159435693708749 08	0
0	7	1.864412880425035 5	2.269562034447317	0.657430076534338 2	7
7	3	3.950259168475957 6	3.527666925007876 5	0.620049008463354 9	3
2	5	3.709566346879519	3.855782920906115	0.829369468733608	2
6	5	2.704645689525551 5	2.126475563444172	0.726855128958102 9	6
3	8	3.154144051143 1755	2.550727197343823 2	0.447853527188390 67	3
1	4	3.246336475606148	2.732756932166130 2	0.630644273846730 6	4

Figure 4.3: Sample graph topology file.

I,I,D,D,D,D,I,I,D,D,I,D,I,I,I,D,D,D,D,D,D,I,I,D,D,D,D,I,D,I,I,D,D,I,D,I,D,I,I,D,D,D,D,D,D,I,I,D,I,D,D,D,I,I,
D,I,I,I,I,D,D,I,I,D,D,D,D,I,I,I,D,D,I,D,I,D,I,I,D,I,I,D,I,I,I,D,D,I,D,I,D,I,I,D,I,I,D,I,D,I,I,D,I,D,I,I,D,I,D,
I,I,D,I,D,I,I,D,D,D,I,D,D,D,D,D,D,D,D,I,I,I,I,I,I,D,D,D,I,I,D,D,D,I,I,D,D,I,I,D,D,I,I,D,D,I,I,D,D,I,I,I,I,I,I,I,I,
D,I,I,D,D,I,I,I,D,I,I,I,I,D,I,I,I,I,D,D,D,I,I,I,I

Figure 4.4: Sample output of the update sequence generator.

2. *Number of processed nodes*: This quantity measures the number of nodes that are processed in edge-weight increase/decrease operations. It is incremented each time a node in the algorithm is removed from the priority queue that stores the nodes for processing.

3. *Time required per update operation*: This quantity is the running time required to update weights and to obtain all the shortest paths.

Although our study considered only the above mentioned three metrics, it is clear that other metrics ([22, 27])can also be proposed to assess the performance of our algorithms. A few of these possible metrics are:

(i) Average time taken for a single execution of the algorithm over several executions.

(ii) Maximum time taken for a single execution of the algorithm over several executions.

(iii) Minimum time taken for a single execution of the algorithm over several executions.

(iv) Average amount of space consumed for a single execution of the algorithm over several executions.

(v) Maximum space consumed for a single execution of the algorithm over several executions.

(vi) Minimum space consumed for a single execution of the algorithm over several executions.

(vii) Time for initialization of the data structures of the algorithms.

(viii) Average number of nodes processed in the distance update phase of the algorithms.

(ix) Average number of edges scanned in the distance update phase of the algorithms.

While the above-mentioned measures (i) through (ix) can also be utilized to compare the performance of the algorithms considered in the study, we suggest that they are not as important (as the ones we have currently utilized) in judging the performance of any dynamic shortest path algorithm. A responsive dynamic shortest path algorithm should be able to adapt itself to topology updates in the least possible time, by processing the least number of nodes, and by scanning the least number of edges for each update operation. From this perspective, the overall time computation measures (i) through (iii) are less important than the time taken per update operation. Furthermore, because of the decreasing costs of storage in the market, we believe that the space complexity measures (iv) through (vi) should not be considered to be significant. Finally, although the performance measures (vii) through (ix) are valid for comparing the RR and FMN algorithms, since LASPA essentially invokes the FMN and the RR algorithms, we believe that these measures are not significant in the current setting. However, we emphasize that this is merely *our* perspective.

4.3.4.3 Experimental Results

This Section reports the results of the experiments that were conducted to examine the performance of LASPA. The performance was compared with respect to the three indicators discussed in the previous Section. Several experiments were performed on

random graph topologies, and random edge-update sequences for different parameters. The experiments showed that LASPA does not perform well at the beginning, i.e., when the algorithm is learning, but after the algorithm has learned, LASPA outperforms both RR and FMN algorithms. The results of the three sets of experiments are summarized below. Only the running average values of the different metrics over the whole sequence of update operations are plotted in the graphs below. The programs kept track of these running average values, and they are cumulative in their nature.

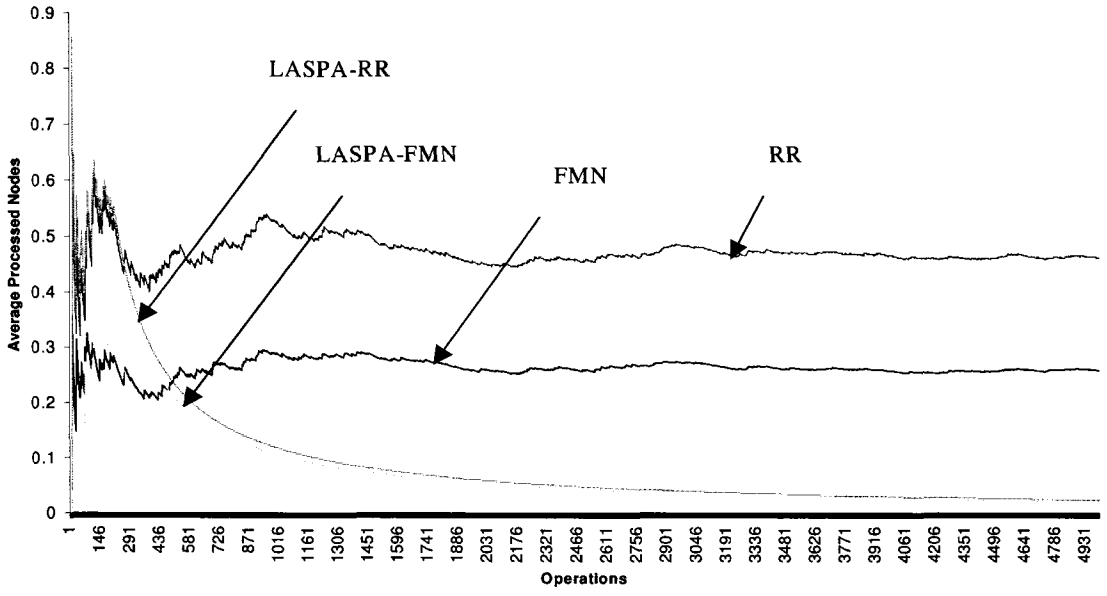


Figure 4.5: Average Processed Nodes

4.3.4.3.1 Experiment Set 1

The implementations of RR, FMN, LASPA-RR, and LASPA-FMN were run on mixed sequences of 5000 modifying edge-update operations performed on a graph topology with 20 nodes, and 50% sparsity. The edge-weights use random real values having a mean between 1.0 and 5.0, and with variances between 0.1 and 0.9. The value of the

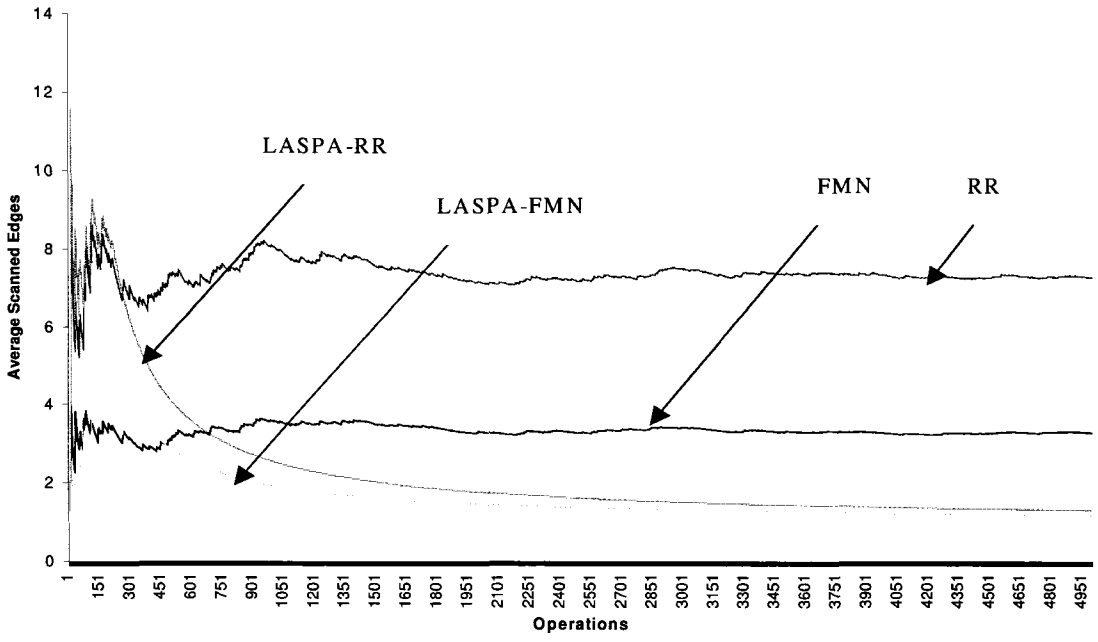


Figure 4.6: Average Scanned Edges

L_{RI} learning parameter, λ, was set to 0.995.

The results of the experiment are shown in Figures 4.5 through 4.7. The results show $75-95\%$ improvement of the performance of LASPA with respect to that of FMN and RR. Inspection of the plots reveals that, initially, LASPA-FMN performs worse than FMN, and LASPA-RR performs worse than RR. This is quite expected, because initially LASPA is unaware of the stochastic behavior of the Environment, and is in the process of learning. But it does not take it too long to converge. After this, LASPA utilizes its knowledge about the Environment, and its performance exceeds that of both FMN and RR. After convergence, the average number of nodes processed, the average number of edges scanned, and the average time spent per update operation are much less for LASPA than for both FMN or RR. For example, as seen from Figure 4.5, when the number of operations is 4375, the average number of processed nodes for RR is

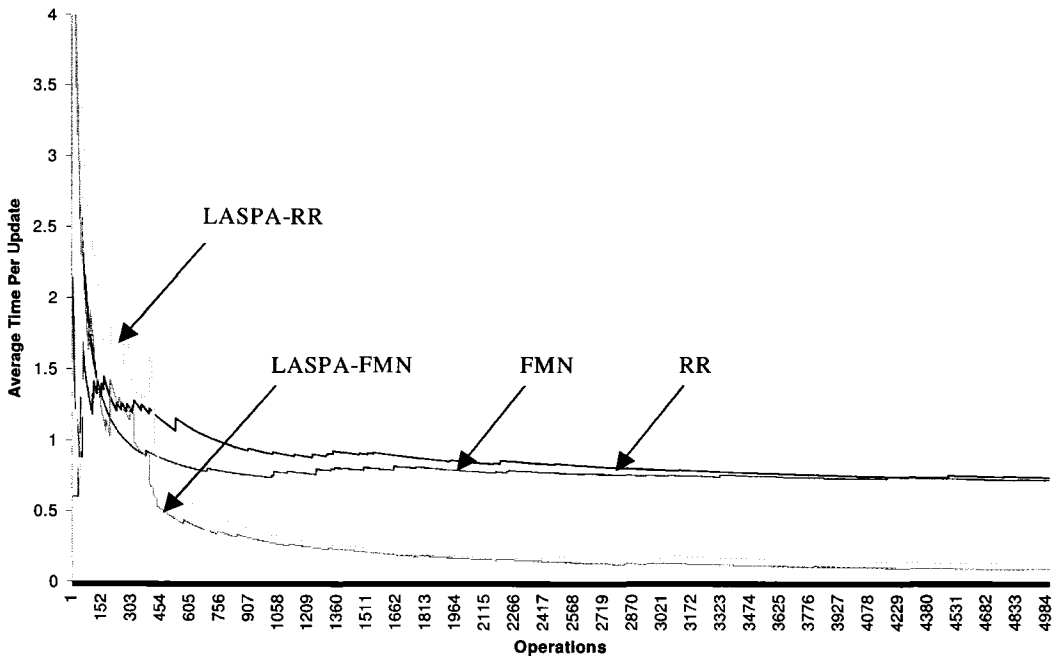


Figure 4.7: Average Time Per Update

approximately 0.46, whereas the average number of processed nodes for LASPA-RR is approximately 0.2.

Experiments were run on other graph topologies as well, and they yielded similar results. LASPA was found to be superior with respect to these three metrics in all the experiments conducted.

4.3.4.3.2 Experiment Set 2

A second set of experiments was conducted to evaluate the algorithms with a variation of the graph structure, specifically, the graph sparsity, and the number of nodes in the graphs. In other words, our aim was to observe whether there was a different trend in performance results when the structures of the graphs were varied, keeping other parameters constant.

Table 4.1: Statistics reporting the *Number of Processed Nodes* with the variation of the graph sparsity.

Sparsity	FMN	RR	LASPA-FMN	LASPA-RR
10%	0.251/0.0/114.0	0.241/0.0/136.0	0.03/0.0/21.0	0.08/0.0/24.0
30%	0.196/0.0/66.0	0.425/0.0/136.0	0.004/0.0/9.0	0.012/0.0/20.0
50%	0.216/0.0/66.0	0.492/0.0/76.0	0.004/0.0/12.0	0.006/0.0/17.0
70%	0.188/0.0/96.0	0.0872/0.0/94.0	0.0021/0.0/17.0	0.005/0.0/18.0
90%	0.161/0.0/64.0	1.174/0.0/112.0	0.0002/0.0/11.0	0.0036/0.0/12.0

Table 4.2: Statistics reporting the *Number of Scanned Edges* with the variation of the graph sparsity.

Sparsity	FMN	RR	LASPA-FMN	LASPA-RR
10%	11.733/1.0/5871.0	18.456/2.0/7626.0	1.013/1.0/241.0	1.0828/1.0/219.0
30%	7.113/1.0/2422.0	19.02/2.0/5973.0	1.128/1.0/303.0	1.259/1.0/866.0
50%	7.119/1.0/1779.0	16.55/2.0/2353.0	1.09/1.0/299.0	1.416/1.0/739.0
70%	7.574/1.0/1616.0	16.89/2.0/2032.0	1.472/1.0/311.0	1.124/1.0/798.0
90%	6.944/1.0/875.0	16.0/2.0/914.0	1.132/1.0/196.0	1.112/1.0/315.0

Table 4.3: Statistics reporting the *Time Per Update* with the variation of the graph sparsity.

Sparsity	FMN	RR	LASPA-FMN	LASPA-RR
10%	3.732/0.0/940.0	6.418/0.0/1320.0	1.872/0.0/510.0	1.87/0.0/520.0
30%	2.534/0.0/680.0	6.256/0.0/990.0	1.83/0.0/390.0	1.734/0.0/630.0
50%	2.606/0.0/330.0	6.278/0.0/2300.0	1.988/0.0/410.0	1.812/0.0/330.0
70%	1.954/0.0/510.0	4.394/0.0/490.0	2.012/0.0/390.0	1.991/0.0/340.0
90%	1.674/0.0/480.0	3.23/0.0/490.0	1.782/0.0/330.0	1.671/0.0/330.0

Table 4.4: Statistics reporting the *Number of Processed Nodes* with the variation of the number of nodes.

No of Nodes	FMN	RR	LASPA-FMN	LASPA-RR
10	0.812/0.0/21.0	1.494/0.0/32.0	0.0/0.0/4.0	0.003/0.0/6.0
30	0.439/0.0/51.0	0.815/0.0/96.0	0.0096/0.0/18.0	0.0096/0.0/16.0
50	0.284/0.0/93.0	0.463/0.0/92.0	0.0048/0.0/9.0	0.039/0.0/108.0
70	0.2616/0.0/87.0	0.42/0.0/117.0	0.0089/0.0/18.0	0.04/0.0/63.0
100	0.412/0.0/102.0	0.614/0.0/88.0	0.041/0.0/21.0	0.06/0.0/41.0
200	0.431/0.0/110.0	0.632/0.0/121.0	0.073/0.0/39.0	0.092/0.0/69.0

Table 4.5: Statistics reporting the *Number of Scanned Edges* with variation of the number of nodes.

No of Nodes	FMN	RR	LASPA-FMN	LASPA-RR
10	5.168/1.0/124.0	10.56/2.0/201.0	1.013/1.0/83.0	1.102/1.0/108.0
30	6.96/1.0/844.0	15.72/2.0/1788.0	1.234/1.0/268.0	1.186/1.0/286.0
50	7.51/1.0/2638.0	15.502/2.0/3003.0	1.2032/1.0/212.0	2.19/1.0/3394.0
70	9.25/1.0/3209.0	18.79/2.0/4125.0	1.826/1.0/299.0	2.131/1.0/937.0
100	11.130/1.0/4311.0	16.421/2.0/5008.0	2.576/1.0/434.0	2.99/1.0/1317.0
200	20.164/1.0/7591.0	23.201/2.0/9217.0	3.849/1.0/784.0	4.71/1.0/2014.0

Table 4.6: Statistics reporting the *Time Per Update* with variation of the number of nodes.

No of Nodes	FMN	RR	LASPA-FMN	LASPA-RR
10	1.754/0.0/610.0	2.558/0.0/160.0	1.122/0.0/60.0	0.699/0.0/80.0
30	2.46/0.0/380.0	3.77/0.0/330.0	1.428/0.0/380.0	1.95/0.0/170.0
50	3.314/0.0/550.0	5.518/0.0/610.0	2.59/0.0/550.0	2.54/0.0/550.0
70	3.924/0.0/2250.0	4.81/0.0/720.0	2.713/0.0/720.0	3.12/0.0/998.0
100	5.612/0.0/4310.0	7.331/0.0/4510.0	3.205/0.0/1210.0	4.1/0.0/1089.0
200	8.106/0.0/8110.0	9.238/0.0/6320.0	5.643/0.0/3090.0	5.13/0.0/427.0

The results of the experiment are listed in Tables 4.1 through 4.6. In the first set of experiments only the sparsity of the graphs was varied in steps, keeping other parameters fixed. These "other parameters" were the means and variances of edge costs, λ, the number of nodes in the graph, and the number of edge-update operations. Each entry in the tables represents the three quantities, namely, the average/minimum/maximum values of the results. Since simple arithmetic averages are shown, no accurate inference can be made about the performance of individual algorithms with increase in graph sparsity. The sensitivity of the performance of the individual algorithms to variation in graph sparsity is seen in Experiment 3. The tables report the simple statistics of the results obtained in two sets of experiments, where:

- The sparsity of graphs was varied, keeping other parameters constant. In our particular example, the mean edge costs of the graphs ranges between $1.0 - 5.0$, and the variance ranges between $0.5 - 0.9$, $\lambda = 0.95$, $N = 100$ and the number of operations = 5000.

- The number of nodes in a graph was varied, keeping other parameters constant. Our experiments were performed on random graphs with similar characteristics, but with a sparsity of 60%.

From Tables 4.1 through 4.3, where the average, minimum, and maximum values are taken together for one algorithm and compared against the others, it can be seen that both LASPA-FMN and LASPA-RR perform far better than both the FMN and RR algorithms across different graph structures with varying graph sparsities. For example, from Table 4.1, we see that the average value of time per update for LASPA-FMN and LASPA-RR at 10% sparsity are 0.03 and 0.08 respectively, whereas those of FMN and RR are 0.251 and 0.241 respectively. This shows that LASPA-FMN and LASPA-RR, on an average require less time per update operation than the FMN and RR algorithms. Similar results can be observed in the Tables 4.4 through 4.6, when the

number of nodes in the graph are varied. The results for a graph with a maximum of 200 nodes have been reported; it took several hours to generate a table entry with graphs of bigger sizes. Indeed, it was infeasible (due to the time taken) to report the details of all table entries for such large graphs.

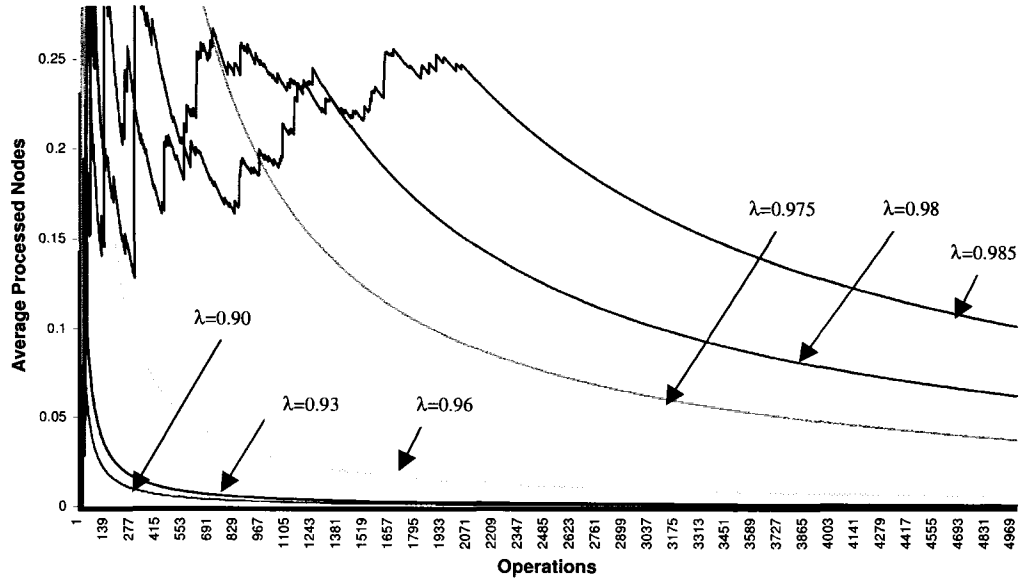


Figure 4.8: Sensitivity of the *Average Processed Nodes* in LASPA-FMN with the variation in the learning parameter.

4.3.4.3.3 Experiment Set 3

Figures 4.8 and 4.9 show the sensitivity of the performance of LASPA-FMN to variation in λ, the learning parameter. Experiments with LASPA-RR yield similar results and are omitted here. Figures 4.8 and 4.9 show that as the value of the learning parameter increases, the average number of nodes processed and the average time per update also increases. This is particularly true after LASPA has undergone processing for a while. Initially, however, the nature of the results is the opposite.

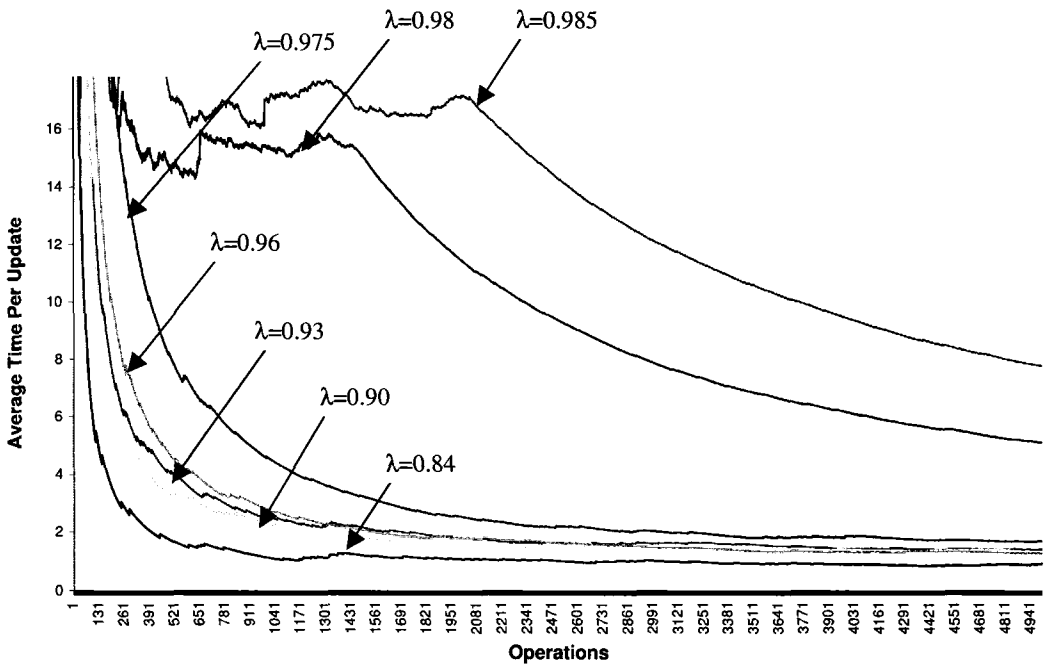


Figure 4.9: Sensitivity of *Average Time Per Update* in LASPA-FMN with the variation in the learning parameter.

From another experiment, we reported the sensitivity of the performance of LASPA to increase in graph sparsity. Figures 4.10 and 4.11 show the variation of average time per update and the average number of processed nodes with the variation in the graph sparsity. It was observed that as the sparsity of the graph increases, the average time to update and the average number of processed nodes decrease.

Similar results can be submitted to show the sensitivity of the other algorithms, like FMN and RR, to the other parameters described in the previous experiments.

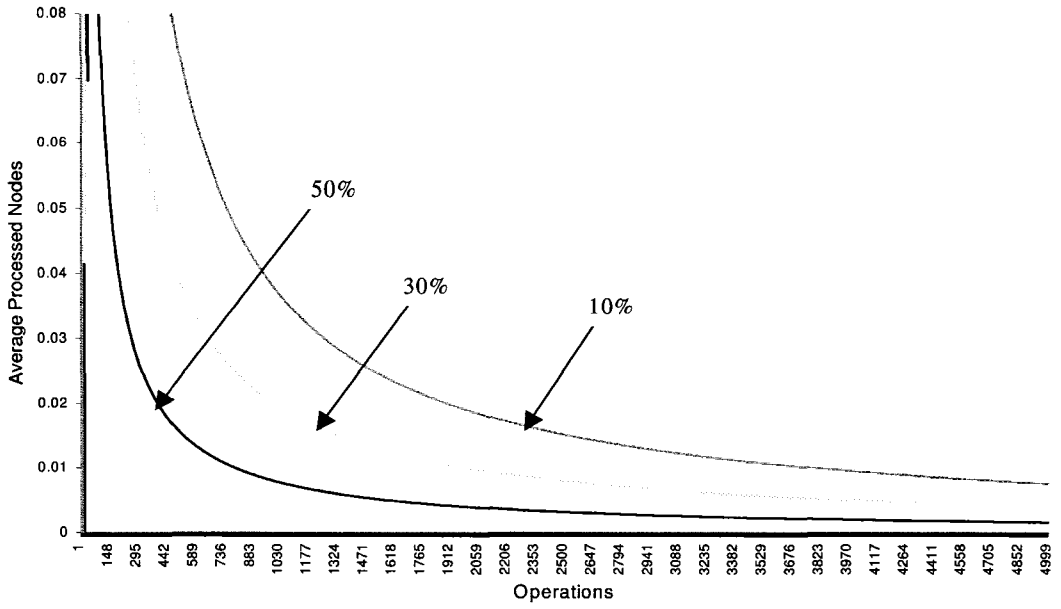


Figure 4.10: Sensitivity of *Average Number of Processed Nodes* of LASPA to the variation in the graph sparsity.

4.4 Alternative Solution: Pursuit Learning Approach

4.4.1 Description

In this Section we briefly present an alternative approach to solving the DSSSP Problem, namely, one which uses the principles of generalized pursuit learning discussed in Chapter 2. The intention of the work presented here was primarily to propose an alternative solution approach by encapsulating the problem within the context of pursuit learning. We propose an efficient algorithm, referred to as the *Generalized Pursuit Shortest Path Algorithm* (GPSPA), for maintaining shortest path routing trees in networks that undergo stochastic updates in their structure. Like the LASPA algorithm,

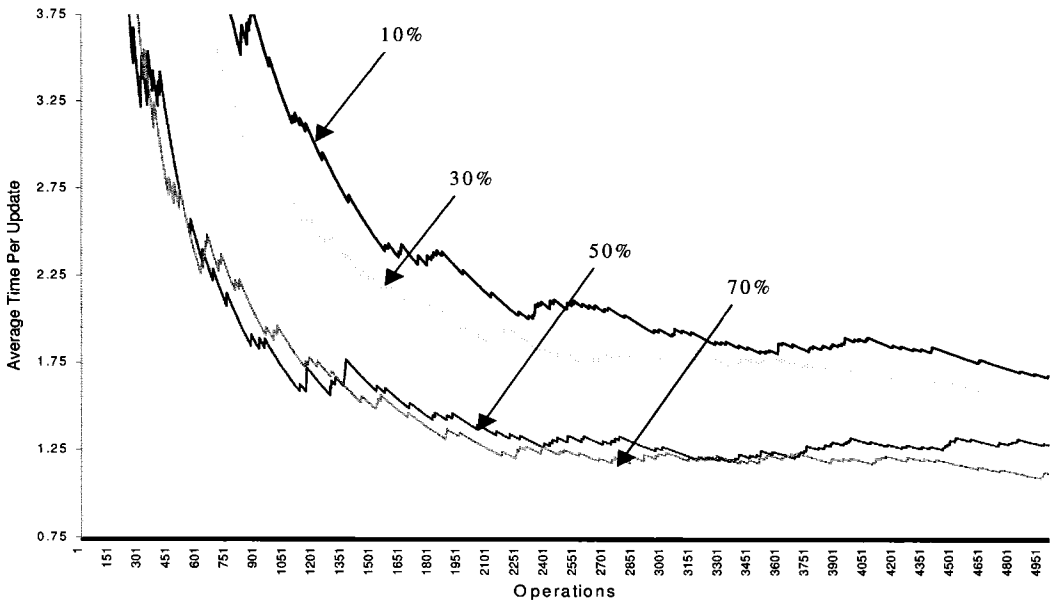


Figure 4.11: Sensitivity of *Average Time Per Update* of LASPA to variation in the graph sparsity.

it has the advantage that it can be used to find the shortest path within the "statistical" average network, which converges irrespective of whether there are new changes in link-costs or not. Although GPSPA is found to be more efficient than LASPA, the implementation of the latter is simpler than that of the former.

It is, indeed, worth re-emphasizing that the development of the two solution approaches has been very much evolutionary. Initially, we proposed to use the simple linear learning approach for solving the DSSSP problem. Finally, our quest for finding an estimator-based learning solution led to the proposition of the GPSPA algorithm.

4.4.2 The GPSPA Algorithm

Our solution model in GPSPA is similar to that used in LASPA, i.e., a team of learning automata interacting with a stochastic graph environment, using a reward/penalty

structure described in Section 4.3.2. The algorithm determines the underlying shortest path of the average link costs, which the system is unaware of.

The proposed algorithm, GPSPA, is similar in its operation to that of LASPA. However, there are a few basic *differences* between the two, which are mentioned below.

1. In addition to maintaining a probability vector, $P(t) = \{p_1(t), p_2(t), \ldots, p_r(t)\}$, for each node of the network, the GPSPA algorithm also maintains a reward-estimates vector $D(t)$.

2. After the shortest paths are calculated using Dijkstra's algorithm, or recalculated using the RR/FMN algorithm following each link-cost update, the algorithm updates the action probability vector, and the reward estimates vector of each node using the following schemes:

   (i) *Action probability vector*: Let i correspond to the outgoing link from a node which is determined to belong to the shortest path link, and j correspond to any other outgoing link from a node. Let $K(t)$ denote the number of actions (outgoing links) that have higher estimates than the chosen action (outgoing link) at the t^{th} iteration. Then the action probability updates follow:

$$\begin{aligned} p_j(t+1) &= p_j(t) - \lambda p_j(t) + \frac{\lambda}{K(t)}, \text{ if } \hat{d}_j(t) > \hat{d}_i(t) \\ p_j(t+1) &= p_j(t) - \lambda p_j(t), \text{ if } \hat{d}_j(t) \leq \hat{d}_i(t) \\ p_i(t+1) &= 1 - \sum_{j \neq i} p_j(t+1) \end{aligned}$$

   In other words, we increase the probability of choosing all the outgoing links with $p_j(t) < 1/K(t)$, whose reward estimates are higher than the reward estimate of the chosen action.

(ii) *Reward estimates vector*: The reward estimates vector for each node is updated according to the following equations for the chosen action:

$$\begin{aligned} W_i(t+1) &= W_i(t) + (1 - \beta(t)) \\ Z_i(t+1) &= Z_i(t) + 1 \\ d_i(t+1) &= W_i(t+1)/Z_i(t+1) \end{aligned}$$

In the above equations, $W_i(t)$ = number of times the i^{th} action has been rewarded up to iteration t, for $1 \leq i \leq r$ (total number of actions), $Z_i(t)$ = number of times the i^{th} action has been chosen up to iteration t, for $1 \leq i \leq r$, and $\beta(t) \in \{0, 1\}$ is the response from the Environment and denotes reward/penalty.

4.4.3 Experimental Results

The GPSPA algorithm was run through the same set of experiments that were used to evaluate the performance of LASPA, as explained in Section 4.3, and its performance was compared with respect to the same three metrics – the average number of processed nodes, the average number of scanned edges, and the average time per update. To avoid redundancy, we do not explain them here again, but rather merely present here some of the interesting results obtained for GPSPA. The results seem to consistent with those observed for LASPA, i.e., like LASPA, GPSPA also does not perform well at the beginning, namely, when the algorithm is learning. However, after the algorithm has learned, GPSPA outperforms the RR and FMN algorithms. The results are summarized below.

4.4.3.1 Experiment Set 1

In this set of experiments, we used a network topology with 20 nodes, and 50% sparsity. The link-costs were random real values having means between 1.0 and 5.0, and variances between 0.5 and 1.0. The value of the learning parameter, λ, was set to 0.995. RR, FMN, GPSPA-RR, and GPSPA-FMN were run on mixed sequences of 5000 link-cost update operations.

The plots of average processed nodes, average scanned links, and average time per link-update operation is shown in Figures 4.12, 4.13, and 4.14 respectively. The plots show a similar behavior to those of the LASPA. Initially, GPSPA performs worse than FMN/RR. After convergence, the average number of nodes processed, the average number of links scanned, and the average time spent per update operation are much less for GPSPA when compared to both FMN and RR. For example, in Figure 4.12, when the number of operations is 3000, the average number of processed nodes for RR is around 0.48, whereas the average number of processed nodes for GPSPA-RR is around 0.02.

Experiments were run on other network topologies as well, and they yielded similar results. GPSPA was found to be superior with respect to these three metrics in *all* the experiments conducted. The details are omitted in the interest of brevity.

4.4.3.2 Experiment Set 2

The results of the second set of experiments are listed in Tables 4.7 through 4.12. Specifically, Tables 4.7 through 4.9 report the simple statistics of the results obtained for two sets of experiments, where the sparsity of networks was varied, keeping other parameters constant, and Tables 4.10 through 4.12 report the results obtained when the number of nodes in a network was varied, keeping other parameters constant.

A comparison of these values against each other demonstrates that both GPSPA-

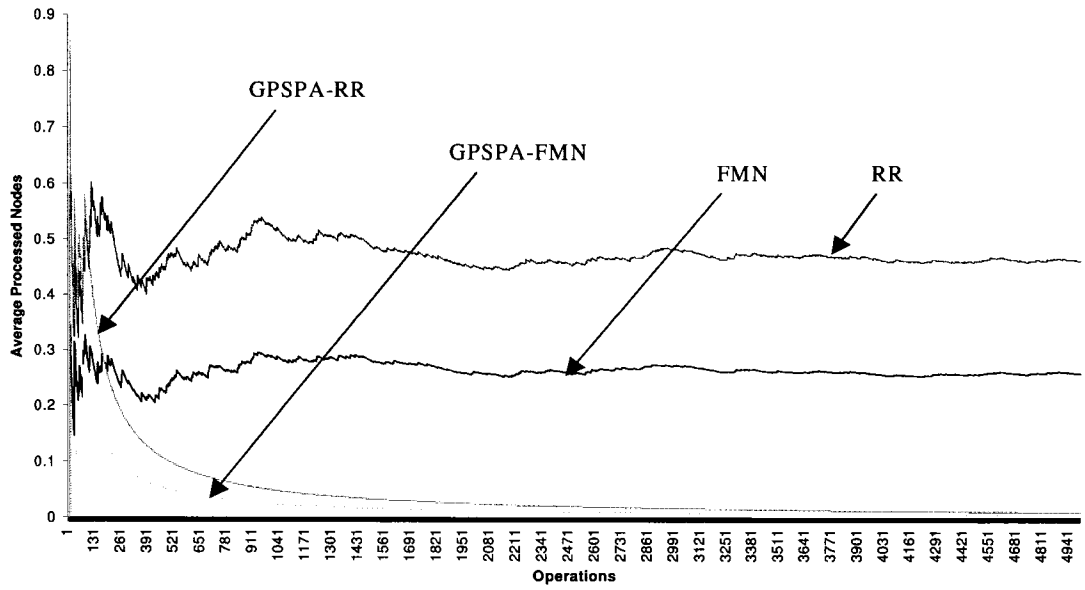


Figure 4.12: Graph showing the *average number of processed nodes* for the two algorithms FMN, RR and their GPL versions, GPSPA-FMN and GPSPA-RR.

FMN and GPSPA-RR perform considerably better than the FMN and RR algorithms. From Tables 4.7 through 4.9, it can be observed that both GPSPA-FMN and GPSPA-RR are significantly better than both the FMN and RR algorithms across different network topology structures with varying link sparsities. For example, from Table 4.7, we see that the average value of time per update for GPSPA-FMN and GPSPA-RR at 10% sparsity are 3.87 and 2.57 respectively, whereas those of FMN and RR are 11.93 and 8.0 respectively. Thus, like LASPA-FMN, and LASPA-RR, GPSPA-FMN and GPSPA-RR, on an average require less time per link-update operation than the FMN and RR algorithms. This is also true for the case when the number of nodes in the network is varied.

Table 4.7: Statistics reporting the *Number of Processed Nodes* with the variation of network link sparsity. The experiments were performed on random network topologies with mean link costs between 1.0-5.0, and with variance ranging between $0.5 - 1.5$. The other parameters were $\lambda = 0.95$, $N = 100$ and the number of operations = 300.

Sparsity	FMN	RR	GPSPA-FMN	GPSPA-RR
10%	0.08/0.0/9.0	0.04/0.0/3.0	0.0/0.0/0.0	0.0/0.0/0.0
30%	0.053/0.0/5.0	0.15/0.0/20.0	0.01/0.0/1.0	0.01/0.0/2.0
50%	0.2/0.0/27.0	0.06/0.0/12.0	0.04/0.0/3.0	0.02/0.0/6.0
70%	0.23/0.0/27.0	0.38/0.0/24.0	0.06/0.0/2.0	0.01/0.0/4.0
90%	0.54/0.0/24.0	1.22/0.0/64.0	0.09/0.0/5.0	0.123/0.0/24.0

Table 4.8: Statistics reporting the *Number of Scanned Links* with the variation of the network link sparsity. The experiments were performed on random topologies with mean link costs between $1.0 - 5.0$, and variance ranging between $0.5 - 1.5$. The other parameters were $\lambda = 0.95$, $N = 100$ and the number of operations = 300.

Sparsity	FMN	RR	GPSPA-FMN	GPSPA-RR
10%	7.19/1.0/710.0	5.93/2.0/279.0	1.0/1.0/1.0	1.03/1.0/2.0
30%	3.98/1.0/260.0	15.097/2.0/1714.0	1.06/1.0/2.0	1.05/1.0/2.0
50%	10.4/1.0/1283.0	5.35/2.0/604.1	1.14/1.0/2.0	1.06/1.0/2.0
70%	7.10/1.0/784.0	14.71/2.0/725.0	1.3/1.0/21.0	2.1/1.0/64.0
90%	5.82/1.0/237.0	16.21/2.0/784.0	3.117/1.0/84.0	3.67/1.0/455.0

Table 4.9: The *Time Per Update* tabulated against the sparsity of the networks The experiments were performed on random network topologies with mean link costs between $1.0 - 5.0$, and with variance ranging between $0.5 - 1.5$. The other parameters were $\lambda = 0.95$, $N = 100$ and the number of operations = 300.

Sparsity	FMN	RR	GPSPA-FMN	GPSPA-RR
10%	11.93/0.0/1540.0	8.0/0.0/270.0	3.87/0.0/390.0	2.57/0.0/110.0
30%	9.0/0.0/220.0	9.17/0.0/220.0	2.27/0.0/500.0	5.33/0.0/490.0
50%	8.73/0.0/170.0	9.1/0.0/220.0	3.83/0.0/380.0	2.33/0.0/160.0
70%	9.97/0.0/220.0	8.6/0.0/160.0	2.97/0.0/500.0	3.16/0.0/110.0
90%	6.97/0.0/220.0	9.73/0.0/710.0	3.93/0.0/610.0	2.23/0.0/60.0

Table 4.10: Statistics reporting the *Number of Processed Nodes* with the variation of the number of nodes. The experiments were performed on random network topologies with mean link costs between $1.0 - 5.0$, and with variance ranging between $0.5 - 1.5$. The other parameters were $\lambda = 0.95$, sparsity $= 50\%$, and the number of operations $= 300$.

No of Nodes	FMN	RR	GPSPA-FMN	GPSPA-RR
10	1.33/0.0/24.0	1.31/0.0/24.0	0.0/0.0/0.0	0.0/0.0/0.0
30	0.38/0.0/21.0	0.73/0.0/27.0	0.02/0.0/1.0	0.01/0.0/4.0
50	0.51/0.0/27.0	0.73/0.0/42.0	0.01/0.0/2.0	0.03/0.0/7.0
70	0.24/0.0/17.0	0.81/0.0/88.0	0.02/0.0/4.0	0.05/0.0/19.0
100	0.07/0.0/7.0	1.45/0.0/210.0	0.05/0.0/5.0	0.16/0.0/36.0

Table 4.11: Statistics reporting the *Number of Scanned Links* with the variation of the number of nodes. The experiments were performed on random network topologies with mean link costs between $1.0 - 5.0$, and with variance ranging between $0.5 - 1.5$. The other parameters were $\lambda = 0.95$, sparsity $= 50\%$, and the number of operations $= 300$.

No of Nodes	FMN	RR	GPSPA-FMN	GPSPA-RR
10	9.81/2.0/161.0	9.63/2.0/161.0	1.0/1.0/1.0	1.0/1.0/1.0
30	6.14/1.0/326.0	14.92/2.0/462.0	1.186/1.0/9.0	1.1/1.0/2.0
50	12.65/1.0/671.0	21.68/2.0/1075.0	1.27/1.0/11.0	1.16/1.0/9.0
70	7.53/1.0/432.0	33.8/2.0/3751.0	1.83/1.0/23.0	1.33/1.0/18.0
100	3.76/1.0/177.0	81.11/2.0/5051.0	2.67/1.0/69.0	1.93/1.0/55.0

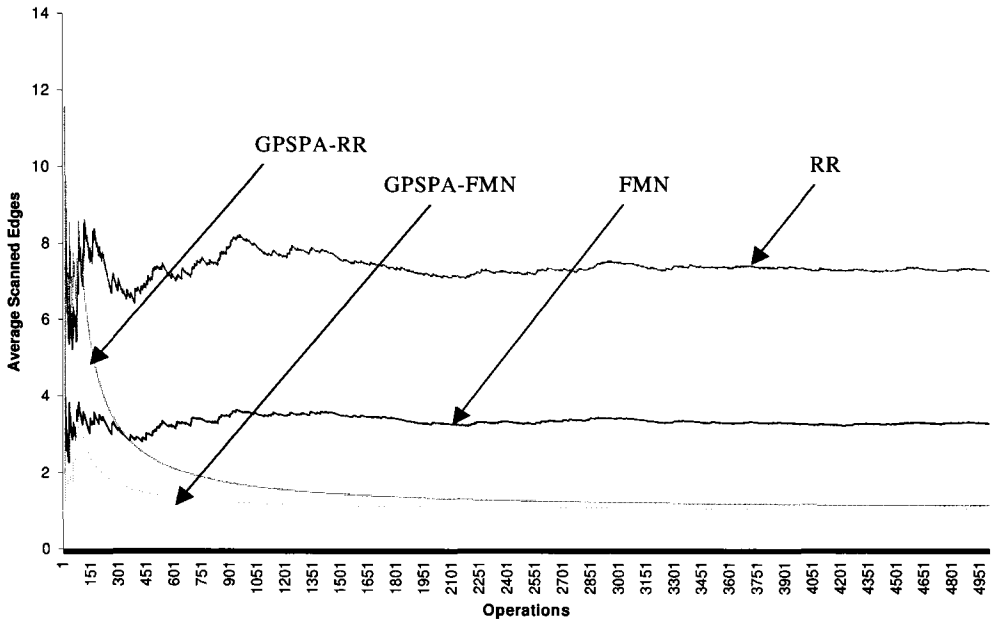


Figure 4.13: Graph showing the *average number of scanned links* for the two algorithms FMN, RR and their GPL versions, GPSPA-FMN and GPSPA-RR.

4.4.3.3 Experiment Set 3

Figure 4.15 shows the sensitivity of the performance of the metric, the time per update operation, of GPSPA-FMN to the variation in the learning parameter.

Results for the metrics, and with GPSPA-RR yield similar results, and are omitted here to avoid repetition. The observations are consistent with those observed for LASPA. From Figure 4.15, we observe that as the value of the learning parameter decreases, the average time per update also decreases. For example, at the 200^{th} operation, the average time per update is around 28ms when $\lambda = 0.97$, and around 8ms when $\lambda = 0.91$.

We had also conducted experiments to measure the sensitivity of the performance of GPSPA to the increase in network sparsity. In those cases too, the performance is

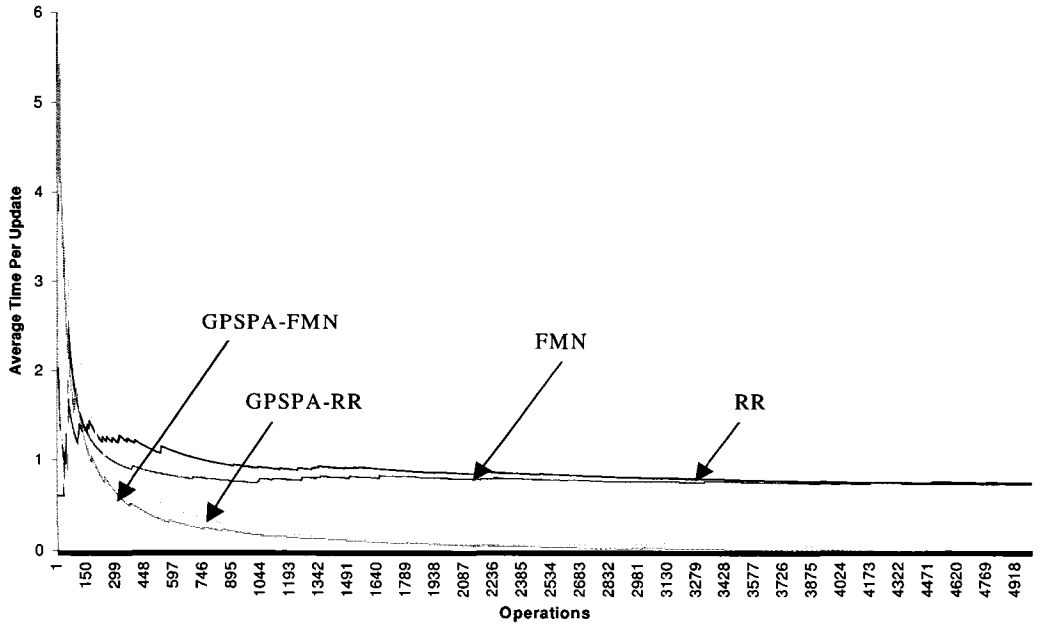


Figure 4.14: Graph showing the *average time per update* for the algorithms FMN, and RR, and their GPL versions, GPSPA-FMN and GPSPA-RR.

exactly as in the case of Figure 4.15. It can be observed that as the sparsity of the links in the network increases, the update time decreases.

In general, as expected, the Pursuit algorithms were better than the L_{RI}-based algorithms, although the improvement was not as marked as in the generic (non-application based) learning problem.

4.5 Conclusions

In this Chapter, we have reported two important solutions for the DSSSP problem, and highlighted their limitations for operating in dynamically changing stochastic networks. We then described our LA-based algorithms to solve this problem, and evaluated them experimentally using synthetic graphs.

Table 4.12: Statistics reporting the *Time Per Update* with the variation of the number of nodes. The experiments were performed on random network topologies with mean link costs between $1.0 - 5.0$, and with variance ranging between $0.5 - 1.5$. The other parameters were $\lambda = 0.95$, sparsity $= 50\%$, and the number of operations $= 300$.

No of Nodes	FMN	RR	GPSPA-FMN	GPSPA-RR
10	3.21/0.0/60.0	3.53/0.0/110.0	1.4/0.0/60.0	1.3/0.0/60.0
30	4.06/0.0/60.0	3.96/0.0/60.0	2.53/0.0/60.0	2.1/0.0/60.0
50	5.06/0.0/60.0	4.1/0.0/110.0	1.5/0.0/60.0	2.5/0.0/60.0
70	8.13/0.0/1050.0	5.3/0.0/440.0	1.8/0.0/60.0	3.4/0.0/110.0
100	11.73/0.0/1370.0	11.63/0.0/1270.0	1.97/0.0/270.0	5.7/0.0/330.0

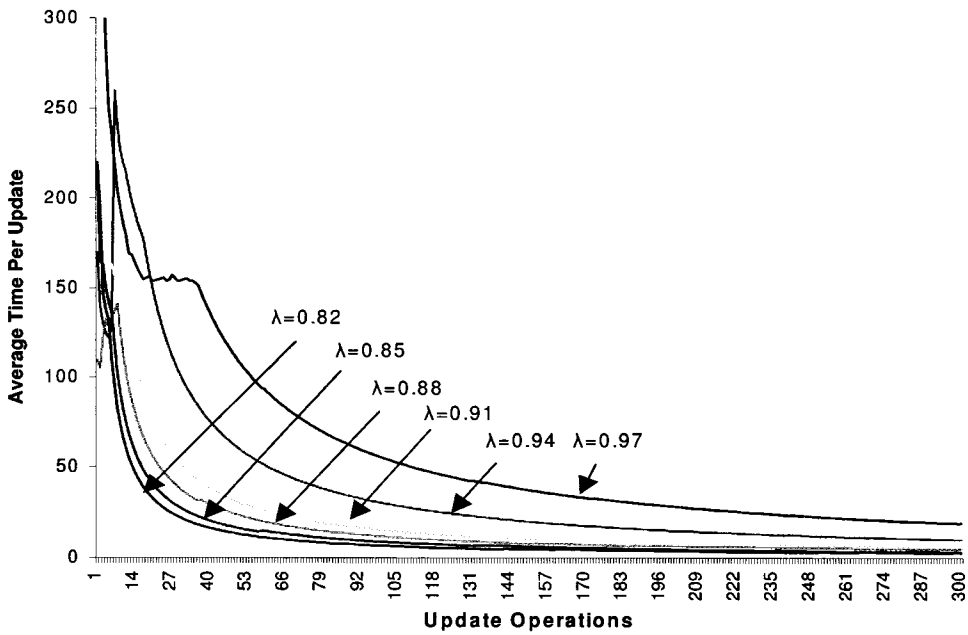


Figure 4.15: Sensitivity of *Average Time Per Update* of GPSPA-RR to variation in λ. The experiments were performed with $N = 100$, sparsity $= 50\%$, mean link costs between 1.0 and 5.0,variances between 0.5 and 1.5, and number of operations $= 300$.

To our knowledge, we have reported the first LA-based solutions to the DSSSP problem. They are also the first LA solutions to the general SSSP Problem. The aim of the problem was to determine the shortest path of the average underlying graph.

The proposed algorithms were designed, implemented, and rigorously experimentally compared to the two currently well-known fully-dynamic algorithms of Ramalingam and Reps [82], and Frigioni *et al.* [31]. The algorithms were tested in stochastic environments where edge-weights changed stochastically, and where graph topologies permitted multiple simultaneous edge-weight updates. The results uniformly show the superiority of the proposed algorithms. Once the algorithms had converged, the average number of processed nodes, scanned edges, and the time per update operation were always superior to the previous ones by, possibly, an order of magnitude. Similar results were also true for different graph topologies of varying sizes, and edge-weights.

Chapter 5

Dynamic All-Pairs Shortest Paths Routing

5.1 Introduction

In this Chapter[1], we report the results of our study of the *Dynamic All-Pairs Shortest Paths Problem* (DAPSP), i.e., the problem of computing and maintaining all-pairs shortest paths information in a graph where the edges are inserted/deleted and where edge-weights constantly increase/decrease [21, 82]. The DAPSP problem is equally important as the DSSSP problem discussed in Chapter 4. Both the problems aim to efficiently maintain shortest path solutions in environments that are representative of most practical situations in daily life, since most real-life environments are dynamically changing. While maintaining consistency with the problem model considered in Chapter 4, in this Chapter, we additionally assume that the changes in the graphs (environments) are limited to only edge-weight increase/decrease. Our aim is to main-

[1]A concise preliminary version of this work has appeared in the *Proceedings of the* 10^{th} *IEEE Symposium on Computers and Communications* (*IEEE ISCC 2005*), Cartagena, Spain, June 27-30, 2005. The extended version of the latter is being reviewed.

tain the shortest paths, without re-computing everything "from scratch" following each topology update.

Similar to what was shown in Chapter 4 for the DSSSP problem, for the DAPSP problem, as well, it can be shown that edge-weight updates can be treated as edge-deletions, and edge-insertions by setting the weights of edges to an infinitely large value [21]. Likewise, an edge-update algorithm for the all-pairs shortest path problem is referred to as being *fully dynamic* if both weight-increase and weight-decrease operations are supported on the edges of the graph. A *semi-dynamic* algorithm handles only weight-increases or weight-decreases, but not both at the same time [21].

Like the static solutions of the single source shortest path problem, the well-known *static* solutions for the all-pairs shortest path problem, e.g., the Floyd Warshall's algorithm [28], the all-pairs adaptations of Bellman-Ford's algorithm [10], or the Dijkstra's algorithm [19], which re-compute the shortest paths "from scratch" each time a topology change occurs in the graph, are certainly inefficient in such dynamic practical scenarios.

Over the last few decades, there has been a lot of research done to solve the DAPSP problem. The earliest papers are [48], [62], and [80]. Many other dynamic algorithms were proposed in the literature (e.g., [24], [82], [83], [81]); however, their worst-case running times were no better than re-computing the all-pairs shortest paths from scratch. Thereafter, a few solutions [3, 32, 35, 40] were proposed whose running times are better than a total re-computation. However, these solutions only work for integer weights. The algorithm proposed by Ausiello and Italiano [3] is applicable for the semi-dynamic case (decrease-only) only, and requires positive integer weights less than a constant "C". The algorithm's amortized running time per insertion operation is $O(Cnlogn)$ [3]. Although Henzinger *et al.* [35] provided a fully-dynamic solution to the all-pairs shortest paths problem, their solution is only for planar graphs with integral values of edge-weights. The running time of their algorithm is $O(n^{4/3}log(nC))$ per update

operation. The first fully-dynamic solution on general graphs was proposed by King [40]. Her solution too only works with positive integer weights less than C, and the running time of the algorithm is $O(n^{2.5}(Clogn)^{1/2})$. Later, Demetrescu and Italiano published two papers [20, 21] containing fully-dynamic algorithms that would perform edge-update operations on general graphs with real-valued edge-weights. If S represents the number of different real values, the amortized running time per update operation for their algorithm is $O(n^{2.5}(Slog^3n))$. Finally, in 2003, Demetrescu and Italiano [21] proposed a remarkable algorithm that solved the same problem for general digraphs with edge-weights that can assume positive real-values but with substantially improved running time per edge-update operation. This algorithm is discussed further in Section 5.2.

The currently acclaimed fully dynamic algorithms (mentioned above) for the DAPSP problem are constrained by the same limitations as those mentioned for the DSSSP problem in Chapter 4, Section 4.1. We address these problems later in the Chapter, and try to design efficient solutions for solving the DAPSP problem.

5.2 Demetrescu and Italiano's Algorithm (DI)

Demetrescu and Italiano [21] devised a *fully dynamic* algorithm for maintaining all-pairs shortest paths in general directed graphs with real-valued non-negative edge-weights. Their solution supports any sequence of edge update operations in $\bar{O}(n^2)$, where $\bar{O}(f(n))$ denotes $O(f(n)\text{polylog}(n))$ is the amortized time per update operation, and n is the number of nodes in the graph. Their solution is the best in terms of the theoretical complexity, when compared to all the previous all-pairs shortest path algorithms. Additionally, their solution works for the general problem, where the edge-weights are non-negative real numbers, and the number of different values an edge-weight can assume is not restricted. Finally, they proposed only one simple algorithm,

update, for supporting both types of edge-update operations [21]. This renders their problem to be quite fascinating.

5.2.1 Notations

Before actually presenting the DI algorithm, we present the notations [21] used in it.

- $G = (V, E, w)$ is a directed graph with edge-weights that are non-negative, and where each weight is a real number.
- w_{uv} is the weight of the edge $(u, v) \in E$.
- π_{xy} is a path from any node x to a node y.
- $w(\pi_{xy})$ is the sum of all the edges in the path π_{xy}.
- $l(\pi_{xy})$ is a path π_{xb}, such that π_{xy} is the concatenation of path π_{xb} and edge (b, y).
- $r(\pi_{xy})$ is a path π_{ay}, such that π_{xy} is the concatenation of edge (x, a), and the path π_{ay}.

5.2.2 Definitions

To understand the functioning of DI algorithm, we present the following definitions, taken from [21].

Definition 5.1: If every proper sub-path of π_{xy} is a shortest path in G, the path π_{xy} is said to be *uniform* in G.

Definition 5.2: If a path π_{xy} is not a shortest path in G, but it used to be a shortest path at some point earlier in time, and none of its edges has been updated since that time, the path π_{xy} is called a *zombie* in G.

Definition 5.3: If every proper sub-path of π_{xy} is a historical shortest path at that time, the path π_{xy} is said to be *potentially uniform* in G.

Thus, from the above definitions, it can be concluded that the *uniformity* of a path π_{xy} can be determined by checking whether $l(\pi_{xy})$ and $r(\pi_{xy})$ are uniform in G. The *potential uniformity* of a path can be determined by checking whether $l(\pi_{xy})$ and $r(\pi_{xy})$ are historical shortest paths in G. Shortest paths are potentially uniform [21].

5.2.3 Data Structures

Every pair of nodes in the graph stores four data structures: the weight $w_{x,y}$; the time of the latest update t_{xy}; a priority queue P_{xy}, such that each item $\pi_{xy} \in P_{xy}$ has priority $w(\pi_{xy})$; and a set $P^{\star}{}_{xy}$ containing elements π_{xy} such that π_{xy} is a historical shortest path in G [21].

Similarly, for each path π_{xy} in P_{xy} and $P^{\star}{}_{xy}$, the algorithm stores the following (adopted from [21]):

- the weight $w(\pi_{xy})$
- $L(\pi_{xy}) = \{\pi_{x'y} = (x', x).\pi_{xy} : \pi_{x'y}$ is potentially uniform in $G\}$
- $L^{\star}(\pi_{x'y}) = \{\pi_{xy} = (x', x).\pi_{xy} : \pi_{x'y}$ is a historical shortest path in $G\}$
- $R(\pi_{xy}) = \{\pi_{xy'} = \pi_{xy}.(y, y') : \pi_{xy'}$ is potentially uniform in $G\}$
- $R^{\star}(\pi_{xy}) = \{\pi_{xy'} = \pi_{xy}.(y, y') : \pi_{xy'}$ is a historical shortest path in $G\}$

The principal idea behind the DI solution is to dynamically maintain, in a suitable manner, the potential uniform paths of a graph in a data structure. To perform an update operation the algorithm, essentially, removes all those paths from the data

structure that contains the updated edge. Thereafter the algorithm performs a modified version of Dijkstra's algorithm to be used with the all-pairs shortest path problem. This is done by extracting at each step a shortest path that was stored in a priority queue, and combining it with existing shortest paths and zombie paths to form new potentially uniform paths [21].

5.2.4 Algorithm DI

The DI algorithm consists of the following set of procedures presented below. They are *update*, *unsmoothed-update*, *cleanup*, and *fixup* (which are adopted from [21]).

Algorithm: update(u,v,w)

Input
(i) G: A directed graph
(ii) (u, v): The edge whose weight is to be updated
(iii) w: The weight of edge (u, v)

Output
(i) Shortest path sub-graph after the update

BEGIN
time $\leftarrow$ time + 1
$t_{uv} \leftarrow$ time
unsmoothed-update(u, v, w)
for *each* $(x, y) \in E$: *time* $t_{xy} = 2^{\lceil log_2(time - t_{xy}) \rceil}$ **do**
unsmoothed-update(x, y, w_{xy})
end
END

Algorithm: unsmoothed-update(u, v, w)

Input
(i) (u, v): The edge whose weight is to be updated
(ii) w: The weight of (u, v)

Output
Updated data structures for both the original and the dummy updates.

BEGIN
cleanup(u, v)
fixup(u, v, w)
END

Algorithm: cleanup(u, v)

Input
(u, v): The edge whose weight is to be updated

Output
Path π_{xy} containing edge (u, v) from the set of potentially uniform paths in G is removed.

BEGIN
if $(u, v) \in P_{uv}$ **then**
$Q \leftarrow (u, v)$
while $Q \neq \phi$ **do**
extract any π_{xy} from Q
remove π_{xy} from P_{xy}, $L(r(\pi_{xy}))$, and $R(l(\pi_{xy}))$
if $\pi_{xy} \in P^{\star}{}_{xy}$ **then**
remove π_{xy} from $P^{\star}{}_{xy}$, $L^{\star}(r(\pi_{xy}))$ and $R^{\star}(l(\pi_{xy}))$
end
add paths in $L(\pi_{xy})$, and paths in $R(\pi_{xy})$ to Q
end
end
END

Algorithm: fixup(u, v, w)

Input

(i) (u, v): The edge whose weight is to be updated

(ii) w: The weight of (u, v)

Output

(i) The weight of edge (u, v) is updated to the new value w.

(ii) A priority queue is maintained with the minimum weight paths for each pair of vertices in the graph.

BEGIN

$w_{uv} \leftarrow w$ // Phase 1

if $w < +\infty$ **then**

$w((u,v)) \leftarrow w$; $l((u,v))\pi_{uu}; r((u,v)) \leftarrow \pi_{vv}$

add(u, v) to P_{uv}, $L(\pi_{vv})$, and $R(\pi_{uu})$

end

$H \leftarrow \phi$ // Phase 2

for *each* (x, y) **do**

add $\pi_{xy} \in P_{xy}$ with minimum w_{xy} to H

end

// Phase 3

while $H \neq \phi$ **do**

extract π_{xy} from H with minimum $w(\pi_{xy})$

if π_{xy} *is the first extracted path for pair* (x, y) **then**

if $\pi_{xy} \notin P^{\star}{}_{xy}$ **then**

add π_{xy} to $P^{\star}{}_{xy}, L^{\star}(r(\pi_{xy}))$, and $R^{\star}(l(\pi_{xy}))$

for *each* $\pi_{x'b} \in L^{\star}(l(\pi_{xy}))$ **do**

$\pi_{x'y} \leftarrow (x', x).\pi_{xy}$

$w(\pi_{x'y}) \leftarrow w_{x'x} + d_{xy}$

$l(\pi_{x'y}) \leftarrow \pi_{x'b}$; $r(\pi_{x'y}) \leftarrow \pi_{xy}$

add$\pi_{x'y}$ to $P_{x'y}, L(\pi_{xy}), R(\pi_{x'b})$, and H

end

for *each* $\pi_{ay'} \in R^{\star}(r(\pi_{xy}))$ **do**

$\pi_{xy'} \leftarrow \pi_{xy}.(y, y')$

$w(\pi_{xy'}) \leftarrow d_{xy} + w_{yy'}$

$l(\pi_{xy'}) \leftarrow \pi_{xy}$; $r(\pi_{xy'}) \leftarrow \pi_{ay'}$

add $\pi_{xy'}$ to $P_{xy}, L(\pi_{ay'}), R(\pi_{xy})$, and H

end

end

end

end

END

5.3 Proposed Solution: Linear Learning Approach

5.3.1 Description

In this Section we present a new solution to the DAPSP Problem using a linear reinforcement-learning scheme. This algorithm, similar to its linear learning-based DSSSP counterpart, can be used to find the all-pairs shortest paths for the "statistical" average graph, and the solution converges irrespective of whether there are new changes in edge-weights or not.

There are two important *contributions* of the proposed algorithm. The first contribution is that not all the edges in a stochastic graph are probed, and even if they are, they are not all probed equally often. Indeed, the algorithm attempts to almost always probe only those edges that will be included in the final list involving all pairs of nodes in the graph, while probing the other edges minimally. This increases the performance of the proposed algorithm. The second contribution is the designing of a data-structure, the elements of which represent the probability that a particular edge in the graph lies in the shortest path between a pair of nodes in the graph. All the algorithms were tested in environments where edge-weights change stochastically, and where the graph topologies underwent multiple simultaneous edge-weight updates. The superiority of the proposed algorithm in terms of the average number of processed nodes, scanned edges and the time per update operation, when compared with the existing algorithms, was experimentally established.

This new algorithm is called the *All-Pairs Shortest Paths using Learning Automata*, or *APLA*. The rationale for why our algorithm works would (probably) rely on the ϵ-optimal property of the L_{RI} scheme, the shortest-path property of Floyd Warshall's

algorithm [28], and the update properties of the DI scheme [21], and is still theoretically open. We empirically tested the APLA algorithm in different sets of experiments, to prove that the proposed scheme performs better than the DI algorithm.

In Chapter 4, we had proposed an LA-based solution to the DSSSP problem. However, the solution model we used for solving the single-source problem could not be used to solve the all-pairs problem considered here. The reason is that in the single-source problem we could maintain a *single* shortest path tree corresponding to the complete graph – that renders updating the action probability vectors much simpler. In the all-pairs problem, such a situation does not arise because an edge that could lie in the shortest path between a pair of nodes may not necessarily lie in the shortest path between another pair of nodes. This renders the solution model used in Chapter 4 inadequate, and inapplicable for solving the all-pairs problem. To overcome this problem, for the all-pairs LA-solution, in this Chapter we propose a *novel* methodology of maintaining action probabilities for the whole graph.

5.3.2 The Benchmark Algorithm

The Benchmark algorithm that we used was the one described in the last Section, namely the one due to Demetrescu and Italiano [21]. We chose their algorithm as a benchmark for the following reasons:

1. It represents the state-of-the-art.

2. It is a fully-dynamic algorithm for the all-pairs shortest paths on general graphs with non-negative real-valued edge-weights that is provably faster than computing the all-pairs shortest paths from scratch, following each update in the topology of the graph.

3. Unlike most algorithms proposed after several years of efforts on dynamic all-pairs shortest path algorithms, this algorithm does not restrict the number of

different values the edge-weights may assume.

4. This algorithm has significantly improved theoretical bounds than the previous algorithms.

5. This algorithm supports both weight-increase and weight-decrease of edges using the same code.

The details of all the above five characteristics of this algorithm are listed in [21].

5.3.3 Motivation for APLA

To achieve our solution, as in any LA-based solution, we have to adequately model the three principal components of the LA system namely, the *Automaton*, the *Environment*, and the *Reward-Penalty* structure. In this context, we mention that in our case, the "system" would imply a *team* of LA interacting with the stochastic graph and playing a *cooperative* game. Incidentally, the Automaton-Environment-Reward/Penalty model we designed for APLA is similar to that designed for the LASPA, and GPSPA algorithms discussed in Chapter 4, with a few differences mentioned below:

- In APLA, after the Automaton chooses an action from its prescribed set of actions, and requests the Environment for the *current* random edge-weight for the edge it has chosen, the system computes the current shortest paths by invoking the DI algorithm. The LA determines whether the choice it made should be rewarded or penalized.

- In APLA, the Reward/Penalty signal to the LA fed back by the Environment is *inferred* by the system, after it has invoked the DI algorithm.

5.3.4 The Linear Learning Solution to DAPSP

For modeling our solution, let us reconsider the abstraction of the graph environment we had in Chapter 4, where we assumed that $G(V, E)$ denotes a dynamically changing directed graph with a set of V nodes and E edges. There are multiple random edge-weight changes occurring in the graph based on a certain probabilistic distribution. Our aim is to determine the underlying *all-pairs* shortest paths of the average weights. However, as before, the system is not aware of the underlying distributions. In the problem, (u, v) denotes an ordered pair of nodes representing a directed edge from u to v with a positive real-valued weight, $w(u, v)$.

The algorithm computes the shortest paths between every possible pair of nodes in the average graph, i.e., from every node s to every other node $v \in V$. The problem is to find an *efficient* algorithm for maintaining shortest paths between all pairs of nodes in the graph.

5.3.5 Data Structure for Maintaining Action Probabilities

One way to approach the above problem could be to reuse the LASPA algorithm [54, 58] for the DSSSP problem that was proposed in Chapter 4. The way the LASPA algorithm works is by maintaining an action probability vector corresponding to every automaton that is stationed at every node in the graph, and then by using the collective responses between the team of interacting automata to maintain a shortest path tree in the "average" graph that will be stable regardless of the continuous randomly changing weights. If we were to use such a strategy for solving the DAPSP problem, we would have to determine a methodology by which we could maintain V shortest path trees corresponding to the V nodes in the graph. However, such a solution would be obviously cumbersome and inefficient.

To address the above problem we propose to have a data structure for maintaining

the action probabilities corresponding to *all* feasible pairs of nodes in the graph. The methodology used to implement the data structure is unimportant to us, which can be a two-dimensional array, a linked-list or any other suitable structure. What is rather important is to *visualize* the data structure as a two-dimensional vector (matrix) (see Figure 5.2, where the structure is shown in the context of an example), whose columns represent all possible source-destination pairs in a particular graph, while the rows represent all possible outgoing edges from each of the nodes in the graph. In such a representation the infeasible pairs of nodes or pairs between the same nodes are pre-processed, and are omitted from further consideration in the vector. At a particular time instant, each of the elements (entries) of the vector represents a probability value corresponding to having an outgoing edge from a particular node (listed as rows in the vector) in the source-destination pair (listed in the column). The usefulness of this data structure will be further clarified below using an example.

5.3.6 The APLA Algorithm

We now describe the proposed linear learning solution to DAPSP, namely, the APLA algorithm. The APLA algorithm is designed for considering only the weight-decrease / increase operations on a graph. The APLA algorithm is designed by taking into consideration the three efficiency criteria, namely, the average number of nodes processed per operation, the average number of edges scanned per operation, and the average time per operation. In other words, the algorithm is designed in such a way that it performs better than the existing algorithms with respect to these criteria.

The variant of the APLA algorithm that we consider in this work is when APLA uses the DI algorithm [21] as a kernel whenever an edge-weight increase/decrease occurs. Informally, the scheme is as follows:

1. Obtain a snapshot of the given graph with each edge having a random cost. This

cost is based on the random call for an edge, where each cost has its own mean and variance. The algorithm maintains an action probability vector, $P(t) = \{p_1(t), p_2(t), \ldots, p_r(t)\}$, for each node of the graph.

2. Run Floyd Warshall's all-pairs static algorithm once to determine the shortest path edges on the graph's snapshot obtained in the first step.

3. Update the action probability vector corresponding to all the nodes such that the outgoing edge from a node that is determined by Step 2 to belong to the shortest path edge between a pair of nodes from that node, has an increased probability than before the update.

4. Randomly choose a node from the current graph. For that node, choose an edge based on the action probability vector. Request the cost of this edge and recalculate the shortest paths using the DI algorithm.

5. Update the action probability vectors such that the edges that belong to the shortest paths between a source-destination pair, have a greater likelihood of being selected, and thus have an increased probability than before the update.

6. Repeat Steps 3-5 above until the algorithm has converged.

We provide below the "bird's eye view" of the pseudo-code of the APLA algorithm.

Algorithm: APLA

Input
(i) $G(V, E) = A$ dynamically changing graph with simultaneous multiple stochastic edge updates occurring on it.
(ii) iters is the total number of iterations.
(iii) λ is the learning parameter.

Output
(i) A converged graph that has all the shortest path information.
(ii) Updated action probability vector.

BEGIN
G' = obtainAGraphInstance(G) // Obtain a snapshot of the graph G
InitializeActionProbability() // Initialize the action probability vector
executeFloydWarshall() // Execute Floyd Warshall's algorithm
updateActionProbabilityFW() // Update the action probabilities after running
// Floyd Warshall's algorithm
// Execute for all iterations.
for $i = 0$ *to iters* **do**
randomVertex = getRandomVertex() // Randomly choose a vertex in the
// current graph.
e = chooseAnEdge(ap) // Choose an edge based on the action probability
// vector for v
currentWeight = getRandom(e) // Obtain a random value of the edge e
// from the environment
oldWeight = getExistingWeight(e) // Obtain the existing weight
if *oldWeight* ! = *currentWeight* **then**
executeUpdate(e) // Execute the Update algorithm of DI, and compute
// shortest paths.
updateActionProbabilityDI() // Update the action probabilities after
// running DI algorithm
end
end
END

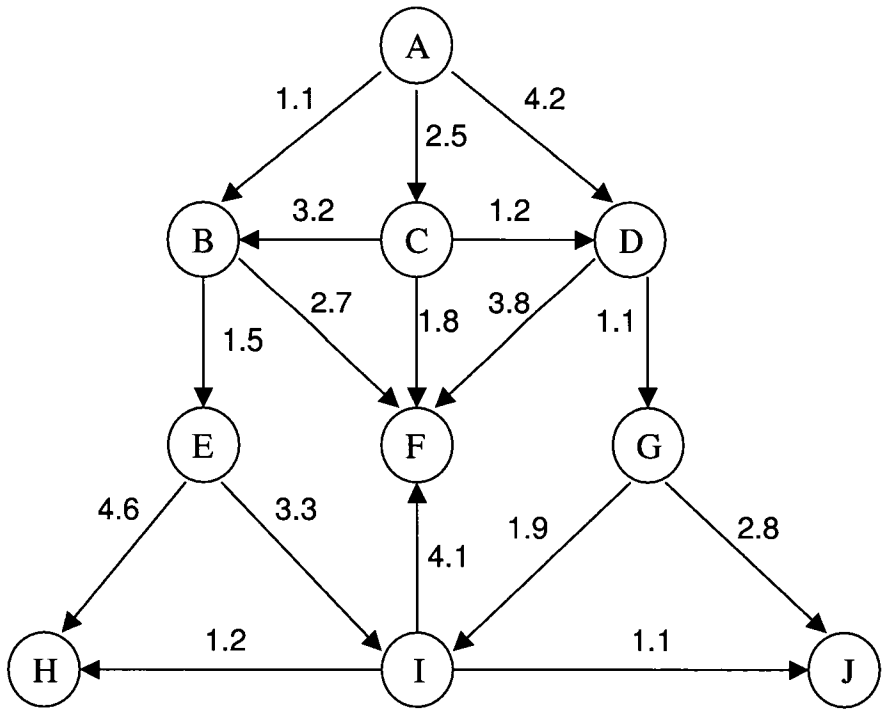


Figure 5.1: A hypothetical graph with 10 nodes. The weights displayed are the average weights, unknown to the algorithm.

5.3.6.1 Example

Let us consider the graph shown in Figure 5.1. In this graph, the weights of the edges are changing randomly in a continuous manner at every time instant. We want to maintain the all-pairs shortest paths in the "average graph" (since the weights of the edges are not constant) using the APLA algorithm. In other words, we want to maintain the shortest paths from A to the rest of the nodes in the graph, then from B to the rest of the nodes in the graph, and so on, for all the nodes in the graph.

We maintain a probability vector as shown in Figure 5.2, whose columns represent the source-destination pairs, whereas the rows represent the outgoing edges from each of the nodes. As stated earlier, at a particular time instant, each of the elements of the

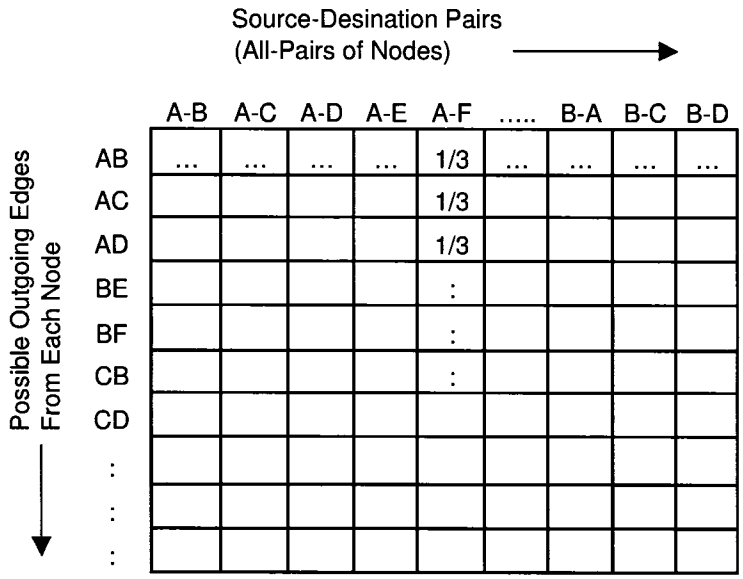


Figure 5.2: The data structure containing the set of action probability vectors for APLA. Each of the entries in the vector represents the probability that a particular edge (row indices) lies in the shortest path between a particular source-destination pair (column indices) in the graph.

vector represents a probability value corresponding to having an outgoing edge from a particular node (listed as rows in the vector) in the source-destination pair (listed in the column). For example, when we consider the node A, and the source-destination pair $A - F$, we want to maintain the probability that each of the outgoing edges from A, i.e., AB, AC, and AD lies in the shortest path between the nodes A and F.

Initialization: Let us consider the source-destination pair $A - F$. Initially, from A, any of the outgoing edges AB, AC, or AD has an equally likely probability of being in the shortest path between A and F. So, probability of choosing AB, AC, or AD is initialized to 1/3, 1/3, 1/3 respectively. Similarly, all the other initial entries (probability values) of the probability vector are computed in the above manner.

Applying Floyd-Warshall's all-pairs shortest path algorithm: Next, we take a snapshot of the graph containing different random weights. We compute the shortest paths between all pairs of nodes in the graph. Then we update the above probability vector (with appropriate probability values) using the L_{RI} scheme. According to the scheme, the probability values of the edges that lie in the shortest path corresponding to the source-destination pair in consideration are increased (rewarded), whereas the probability values of the other edges are decreased linearly. So in our example, the probability value corresponding to the edge AB (in the path $A - F$) is increased (because edge AB lies in the shortest path between A and F) from 0.33 to, say, 0.40, whereas the probability values corresponding to the other edges are decreased to, say, 0.3 and 0.3 respectively.

Applying DI all-pairs shortest path algorithm: In this case, the exact same procedure stated above for applying Floyd-Warshall's algorithm is applied here at each successive iteration. The probability values are updated after applying the DI algorithm, depending on which edges lie in the shortest paths between a source-destination pair.

5.3.7 Justification: Why the Algorithm Works

The formal reasoning of why the above algorithm works follows the pattern of arguments similar to those provided for LASPA in Section 4.3.3. Therefore, instead of describing them again, we briefly mention below some of the properties of APLA that are necessary for understanding the theoretical platform on which the algorithm is based. Like LASPA, the theoretical underpinning of APLA is dependent on the ϵ-optimal property of the L_{RI} scheme, the shortest-path property of Floyd Warshall's

algorithm, and the update properties of DI.

1. *Correctness*: In the APLA algorithm, the shortest paths are calculated when executing Floyd Warshall's algorithm on the graph's snapshot taken at the initialization step, and when executing the edge-weight update algorithm of DI. Floyd Warshall's algorithm is well known to find the optimal all-pairs shortest paths on a given graph topology. Since it is initially executed on the static graph snapshot, it necessarily calculates the *initial* optimal all-pairs shortest paths. Furthermore, this means that, if the edge-weight update operates correctly, the updated all-pairs shortest paths obtained after invoking the DI scheme, will also be accurate.

2. ϵ-*optimality*: The proof that the all-pairs shortest paths are found with a probability as close to unity as desired, *probably* follows because of the fact that the action probability updating scheme, the L_{RI}, is ϵ-optimal (this result would be certainly true if the choices of, and responses to the individual automata are independent). Thus, the probability that a particular edge lies in the shortest path between a particular source-destination pair in a graph, will be arbitrarily close to unity. Therefore, as the number of iterations tends towards infinity, the probability of choosing the optimal action for each pair of nodes tends towards unity, and the overall path costs will tend to the shortest path costs of the mean edge-weights – which has also been confirmed experimentally.

3. *Computational complexity*: The DI algorithm continues to recalculate all the affected shortest paths for *every change* in the edge-weights for the entire duration of execution of the algorithm. However, in such an environment, after convergence, the APLA algorithm will already have attained to the list of optimal all-pairs shortest paths for the graph, and therefore, will do nothing in terms of re-computing the all-pairs shortest paths for every change in edge-weight. Hence,

after convergence, the performance of APLA in terms of the *average number of processed nodes*, the *average number of scanned edges*, and the *average time per update* operation will be superior to the DI algorithm.

5.3.8 Experimental Details

5.3.8.1 Experimental Design

The experimental design made to assess the performance of APLA is similar to the ones used to evaluate the performance of LASPA, and GPSPA in Chapter 4. As in the case of the DSSSP algorithms, three sets of experiments were conducted for evaluating the performance of APLA as well. The purpose of the first set of experiments was to compare the performance of APLA with that of DI for a graph with a fixed number of nodes and sparsity of edges. The purpose of the second set of experiments was to compare the performance results of APLA and DI with the variation in the number of nodes and sparsity of edges in the graph, while the purpose of the third set of experiments was to analyze the sensitivity of the performance of APLA to the variation of certain parameters, while keeping other parameters constant.

The random graph topology generator and the sequence generator used were the same as those described in Chapter 4, Section 4.3.3.1.

As before, the algorithms in this Chapter were also tested both on manually entered, and randomly generated graphs. However, for the sake of permitting a greater flexibility in changing the experimental variables, the results presented here were obtained using random synthetically generated graphs only.

All the algorithms were tested with graphs of similar nature, i.e., graphs with directed edges, positive real-valued edge-weights that are normally distributed, and with stochastic edge-updates. Only the two edge operations, namely, edge-weight increase/decrease were considered.

The metrics that were used for evaluating the performance of the algorithms were the *number of scanned edges*, *number of processed nodes*, and the *time required per update operation*, as described in Chapter 4, Section 4.3.4.2. The set of nine alternative metrics that were shown in Section 4.3.4.2 are also valid here, but as in the case of the DSSSP problem, we opted to use the three above-mentioned metrics, as we considered them to be more important.

5.3.8.2 Experimental Results

In this Section we report the results of the experiments conducted to examine the performance of APLA.

Our observations regarding the general nature of performance shown by APLA are consistent with those obtained for the case of the LA-based solutions to DSSSP described in Chapter 4. For instance, from the several experiments that were performed using the different randomly generated graph topologies, and randomly generated edge-update sequences, we observe that APLA does not perform well at the beginning, i.e., when the algorithm is learning, but after the algorithm has learned, APLA outperforms the DI algorithm. The results are summarized below.

5.3.8.2.1 Experiment Set 1

The purpose of the first set of experiments was to compare the performance of the APLA, and the DI algorithms by running them on mixed sequences of 1000 modifying edge-update operations performed on a graph topology with 20 nodes, and 50% sparsity. In the graph topologies, the edge-weights chosen were random real values between 3.0 and 8.0. While running the APLA algorithm, the value of the learning parameter, λ, was set to a fixed value of 0.90.

Figures 5.3 through 5.5 show the summary of the results obtained from the experiments. The experiments reveal that APLA improves over the performance of DI

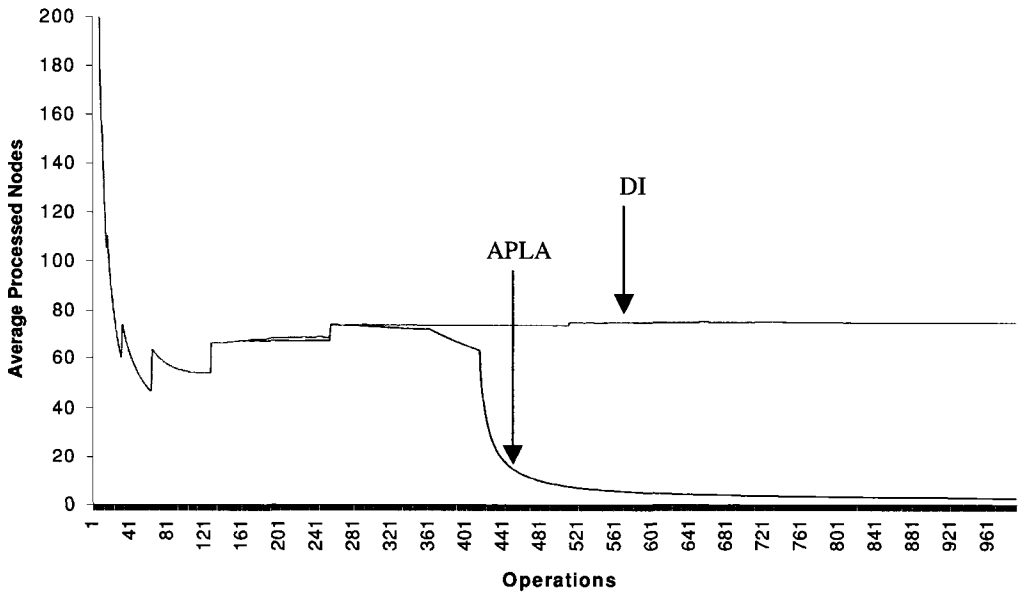


Figure 5.3: Average Processed Nodes

by about $40 - 95\%$. The general observation of comparison between the performance shown by the APLA algorithm, and the LASPA algorithm in Chapter 4 reveals that both the LA-based dynamic shortest path algorithms show worse performance compared to their benchmark algorithms at first, and after the algorithms have learned, the performance of these algorithms improve significantly. As mentioned in Chapter 4, this is not surprising if we consider the fact that initially the APLA algorithm is unaware of the stochastic behavior of the Environment, and is in the process of learning. But after convergence, these algorithms utilize their knowledge about the environment, and their performances exceed those of their non-learning counterpart algorithms.

As seen in the case of LASPA, here also we see that after convergence, the average number of nodes processed, the average number of edges scanned, and the average time spent per update operation of APLA are much less than those of the DI. For example, as seen from Figure 5.3, when the number of operations is 670, the average number of

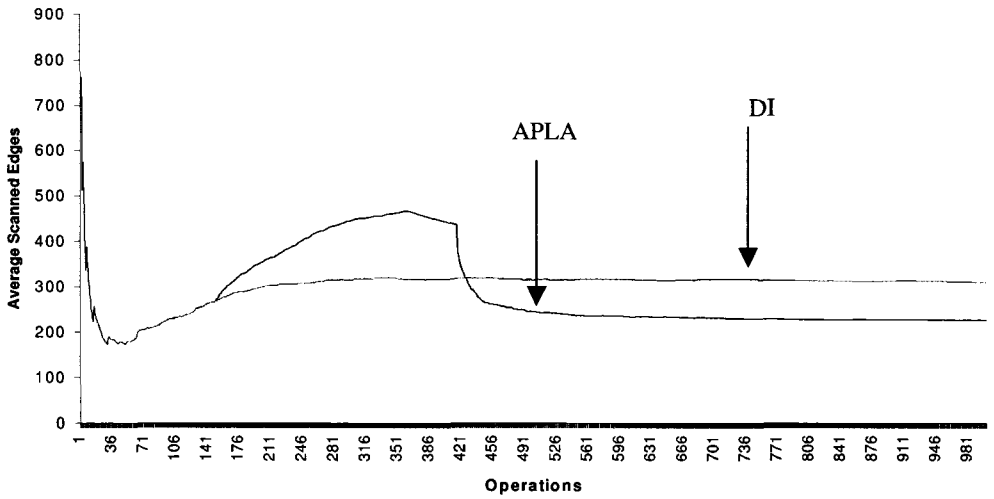


Figure 5.4: Average Scanned Edges

processed nodes for DI is approximately 79, whereas the average number of processed nodes for APLA is approximately 2.

5.3.8.2.2 Experiment Set 2

From the second set of experiments, in which we intended to evaluate the algorithms with the variation of the graph structure, we obtained the results shown in Tables 5.1 through 5.6. As before, we varied the graph structure by first varying the graph sparsity while keeping the other parameters constant, and then by varying the number of nodes in the graph. However, in this set of experiments, while varying the graph sparsity, the edge weights were chosen in the range between $3.0-8.0$, λ was set to 0.90, N was set to 20, and the number of operations were chosen to be 1000. Similarly, while varying the number of nodes in the graph, the edge weights, the learning parameter, λ, and the number of operations were kept the same as before, whereas the value of sparsity was fixed at 50%.

From the results obtained in this set of experiments, it can be seen that APLA

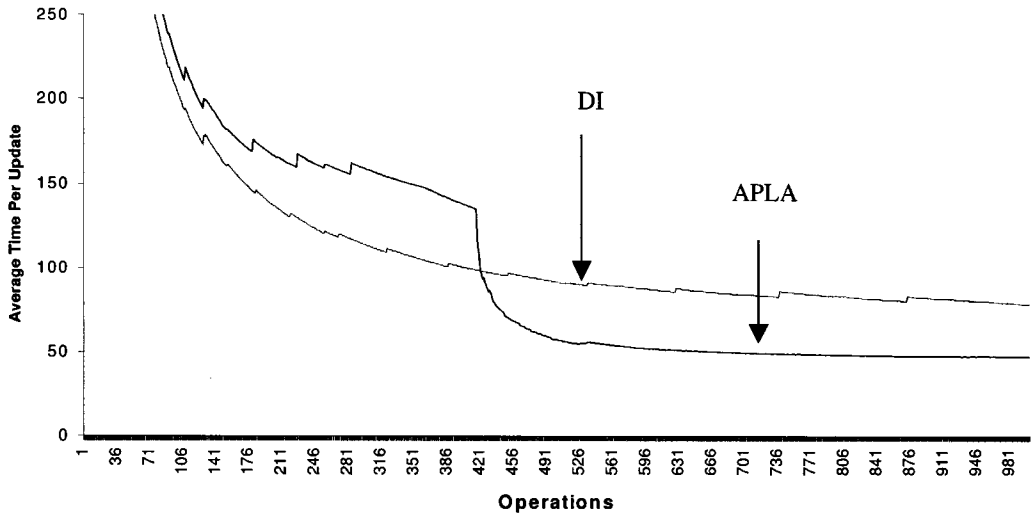


Figure 5.5: Average Time Per Update

performs much better than the DI algorithm across different graph structures with varying graph sparsities. For example, from Table 5.1, we see that the average value of time per update for APLA at 10% sparsity is 18.01, whereas that of DI is 51.03. This shows that APLA, on an average requires less time per update operation than the DI algorithm. Similar results can be observed in the Tables 5.4 through 5.6, when the number of nodes in the graph is varied.

5.3.8.2.3 Experiment Set 3

Figures 5.6 and 5.7 show the sensitivity of the performance of APLA to variation in λ, the learning parameter. It can be seen from these plots that as the value of the learning parameter increases, the average number of nodes processed, and the average time per update for the linear learning based APSP algorithm also increases. This is particularly true after APLA has undergone processing for a while.

Similarly, from Figures 5.8 and 5.9 it can be seen that with the increase in the values of the graph sparsity, the average time to update, and the average number of

Table 5.1: Statistics reporting the *Number of Processed Nodes* with the variation of the graph sparsity.

Sparsity	**DI**			**APLA**		
	Average	Minimum	Maximum	Average	Minimum	Maximum
10%	51.03	9.0	2140.0	18.01	4.0	1161.0
30%	35.39	9.0	1417.0	16.84	4.0	703.0
50%	34.5	9.0	1140.0	16.67	4.0	736.0
70%	29.7	9.0	810.0	15.27	4.0	529.0
90%	21.04	9.0	629.0	12.79	4.0	352.0

Table 5.2: Statistics reporting the *Number of Scanned Edges* with the variation of the graph sparsity.

Sparsity	**DI**			**APLA**		
	Average	Minimum	Maximum	Average	Minimum	Maximum
10%	436.6	12.0	1822.0	373.6	12.0	1365.0
30%	395.21	12.0	1450.0	339.1	12.0	1005.0
50%	364.7	12.0	1793.0	321.0	12.0	1190.0
70%	317.0	12.0	1105.0	288.0	12.0	927.0
90%	261.42	12.0	850.0	137.0	12.0	675.0

Table 5.3: Statistics reporting the *Time Per Update* with the variation of the graph sparsity.

Sparsity	**DI**			**APLA**		
	Average	Minimum	Maximum	Average	Minimum	Maximum
10%	227.7	0.0	3410.0	103.52	0.0	2937.0
30%	123.0	0.0	3100.0	86.41	0.0	2691.0
50%	73.8	0.0	2800.0	73.93	0.0	2630.0
70%	69.1	0.0	1972.0	37.02	0.0	1412.0
90%	62.7	0.0	1438.0	21.3	0.0	760.0

Table 5.4: Statistics reporting the *Number of Processed Nodes* with the variation of the number of nodes.

No of Nodes	DI			APLA		
	Average	Minimum	Maximum	Average	Minimum	Maximum
5	15.12	6.0	60.0	6.01	4.0	89.0
10	22.74	9.0	270.0	8.92	4.0	270.0
15	28.76	9.0	630.0	12.03	4.0	630.0
20	34.47	9.0	1140.0	17.8	4.0	1140.0
25	52.31	9.0	2011.0	24.8	4.0	1656.0
30	79.08	9.0	2612.0	31.3	4.0	1931.0

Table 5.5: Statistics reporting the *Number of Scanned Edges* with the variation of the number of nodes.

No of Nodes	DI			APLA		
	Average	Minimum	Maximum	Average	Minimum	Maximum
5	77.15	12.0	146.0	33.7	12.0	109.1
10	162.18	12.0	394.0	69.9	12.0	283.0
15	241.0	12.0	697.0	188.01	12.0	648.0
20	344.7	12.0	1793.0	321.0	12.0	1190.0
25	392.0	12.0	2104.0	363.5	12.0	1456.0
30	511.8	12.0	2497.0	402.8	12.0	1821.0

Table 5.6: Statistics reporting the *Time Per Update* with the variation of the number of the nodes.

No of Nodes	DI			APLA		
	Average	Minimum	Maximum	Average	Minimum	Maximum
5	6.5	0.0	110.0	2.9	0.0	73.8
10	17.4	0.0	710.0	5.06	0.0	390.0
15	40.2	0.0	1210.0	25.09	0.0	770.0
20	73.8	0.0	2800.0	43.93	0.0	1630.0
25	152.1	0.0	3120.0	112.5	0.0	2710.0
30	249.7	0.0	5622.0	193.2	0.0	3301.0

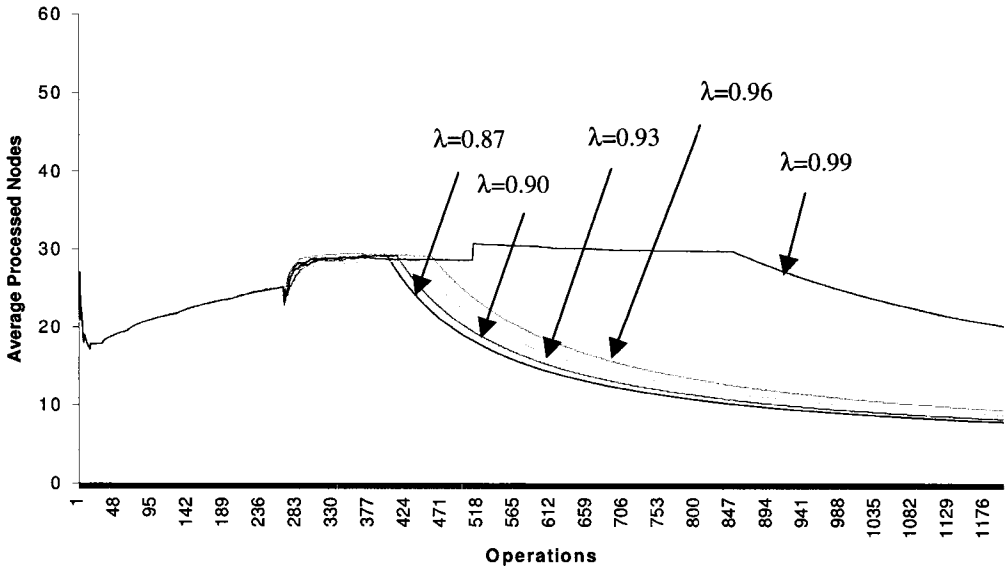


Figure 5.6: Sensitivity of the *Average Processed Nodes* in APLA with the variation in learning parameter.

scanned edges decrease.

5.4 Alternative Solution: Pursuit Learning Approach

5.4.1 Description

As we did in the case of the DSSSP problem, in addition to the linear learning perspective, we also approached the DAPSP problem from a *Pursuit learning* perspective. In this Section, we briefly introduce this alternative solution approach. This algorithm is called the *All-Pairs Generalized Pursuit Learning Algorithm* (*APGP*). As it is a Pursuit algorithm, it "*pursues*" the action that is estimated to be the best at that time instant. This action is the one corresponding to the maximal estimate. In particular,

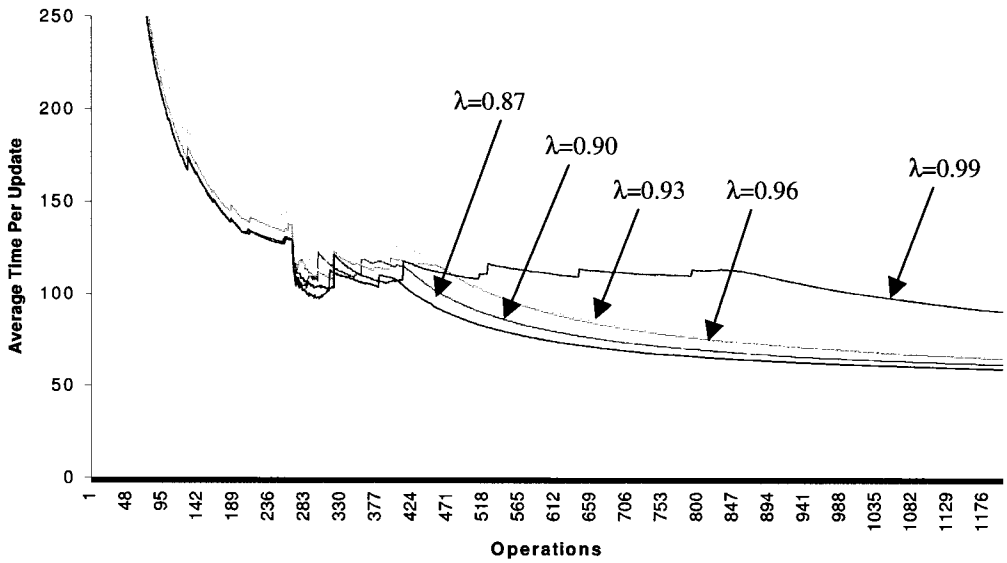


Figure 5.7: Sensitivity of the *Average Time Per Update* in APLA with the variation in learning parameter.

similar to the GPSPA algorithm, here we have used the *Generalized Pursuit Learning* approach proposed by Agache and Oommen [5].

5.4.2 The APGP Algorithm

We now describe the proposed solution to DAPSP, named as the APGP algorithm. The graph model we consider is exactly the same as described in Section 5.3.4. In such an environment, the proposed algorithm determines the underlying set of all-pairs shortest paths of the average edge weights, which is unknown to the system, by *pursuing* the best-estimated action at all time instants.

The APGP algorithm uses as its kernel the DI algorithm [21] whenever an edge-weight increase/decrease occurs. It is similar to its linear learning counterpart as far as the Automaton-Environment-Reward/Penalty structure is concerned. Therefore, we

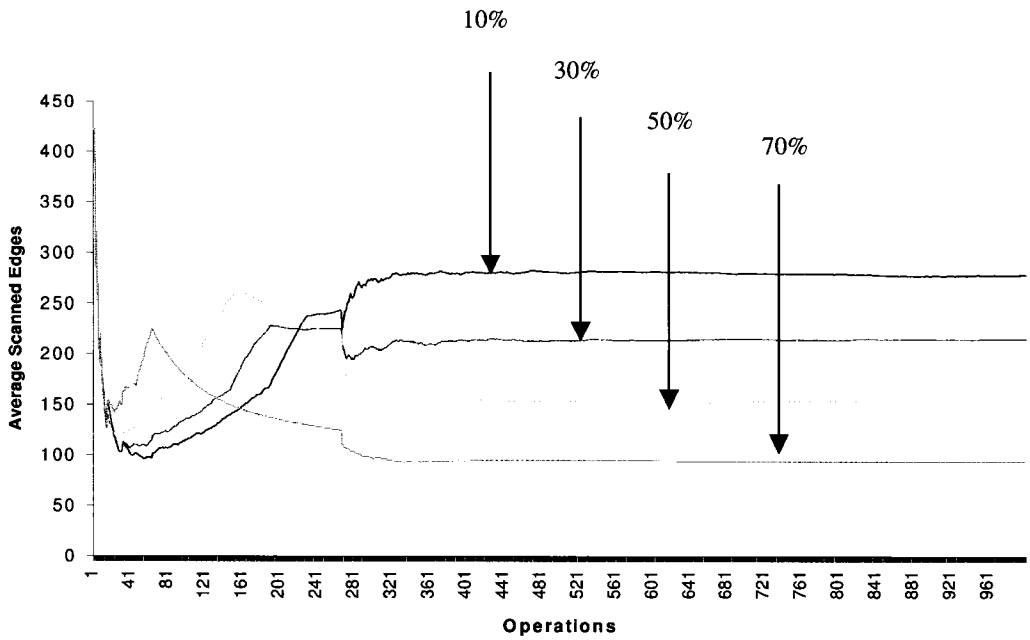


Figure 5.8: Sensitivity of the *Average Number of Scanned Edges* of APLA to variation in Graph Sparsity.

do not formally describe the items again. However, we highlight the major difference with the APLA algorithm, which consists of the way the algorithms learn the underlying shortest paths. Specifically, the APGP algorithm maintains a reward-estimates vector, $D(t)$, in addition to the probability vector, $P(t)$. Therefore, after the shortest paths are first calculated using Floyd-Warshall's algorithm, or recalculated using the DI algorithm following each link-cost update, the algorithm updates the action probability vector, and the reward estimates vector of each node using the Schemes 2(a), and 2(b) provided in Section 4.4.2 for the GPSPA algorithm. The details of this are omitted to avoid repetition.

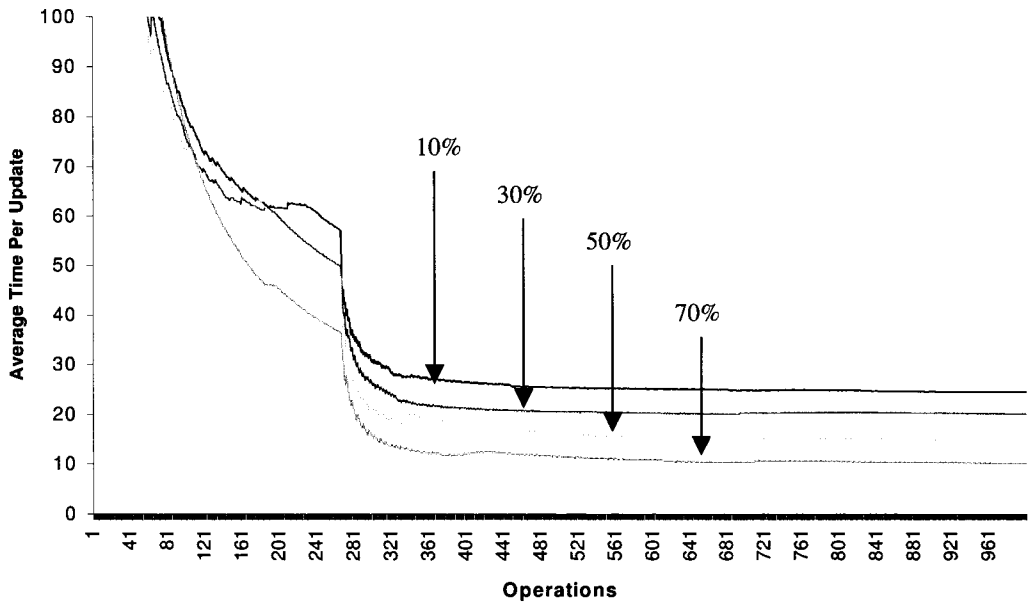


Figure 5.9: Sensitivity of the *Average Time Per Update* of APLA to variation in Graph Sparsity.

5.4.3 Experimental Results

In this Section we present the results of the experiments that were conducted to examine the performance of APGP. Like all the other LA-based shortest path algorithms discussed so far in the Thesis, the performance for APLA was also compared with respect to the three performance indicators, viz., processed nodes, scanned edges, and the time per update operation, using three sets of experiments on random graph topologies, and random edge-update sequences for different parameters.

The overall observation with APGP is consistent with the other LA-based single-source, and all-pairs dynamic shortest path algorithms discussed earlier. The experiments with APGP also show that APGP does not perform well at the beginning, i.e., when the algorithm is learning, but after the algorithm has learned, APGP outperforms the DI algorithm. The results of the three sets of experiments are summarized

below.

5.4.3.1 Experiment Set 1

The experimental parameters for this set of experiments were the same as those used in Section 5.3, i.e., a graph topology with 20 nodes and 50% sparsity, and with mixed sequences of 1000 modifying edge-update operations. The edge-weights are random real values between 3.0 and 8.0, and the value of the learning parameter, λ, was set to 0.90.

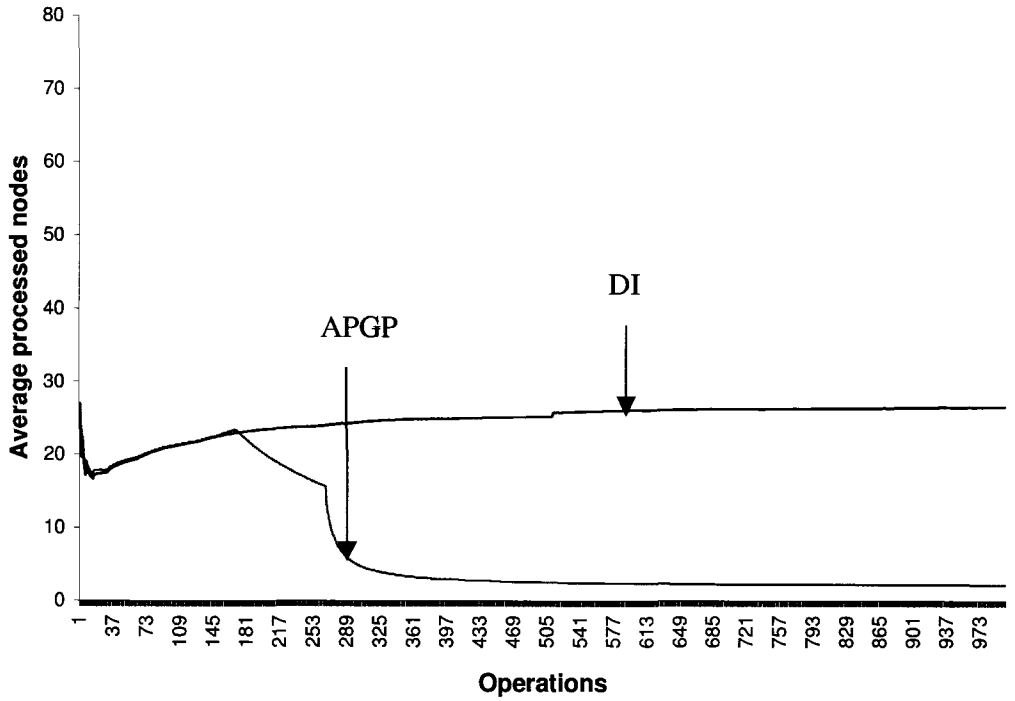


Figure 5.10: *Average Processed Nodes* for APGP and DI algorithms.

The results of the Experiment Set I are shown in Figures 5.10 through 5.12. The results show a 40 – 95% improvement of the performance of APGP with respect to that of DI. As with the other LA-based shortest path algorithms, here too, we see that

initially, APGP performs worse than the DI, but after it has learned, the performance of APGP exceeds that of DI. For instance, consider Figure 5.10. When the number of operations is 650, the average number of processed nodes for DI is approximately 28, whereas the average number of processed nodes for APGP is approximately 2.

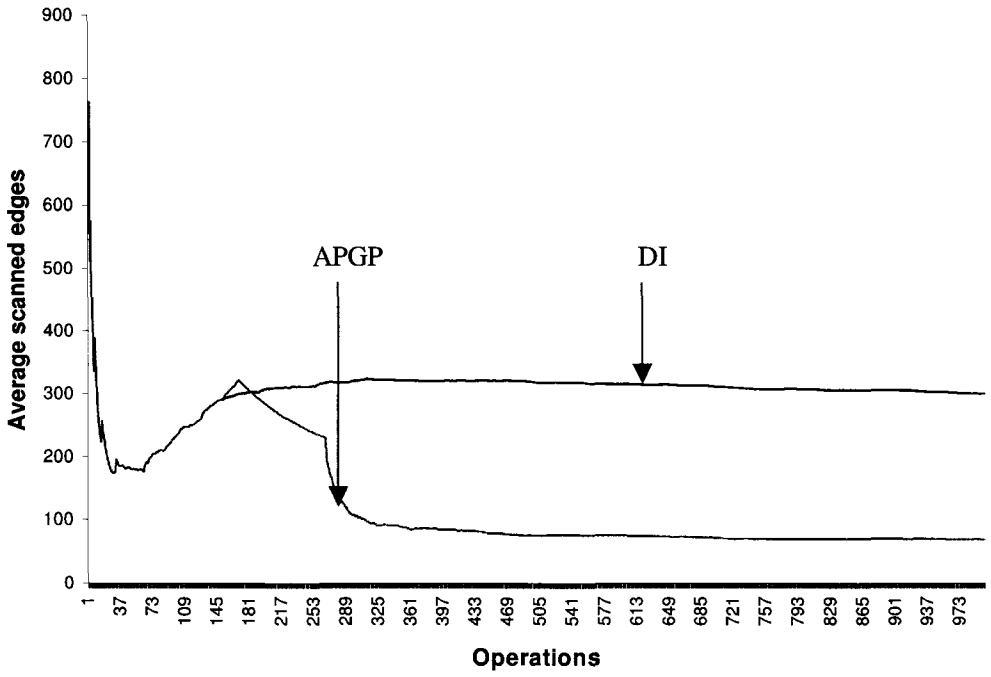


Figure 5.11: *Average Scanned Edges* for APGP and DI algorithms.

5.4.3.2 Experiment Set 2

This set of experiments, and the different parameters used are similar to those described for Experiment Set 2 in Section 5.3. Therefore, without elaborating the details, we mention below the results obtained.

The results of the experiment are summarized in Tables 5.7 through 5.12. Tables 5.7 through 5.10 describe the results when the sparsity of graphs was varied, keeping

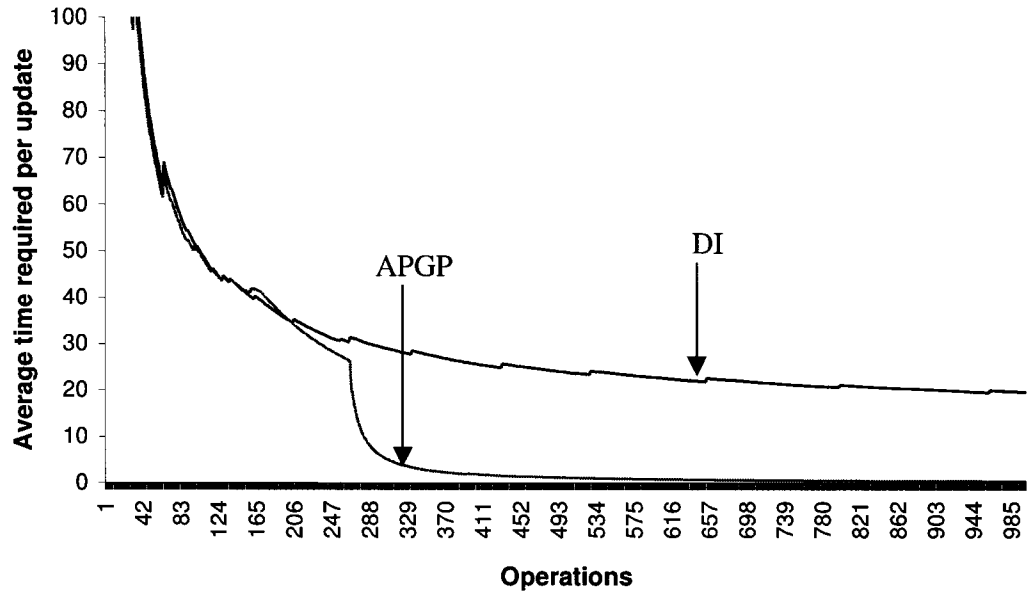


Figure 5.12: *Average Time Per Update* for APGP and DI algorithms.

other parameters constant, with the edge weights set to assume real numbered values in the range of $3.0 - 8.0$. The other parameters were: $\lambda = 0.90$, $N = 20$, and the number of operations $= 1000$.

Tables 5.11 through 5.12 describe the results for the case when the number of nodes in a graph was varied, keeping the other parameters constant. In this case, the experiments were performed on random graphs with edge weights between $3.0 - 8.0$, $\lambda = 0.90$, sparsity $= 30\%$, and the number of operations $= 1000$.

Our observations from this set of experiments follow the similar trend with those of same set of experiments discussed in the case of APLA in Section 5.3, i.e., APGP performs much better than the DI algorithm across different graph structures with varying graph sparsities. For example, from Table 5.7, we see that the average value of time per update for APGP at 10% sparsity is 14.98, whereas that for DI is 51.03. This shows that on an average APGP processes a lesser number of nodes per update

Table 5.7: Statistics reporting the *Number of Processed Nodes* with the variation of the graph sparsity.

Sparsity	**DI**			**APGP**		
	Average	Minimum	Maximum	Average	Minimum	Maximum
10%	51.03	9.0	2140.0	14.98	2.0	1140.0
30%	35.39	9.0	1417.0	14.16	2.0	1140.0
50%	34.5	9.0	1140.0	12.84	2.0	1140.0
70%	29.7	9.0	810.0	11.85	2.0	1140.0
90%	21.04	9.0	629.0	8.75	1.0	1140.0

Table 5.8: Statistics reporting the *Number of Scanned Edges* with the variation of the graph sparsity.

Sparsity	**DI**			**APGP**		
	Average	Minimum	Maximum	Average	Minimum	Maximum
10%	436.6	12.0	1822.0	184.48	1.0	1052.0
30%	395.21	12.0	1450.0	158.45	1.0	10.24
50%	364.7	12.0	1793.0	124.21	1.0	768.0
70%	317.0	12.0	1105.0	97.21	1.0	1239.0
90%	261.42	12.0	850.0	34.42	1.0	1701.0

operation than the DI algorithm. Similar results can be observed in the Tables 5.10 through 5.12, when the number of nodes in the graph is varied.

5.4.3.3 Experiment Set 3

Figures 5.13 through 5.15 show the plots of the variation of the sensitivity of the performance of APGP to variation in λ, the learning parameter. The general observations in this case are also consistent with those of the results obtained by conducting the similar set of experiments with the other LA-based shortest path algorithms examined in Chapters 4 and 5.

The plots in Figures 5.13 through 5.15 show that after APGP has undergone pro-

Table 5.9: Statistics reporting the *Time Per Update* with the variation of the graph sparsity.

Sparsity	**DI**			**APGP**		
	Average	Minimum	Maximum	Average	Minimum	Maximum
10%	227.7	0.0	3410.0	7.73	0.0	601.0
30%	123.0	0.0	3100.0	7.00	0.0	851.0
50%	73.8	0.0	2800.0	6.24	0.0	1012.0
70%	69.1	0.0	1972.0	5.03	0.0	481.0
90%	62.7	0.0	1438.0	3.0	0.0	560.0

Table 5.10: Statistics reporting the *Number of Processed Nodes* with the variation of the number of nodes.

No of Nodes	**DI**			**APGP**		
	Average	Minimum	Maximum	Average	Minimum	Maximum
5	15.12	6.0	60.0	2.27	2.0	60.0
10	22.74	9.0	270.0	4.36	2.0	270.0
15	28.76	9.0	630.0	7.81	2.0	630.0
20	34.47	9.0	1140.0	12.83	2.0	1140.0
25	52.31	9.0	2011.0	19.11	2.0	1800.0
30	79.08	9.0	2612.0	26.43	2.0	2033.0

Table 5.11: Statistics reporting the *Number of Scanned Edges* with the variation of the number of nodes.

No of Nodes	**DI**			**APGP**		
	Average	Minimum	Maximum	Average	Minimum	Maximum
5	77.15	12.0	146.0	5.91	1.0	78.0
10	162.18	12.0	394.0	30.26	1.0	310.0
15	241.0	12.0	697.0	68.33	1.0	603.0
20	344.7	12.0	1793.0	126.02	1.0	891.0
25	392.0	12.0	2104.0	197.0	1.0	1249.0
30	511.8	12.0	2497.0	283.0	1.0	1467.0

Table 5.12: Statistics reporting the *Time Per Update* with the variation of the number of nodes.

No of Nodes	DI			APGP		
	Average	Minimum	Maximum	Average	Minimum	Maximum
5	6.5	0.0	110.0	0.089	0.0	20.0
10	17.4	0.0	710.0	0.479	0.0	60.0
15	40.2	0.0	1210.0	1.812	0.0	181.0
20	73.8	0.0	2800.0	5.651	0.0	501.0
25	152.1	0.0	3120.0	14.270	0.0	1892.0
30	249.7	0.0	5622.0	37.152	0.0	2308.0

cessing for a while, as the value of the learning parameter increases, the average number of nodes processed, and the average time per update also increases. For instance, in Figure 5.13 we can see that when $\lambda = 0.87$ and number of operations is 408, the average number of processed nodes is approximately 6; that when $\lambda = 0.90$ and number of operations is 408, the average number of processed nodes is approximately 7, and this value increases with λ until $\lambda = 0.98$ and the number of operations is 408, when the average number of processed nodes is approximately 26.

Similarly, by closely observing the plots in Figures 5.14 and 5.15, it can be observed that as the value of λ increases, the average number of scanned edges, and the average time per update also increases. It may also be observed that the rate of change of the average number of processed nodes, the average number of scanned edges, and the average time per update operation is much more rapid (or more prominent) at higher values of λ than at the lower values.

From another experiment, we reported the sensitivity of the performance of APGP to the increase in graph sparsity. Figures 5.16 through 5.18 show the variation of average number of processed nodes, the average number of scanned edges, and the average time per update with the variation in the graph sparsity. It was observed that as the sparsity of the graph increases, the average time to update, and the average number

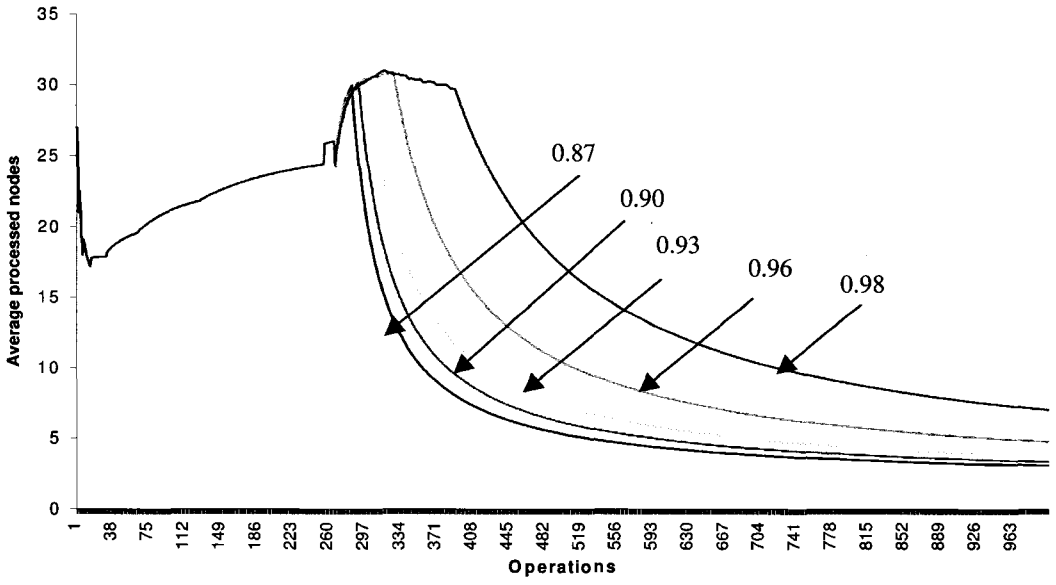


Figure 5.13: Sensitivity of the *Average Processed Nodes* in APGP with the variation in learning parameter.

of scanned edges decrease. Similar to the observation made in the last paragraph about the variation in the value of λ, it was observed in this class of experiments that the rate of change of the average number of processed nodes, the average number of scanned edges, and the average time per update operation are more prominent at lower values of the sparsity than at higher values.

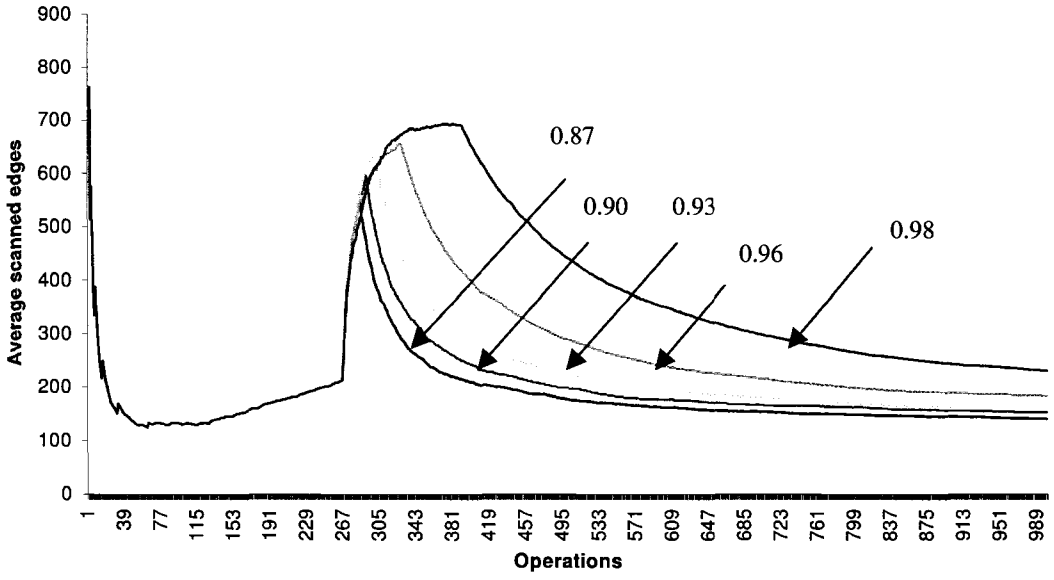


Figure 5.14: Sensitivity of the *Average Scanned Edges* in APGP with the variation in learning parameter.

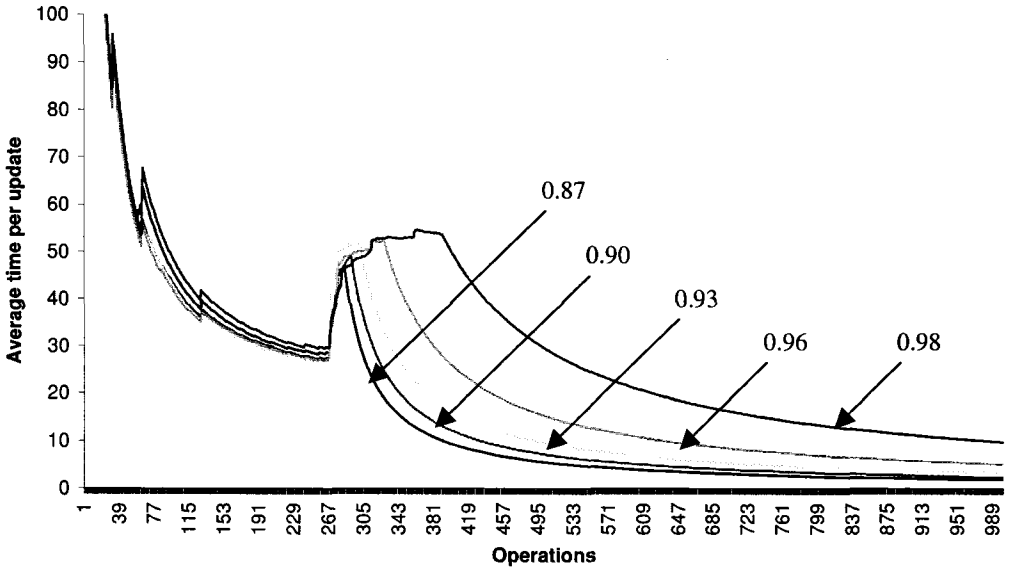


Figure 5.15: Sensitivity of the *Average Time Per Update* in APGP with the variation in learning parameter.

5.5 Conclusions

In this Chapter, we first introduced the problem of computing and maintaining all-pairs shortest paths information in a graph in which the edges are inserted/deleted and edge-weights constantly increase/decrease. We also described the functionalities of one of the most recent solutions to this problem, the recently proposed fully-dynamic algorithm by Demetrescu and Italiano [21].

Subsequently, we presented our first linear-learning solution, which is also the first reported LA solution to the DAPSP Problem. Finally, we proposed an alternative solution to the same problem using the Pursuit Learning approach.

Both the proposed solutions were implemented, and rigorously experimentally compared to the DI algorithm. All the algorithms were tested in stochastic environments where edge-weights changes stochastically, and where graph topologies underwent mul-

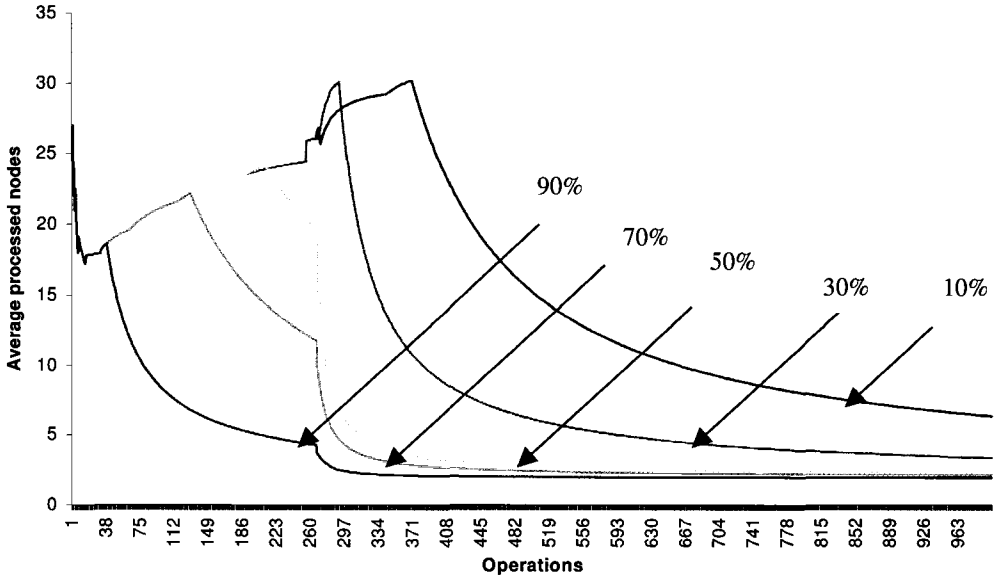


Figure 5.16: Sensitivity of the *Average Processed Nodes* of APGP to variation in Graph Sparsity.

tiple simultaneous edge-weight updates. The results also showed the superiority of the proposed algorithms with respect to the average number of processed nodes, scanned edges, and the time per update operation, over its competitor by, possibly, an order of magnitude. The results reported were for different graph topologies of varying size and edge-weights.

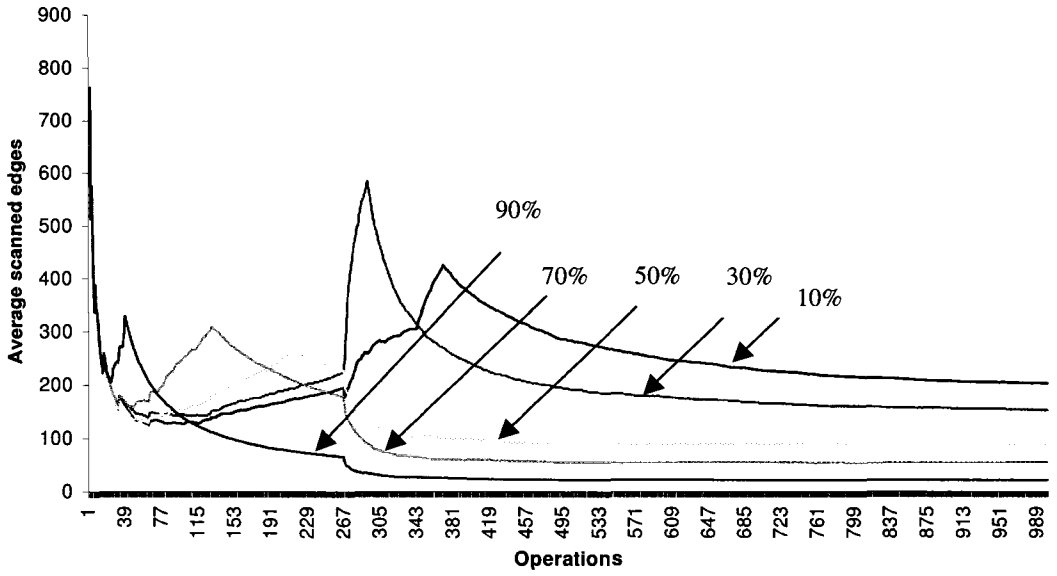


Figure 5.17: Sensitivity of the *Average Number of Scanned Edges* of APGP to variation in Graph Sparsity.

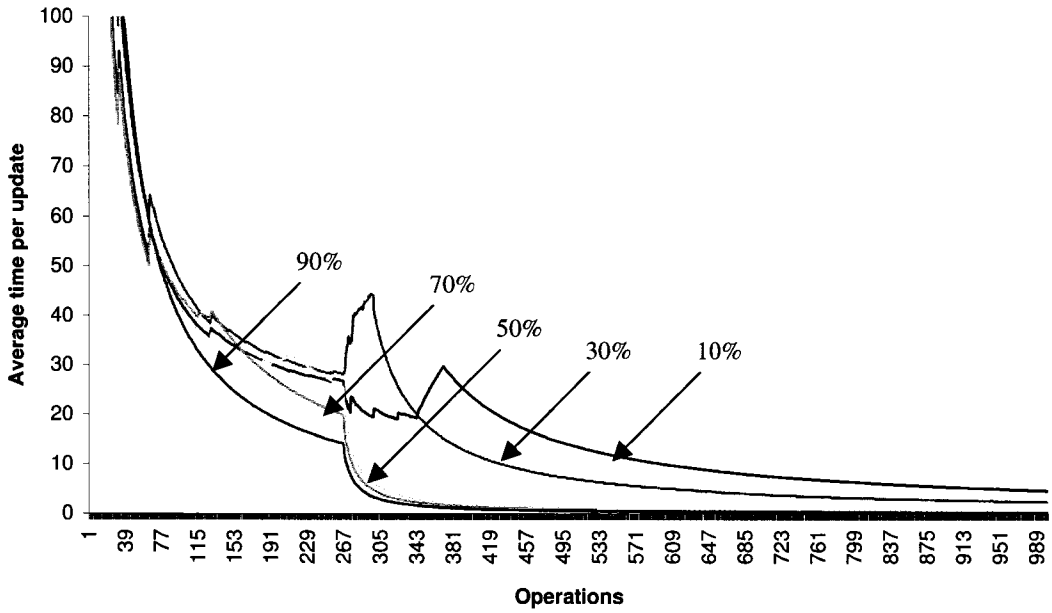


Figure 5.18: Sensitivity of the *Average Time Per Update* of APGP to variation in Graph Sparsity.

Chapter 6

QoS Routing And TE Algorithms

In this Chapter[1], we discuss how we can use learning strategies to enable the computation of bandwidth guaranteed paths in networks, and dynamically routing path setup requests between a pair of ingress-egress routers by satisfying the requested bandwidth.

In Sections 6.1 through 6.3, we first introduce the fundamental concepts of QoS routing, and the different QoS routing algorithms widely used in routers. Then, in Section 6.4 we discuss the algorithms proposed under the relatively recently evolved discipline of TE[2]. Finally, in Section 6.5, we present our proposed algorithm, and the

[1]The results reported in this Chapter have been submitted for publication. A preliminary solution for the problem has appeared in the *Proceedings of the* 3^{rd} *International Conference for Upcoming Engineers* (ICUE 2004), IEEE Toronto, Ontario, May 13-14, 2004, and has won the **Best Technical Paper Award** in the Communications Session of the conference.

[2]QoS or TE Routing and often used in similar contexts in the literature, and they seem to have fine lines of differences. These two terminologies are very interrelated and their distinction is often ambiguous. On being asked the differences between QoS Routing and TE, Professor Roch Guerin, in a public forum (*http://cell.onecall.net/mhonarc/mpls/1999-Jan/msg00038.html*), presented his views as follows: "Specifically, differences can/will emerge along two axes: time scale and requirements. In my mind, traffic engineering typically operates at a relatively long time scale and is primarily concerned with ensuring proper provisioning of links and assignment of traffic to meet some relatively static traffic estimates, e.g., average load or 99 load percentile over the busiest hour of the day/week/month/etc. In contrast, QoS routing often targets balancing of load at a finer time scale, e.g., to take advantage of "short term" fluctuations in traffic patterns. The same kind of routing algorithm can be applied to both, and I expect that initially the focus will be mostly on balancing longer term load assignments than doing finer grain optimization. If anything because requests will not be for dynamic setup and take down of individual flows, but rather will be dealing with traffic aggregates that are less subject

results of its experimental evaluation.

6.1 QoS Routing

A network is said to support QoS, if it has the capability of treating different packets differently. QoS technology has enabled service providers to support different levels of service to different customers, thereby capacitating them with the option to provide better levels of paid services to some customers than to others. For example, some groups of customers may be concerned with a service that guarantees packet delivery, even if that means paying a higher price for these services, others may just as well be satisfied with relatively less reliable data transfer by paying less for their subscribed services. Networks that transport multimedia traffic, i.e., voice, data, and video, needs differential treatments of different packets – while voice traffic is highly sensitive to time delay and the orderly delivery of packets, data traffic is relatively less sensitive to these, and, video-conferencing traffic requires a dedicated connection for a fixed amount of time for the real-time, orderly delivery of packets [17, 34, 76].

Typical QoS-routing based performance metrics are bandwidth, delay, and throughput. While some applications require bandwidth guarantees, some others mandate the satisfiability of strict end-to-end delays, and others still require a high throughput, or a combination of both of these criteria [51].

---

to fluctuations. The other dimension where I expect differences to emerge between traffic engineering and QoS routing is in terms of supporting the selection of paths satisfying multiple requirements, e.g., satisfying bandwidth and delay constraints. One could consider extending traffic engineering to also include such capabilities, but I think it is somewhat outside its traditional scope, at least at the network level (it is part of link level traffic engineering, e.g., local call admission)."

For the purpose of this Thesis, without unnecessarily getting bogged down into debating the exact differences between the two, we will use QoS Routing algorithms to denote those that were proposed earlier, and refer to them as "conventional" (e.g., WSP, SWP). We will denote TE algorithms as those relatively recently proposed "sophisticated" algorithms (like MIRA, etc.) which considered different network and traffic characteristics to obtain higher performance routing as compared to the conventional ones. However, it should be remembered that the TE algorithms that are presented in this Thesis are also essentially routing algorithms with QoS considerations.

Routing protocols in the pre-QoS era did not consider QoS requirements of connections (e.g., delay, bandwidth, and throughput). Furthermore, optimization of resource utilization was not a primary goal. As a result, while there were flow requests that were rejected because of non-availability of sufficient underlying resources, there were some other resources that remained available. To address such deficiencies with conventional routing protocols, QoS routing algorithms were devised that could locate network paths which satisfy QoS requirements, and which made better use of the network. Routing of QoS traffic requires stringent performance guarantees of the QoS metrics (e.g., delay, bandwidth, and throughput) over the paths selected by the routing algorithms. Accordingly, whereas the traditional shortest path algorithms, e.g., Dijkstra's or Bellman Ford's, indeed, have the potential of selecting a feasible path for routing, QoS routing algorithms must consider multiple QoS and resource utilization constraints, typically making the problem intractable. QoS routing is, thus, different from that of routing in traditional circuit-switched and packet-switched networks [34, 51].

In ATM Networks, for example, a connection is accepted or not by the *Connection Admission Control* (CAC), depending or whether or not sufficient resources (e.g., bandwidth) are available. The CAC operates by taking into account factors such as, the incoming QoS requests, and the available resources in the network. The CAC operates the QoS routing algorithms, which identify whether the different possible candidate paths satisfy the QoS requirements or not. The network architects, and the engineers want to choose such a routing algorithm which will help the service providers for maximizing the network resources (e.g., the amount of bandwidth to be routed), while satisfying the requirements from the customers. Therefore, the design of any good QoS routing algorithm takes into account factors such as, satisfying the QoS requirements, optimizing the consumption of the network resources (e.g., buffer space, link bandwidth), and balancing the traffic load across different paths. In addition to these, a good QoS routing algorithm should characterize itself by its ability to adapt to the

periodic dynamic behavior of the network [104].

6.2 Conventional QoS Routing Algorithms

Several QoS-based path selection algorithms such as, the *Shortest-Widest Path* (SWP) [106], the *Widest-Shortest Path* (WSP) [34], and the utilization-based algorithms [60], have been proposed in the literature, and their performance comparisons have also been made [51]. In this Section, we present an overview of these algorithms.

6.2.1 Shortest-Distance Path / Min-Hop Routing Algorithm

The shortest-distance calculation based algorithms do not strictly fall into the QoS-based routing category. These are, indeed, the predecessors of the QoS algorithms that were used in popular routing protocols such as the *Open Shortest Path First* (OSPF). However, we specifically provide a brief review of these schemes to facilitate the understanding of the other QoS algorithms, and to demonstrate that these algorithms can be considered as special cases of the QoS algorithms discussed below.

Popular shortest-distance calculation algorithms are: (i) the Dijkstra's algorithm [19], and (ii) the Bellman-Ford's algorithm [10]. Dynamic shortest-path algorithms, like the Ramalingam-Reps' algorithm [82], and Frigioni *et al.*'s algorithm [31] have already been discussed in the previous Chapters. The principal idea in shortest-path algorithms is to compute the shortest paths from a source node to all other nodes in a network. *Min-Hop Routing Algorithm* (MHRA) is, essentially, a shortest path algorithm where all links in the network have equal weights. In such a network, MHRA essentially computes the shortest path by computing the number of hops. Therefore, the shortest distance between two nodes is the least number of hops by which one can reach a destination node from a source node.

6.2.2 SWP Algorithm

The SWP algorithm was proposed by Wang and Crowcroft [106]. They proposed two variants of the algorithm: the *distance-vector-based SWP*, and the *link-state based-SWP*, depending on whether the SWP algorithm is based on *distance-vector-based routing* or on *link-state-based routing*. Essentially, in both variants of the SWP algorithm, the algorithm finds a path with the widest path, i.e., the path with the maximum bottleneck bandwidth. When there is more than one choice available, the algorithm chooses the path that has the minimum length, i.e., the one that has the shortest propagation delay [106]. If still there is a tie with one or more such path(s), one of the prospective paths is randomly chosen [51].

In distance-vector routing (see p. 285 of [76]), each node in the network is aware of the link-costs of its immediate neighbors. When this information is exchanged by each of the nodes with its immediate neighbors, all the nodes are made aware of the distances to the rest of the nodes. When the algorithm stabilizes, upon updating the routing tables of each of the nodes, each node knows the costs, and next-hop nodes for rest of nodes in the network. Using this concept, the *distance-vector-based SWP* algorithm can be formally stated as shown below (taken from [106]).

Algorithm: Distance-vector-based SWP

Input
A network with given length and width of the links.

Output
The shortest distance path with the maximum bandwidth.

BEGIN
Initially, $h = 0$, and ${B_1}^h = 0$, for all $i \neq 1$
Find set K so that $\text{width}(1, \ldots, k, i) = \max_{1 \leq j \leq N} \left[\min[{B_j}^{(h)}, b_{ji}]\right]$, $i \neq 1$
if *K has more than one element* **then**
Find $k \in K$ so that $\text{length}(1, \ldots, k, i) = \min_{1 \leq j \leq N} \left[{D_j}^{(h)} + d_{ji}\right]$, $i \neq 1$
end
${B_i}^{(h+1)} = \text{width}(1, \ldots, k, i)$ and ${D_i}^{(h+1)} = \text{length}(1, \ldots, k, i)$
if $h \geq A$ **then**
Exit and end the algorithm
end
else
$h = h + 1$
Go to Step 2
end
END

In the above algorithm, the widest of all paths from node 1 to each node i is computed. Then, the algorithm selects a path with the minimum length, if more than one widest path is obtained through the previous step. Then, the algorithm updates the width and length of the shortest-widest path from node 1 to each node i in the network [106].

In the case of link-state routing (see pp. 292 of [76]), each node is made aware of the link-state information (i.e., whether the links to its neighbors are active or dead and how much the cost of each link is) by a procedure called *flooding*, so that each of the nodes can find the shortest paths to all the remaining nodes in the network. Upon stabilization of the algorithm, after flooding has disseminated all the link-state information to all the nodes in the network, each node is capable of calculating the

shortest path to the remaining nodes in the network. The algorithm maintains two lists for a node, called the *Tentative*, and *Confirmed* lists, each of which contains entries which store information about the destination node, the cost to reach the neighbor of the node, and the next hop of that node. The entries labeled *Tentative* are those that are being processed but are not yet confirmed to belong to the shortest path routes. Using this concept, the *link-state-based SWP* algorithm [106] is outlined below (taken from [106]).

Algorithm: Link-state-based SWP

Input
A network with given length and width of the links.

Output
The shortest distance path with the maximum bandwidth.

BEGIN
Initially, $L = \{1\}$, $B_i = b_{ij}$, and $D_i = d_{1i}$ for all $i \neq 1$.
Find set $K \notin L$, so that $B_K = \max_{i \notin L} B_i$
if *K has more than one element* **then**
Find $k \in K$, so that $\text{length}(1, \ldots, k, i) = \min_{j \in K} [D(1, \ldots, j, i)]$
end
$L \leftarrow L \cup \{k\}$
if *L contains all nodes* **then**
Exit and end the algorithm
end
for *all $i \notin L$* **do**
Set $B_i \leftarrow \max [B_i, \min[B_k, b_{ki}]]$
end
Go to Step 2
END

In the link-state-based SWP algorithm, the number of nodes, which has the maximum width among the tentatively labeled nodes, is determined first. If that results in more than one node, the minimum length path is then chosen, and labeled permanently. Then, the temporarily labeled nodes are updated around the newly perma-

nently labeled node found in the previous step [106]. Thus, in simplified words, the link-state-based SWP algorithm can be stated in the following manner:

1. First, from the list of tentatively labeled nodes, find the nodes with the maximum width.

2. If there are several such nodes satisfying Step 1, select the node with the minimum length path, and label it as *Confirmed.*

3. Finally, update the maximum width (or the link residual capacity), and the length for the shortest-widest path for the nodes labeled as *Confirmed.*

The SWP algorithm has the following properties [106]:

1. The time complexity of the SWP algorithm is proportional to that of the corresponding shortest path algorithm.

2. The SWP algorithm can be considered as a general case of the shortest-path algorithm when all links are considered to have different link capacities. In other words, if all link capacities are considered the same, the shortest path algorithm can be viewed as the particular case of the SWP algorithm.

3. A path's width is decided by the bottleneck link, i.e., the link in a path with the minimum residual bandwidth.

4. The paths computed using SWP algorithms do not form a loop. This property of SWP is analytically *appealing and proven* [106].

6.2.3 WSP Algorithm

The WSP algorithm was proposed by Guerin *et al.* [34]. Unlike the SWP algorithm, WSP first attempts to compute the shortest path. If there exists more than one

alternative, the algorithm chooses the one with the largest residual bandwidth in the bottleneck link (i.e., the widest path). If there is still a tie with one or more such path(s), one of the prospective paths is randomly chosen [51].

This process can be described into four principal steps stated below [34, 51, 108]:

1. For a route request with b units of bandwidth, scan and discard all links, e, whose residual bandwidth $r_j(e) < b$.

2. Apply a suitable shortest path algorithm (e.g., Dijkstra's), and compute the shortest path. If all paths have same edge lengths (costs), then select the path with the least number of hops.

3. If Step 2 results in more than one choice, select the widest path with the largest residual bandwidth.

4. If it turns out that there is more than one path with the same widest path, randomly select one path from amongst the available alternatives.

The major limitation of WSP lies, principally, in the fact that the algorithm is essentially a shortest path greedy algorithm. Note that the notion of widest paths arises only when there are multiple shortest paths. On the other hand, since the SWP algorithm first selects a path, which has the largest residual bandwidth, it can select very long paths to satisfy its requirement. The choice of the algorithm should depend on the network environment and requirements. If we are provided with a meshed network with intricately connected equal-length paths, WSP could be a better alternative than the SWP. Whereas, if we have either (i) a general non-meshed (or even small) network, where propagation delays are small, or (ii) network resources are inexpensive, SWP could be a better alternative of choice [108].

The difference between WSP, and SWP clearly lies in the fact that whereas WSP attributes higher priority to minimizing the distance or the hop-count, SWP pays more

emphasis on balancing network loads [108].

6.3 SELA Routing Algorithm

The SELA Routing Scheme is a QoS-based scheme that was proposed by Vasilakos *et al.* [104] for use in CAC problems in ATM networks. It uses the reinforcement learning scheme called SELA (see Chapter 2) for optimizing the network revenue while ensuring that the QoS requirements (e.g., *cell loss ratio*, *maximum cell transfer delay*, and *cell delay variation*), for the different connections are satisfied in the ATM Networks [104].

In SELA routing, an automaton operates at each source node. The purpose of the automaton is to choose one of the different possible paths, represented as the different actions of the LA operating at the source node, for routing requests between pairs of source and destination nodes. The SELA algorithm maintains a table containing the current link utilizations for the entire network. This table helps to compute the feedback of SELA to the different possible actions. The algorithm reserves trunks for network conditions where uncontrolled alternate path routing can lead to unacceptably increased rejections of connection setup requests [6, 104]. As a result, the algorithm tries to improve its QoS performance by discouraging the loading of the congested links, unless the link happens to be in the shortest path (in terms of the number of hops) between the source and the destination nodes [104].

The algorithm defines a parameter ρ_{TRT} , which represents a predefined route utilization threshold that any route can have. The SELA algorithm routes a request through a over-utilized path only if the maximum expected utilization that the links comprising a particular route can assume ($\rho_{\text{exp}}^{\text{route}}$) exceeds ρ_{TRT}, but that over-utilized path happens to be a shortest distance path. If the path is not a shortest distance path, the algorithm looks for an alternate path [104].

The SELA algorithm utilizes a feedback function $b(t)$ for any link in a set of links

comprising a route as follows [104]: $b(t) = 1.0 \times \text{MAX\_NO\_HOPS} - \sum_{i=0}^{n} \rho_i$ where

$$\rho_i = \begin{cases} 1, & \text{if } \rho^i{}_{\text{exp}} > \rho_{TRT} \\ \rho^i{}_{\text{exp}}, & \text{otherwise.} \end{cases}$$

Here, $\rho^i{}_{\text{exp}}$ is the expected utilization of the i^{th} of the n available link segments comprising any route, and MAX_NO_HOPS is the maximum number of hops that is possible between any pair of source-destination nodes.

The structure of the function $b(t)$ demonstrates that increasing congestion would reduce its value. Similarly, the value of $b(t)$ increases with increasing number of hops in a route, thereby signifying that the algorithm favors minimum-hop routes. The SELA routing algorithm [104] is formally presented below (most Sections of text in the algorithm below are taken verbatim from [104]):

Algorithm: SELA Routing

Input
A network with incoming bandwidth routing requests

Output
Requests routed through different paths

Algorithm
BEGIN
At each source node
1. Find k-shortest (minimum-hop) routes for each source-destination pair.
2. Inform the routing tables at each node of the shortest paths information.
3. Represent each of the possible routes between a source-destination pair to one of the k actions of SELA.

... (Continued to next page)

Algorithm: SELA Routing (Continued from previous page)

Online operation

1. Whenever there is a new call-request at the source node, estimate the expected utilization of the links of the candidate routes.
2. Run the SELA algorithm computing the environmental feedback.
3. Select the first optimal action.
4. If that action corresponds to a minimum-hop route and this route is accepted by the CAC, then choose that action as the selected one and establish the call over the corresponding route.
5. Else if that action corresponds to an alternate route which is accepted by the CAC and whose utilization ($\rho_{\text{exp}}^{\text{route}} < \rho_{TRT}$), then choose that action as the selected one, and establish the call over the corresponding route.
6. Else if none of the above stands, then repeat Step 3 concerning the next optimal action.
7. If all actions have been tried without any of them being chosen as the selected one, then the call request is rejected.

END

The SELA Routing algorithm is stated to have the following properties [104]:

1. *Increases QoS performance and good fault-tolerance*: The algorithm uses a simple probabilistic approach and adapts to the past history of the links. This helps to improve the performance of the algorithm (in terms of satisfying the QoS requirements). In addition to that, because of the high speed of convergence, if a node or a link fails in a network, or if for any reason the topology changes, SELA restarts, and due to its high-speed of convergence, adapts to the new topology very fast [104].

2. *Small computational overhead*: SELA routing has very small computational overhead due to the minimum amount of feedback it requires to operate, and a lesser number of control messages than the traditional routing schemes (e.g., SWP, WSP) [104].

6.4 TE and MPLS

Recently TE-based routing has become popular. TE mandates to optimize the performance of traffic handling, and resource utilization on existing physical network topologies. This is, in principle, engineered by minimizing the over-utilization of network capacity, and distributing the traffic load on costly network resources such as, links, routers, switches, and gateways [8, 75]. In the context of routing, TE is of great usefulness because traditional routing techniques are based on greedy shortest path computation techniques that lead to the over-utilization of certain network resources, even when other resources remain under-utilized.

Multiprotocol Label Switching (MPLS) has recently emerged to many professionals as a *de facto* standard in TE. MPLS is an IETF standard which merges the Layer 2 information of bandwidth, latency, and utilization of network links, with the control protocols used in Layer 3 Internet Protocol (IP), in order to simplify the exchange of IP packets. At the heart of the idea is the usage of a *label* (or a *tag*) to calculate shortest paths to all destinations within an autonomous system, thereby expediting the forwarding of packets. A label can be perceived as a simplified representation of an IP packet's header, with the additional advantage of enabling core backbone networks to operate at high speed because of the exclusion of the need to re-examine each packet's IP header in detail. This, in turn, permits the differentiating between packets on a individual basis, and facilitates the support of QoS. The destination of a packet is determined by observing the label, and not the IP address it is destined to. MPLS helps network operators manage network route failures, and makes the system to decongest bottleneck links by providing a *de tour* route for the incoming traffic. It can also help service providers manage the provisioning of different kinds of traffic according to different plans, service, and policies of the customers [8, 75].

To pose our problem in the right perspective, we review the basic MPLS terminolo-

gies using a hypothetical network as shown in Figure 6.1, and describe below, in brief, how a packet is transmitted. In actuality, the steps involved are much more complex. In Figure 6.1, a *Label Switched Router* (LSR) is a backbone router in the physical network topology that runs the existing Layer 3 IP protocol. The particular cases of LSR, which are the edge routers are called the *Label Edge Routers* (LERs). These routers often serve as the ingress and egress routers, which can be thought of as synonymous to the source, and destination routers for packets passing through the MPLS network. Packets are tagged with fixed-length labels at the ingress router, and untagged off the labels at the egress routers. The paths (or routes) along which the labeled packets are transmitted are termed as the*Labeled Switched Paths* (LSPs), and the protocol that monitors the negotiation, and the exchange of labels is called the *Label Distribution Protocol* (LDP). In a typical transmission of a packet through the MPLS network, after the backbone routing protocol (such as the OSPF) ascertains the reachability of a destination node, the LDP protocol establishes a mapping between a label, and the destination node; the ingress node then receives a packet, and labels it with a fixed length label (as described earlier), and transmits it towards the egress node, which, upon receipt of the labeled packet, removes its label, and delivers it [8, 75].

TE and MPLS are entire new disciplines by themselves, that have recently gained much attention in the networking community. Since their detailed understanding is not required over here, we only provide a brief overview. A more detailed understanding of how traffic engineering is done with MPLS can be obtained by reading the excellent book by Osborne and Simha [75], or a white paper on the topic [8].

During the LSP setup phase mentioned above, the intermediate LSRs between the ingress-egress nodes are specified. During this phase, the paths for a given flow are explicitly specified, enabling service providers for engineering the incoming traffic to be routed, and also supporting QoS, optimizing network utilization, and minimizing the number of rejected LSP setup requests, as this is what is typically mandated by

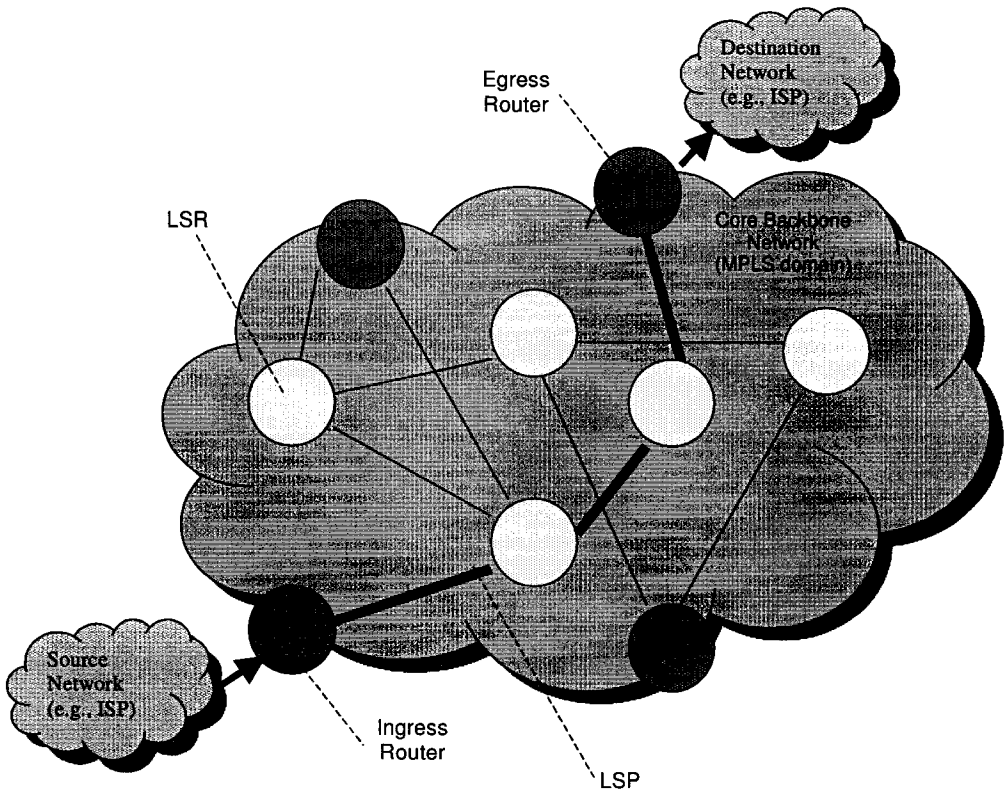


Figure 6.1: A hypothetical MPLS backbone network.

the traffic engineers [8, 75, 92].

The traditional routing algorithms described in the previous Chapters that forward packets based on the information about the destination address only are short-sighted, and are, therefore, constrained by the following limitations (as identified by Suri *et al.* [92]):

1. They do not take into consideration the current flows, or expected future flow demands in the network. Those algorithms are greedy, in the sense that they would route a request through the default shortest path, the shortest-widest path, or the widest-shortest path, and would reject a demand when the pre-computed

routes using those algorithms are congested, even when other alternate routes are free to accept more requests.

2. They do not take into consideration information about network infrastructure, network topologies, or traffic profiles to avoid loading bottleneck links in a network that might lead to rejection of future demands.

3. The previous algorithms will perform negatively when they operate in an online routing situation, where the tunnel setup requests arrive one at a time, and the future demand is unknown. Those algorithms require the knowledge of future demands to operate successfully.

4. The previous algorithms are not adaptive to possible link-failures. Therefore, in a situation where a link fails, those algorithms will not be able to route requests through alternative routes.

6.5 Online Traffic Engineering Routing Algorithms

As mentioned earlier, online Routing using TE principles has recently drawn considerable attention. We mention here a few TE Algorithms proposed in the literature (e.g., [41], [92], [109], [86], [37]). Of all the online TE routing algorithms, we believe that the one that has attracted the most attention is the *Minimum Interference Routing Algorithm* (*MIRA*) designed by Kar *et al.* [41][3].

In addition to the MIRA algorithm, there are a few other TE routing algorithms[4] that were proposed by other researchers, some of which are: (i) the *Profile Based*

[3]MIRA algorithm is available in both a conference version [42] of Kodialam and Lakshman, and also as a journal paper by Kar, Kodialam, and Lakshman [41]. For the sake of our convenience, we shall refer to these pieces of literature interchangeably.

[4]We do not intend to exhaustively list all the available pieces of literature in these areas. Although several algorithms have been proposed, in the interest of brevity and conciseness, we merely highlight the "prominent" algorithms.

Routing (PBR) [92, 93], (ii) the *Dynamic Online Routing Algorithm* (DORA) [14], (iii) Iliadis and Bauer's algorithm [37], (iv) Wang *et al.*'s Algorithm [109], and (v) Subramanian and Muthukumar's algorithm [88].

Kar *et al.* [41] identified a set of properties that a "practically-useful" TE algorithm (based on MPLS) should have. Some of these properties are:

1. The algorithm should be based on an *online* routing model, where LSP setup requests arrive one at a time (not all at once), and the future demand is unknown. On the other hand, in an *offline* model, all LSP setup requests are known *a priori*, and there are no demands for future LSP setup requests.

2. It should be able to use knowledge about the *physical locations* of the ingress-egress router pairs through which an LSP is set up.

3. The algorithm should be able to *adapt to possible link failures* in the network. In other words, a good TE algorithm should be able to reroute post-failure requests through alternate routes.

4. It should be able to route the requested bandwidth *without splitting the demands* as much as possible through multiple paths. This is necessary because it often occurs that the nature of the traffic does not allow a demand to be split. Splitting traffic is, however, a common practice in scenarios like load balancing, and network performance improvement.

5. The algorithm should, if possible, support a *distributed implementation*, where instead of performing the route computation in a centralized server, the computations of each LSP's route request is distributed at the local ingress node.

6. It is quite desirable that such an algorithm is capable of using different *policy constraints*. For example, service level agreements might impose a restriction that

LSPs with less than a threshold value of flow guarantees should not be accepted [41, 92].

7. Such an algorithm should operate under strict bounds of computational complexity. The algorithms should be very fast, and execute within a fixed time constraint. The amount of computation involved with LSP setup requests should be minimized so that the algorithm can be implemented on a router or a route-server [41, 92].

6.5.1 The MIRA Algorithm

The most influential online routing algorithm in TE was the MIRA algorithm of Kodialam and Lakshman [42]. MIRA is online because it does not require *a priori* knowledge of the tunnel requests that have to be routed. The algorithm is targeted towards applications such as the setting up of LSPs in MPLS networks. The algorithm helps service providers setup bandwidth guaranteed tunnels at ease in their transport networks. The terminology, "minimum interference", used in the nomenclature of the algorithm indicates that the tunnel setup request is to be routed through a path (or a path segment) that must not *interfere too much* with future tunnel setup requests. The algorithm aims to protect the *critical links* in a network from being overloaded, and thereby reducing the chances of rejection of future requests. The critical links are identified by the algorithm to be those that, if congested because of heavy loading, might lead to rejection of requests. One characteristic that is particularly attributable to MIRA is that it is the first algorithm that uses information about ingress-egress pairs. Unlike its predecessor algorithms, MIRA uses any available information about ingress-egress nodes for potential future demands. MIRA is based on the core concepts of the *max-flow*, and the *min-cut* computations that are described below [4, 42, 108].

Consider a hypothetical network as shown in Figure 6.2. We demonstrate the

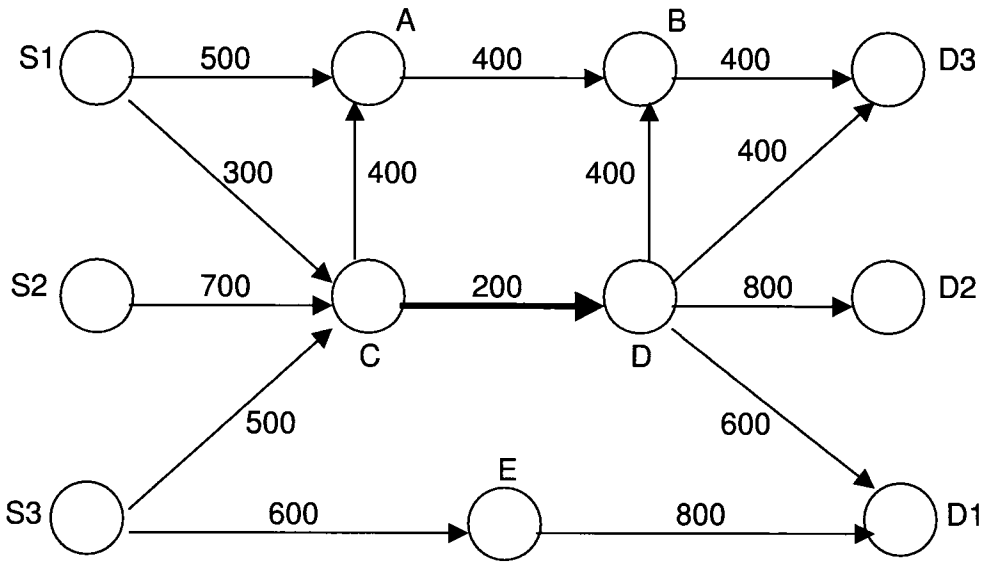


Figure 6.2: Hypothetical network for demonstrating the functioning of MIRA.

concepts of max-flow, min-cut, critical-links, and the MIRA algorithm using this sample network. While the original paper of MIRA [41] presents the concepts in an abstract manner, we attempt to clarify the individual steps in a lucid manner with the help of Figure 6.2. In Figure 6.2, $(S1, D1)$, $(S2, D2)$, and $(S3, D3)$ are three pairs of ingress-egress nodes. A, B, C, D, and E are five intermediate nodes in the network. The figures written alongside the links connecting two or more nodes indicate the maximum capacity of those links.

- **Max-flow:** A *flow* in a communication network can be defined as the rate at which information flows in the network. A *max-flow* can be defined as the maximum rate at which information can be transmitted over a link in the network by satisfying the capacity constraints. Intuitively, in a max-flow problem, we aim to design the network in such a way that the flow should not violate the capacity constraints and the algorithm has to transmit the maximum possible flow from a source node to a sink node [4, 18, 108]. In the network of Figure 6.2, we need

to transmit as much flow as possible from $S1$ to $D1$, $S2$ to $D2$, and from $S3$ to $D3$. When a demand is routed through a link, its maximum flow decreases, and the used capacity is unavailable to future demands.

- **Critical links:** If information has to pass through a minimum-hop route, there would be *interference* at the bottleneck link. The link CD in Figure 6.2 is an example of a bottleneck link when information is transmitted at the same time from $S1$ to $D1$, $S2$ to $D2$, and $S3$ to $D3$ through the minimum hop routes. MIRA attempts to minimize this interference due to routing of requests through the bottleneck links. The bottleneck links are called *critical links* and are computed using the min-cut concept described below. The set of critical links between a pair of source and sink nodes is the minimum cut between that pair. MIRA defers the loading of critical links as much as possible [41, 108].

- **Cut:** The phrase "cut" in "Flow Networks" indicates a partitioning of the nodes in the network into two sub-sets [4]. As illustrated by Ahuja *et al.* [4], if N is a set of nodes in a flow network, a cut involves partitioning all the nodes of the network into two sub-sets, S and $S' = N - S$, such that $S \cup S' = N$.

- **s-t cut and min-cut:** A cut is referred to as an s-t cut if s belongs to S (i.e., one sub-set of the nodes), and t belongs to S' (i.e., the other sub-set of the nodes N). The capacity of all the forward arcs in the cut is referred to as the *capacity* of the s-t cut. The minimum capacity of the capacities of all s-t cuts is referred to as the *min-cut* [4].

- **Max-flow Min-cut Theorem:** Ahuja *et al.* in their classic book on Network Flows enunciate the theorem as: "the maximum value of the flow from a source node s to a sink node t, equals the minimum capacity among all $s - t$ cuts" [4]. For example, in Figure 6.2, the edge CD is a *min-cut* edge.

6.5.1.1 The Algorithm

After having reviewed these preliminary concepts, we shall utilize them to understand the MIRA algorithm [42, 41] after presenting their system model.

Let us suppose that we have a network represented by a set of N nodes, L links, and a set B representing the bandwidth between the links. Of all the N nodes, only a subset of these nodes are ingress-egress nodes, of which, again, only a sub-set of all possible ingress-egress nodes will lead to setting up an LSP. A request for setting up an LSP is processed either directly by a route server, or indirectly via an ingress node, both of which finally, through some interactions, know the explicit route from the ingress node to the egress node [42, 41].

A request for setting up an LSP is denoted by [42] as the triple (o_i, t_i, b_i), where o_i represents the ingress router, t_i represents the egress router, and b_i the amount of bandwidth requested for LSP i. They assume that all the QoS requirements are converted into an effective bandwidth requirement, that all the requests for setting up an LSP come one at a time, and that the future demands are unknown. In such a model they intend to devise an algorithm that will help to determine a path along which each demand for an LSP is routed along such a path so as to make optimum use of network resources [42].

Suppose there is a set of distinguished node pairs P, which can be perceived as potential ingress-egress router pairs. Suppose the pair $(s, d) \in P$. All LSP setup requests occur between these pairs. Assume now that a request of D units of bandwidth arrives for the ingress-egress pair $(a, b) \in P$ [42].

Suppose now that the max-flow problem is solved between the ingress-egress pair (s, d) with the link capacities reflecting the current residual capacities of those links. The max-flow value determines the maximum amount of bandwidth that can be routed from s to d. Kodialam and Lakshman [42] define the *interference* between a path in

(a, b), and an ingress-egress pair (s, d) as the reduction in the max-flow value in the ingress-egress pair due to the routing of bandwidth on that path. Since they do not make any assumptions about the knowledge of future demands, the demand of D units between a and b should be routed in such a way as to maximally utilize the smallest max-flow value for the ingress-egress pairs in $P \backslash (a, b)$ [42].

To solve the problem, Kodialam and Lakshman's approach [42] is to compute the weights for the links in the network and route a demand along the weighted shortest path. The problem is to estimate the weights for all those links so that the current flow does not interfere too much with the future demands [42]. Without delving much into the procedure[5] about how they estimate those link weights, we give below the final equation that computes these weights for the links [42]:

$$w(l) = \sum_{(s,d) \in P \backslash (a,b)} \alpha_{sd} \frac{\partial \theta_{sd}}{\partial R(l)} \tag{6.1}$$

In Equation 6.1,

- α_{sd} represents the weight for the ingress-egress pair (s, d),
- θ_{sd} represents the maxflow value for the ingress-egress pair,
- $\frac{\partial \theta_{sd}}{\partial R(l)}$ represents the rate of change of the value of maximum flow between (s, d), with change in the residual link capacity $R(l)$.

Equation 6.1 can be further simplified by considering the notion of critical links. For an ingress-egress pair, a *critical* arc belongs to any mincut for that ingress-egress pair. The mincut is always computed with the present value of the residual capacities on those links. If Csd represents the set of critical links for an ingress-egress pair (s, d),

[5]Kodialam and Lakshman [42] utilize the concepts of Integer Programming for deriving these equations.

by the max-flow min-cut theorem it can be shown that[6] [42]

$$\frac{\partial \theta_{sd}}{\partial R(l)} = \begin{cases} 1, & \text{if } l \in C_{sd} \\ 0, & \text{otherwise.} \end{cases}$$

which implies

$$w(l) = \sum_{(s,d):l \in C_{sd}} \alpha_{sd} \tag{6.2}$$

Equation 6.2 can be used for computing the set of critical arcs for all ingress-egress pairs, which is essentially the problem of determining the weights of the arcs as described by Equation 6.1. Kodialam and Lakshman further show[7] that the set of critical links for a pre-determined set of ingress-egress pairs of nodes can be determined by a single execution of the max-flow algorithm between the ingress-egress pair [42].

For practical implementations, the value of α_{sd} can be chosen to signify the importance of an ingress-egress pair (s, d). The weights can be chosen to be inversely proportional to the max-flow values, i.e., $\alpha_{sd} = \frac{1}{\theta_{sd}}$. In other words,those critical arcs that have lower max-flow values are assigned more weights than the ones with higher max-flow values [42].

The algorithmic steps of MIRA are presented below (adapted verbatim from [42]).

[6]This is not discussed, in detail, in this Thesis. Interested readers are referred to [42] for obtaining further details of the proofs/reasons they use to establish their claims.

[7]We do not discuss this in much detail in this Thesis. Interested readers are referred to [42] for obtaining further details of the proofs/reasons they use to establish their claims.

Algorithm: MIRA

Input

(i) A graph $G(N, L)$, and a set B of all residual link capacities.

(ii) An ingress node a, and an egress node b, between which a flow of D units have to be routed.

Output

(i) A route between a and b having a capacity of D units of bandwidth.

Algorithm

BEGIN

1. Compute the maximum flow values for all $(s, d) \in P \backslash (a, b)$.
2. Compute the set of critical links C_{sd} for all $(s, d) \in P \backslash (a, b)$.
3. Compute the weights $w(l) = \sum_{(s,d):l \in C_{sd}} \alpha_{sd} \quad \forall l \in L$
4. Eliminate all links which have residual bandwidth less than D and form a reduced network.
5. Using Dijkstra's algorithm compute the shortest path in the reduced network using $w(l)$ as the weight on link l.
6. Route the demand of D units from a to b along this shortest path, and update the residual capacities.

END

From the above, we see how MIRA prevents loading the critical links, and routes requests along non-critical links, even if such a path is not a minimum-hop shortest distance path.

6.5.1.2 Limitations of MIRA

Although MIRA was quite influential to the TE community (because it proposed an online algorithm for routing bandwidth-guaranteed tunnels using the idea of minimum interference), it was not "robust". Suri *et al.* [92] identified the limitations of MIRA, as follows:

1. MIRA fails to identify the effect of interference on a cluster of ingress-egress nodes. It just computes the effect of interference on single pairs of ingress-egress nodes.

2. MIRA does not keep track of the expected bandwidth requests between a pair of nodes, thereby rejecting requests even when there might be sufficient residual capacity available to route the request between the affected pair.

3. MIRA is computationally expensive in complex networks where thousands of max-flow/min-cut computations have to be performed for every request.

6.5.2 Summary of Other Selected TE Algorithms

6.5.2.1 PBR algorithm

Suri *et al.* proposed the PBR algorithm, which was designed to work on a group of ingress-egress nodes. The PBR algorithm, however, uses the *traffic profile* of a network in a pre-processing step to roughly predict the future traffic distribution.

PBR has two main phases of processing: (i) the offline pre-processing phase, and (ii) the online routing phase. These are described below in further detail.

Offline pre-processing phase: This phase is based on the solution to the multi-commodity flow computation problem [4] on a set of traffic profiles. The traffic profiles can, typically, be obtained from a number of sources, e.g., the *Service Level Agreements* (*SLAs*) between the customers and network service providers, historical trend statistics, or by suitably monitoring the network for a predetermined period of time.

Using the same notations that were used by the original authors [93, 92], let us assume that we have a capacitated network, $G = (V, E)$, with the function $cap(e)$ denoting the capacity of a link $e \in E$, and the set of links in the network with V vertices. We also assume that we have a table of traffic profiles obtained by using one of the methods mentioned in the last paragraph. Let us denote these traffic profiles by, $(classID, s_i, d_i, B_i)$, where $classID$ denotes the traffic class that an ingress-egress node belongs to, s_i and d_i denote the ingress-egress pairs, and B_i is the aggregated

bandwidth between the above ingress-egress pairs for that class. The term "class" can, typically, be perceived as, for example, having requests between an ingress-egress pair to belong to one class, or as having all requests that has the same ingress or egress node to belong to one class [108]. In multi-commodity flow pre-processing, each of these classes is classified as a distinct *commodity*. The requested bandwidth is treated as an aggregated bandwidth between a pair of ingress-egress nodes of the identified class (as mentioned above), and not as a single flow. Therefore, each profile (and *not* each flow) can be split, and consequently, several flows belonging to a traffic profile, can be routed on distinct paths.

Satisfying all bandwidth route requests may not always be possible. Thus additional edges, called *excess edges*, are added to the network. This is done to avoid the situation where there is no feasible solution to the multi-commodity flow problem. The excess edges are assigned an infinite capacity, so as to discourage routing of flows through these routes as much as possible. All the rest of the network edges are assigned a cost of unity. Thus, the cost function can be expressed as [92]:

$$\text{cost}(e) = \begin{cases} 1, & \text{if } e \text{ is not an excess edge} \\ \text{cost}(e) = \infty, & \text{if } e \text{ is an excess edge} \end{cases}$$

If there are k commodities, the multi-commodity problem to be solved for the transformed network from the last step can be expressed as [92]:

$$\text{minimize} \sum \left(cost(e) \sum_{i=1}^{k} x_i(e) \right)$$

subject to the constraints that (taken from [92]):

1. All edges satisfy the capacity constraints. In other words, if e is not an excess edge, then the following inequality holds true: $\sum_{i=1}^{k} x_i(e) \leq \text{cap}(e)$

2. All nodes conserve the flow for each node, except at the ingress-egress nodes corresponding to that flow.

3. B_i is the amount of commodity i reaching its destination d_i.

The $x_i(e)$ values that can be obtained from the multi-commodity flow computation, are used for pre-allocating the capacity of the edges of the network. The incoming requests in the online phase of the algorithm, described below, will use these capacities as thresholds so as to route flows belonging to the traffic class i.

Online routing phase: The online phase of the PBR algorithm is much simpler than the offline phase. As before, we use the terminologies of the original authors [93] to explain the online phase of this algorithm. The reduced graph, with residual capacities from the offline pre-processing phase, is used as the input to this phase. Let this residual capacity of an edge e of class j be denoted by $r_j(e)$. The online routing phase takes this residual graph to process an online sequence of incoming LSP setup requests (id, s_i, d_i, b_i), where id, s_i, d_i, and b_i are respectively the ID of the request to be routed, the ingress and egress node pairs between which the request is to be routed, and the bandwidth that is to be routed. While routing the requests, given a choice, the requests are routed through the minimum hop shortest paths between the ingress-egress pairs of nodes.

PBR was identified to have the following limitations [108]:

- PBR is based on the assumption of the splitting of a group of flows, even if it is not a single flow. This will still remain problematic when an individual flow has a very high demand.

- The performance of the algorithm will be limited by the accuracy of information provided in the traffic profiles in the pre-processing phase of the algorithm.

6.5.2.2 DORA algorithm

DORA is another TE routing algorithm, proposed by Boutaba *et al.* [14], for the routing of bandwidth guaranteed tunnels. DORA attempts to accommodate more future path setup requests by evenly distributing reserved bandwidth paths across all possible paths in the entire network, thereby balancing the load in an efficient manner. It does so by avoiding, as much as possible, routing over links that have a greater potential to coincide with the link segments of any other path, and those that have low running estimate of the residual bandwidth [14].

DORA is designed to operate in two phases. In the first phase, DORA computes, for every source-destination pair, the *Path Potential Value* (PPV), which stores in an array the potential of a link to be more likely to be included in a path than other links. In other words, the algorithm tries to preprocess a set of links that are more likely to be included among the different paths that packets could travel between any pair of source-destination nodes. The PPV for any pair of nodes (S, D) is denoted as $PPV_{(S,D)}$. If a path could be constructed over a link L for any pair of source-destination nodes, $PPV_{(S,D)}$ is reduced by unity, whereas if different paths could be constructed over the same link L for different source-destination pairs, the value of $PPV_{(S,D)}$ is increased by unity [14].

In the second phase, DORA first preprocesses, and removes those links that have a required bandwidth exceeding the residual bandwidth. It then provides weights to the different links in the network by taking into consideration the PPV arrays, and the current residual bandwidths. Since there are two parameters that may influence the weights that can be assigned to the links namely, the PPV and the current residual bandwidth, they introduce a new parameter called *Bandwidth Proportion* (BWP), which determines the proportion of influence either of these two parameters could have on the weight of the link.

6.5.2.3 Wang *et al.*'s algorithm

Wang *et al.* [109] also proposed an algorithm for setting up bandwidth guaranteed paths for MPLS TE, which is based on the concept of the critical links, the degree of their importance, and their contribution in routing future path setup requests. Thus, like MIRA, their algorithm essentially belongs to the class of minimum interference algorithms. Their algorithm takes into account the position of source-destination nodes in the network. Specifically, for any path setup request, their algorithm first considers the impact of routing this request on future path setup requests between the same pair of source-destination nodes. To do this, they assign weights to the links that may be used by future demands between that pair of source-destination nodes. For any source-destination pair of nodes, (s, d), they compute the maximum network flow, $\theta^{(s,d)}$, between s and d, and $f_l^{s,d}$, which represents the amount of flow passing through link l. Then, for all links in the paths between s, and d, they compute each of those links bandwidth contribution, $\frac{f_l^{s,d}}{\theta^{(s,d)}}$, to the maxflow between s, and d [109].

In addition to the link's bandwidth contribution, since they also consider the link's residual bandwidth $R(l)$, they propose a parameter, $\frac{f_l^{s,d}}{\theta^{(s,d)}R(l)}$. They use this normalized bandwidth contribution of any particular link l to compute the overall weight of a link l as $w(l) = \sum_{(s,d)\in L} \frac{f_l^{s,d}}{\theta^{(s,d)}R(l)}$, where $w(l)$ represents the total weight contribution of all the links between a pair of source-destination nodes. Subsequently, they assign the weights to the different links using the above formula, and then, in the latter phases of their algorithm, eliminate those links that have requested bandwidth for any request greater than the residual bandwidth in that link. They then run Dijkstra's algorithm on the reduced network (with the weights assigned using the formula above) to route a request through the shortest path between a pair of source-destination nodes in a network.

6.6 Proposed Solution

6.6.1 Description

Now we present our proposed solution – an efficient adaptive online routing algorithm for the computation of bandwidth-guaranteed paths in MPLS-based networks, using a learning scheme that attempts to compute an optimal ordering of routes.

This work has two-fold contributions. The *first* is that we propose a new class of solutions other than those available in the literature. The most popular previously-proposed MPLS TE solutions try to find a superior path to route an incoming path setup request. Our algorithm, on the other hand, tries to learn an optimal ordering of paths through which requests could be routed according to the *rank* of the paths in the order learnt by the algorithm using the theory of *Random Races* (*RR*) [64]. Our proposed algorithm, the *Random Races-Based Algorithm for Traffic Engineering* (*RRATE*), is inspired by the computation, and deferred loading of critical links in MIRA [41], and the construction of a feedback function to estimate the feedback response of the Environment to a selected path of SELA [104]. However, RRATE is different in its operation from both MIRA and SELA, and it performs much better than both of these algorithms, as will be seen shortly. In particular, our solution is based on a novel approach using the principle of *Learning Multiple Race Tracks* (LMRT) to achieve the optimal *ordering* of competing candidate paths for selection between a pair of nodes.

The *second* contribution of our work is that we have proposed a routing algorithm that has better performance than the important algorithms in the literature. Our conclusions are based on three important performance criteria: (i) the rejection ratio, (ii) the percentage of accepted bandwidth, and (iii) the average route computation time per request. While some of the previously proposed algorithms were designed to achieve low rejections and high throughput of route requests, they were unreasonably

slow. Our algorithm, on the other hand, in general, rejects the least number of requests, achieves the highest throughput, and computes the routes in the fastest possible time, as compared to the algorithms we used as benchmarks for comparison.

Our proposed solution was inspired by the previous works mentioned earlier in this Chapter, e.g., [104], and [42]. Some of the concepts used in our solution are taken from these previous works.

6.6.2 Random Races and Traffic Engineering Routing

Most of the algorithms that attempt to achieve adaptive TE routing do so by trying to learn the most suitable action (path) to be taken at any particular time instant. These algorithms are, indeed, commendable, and are the basis for exceptionally good techniques such as the SELA [104] and MIRA [41]. We advocate that there is, possibly, a superior strategy. Rather than attempting to merely learn the best action/path, it should be advantageous if a learning mechanism could *rank* the paths in the respective order of their optimality. This problem is far from trivial because unlike the problem faced in the traditional learning paradigm where the solution space is of size r (where there are r actions), in this case, the solution space is of magnitude $r!$.

What we advocate is as follows. The learning mechanism has a current understanding of the ranking of the paths. It proceeds to choose the paths in the order of their rankings till one accepts the request, and permits the traffic to proceed. At this juncture, the learning mechanism treats this admission control response as a reward *for that particular path.* But rather than merely increase the probability of choosing that path, it uses this response to update the understanding of the *optimal ordering.*

Observe that this paradigm now has two consequences. Firstly, the issue of choosing an action based on an action probability vector is not as prominent as it is traditionally encountered in the field of LA. Secondly, the response of the system is used to infer

the information about the *ordering* and not merely about a single action.

Although easy to explain, there is no solution for the general action-ranking problem. Rather, as mentioned earlier, we utilize an earlier reported solution to this problem for the restricted case when the Environment is "suggestive", as was explained in Chapter 3.

In a suggestive Environment, the latter, at every time instant, provides to the learning mechanism a proposal as to what action it thinks is the best. In a non-suggestive Environment the learning mechanism would propose a path or an ordering of the paths, and expect a response. It would have been desirable if a solution for a non-suggestive Environment was available – unfortunately there is none. It turns out, however, that if the problem is modeled correctly, the restricted solution for the suggestive Environment suffices. This is, indeed, because the response that can be inferred can be directly used to rank the actions because this response contains crucial information about the critical links and the maximum expected residual bandwidth. Our experiments also conform to this.

6.6.3 Motivation to the LMRT-Based QoS/TE Routing Solution

Our solution to the QoS/TE routing problem uses RR, the suggestive Environment, and a sequence of responses, which permit the ranking of the paths. We explain each of these below. In this context, we mention that in our case, the "system" would imply a *team* of learning machines interacting with the stochastic graph, and playing a *cooperative* game, where at each time instant the machines are not just choosing the best action, but the ranking of the actions. We shall now clarify how we have achieved this.

The Racers: We propose to station an LM corresponding to every ingress-egress pair of nodes in the graph, whose task is to rank all the outgoing paths (the racers) from an ingress node to an egress node using some critical information[8]. This invokes a game of LMs operating in a sequential fashion. At every time instant, the LM asks the Environment to suggest the best path. The Environment suggests a suitable path from all the possible paths between a ingress-egress pair of nodes. The intention, of course, is that the Environment guesses that the loading of *this* suggested path would probably minimize the number of rejections of the future requests, and maximize the availability of residual bandwidth for potential future requests.

The Suggestive Environment: The Environment changes continuously, and stochastically. These changes are based on a distribution that is unknown to the Racers, but assumed to be known to the Environment. In a religious LM-Environment feedback, the Environment suggests an LM with a signal indicative of the best action at a particular time instant.

The Feedback - Reward/Penalty: In a typical feedback loop, the Environment provides the machine with a reward or a penalty for a particular action. Unfortunately, within this present learning paradigm such a feedback response is not available, and even if it were, its applicability is uncertain. Rather what we want is a strategy by which we can infer the optimality of the paths by processing certain crucial pieces of information about them. In the context of QoS/TE routing, we believe that the pieces of information that are crucial involve the number of critical paths, and the maximum residual bandwidth. We shall use a function of various statistics of the available paths to infer how the system perceives them, and from them compute a ranking based on

[8]SELA [104], for example, uses some of the information contained in the residual bandwidths of the paths, and MIRA [41] uses pieces of information contained in critical paths.

their performance in the RR. The statistics that we infer from the paths involve two components, as explained below:

- *Information about the critical paths*: The determination of the critical paths is useful to reduce the excessive loading of those paths that are more likely to be congested due to future incoming requests. In our RR-based solution, the non-critical paths are loaded prior to the critical ones.

- *Information about the residual bandwidths*: Residual bandwidth determines the amount of bandwidth remaining on a path if a flow is routed through it. The determination of the residual bandwidths helps to obtain an "estimate of the potential utilization" of the different paths between an ingress-egress pair. The idea is that the lesser the residual bandwidth in a path, the less likely that path would be as a candidate path for routing future requests.

Rather than use each of these pieces of information directly, clearly, the optimality of the ranking will be a function of the two, and we have chosen to use a simple linear function weighted by two user-defined constants k_1 and k_2. We will see in the next Section how we use these pieces of information to construct a feedback function, $\beta(t)$, which helps to determine the feedback to the path suggested by the Environment.

6.6.4 Benchmark Algorithms for Comparison

For evaluating the performance of our proposed algorithm, we selected five popular algorithms as benchmarks for comparison:

(i) Shortest Path Algorithm

(ii) Shortest Widest Path Algorithm [106]

(iii) Widest Shortest Path Algorithm [34]

(iv) Minimum Interference Routing Algorithm [KKL00]

(v) Stochastic Estimator Learning Automata Routing Algorithm [104]

In terms of the practical use in routers, the most widely used algorithm for routing in MPLS, and other TE domains are algorithms (i), (ii), and (iii). Algorithms (iv) and (v) are still primarily of interest in the research community, and have seldom been deployed in practice, to our knowledge.

6.6.5 The LMRT Solution to the Problem

Our solution to the QoS/TE Routing problem is as follows. We first station an LM corresponding to each ingress-egress (IE) router pair. For each IE pair LM, we maintain an r-action vector, $A = \{\alpha_1(t), \alpha_2(t), \ldots, \alpha_r(t)\}$ representing the set of actions (or the possible paths between an IE pair) offered at time instant t by the Environment to the LM maintained for the IE pair. When the LM enquires of the Environment, the Environment responds by suggesting one of the paths as explained presently. The best path has the highest probability of being suggested, but an imperfect (noisy) Environment may also suggest other less optimal paths, although with a correspondingly lower probability. The output of a system, at every time instant, is the permutation of the paths that is possibly the optimal ordering of the paths at that instant. On convergence, the RR for each of the IE pairs reports an ordering which it considers to be the optimal ordering of the paths based on a certain criterion described below. At that time, the system manager will asymptotically choose those actions that are learnt by the LM stored for each of the IE pairs. Thus in this model, the number of LM required is equal to the number of possible IE pairs in that network.

We propose that if at time instant t, the action α_i is suggested by the Environment, the environmental feedback $\beta(t)$ corresponding to this suggested action is computed as follows: If C_L is the number of critical links (described below) in the path, and R

is the maximum expected residual bandwidth in the path if the request were routed through that path, then $\beta(t) = -k_1 C_L - \frac{k_2}{R}$, where $k_1, k_2 > 0$.

The purpose is to minimize the number of critical links in the path, however, by also weighting this goal against the residual bandwidth of the path.

The algorithm intends to protect the *critical links* in a network from excessive loading, and thereby reducing the chance of rejection of future requests. The critical links are identified by the algorithm to be those that, if congested because of heavy loading, might lead to the rejection of requests. The determination of the critical links is based on the concepts of the *maxflow* and the *mincut* computations [4] explained earlier.

6.6.6 The RRATE Algorithm

The RRATE algorithm is formally presented below. The algorithm is run on a network having a certain number of nodes, and links connecting certain pairs of nodes. A set of incoming bandwidth routing requests are run through the network. RRATE tries to efficiently route the requests. The efficiency of RRATE is evaluated on different networks, and their results are presented afterwards.

Algorithm: RRATE

Input
A network with incoming bandwidth routing requests

Output
Requests routed through different paths

Parameters
N: A predefined number, which denotes the maximum number of rewards any of the actions can assume.

Method
BEGIN
Offline operation
1. Determine the k-shortest paths between each of the IE router pairs.
2. Maintain an RR corresponding to each IE-pair. Each path corresponds to the different actions of the racers.
3. Specify a threshold bandwidth utilization (ρ_{Thresh}) that any link in the network can have at any time instant. This can better be specified as a percentage of the maximum possible bandwidth of a link, rather than a fixed value of bandwidth.

... (Continued to next page)

Algorithm: RRATE (Continued from previous page)

Online operation

Pre-convergence

4. Assume that $x_i(n)$ represents the number of rewards received by action i at the n^{th} time instant. Initially, set, $\forall x_i$, $x_i(0) = 0$.
5. For each incoming bandwidth setup request for each path between an IE pair, compute $\beta(t)$ using $\beta(t) = -k_1 C_L - \frac{k_2}{R}$
6. Select the action with the highest value of β.
7. If that action is accepted by the CAC, and the expected utilization is less than the threshold bandwidth utilization specified for all the paths in the network (ρ_{Thresh}), then select the action and route the request through that path.
8. Else if the action has an expected utilization greater than the threshold utilization, choose the action with the next highest value of β.
9. Increase $x_i \leftarrow x_i + 1$, if the call is accepted for the path i, otherwise repeat Steps 7 and 8 with the next highest value of β.
10. If all the actions have been tried without any of them being selected, then reject the request.
11. Repeat Steps 5-10 until any one of the values of x_i equals N.
12. Sort the x_i values in the decreasing order of their respective values.

Post-convergence

13. Choose the actions one at a time in the order determined in Step 12.
14. Route requests through the first path of those allowed by the CAC.

END

6.6.7 Experimental Details

6.6.7.1 Experimental Design

Several interesting experiments were designed to evaluate the performance of our proposed algorithm, RRATE, with respect to the five other benchmark algorithms: SP, SWP, WSP, SELA, and MIRA. In this Thesis we report only some of the results. Also, for the purpose of the evaluation of RRATE, we have limited ourselves to scenarios where there are no link/node failures in the network. We report the results of the

following sets of experiments:

1. **Experiment Set 1:** Comparison of the performance of RRATE with the chosen benchmark algorithms, under the following network conditions:

   (i) Moderately loaded networks of fixed size and density using three sets of requests of varying sizes.

   (ii) Marginally loaded networks (compared to (i)) on a fixed size network using three sets of requests of varying sizes.

   (iii) Heavily loaded networks (compared to (i)) on a fixed size network using three sets of requests of varying sizes.

2. **Experiment Set 2:** Comparison of the variation of the performance of RRATE with the chosen benchmark algorithms, and with the variation of the network density. Since in our abstraction we represent a network using a directed graph, we increase the density of the network by increasing the number of edges in the graph.

Network Topologies used: The topologies used in the study were generated using the BRITE network topology generator (http://www.cs.bu.edu/brite/), and can be found in http://www.scs.carleton.ca/~smisra_/TE/topol. Table 6.1 below summarizes the size of some of the network topologies used in the study. In the interest of completeness, we show in Figure 6.3, some of the relevant fields that BRITE yields for a topology having 15 nodes and 23 edges.

Request Sequence Generator: This module is a random generator of a sequence of bandwidth requests. According to Zipf's law [113], if a is a parameter close to unity,

Table 6.1: Network topologies used in the experiments.

Topology	Number of Nodes	Number of Links
1	30	57
2	15	23
3	15	27
4	15	39
5	15	78
6	15	87
7	15	95

and P_n is the frequency with which an n^{th} ranked item occurs then, $P_n = 1/n^a$. Thus, the generator generates small request sizes more frequently than larger ones.

6.6.8 Performance Metrics

To compare the performances of our proposed algorithm, RRATE, with those of the other algorithms, the following performance metrics were used.

1. **Rejection ratio of requests:** It measures the ratio between the number of incoming bandwidth setup requests that are dropped because a suitable tunnel could not be found to route the request between an ingress-egress pair of nodes, and the total number of requests routed.

2. **Percentage of accepted bandwidth:** It measures the percentage of the total amount of bandwidth that is routed by the algorithm.

3. **Average route computation time per request:** It measures the average amount of time it takes for the algorithm to determine the route through which a request could be routed, and actually route it.

We choose the above performance metrics because we believe they characterize the most important aspects of any QoS routing algorithm. The idea is that the best routing

Topology: (15 Nodes, 23 Edges)
Nodes: (15)

NodeID	xpos	ypos	indegree	outdegree
1	257	507	3	3
2	149	840	9	9
3	966	477	8	8
4	552	925	3	3
5	296	53	6	6
6	207	79	2	2
7	199	501	4	4
8	316	367	3	3
9	49	840	1	1
10	162	98	1	1
11	987	372	1	1
12	533	170	1	1
13	815	434	1	1
14	488	780	1	1

Edges: (23)

EdgeID	From	To	Euclidean Length	Delay	Bandwidth
1	2	1	350.075706097981	1.16772686155427	0.5628503508978068
2	3	0	767.387776811697	2.55973009438315	1.6702838524325636
4	5	1	455.672031180321	1.51995828787768	0.6151051530610663
5	1	3	709.634412919779	2.36708560867058	1.3713760439847438
7	3	2	894.012304165887	2.98210405335109	0.5085794120878812
9	0	2	630.201555059966	2.10212611506046	0.6369035611898777
13	4	2	411.866483219987	1.37383870817719	0.6490692141241456
14	7	5	458.380846022169	1.52899392159548	0.5604856407936213
18	11	2	959.827067757520	3.20163847403232	0.7050059471747464
21	3	7	767.375397051534	2.55968879994817	0.6932073822926373
22	8	2	501.615390513488	1.67320883874099	0.5172866591871618
23	2	5	800.611016661649	2.67055089378415	0.9866115561128445
27	8	5	314.636297969576	1.04951372048717	0.6458147206142891
28	2	7	342.667477301246	1.14301567019823	1.9126407392278344
30	6	3	857.021003243211	2.85871435512634	0.5351001034016294
31	5	4	908.801408449612	3.03143519524301	0.6911380490460441
33	13	6	704.051844681909	2.34846416543911	0.7172085711117819
34	3	9	986.234252092270	3.28972335952584	0.5820176858644273
35	12	3	530.789977297989	1.77052478517651	0.9518159508698532
36	10	8	309.962901005910	1.03392494619030	1.0763454528089278
37	14	2	344.268790336852	1.14835707553674	1.4587413993228968
38	3	5	792.890913051726	2.64479940002935	1.3465965652301161
40	4	7	551.710975058499	1.84030972206278	1.3751569126426475

Figure 6.3: A network topology (shown in the BRITE output format) for a graph with 15 nodes, and 23 edges.

algorithm should be able to route the maximum amount of bandwidth by taking the least amount of route computation time, and by dropping the least number of requests. However, for the sake of interested readers, we provide below other metrics that could be used as well:

(a) Number of accepted requests

(b) Amount of rejected bandwidth

(c) Computational time complexity of the algorithm

(d) Computational space complexity of the algorithm

(e) Bandwidth duration product, representing the earned network revenue.

Options (a) and (b) can be computed using the metrics (1) and (2) above. We consider the average route computation time to be more important for a QoS routing algorithm than options (c). Option (d) is not as important because of the decreasing prices of the storage devices. Option (e) is used in [104], but we believe that it should not be used as a *fundamental* QoS performance metric.

6.6.9 Experimental Results

6.6.10 Experiment Set 1

Our first set of experiments compared the performances of RRATE with the five selected benchmark algorithms on a topology of fixed size and density using three different sets of LSP setup requests under different loading conditions. Topology 1 of Table 6.1, containing 30 nodes and 57 links, was used in the study. The chosen requests were of sizes 1,000, 2,500, and 5,000 respectively.

Tables 6.2-6.4 respectively show the performance results of the algorithms under three different loading conditions:

(i) Moderate

(ii) Heavy compared to the loading conditions in (i), and

(iii) Marginal compared to the loading conditions in (i)

Observations on rejection ratio: In terms of the rejection ratio (signifying the rate of rejections) of the algorithms, it can be seen that RRATE almost consistently outperforms all the other algorithms (see Table 6.2). For example[9], for a request size of 1,000 requests under moderately loading conditions, the rejection ratio of RRATE is 0.368, whereas that of MIRA (which is well-known in the literature for its strong capabilities for minimizing the number of rejections) is 0.430. This shows an improvement of almost 15%[10]. On the other hand, when all the algorithms were run on a request size of 5000 under marginally loaded network conditions, RRATE attains an even better performance improvement of about 21% over MIRA. In other cases, RRATE achieves, on an average of $8 - 12\%$ performance improvement over MIRA. The SELA routing algorithm shows slightly better performance as compared to MIRA in terms of the number of rejections. However, it is interesting to note that our algorithm performs even better than SELA. Among WSP, SWP, and SP, WSP is found to perform the best, while SWP was the worst. But, in general, our algorithm performs considerably better than the WSP. For example, under high loading conditions on a request size of 1,000, RRATE achieves about 17% improvement over the WSP.

When we compare each of these algorithms, the general observation (with certain exceptions) is that the rate of rejections increases with the increase in the size of the requests. In other words, it appears that the percentage of rejections on a set with 2,500 requests is more likely to be greater than a set with a size of 1,000 requests. For example, from Table 6.2, we see that under moderate loading conditions, RRATE

[9]All such comments in this paper are based on generic "overall" observations. There are, of course, exceptions to these observations.

[10]Although in an absolute sense, this figure may not sound too appealing, this is a significant improvement when compared with the results presented in the literature (e.g., [14], [37], [92], [109]).

rejects 36.8% of the requests on a set of 1,000 requests, it rejects 41.8% of the requests on a set of 2,500 requests, and it rejects 42.8% of the requests on a set of 5,000 requests. Similar observations can also be made for a network under heavily and marginally loaded conditions.

Under marginally loading conditions, on the average, RRATE achieves better performance improvement over the other algorithms than under heavier loading conditions. For example, under marginal loading conditions, the maximum performance improvement of RRATE over MIRA is 21%, whereas that under normal and high loading conditions, it is approximately 14% and 9% respectively.

Observations on percentage accepted bandwidth: In terms of the percentage of the accepted bandwidth by the different algorithms, under all loading conditions, Table 6.3, shows that, on an average, RRATE accepts more bandwidth compared to the other algorithms. For example, the percentage accepted bandwidth of RRATE is 56.19%, whereas that of MIRA is 51.14%, which is a 10% improvement of performance of RRATE over MIRA. The improvement over SELA is lower than that over MIRA. However, the improvement over WSP, SWP, and SP is quite significant.

It can also be observed that, in terms of the percentage of accepted bandwidth, the amount of improvement of RRATE over the other algorithms is more when the set of requests is smaller sized. For example, from Table 6.3, we see that RRATE achieves about 10% improvement on a set of 1,000 requests, 6% improvement on a set of 2,500 requests, and about 3% improvement on a set of 5,000 requests.

Observations on the average route computation time per request: The most interesting improvement of RRATE over the other algorithms is with respect to the average route computation time per request. MIRA is observed to be the slowest of all algorithms that we had considered in the study. With respect to the route computation time

Table 6.2: Comparison of the *rejection ratios* of the algorithms at different loading conditions (moderate/heavy/marginal) on a network of 30 nodes and 57 links using a set of 1,000 requests, 2,500 requests, and 5,000 requests

Loading Condtion	Num. Req.	RRATE	MIRA	SELA	WSP	SWP	SP
Moderate	1000	0.368	0.430	0.404	0.447	0.612	0.475
	2500	0.418	0.443	0.421	0.475	0.626	0.491
	5000	0.428	0.439	0.470	0.450	0.619	0.470
Heavy	1000	0.568	0.628	0.581	0.625	0.687	0.643
	2500	0.655	0.670	0.668	0.676	0.728	0.683
	5000	0.569	0.580	0.581	0.617	0.678	0.626
Marginal	1000	0.069	0.078	0.112	0.150	0.534	0.165
	2500	0.081	0.096	0.128	0.158	0.535	0.184
	5000	0.085	0.108	0.125	0.176	0.558	0.198

(Table 6.4), RRATE is about 200 – 600% faster than MIRA, about 30 – 35% faster than SELA, and almost similar in speed with WSP, SWP, or SP. MIRA seems to work better under heavy loading conditions. It was also observed that whereas the average route computation time for MIRA seems to increase with the increase in the size of the requests, RRATE seems to achieve consistent performance irrespective of the size of the requests.

Overall observations: From the study of a graph of fixed size and density, the general observation is that RRATE outperforms the other algorithms with respect to the performance criteria we considered in this study. RRATE can reject fewer requests and accept more bandwidth than the other algorithms, in much faster (MIRA, SELA), or is of comparable time (WSP, SWP, and SP). This is a significant improvement because MIRA is well known in the literature to have strong capabilities in reducing the number of rejections, but RRATE outperforms MIRA with respect to the number of rejections and the computation time.

Table 6.3: Comparison of the *percentage of accepted bandwidth* of the algorithms at different loading conditions (moderate/heavy/marginal) on a network of 30 nodes and 57 links using a set of 1,000 requests, 2,500 requests, and 5,000 requests

Loading Condtion	**Num. Req.**	**RRATE**	**MIRA**	**SELA**	**WSP**	**SWP**	**SP**
Moderate	1000	56.19	51.14	54.46	52.81	35.33	51.14
	2500	54.22	51.10	54.40	50.94	35.85	49.48
	5000	55.07	53.39	51.10	53.13	35.73	53.81
Heavy	1000	30.89	26.69	29.84	28.71	25.79	28.01
	2500	29.01	27.89	28.29	29.41	23.74	28.39
	5000	38.01	36.35	38.81	36.08	30.04	35.29
Marginal	1000	87.16	86.54	83.54	82.73	44.07	81.30
	2500	88.62	87.59	85.72	83.00	44.05	80.20
	5000	88.72	87.25	86.10	81.10	44.15	79.12

Table 6.4: Comparison of the *average route computation time per request* of the algorithms at different loading conditions (moderate/heavy/marginal) on a network of 30 nodes and 57 links using a set of 1,000 requests, 2,500 requests, and 5,000 requests.

Loading Condtion	**Num. Req.**	**RRATE**	**MIRA**	**SELA**	**WSP**	**SWP**	**SP**
Moderate	1000	0.003	3.447	0.326	0.004	0.002	0.002
	2500	0.003	3.359	0.326	0.004	0.002	0.002
	5000	0.003	3.555	0.324	0.003	0.002	0.002
Heavy	1000	0.003	1.908	0.316	0.002	0.003	0.002
	2500	0.003	1.940	0.307	0.002	0.002	0.002
	5000	0.004	2.794	0.362	0.003	0.002	0.002
Marginal	1000	0.003	6.384	0.311	0.002	0.002	0.002
	2500	0.004	6.553	0.349	0.003	0.002	0.002
	5000	0.003	6.572	0.376	0.003	0.003	0.003

6.6.11 Experiment Set 2

The results reported in the first set of experiments were for a graph of fixed size and density. In Experiment Set 2, we were interested in observing the variation in performance of RRATE and the other algorithms, with the variation of the network density. Topologies 2-7 listed in Table 6.1 were used in the study. The results reported here are for experiments run on a request size of $1,000$[11].

Figure 6.4 shows the plots of the rejection ratios of all the six algorithms considered in the study. We see from the figure that there is a general downward trend of the plots for all the algorithms signifying that as the network becomes denser the rate of rejections decreases. This is intuitive because as the network becomes denser, the algorithms are more likely to find alternate paths to route requests, thus reducing the chance of rejections due to the non-availability of suitable paths between a pair of ingress-egress nodes. However, one could argue that it is the structure of the network, and therefore the paths connecting the different nodes, and not the number of edges present in a network, that determines whether alternate paths can be found for routing LSP setup requests. Perhaps, that is the reason why the downward trends in the slopes of the plots are not consistent.

From both Figures 6.4 and 6.5, we see that, in general, RRATE achieves better performance as compared to the other algorithms for all the six different network topologies considered in this study. This observation is consistent with our observation in Experiment Set 1, where we used only a single network of fixed size and density.

Observations similar to the above are seen also with respect to the average route computation time per request (Table 6.5). RRATE consistently performs better than MIRA and SELA, and performs similar to those of SWP, WSP, and SP when the average route computation time per request is the criterion.

[11]Similar trends were observed for sets with $2,500$ and $5,000$ requests. These are omitted in the interest of brevity.

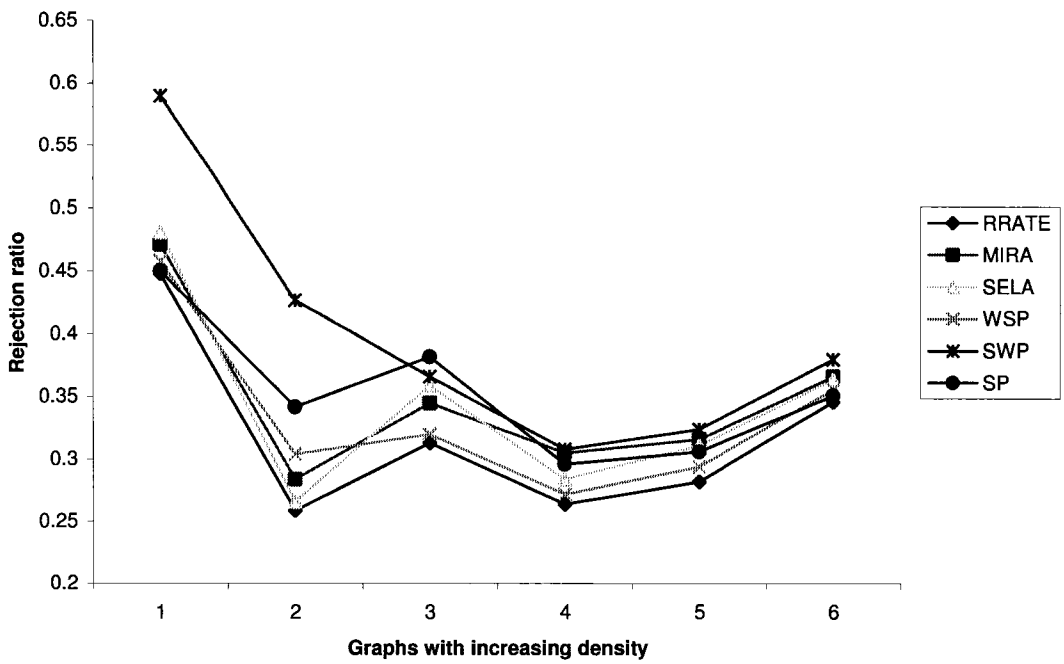


Figure 6.4: Comparison of the *rejection ratios* of the different algorithms with the variation of the network density. In this plot, the density of the network increases along the abscissa. The networks considered are the ones whose topology are listed as 2 to 7 in Table 6.1.

Table 6.5: Comparison of the *average route computation time per request* (in seconds) with the variation of the network density. Here a graph topology of higher ID is denser than the corresponding topology of lower value.

Topology ID	RRATE	MIRA	SELA	WSP	SWP	SP
2	0.003	0.236	0.029	0.002	0.002	0.001
3	0.002	0.46	0.039	0.003	0.002	0.002
4	0.003	0.723	0.053	0.004	0.002	0.001
5	0.004	2.126	0.066	0.002	0.003	0.002
6	0.006	2.478	0.068	0.002	0.003	0.003
7	0.005	2.653	0.07	0.003	0.003	0.002

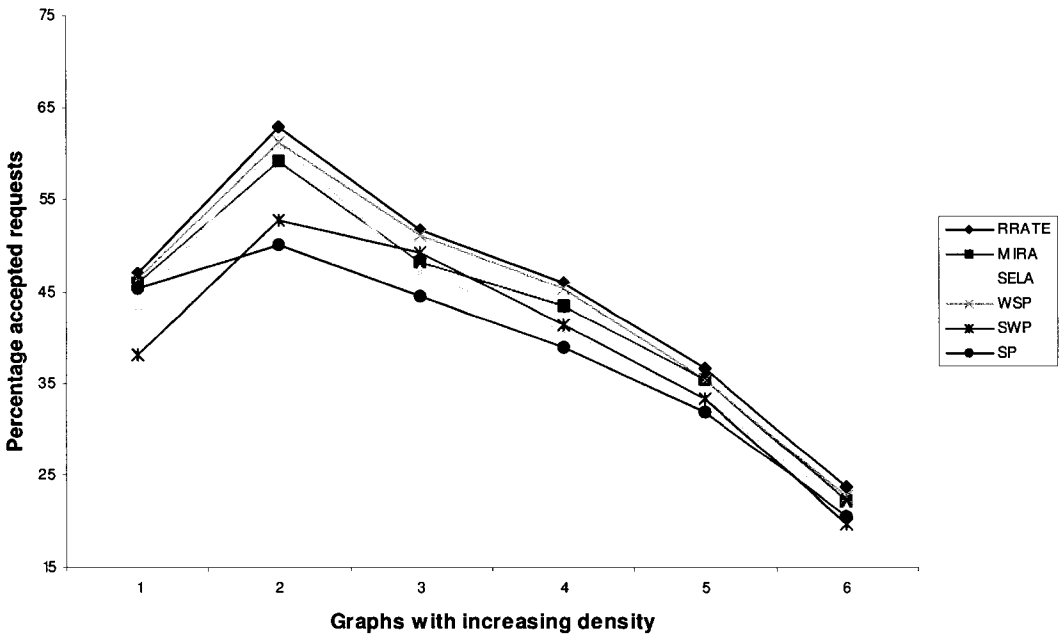


Figure 6.5: Comparison of the *percentage accepted requests* of the different algorithms with the variation of the network density. In this plot, the density of the network increases along the abscissa. The networks considered are the ones whose topology are listed as 2 to 7 in Table 6.1.

6.6.12 Conclusions

In this Chapter, we first discussed the importance of QoS routing, and TE in networks, and described, with examples, few popular conventional QoS routing and TE algorithms.

We then proposed a new TE algorithm using the concepts of RR. We experimentally compared its performance with five other algorithms, and demonstrated that RRATE consistently performs considerably better than the existing algorithms with respect to three important performance comparison criteria.

As future work, we plan to test the performance of our algorithm on actual, typical, atypical, and more complex networks. Although the three performance criteria cho-

sen in our work are fundamental for evaluating the performance of any TE algorithm, we also plan to test the performance of our algorithm with respect to other performance criteria. Our algorithm was tested using simulation on synthetic networks only. Although synthetic networks allow us the flexibility to consider different network scenarios, it would be interesting to see how our algorithm would perform on real networks under real traffic conditions. Finally, in our study we have considered idealistic situations where there are no network link failures. In our future work, we are interested to consider situations were there are realistic link-ups and downs occurring in a network. We would be interested in studying how our algorithm can perform rerouting of paths if a desired path fails.

Chapter 7

Fault Tolerant Routing Algorithms for Mobile *Ad Hoc* Networks

7.1 Introduction

Mobile *Ad Hoc* Networks (MANETs) are characterized as a cooperative engagement of mobile nodes forming networks with continuously changing infrastructures, with the absence of centralized network managers, access points, fixed base stations, or a backbone network for controlling the network management functions. They do not possess designated routers for making routing decisions. All nodes in MANETs take part in routing by acting as routers for one another. However, several hops are normally needed in such networks for transmission of data from one node to another because of the limited wireless transmission range of operation of the mobile nodes [16, 61, 77].

The above-mentioned characteristics of MANETs, particularly those arising due to the mobility of nodes, and the continuously changing network infrastructures, pose several challenges. Due to the continuously changing infrastructures, the routes that were once considered to be the "best" routes may no longer remain the same at a later

time instant. This, therefore, requires a continuous recomputation of routes, and there is no permanent convergence to a fixed set of routes in such networks. So, any routing protocol that needs to operate in MANET network environments should take these issues into consideration [16].

Designing routing protocols poses further challenges when one needs to design routing schemes in the presence of adversarial environments in MANET networks. This is what we address in this Chapter. Specifically we discuss fault-tolerant routing schemes where there are malfunctioning nodes in the network. Most existing MANET protocols were postulated considering scenarios where all the mobile nodes in the *ad hoc* network function properly, and in an idealistic manner. However, adversarial environments are common in MANET environments, and there are misbehaving nodes that degrade the performance of these routing protocols [111]. The need for fault tolerant routing protocols was identified to address routing in adversarial environments in the presence of faulty nodes by exploring network redundancies in networks [110, 111].

Despite the challenges that we have seen above, it is worthwhile to note a few applications of MANETs which have made them popular. One of the popular application domains of MANETs is communications in moving battlefields [61]. Other applications may be found in rural regions where building up fixed wireline or wireless infrastructures can be costly or difficult.

Our only contribution in this Chapter is in proposing a weak-estimation-based learning scheme for route estimation in MANETs. Other works mentioned in this Chapter were either taken directly from or inspired by the previous works such as, [110], [111].

7.2 Mobile Ad hoc Network Routing Protocols

For routing, transmission of data from one node to another is direct if the source, and destination nodes are neighbors (i.e., they are within the wireless range of each other), or indirect through a series of multiple hops (with intermediate nodes between the source, and the destination nodes serving the purpose of routers for relaying the information in between), if those source and destination nodes are not within their range of operation [61]. The dynamic nature of the topology of MANETs created due to the continuously moving in and out of nodes in the networks makes such routing considerations difficult. The following characteristics [61] of MANETs make the routing in them further challenging

1. The terrain in which the mobile nodes may operate in MANETs may pose to be hostile with hazardous conditions subjecting the nodes, and the links between them subject to frequent node failures [61].

2. The medium of transmission of information in MANETs is wireless. Wireless media are unreliable, insecure, and are very susceptible to different kinds of errors and unwanted noises [61].

3. MANETs operate with battery-powered nodes, which are normally low powered, and resource constrained. If the region of operation of the nodes is in a hostile terrain, the frequent recharging of the nodes may not always be feasible. Thus all routing algorithms should be energy-efficient, of low complexity, and capable of operating under limited bandwidth [61].

The different types of errors [61] that can occur in MANETs are listed below:

1. Transmission errors

2. Node failures

3. Link failures

4. Route breakages

5. Packet loss due to congested nodes/links

MANET routing protocols available currently may be classified into two categories [61]:

1. Unipath routing protocols

2. Multipath routing protocols

These are further described below.

7.2.1 Unipath Routing Protocols

In unipath routing protocols, the transmission of messages between a source-destination pair of nodes takes places through a unique path. All the unipath routing protocols may be classified to be either table-based, or to be of an on-demand nature [61]. Table-based protocols are characterized by their ability to maintain routing tables that store information about routes from one node in the network to the rest of the others. Obviously, this requires that the nodes in the network maintain the table up-to-date by exchanging routing information between the participating nodes. Although, in general, the table-based protocols may be easy to implement, the major limitation with these protocols is that due to the highly mobile, and dynamic nature of *ad hoc* networks, the maintenance of routing information in these table is challenging [61].

On-demand routing protocols, on the other hand, alleviate the above problems, and make routing in them more scalable to highly dynamic, and large networks. As the name suggests, on-demand routing protocols are characterized by the computation of routes on an "as-required" basis. In on-demand routing protocols, there is initially a

route discovery phase in which a route is found between two nodes. The route discovery phase is normally followed by a *route maintenance* phase in which a broken link in a route is repaired, or a new route found [61, 77].

There are many unipath routing protocols that have been proposed (e.g., [39], [77]). Of all the different protocols that are available in the literature, the *Ad Hoc On-Demand Distance Vector* (AODV) routing protocol[1] [77], and the *Dynamic Source Routing* (DSR) protocol [39] are the most popular ones. We briefly discuss these protocols below, with enough depth so as to set the context for the fault-tolerant routing problem we discuss in this Chapter.

7.2.1.1 AODV

As the name suggests, AODV is classified as a unipath on-demand distance vector routing protocol. Therefore it functions by using both a route discovery phase, and a route maintenance phase. It uses multihop routing in the intermediary nodes between the source-destination nodes. In AODV, every mobile node functions as a specialized router. Routing tables are maintained in the intermediary nodes, with routing information obtained on an "as-required" basis with no (or little) assumption of the presence of periodic advertisements by the nodes [61, 77].

AODV is shown to be scalable with the increase in the number of mobile nodes in a MANET. It is characterized by its ability to provide loop-free route information while maintaining broken links by repairing existing links or introducing new ones. Because there is no assumption on the presence of periodic advertisements by the nodes, there is little requirement on the amount of bandwidth that should be available to the mobile nodes as compared to protocols that require the presence of advertisements. Finally, it is worth mentioning that AODV works under the assumption that the links are

[1]There are multipath extensions of the AODV protocol that have also been proposed. See, for example, [52], or [107]

symmetric, and the communication can be synchronous, meaning that both the nodes on either side of a link are capable of talking to each other [77].

Perkins and Royer [77] observed that there are normally nodes and paths in a network that are not frequently active. Those nodes not only seldom maintain any routing information, but also seldom take part in the periodic routing advertisements of routing information. Furthermore, two nodes really need to share routing information only when they need to communicate with each other, or one of them is acting as an intermediary node to relay information destined to another node in the network [77]. There are many ways to determine the local connectivity amongst the mobile nodes. One of the most common ways is by sending local, and not system-wide, broadcast *hello* messages. This will help the routing tables maintained by the nodes in the neighborhood to be updated quickly, and the response time to be optimized to local movements thereby providing fast response time for new route establishment requests [77].

In AODV, there are primarily two phases of operation: (1) the *path discovery* phase, and (2) the *path maintenance* phase [77].

When one node needs to communicate with another node in a network for which there is no routing information in its table, the *route discovery* phase is triggered. The source specifies the destination node to which information needs to be transmitted, and floods the network with a *route request* (RREQ) packet, which contains the information about the source address, the source sequence number, the broadcast identification number (which is incremented every time the source node starts a new route discovery request), the destination address, the destination sequence number, and the hop count. Any of the nodes that receives the request checks to see if it is identified as the destination node by the RREQ packet, or if it can serve as an intermediary node to transmit information to another node in the network. If that is the case, that node generates a unicast *route reply* (RREP) packet that is sent back along the reverse path

in which the RREQ packet was originally sent by the source node. Once the source receives the RREP packet, it then knows where to transmit the packet. If none of the above cases holds true, i.e., the node that received the packet is neither the destination node, nor can it serve as an intermediary node to the destination node, it broadcasts the RREQ packet again. Obviously, by doing so, multiple copies of a RREQ packet may be received by the nodes in the network. Those multiple copies are discarded [61, 77].

The *route maintenance* phase is triggered whenever a broken link is detected by any node, and when that node attempts to forward a packet to the next-hop. In the route maintenance phase, once it is determined that the next hop is unreachable, the upstream node sends an unsolicited RREP packet with a new sequence number which is greater than the previously known sequence number by unity. It also sends a hop count of ∞ to all the neighboring upstream nodes, which in turn replay that information to their active neighbors, until all active source nodes are notified [77].

Once a notification is received of a broken link, the source node starts a discovery process only if it is determined by that node that there is a need for the identification of a route to the destination node. The source node makes some decision about whether or not it wants to rebuild an alternative route to the destination node (because of the broken link). If it does, a RREQ packet is sent out with a destination sequence number which is greater than the previously known sequence number by unity [61, 77].

In a summary, AODV sends broadcast discovery messages only when required, distinguishes between neighborhood detection and general topology maintenance, and selectively disseminates information about changes to local connectivity only to those nodes that might need the topology/connectivity change information [77].

7.2.1.2 DSR

Like AODV, DSR is a unicast dynamic on-demand routing protocol. It is a source routing protocol, where the source explicitly provides a packet with the complete information of the route to follow, which is used by the intermediary nodes to forward the packet to the right destination node [61].

DSR only routes packets between hosts that want to communicate with one another. Like AODV, DSR also has a *route discovery* phase, and a *route maintenance* phase. When two hosts need to communicate with each other, the sender host determines a route based on the information stored in its cache, or based on the results of a route discovery, depending on whether or not the information about the destination node is already available to the source node [39].

Briefly, in DSR the mechanism of transmission of a packet from a source node to a destination node requires that the sender determines, and stores in the packet's header the *source route*, where the address of each host in the network is explicitly provided until it can reach the intended destination node. The source finds out the complete route to the destination from a *route cache* that stores the routing information to different nodes in the network. If such an entry is found, the sender uses this route to send the packet. If not, a route discovery exercise, similar to the one discussed for the AODV protocol is initiated by the source route. After the next destination is successfully identified, the packet is then sent to the first hop in the identified sequence of nodes by the source. The first hop node first determines whether it is the final destination. If it is, the packet is considered to be delivered. If not, the next hop is scanned from that sequence of identified nodes to the destination, and the packet is forwarded to the next identified hop. The process continues until the packet is considered to be delivered [39].

Like in AODV, a route maintenance exercise may be initiated whenever a broken

link is detected, which, in turn, may be because any of the nodes along a route fails or is powered down. In such a case, an error message is relayed back to the source node with the information of that particular link which failed. Each of the intermediate nodes (including the source node) that receives that error message deletes all the routes containing that link from their route cache. A route discovery may be initiated afterwards to find out new routes [39, 61].

DSR is characterized by its ability to rapidly adapt itself to routing changes in environments where there are frequent and rapidly occurring host movements. One of the important aspects of DSR is that there is no requirement for the periodic route advertisements, as is frequently required in many routing protocols. This reduces the overall overhead on the network bandwidth, especially because most mobile nodes in *ad hoc* networks are operated over same battery power, and there are often situations in *ad hoc* networks when there are no periodic routing advertisements taking place [39]. Hence it has popularized itself as a suitable protocol for *ad hoc* networks.

7.2.2 Multipath Routing Protocols

Multipath routing protocols proposed in the literature (see, for example, [52], [67], [107]) are of different types, of which some are based on the ideas of AODV and DSR. However, any multipath routing protocol shares a common characteristic of multipath routing, i.e., they discover multiple routes between a pair of source-destination nodes. The multipath routing protocols take advantage of the inherent redundancy observed in networks to be able to find multiple routes from on source node to a destination node. This becomes advantageous for *ad hoc* networks because they are characterized to be very dynamic, and unpredictable in nature [61].

In multipath routing, multiple redundant packets are sent along different paths between a pair of source-destination nodes. This brings in more reliability in information

transmission [112], meaning that there is a much greater chance than in unipath routing for guaranteeing that at least one of the paths will be able to successfully delivery the packet thereby being very successful as a fault-tolerant routing algorithm providing route resilience when there are route failures in the network. However, the disadvantage of multipath routing is that when redundant packets are sent through different routes, they introduce an unnecessary overhead on the capacity of the network [61, 87]. This is disadvantageous especially when we take into account the fact that energy-efficiency is an important concern in wireless *ad hoc* networks [87]. This is because most mobile nodes in such environments are battery powered, and are thus resource constrained.

Some of the multipath routing algorithms are also capable of providing load balancing in the network by carefully selecting a mechanism to split traffic along different routes so as not to overload any single route. This is quite often advantageous in wireless network environments as it might sometimes be difficult to guarantee the reservation of a large amount of bandwidth through a single path, whereas it might be possible to reserve small amount of bandwidth over multiple routes through many paths taken together [61].

The multipath routing algorithms, in general, have three phases: *route discovery, route maintenance*, and *traffic allocation.* The overall route discovery and route maintenance strategies in multipath routing are similar to those used in unipath routing, except that in a multipath routing protocol multiple routes are discovered or maintained between a pair of source-destination nodes [61].

Two important issues arise in multipath routing, viz., the number of paths that would be considered to be optimal, and the selection mechanism of the paths. Nelakuditi and Zhang [67] published an interesting paper that addressed these issues. They proposed a hybrid approach that uses the idea of exchanging link state metrics to identify a set of "good" paths. Without delving deeply into their approach, because it is not directly related to our work, we review below some of the commonly used

approaches for multiple path selection.

The multiple paths discovered in multipath routing may take different forms: *node disjoint, link disjoint*, or *non-disjoint* routes. In node disjoint routes there are no overlapping nodes or links. In link disjoint routes, there are no overlapping links, whereas in non-disjoint routes there may be overlapping nodes or links. The advantage of having disjoint routes is that they provide greater fault-tolerance, in the sense that if one of the nodes/links fail, it is quite unlikely that that failure will affect any of the other routes. Route maintenance in multipath routing is similar to that in unipath routing except that decision has to be taken when a route discovery needs to be triggered when a broken link is identified. This is because triggering a route discovery every time a failure is identified introduces more traffic, and a degraded performance of the network. On the other hand, waiting for all disjoint routes between a pair of source-destination nodes to fail before a decision for route discovery is made, might result in an unreasonable amount of delay [61].

7.3 Fault Tolerant Routing in Mobile Ad Hoc Networks

We have seen earlier in this Chapter that reliability of the correct transmission of messages is an important concern for MANETs, as such networks are characterized by continuously, rapidly changing network infrastructures due to node mobility. Hence mechanisms are required that would guarantee the delivery of packets in adversarial environments, and in the presence of node/link failures. Without a mechanism that "tolerates" route failures due to malfunctioning nodes while making routing decisions, the performance of *ad hoc* network protocols may be poor and routing decisions made by those protocols would be erroneous. The well-known MANET routing algorithms

(e.g., DSR, multipath routing) are unsuitable as fault-tolerant routing algorithms for MANETs. Since DSR chooses the shortest path route for packet transmission in adversarial environments, it can be shown that DSR will achieve a low packet delivery rate [110]. On the other hand, multipath routing algorithms are strong in their fault-tolerance ability, because they send multiple copies of packets through all possible (disjoint) routes between a pair of source-destination nodes. However, the disadvantage with multipath routing algorithms is that they introduce an unnecessary amount of overhead on the network [110].

Xue and Nahrstedt [110] confirmed that devising a fault-tolerant routing algorithm for *ad hoc* networks is inherently hard. This is because the problem itself is NP-complete due to the unavailability of correct path information in these environments. They designed an efficient algorithm, called the *End-to-End Fault Tolerant Routing Algorithm* (E2FT) [110], that is capable of significantly lowering the packet overhead, while guaranteeing a certain packet delivery rate.

7.3.1 Xue and Nahrestedt's Solution

7.3.1.1 Problem Model

We consider a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consisting of $\mathcal{V}$ mobile nodes and $\mathcal{E}$ bi-directional links connecting different nodes. If there are n mobile nodes in a path, the length of any path p is denoted by $L(p)$, where, $p = \{v_1, v_2, \ldots, v_n\}$, where $v_1, v_2, \ldots, v_n \in V$, and $(v_i, v_{i+1}) \in \mathcal{E}$, $i \in \{1, 2, \ldots, n\}$. The multipath routes between a pair of source-destination nodes is denoted by $\pi = \{p_1, p_2, \ldots, p_m\}$, where m is the number of paths between any pair of source-destination nodes. In such a model, $L(\pi) = \sum_{i=1}^{m} L(p_i)$ is used to represent the length of the multipath route [110].

The *packet delivery probability of a path* is represented as $\gamma(p) = \prod_{i=1}^{m} \gamma(v_i)$. If there are m paths in a multipath route between a pair of source-destination nodes,

the packet *delivery probability of a multipath route*, $\gamma(\pi)$, determines the probability that when multiple copies of the packets are sent along all the m paths between the source-destination pair, at least one copy is received. $\gamma(\pi)$ is calculated ([110]) as:

$$\gamma(\pi) = 1 - \prod_{i=1}^{m}(1 - \gamma(p_i))$$

The problem that we address here is that of determining how to provide a fault-tolerant routing mechanism, which would route packets through mobile nodes in the above environment (i.e., in the presence of faulty nodes) by providing a certain packet delivery rate guarantee, and at the same time by routing the least number of duplicate packets through multiple routes between a pair of source-destination nodes. Recall from Section 7.2 that "blind" multipath routing algorithms are capable of achieving a high packet delivery rate guarantee, because they utilize the benefits of network redundancy. However, their disadvantage is that they route duplicate packets through multipath routes to provide such a high packet delivery guarantee. We need a solution that would provide a certain optimum packet delivery rate guarantee, and at the same time reduce the amount of routing "overhead" posed on the network by packet duplication mechanisms adapted by the existing "blind" multipath routing algorithms.

7.3.1.2 Description of E2FT

The E2FT algorithm [110] involves two major phases: a *route estimation* phase, and a *route selection* phase. The route estimation phase is used to estimate the packet delivery probability of all the routes at our disposal at any time instant, whereas the route selection phase is used to select those routes that are confirmed to have satisfied a certain optimization constraint, and drop those routes from further consideration that are estimated to be unnecessary among all the available multipath routes between a pair of source-destination nodes.

The *route estimation* algorithm works as follows. N packets are sent along a path p. The source node estimates the fraction of packets delivered, $\hat{\gamma}(p)$ from the number of packets N' that are received along that path so that (see [110])

$$\hat{\gamma}(p) = \frac{N'}{N}$$

The number of packets to be sent depends on the level of accuracy desired through the estimation process – better estimation is achieved by sending a large number of packets, with a tradeoff of the overall high network overhead. In the E2FT algorithm [110], the accuracy of the estimation is achieved progressively through iterations. If B packets are sent, and B' packets received at the i^{th} iteration, the packet delivery probability of a path p in the i^{th} iteration can be calculated using the following scheme [110]:

$$\hat{\gamma}(p)^i = (1 - \frac{1}{i})\ \hat{\gamma}(p)^{i-1} + \frac{1}{i}\ \frac{B'}{B} \tag{7.1}$$

Note that, initially, since there are no packets sent, $\hat{\gamma}(p)^0 = 0$.

However, the above scheme does not take into account the differential behavior of paths with the different values of N, leading to different accuracies. Xue and Nahrstedt [110] provide a better estimation formula as follows:

If α represents a parameter indicating the level of confidence of an estimation, the α-estimation $\hat{\gamma}_\alpha(p)$ of a path p is computed as follows [110]:

$$\hat{\gamma}_\alpha(p) = \max\{\hat{\gamma}(p) - \frac{1}{\sqrt{4N(1-\alpha)}}, 0\} \tag{7.2}$$

The *route selection* algorithm has two phases: *Confirmation*, and *Dropping* [110]. Using the estimation procedure stated in Equation 7.1, with an increasing number of iterations, the route selection process uses the estimation results to select those routes (from the different candidate multipath routes) that are most likely to be able

to deliver packets with a minimum expected packet delivery probability guarantee, $\bar{\gamma}$, and removes those that are considered unnecessary [110].

The route selection algorithm works as follows [110]. At the beginning, since no estimation results are obtained, all paths between a pair of source-destination nodes are selected to route the packets. By using the estimation scheme presented in Equation 7.2, when the estimation of the paths are obtained to be accurate enough, the paths are reviewed to either be confirmed as one of the routes that "wins" the selection process, and be permanently used for routing all future requests, or be "knocked out" from further consideration of routing. For a path to be confirmed, the following condition [110] should be satisfied:

$$\hat{\gamma}(p) \geq \bar{\gamma} \tag{7.3}$$

Once a path is confirmed, it is considered to be solely used for routing future requests, and no further estimation is carried out on that path [110].

The dropping algorithm selects a path, p_{min}, from all the available paths, π, with the minimum packet delivery estimation value and also checks to see if the following dropping condition [110] is satisfied:

$$\hat{\gamma}_{\alpha^{1/m}}(\pi^{'}) \geq \bar{\gamma} \tag{7.4}$$

where

$$\hat{\gamma}_{\alpha^{1/m}}(\pi^{'}) = 1 - \prod_{p \in \pi^{'}} (1 - \hat{\gamma}_{\alpha^{1/m}}((p))), \text{ and } \pi^{'} = \pi - \{p_{min}\}$$

The above path estimation, and path selection algorithms were presented with the assumption that the set of paths between source-destination nodes does not change with time. However, for realistic *ad hoc* network environments, we need to consider cases where the nodes are mobile. With the consideration of the mobility of nodes,

new situations may arise, such as when the probability of delivery of packets through a set of multipath routes changes with the deletion of a path which was in use earlier, and the route to be used needs to be determined again. In such a situation the selected path can be broken, since the estimation procedure used in the static case was based on the end-to-end performance of a path, and not on the nodes constituting it. The estimation results will thus be useless in such situations [110].

Xue and Nahrstedt [110], proposed an optimization of their algorithm for environments to account for mobile nodes. Based on the packet delivery probability, $\hat{\gamma}(p)$, of a path p, they proposed to compute $\hat{\gamma}(v)$ of each node v in the path. When a path p is broken, they proposed to maintain $\hat{\gamma}(v)$ for each node v in the path p. Those values will be later used for estimating the $\hat{\gamma}(p')$ for a new path p'. If a new path is denoted as $p' = \{v_i\}$, where $i = 1, 2, \ldots, l$, the packet delivery estimation of p' is equal to (see [110])

$$< \prod_{i=1}^{l} \hat{\gamma}(v_i), \min_{i=1,2,\ldots,l}\{n_{v_i}\} >$$

The following theoretical result/propositions[2] are presented, without proof, for Xue and Nahrstedt's algorithm [110]. Interested readers are referred to [110] to obtain the proof of these results.

Lemma 7.1: (Property of α-estimation) $\hat{\gamma}_\alpha(p)$ satisfies: $\Pr\{\gamma(p) \geq \hat{\gamma}_\alpha(p)\} \geq \alpha$.

Lemma 7.2: If a confirmation procedure is used and path p is confirmed, then the packet delivery probability $\gamma(\pi)$ of the resulting path set $\pi = \{p\}$ satisfies: $\Pr\{\gamma(p) \geq \bar{\gamma}\} \geq \alpha$, where $\bar{\gamma}$ is the expected packet delivery probability guarantee.

[2]These lemmas and theorems are enunciated verbatim from [110].

Lemma 7.3: If a dropping procedure is used and π is the resulting path set, then π satisfies: $\Pr\{\gamma(\pi) \geq \bar{\gamma}\} \geq \alpha$, where $\bar{\gamma}$ is the expected packet delivery probability guarantee.

Theorem 7.1: The packet delivery probability of E2FT has a lower bound of $\min\{\gamma(\Omega), \bar{\gamma}\}$, where Ω is a given path set, and $\bar{\gamma}$ is the expected packet delivery probability guarantee.

Theorem 7.2: The number of packets N required to confirm a good path is upper bounded by: $\frac{1}{4(1-\alpha)(\hat{\gamma}(p)-\bar{\gamma})^2}$, where $\bar{\gamma}$ is the expected packet delivery probability guarantee.

7.4 Proposed Solution

The *objective* of our work was to propose a routing algorithm for MANETs, which will be able to minimize the overhead by sending the least possible number of redundant packets, while guaranteeing a certain rate for the delivery of packets. The reader should recall that there is a tradeoff between the rate of delivery of packets and the overhead. It is possible to achieve a very high packet delivery rate if the number of packets sent is not a concern (e.g., by using the multipath routing scheme). On the other hand, it is possible to achieve a very low overhead if we do not care about the number of packets that are successfully delivered (e.g., by using the DSR scheme). Thus, attempting to increase one will decrease another, and *vice versa*. What is challenging is how we can achieve a "balance" between the two. In other words, we need an algorithm that will be able to minimize the overhead by guaranteeing a certain level of packet delivery percentage.

Another *objective* of our work was to propose an algorithm that would be efficient in non-stationary environments, i.e., environments where the fault probability of a mobile node increases as it moves away from the center of the network in which it is supposed to operate. In other words, as a node moves away from the center of the region of operation, it is more likely to drop packets. Such an environment was not considered in the work by Xue and Nahrstedt [110].

To achieve our objectives, we used a powerful technique, which is essentially a learning scheme for estimation in *non-stationary* environments.

7.4.1 Weak Estimation Learning

In statistical problems involving random variables, the quality, reliability, and accuracy of the estimation are important considerations. Traditionally, there have been different estimates schemes proposed in the literature, which can broadly be classified as either belonging to the *Maximum Likelihood Estimator* (MLE) class of algorithms [23, 25, 36], or as belonging to the Bayesian family of algorithms [12, 23, 25]. Although the above estimation schemes have been proven to be quite efficient, they work under the premise that the underlying distribution in the environment is stationary, i.e., the estimated parameter does not vary with time. Oommen and Rueda [72, 73] studied this problem, and thereafter proposed a novel estimation scheme for learning in non-stationary environments. They considered the case where the Bernoulli trials yielding binomially distributed outcomes of random variables changed with time to new random values [72, 73].

In our fault-tolerant routing solution, we have used this efficient procedure [72, 73] for the estimation of the packet delivery probability through available paths. It is called the *Stochastic Learning Weak Estimator* (SLWE) scheme[3] [72, 73], and is based on the

[3]The term "weak" used in the SLWE estimator scheme, refers to the weak convergence of the random variable with respect to the first and second moments only.

stochastic learning paradigm discussed in Chapter 2. It uses a learning parameter, λ, which does not influence the mean of the final estimate. On the other hand, the variance of the final distribution[4], and the speed of convergence decrease with the increase in the value of the learning parameter [72, 73].

Prior to Oommen and Rueda's solution [72, 73], other solutions [11, 33, 38, 43, 84] were proposed for the estimation in non-stationary environments. The most well-known, and commonly used scheme to obtain estimates in non-stationary environments used the *sliding window* methodology [38]. The sliding window scheme has problems involving the optimum width[5] of the window, the maintenance, and the updating of the observations obtained during the entire sliding window [72]. The problems with the methodologies involving the detection of change-points during estimation are reported in [72]. Whereas almost all of the prior estimation schemes for non-stationary environments are additive in nature, the weak estimator based learning scheme of Oommen and Rueda [72] is multiplicative in nature. In other words, for a particular event that occurs at a specific time instant, the weak-estimation based learning scheme multiplies with a fixed constant, the estimate obtained till that time instant [72].

We now present below the weak estimation scheme [72, 73]. Let us consider a binomially distributed random variable X, such that:

$$X = \begin{cases} 0 & \text{with probability } s_0 \\ 1 & \text{with probability } s_1 \end{cases}$$

such that $s_0 + s_1 = 1$, where $S = [s_0, s_1]^T$

Assume that the at any time t, X assumes the value $x(t)$. In order to estimate s_0 and s_1, WELA keeps track of the running estimate $p_i(t)$ of s_i at time t, where $i = 0, 1$.

[4]In this Chapter, we illustrate the case of binomial distributions only. The results are extensible for the case of multinomial distributions [72] as well, but we omit discussing them here.

[5]The estimates are normally weak for small window sizes. The estimates obtained in between the change of parameters have a marked influence on one another when the window size considered is large.

In such a setting, the value of p_0 is updated using the following multiplicative scheme [72, 73]:

$$p_0(t+1) = \begin{cases} \lambda \times p_0(t), & \text{if } x(t) = 1 \\ 1 - \lambda \times p_1(t), & \text{if } x(t) = 0 \end{cases}$$

where λ is a constant ($0 < \lambda < 1$), called the learning parameter, and $p_1(t+1) = 1 - p_0(t+1)$.

We now present below, without proof[6], some of the interesting results concerning WELA[72][7].

Theorem 7.3: Let X be a binomially distributed random variable, and $P(t)$ be the estimate of S at time "t". Then, $E[P(\infty)] = S$.

Theorem 7.4: If the components of $P(t+1)$ are obtained from the components of $P(t)$ as per Equation 23, $E[P(t+1)] = M^T E[P(t)]$, where M is a stochastic matrix. Thus the limiting value of the expectation of $P(.)$ converges to S, and the rate of convergence of P to S is fully determined by λ.

Theorem 7.5: Let X be a binomially distributed random variable governed by the distribution S, and $P(t)$ be the estimate of S at time "t", obtained by Equation 23. Then, the algebraic expression for the variance of $P(\infty)$ is fully determined by λ.

7.4.2 Proposed Scheme

We use the above-mentioned weak-estimation learning scheme to propose a new fault-tolerant routing algorithm, named *Weak-Estimation-Based Fault Tolerant Routing Algorithm* (WEFTR), which is capable of efficiently estimating the probability of delivery

[6]Interested readers are referred to [72] for obtaining the proof of these results

[7]These theorems are enunciated verbatim from their original source [72].

of packets through the paths available at any moment[8]. Like the E2FT algorithm [110], the WEFTR algorithm involves, among other steps, *a route estimation* phase, and a *route selection* phase. The route estimation phase is used to estimate the packet delivery probability of all the routes at the disposal at any time instant, whereas the route selection phase is used to select those routes that are confirmed to have satisfied a certain optimization constraint, and drop the unnecessary multipath routes between a pair of source-destination nodes.

The *route estimation* algorithm works as follows. N packets are sent along a path p. The source node estimates the fraction of packets delivered, $\hat{\gamma}(p)$ from the number of packets N' that are received along that path[9] so that (see [110]) $\hat{\gamma}(p) = \frac{N'}{N}$.

In the WEFTR algorithm, the accuracy of estimation of the packet delivery probability is refined with the increase in the number of iterations. In every iteration, a set of packets is transmitted through each of the multipath routes between a pair of source-destination nodes. We can have two possible scenarios for any path: the nodes in a path either forward the packets correctly, or they do not. So, we use a binomial estimation scheme (based on SLWE) as follows.

$$\hat{\gamma}_0(p) = \begin{cases} \lambda \times \hat{\gamma}_0(p) & \text{if the path does not forward the packet correctly} \\ 1 - \lambda \times \hat{\gamma}_1(p) & \text{if the path forwards the packet correctly} \end{cases}$$

where λ is the learning parameter such that $0 < \lambda < 1$, and $\hat{\gamma}_1(p) = 1 - \hat{\gamma}_0(p)$.

The *route selection* algorithm works similar to that of the E2FT algorithm [110], except that instead of the α-estimation result, we use the estimation result obtained using the above Equation. In our selection algorithm, we however, use the same *Con-*

[8]This has been experimentally verified in Section 7.4.4.

[9]Like Xue and Nahrstedt [110], we have restricted this work to source routing algorithms, where the source node decides which path to select to route packets. The source node is also assumed to know all possible multipaths between a pair of source-destination nodes. We also assume that in this problem the destination node informs the source node of the number of packets that are received by it.

firmation, and *Dropping* procedures of Xue and Nahrstedt [110]. For a path to be confirmed, the following condition [110] should be satisfied: $\hat{\gamma}_{WE}(p) \geq \bar{\gamma}$, where $\bar{\gamma}$ is the minimum packet delivery probability required for a path to be confirmed, and $\hat{\gamma}_{WE}(p)$ is the packet delivery probability estimate using the weak-estimation scheme presented in the above Equation. Recall that once a path is confirmed, it is considered to be solely used for routing future requests, and no further estimation is carried out on that path[110].

The dropping algorithm selects a path, p_{min}, from all the available paths, π, with the minimum packet delivery estimation value also being checked to see if the following dropping condition [110] is satisfied:

$$\hat{\gamma}_{WE^{1/m}}(\pi^{'}) \geq \bar{\gamma} \tag{7.5}$$

where

$$\hat{\gamma}_{WE^{1/m}}(\pi^{'}) \;=\; 1 - \prod_{p \in \pi^{'}} (1 - \hat{\gamma}_{WE^{1/m}}(p)), \text{ and } \pi^{'} = \pi - \{p_{min}\}$$

We present below a high level sketch of the proposed algorithm.

Algorithm: WEFTR

Input

(i) A graph (network) with a set of nodes, and a set of links connecting the nodes.
(ii) The nodes are mobile, and links connecting them can be reset with the change in the position of the nodes.
(iii) Some of the nodes in the network are faulty with a certain packet delivery rate dependent on the distance of the node from the center of the area of mobility of the mobile nodes (simulation area).

Output

All the incoming packets are delivered from the source node to the destination node (with the intention of maximizing the packet delivery rate, and minimizing the network overhead).

... (Continued to next page)

Algorithm: WEFTR (Continued from previous page)

BEGIN

Initialization

Initialize a vector WEFTR_MP that stores all the paths in use, and WEFTR_Nodes that stores all the nodes in the graph, with information about their estimated packet delivery probability.

At each second, do the following

Case1: If the second is a simulation pause then

Step 1

Save the estimated packet delivery probability of each node in the vector WEFTR_Nodes.

Step 2

Update the edges and probabilities in the graph to reflect the current position of the nodes and calculate the new paths from the source to the destination.

Step 3

Use the values stored in WEFTR_Nodes in order to calculate the estimated packet delivery probability of each path.

Case2: At each second

Step 4

Try to confirm or drop paths. Paths dropped are removed from the WEFTR_MP vector.

Step 5

Use all the paths in the WEFTR_MP vector to send the packets, and calculate the number of packets that are received for each path, and the total number of non-duplicated packets that are received.

END

7.4.3 Experimental Details

In order to determine how the performance of the proposed algorithm compares with the rest of the other algorithms, we simulated an *ad hoc* network with mobile nodes, and dynamically changing topologies, and ran our proposed algorithm along with the other

benchmark algorithms (described in the next section) in the simulated environment.

7.4.3.1 Simulation Environment

The simulated environment that we considered consists of a flat square of length 500 meters. There are 50 nodes in the network, each having a different data delivery probability which decreases as the node moves away from the center of the square, and increases as it moves closer to it. Each node moves randomly, and we have assured that we have the same enviroment for all runs by using the same seed for the random number generator. If after a random move, a node reaches the edge of the square, then the move is canceled and a new random move for the same is done until it lands in a valid position. In our simulated *ad hoc* network, we assumed that the maximum speed with which the mobile nodes can travel is $20m/s$. Observe that the nodes move at each time unit, but the links between them are only recalculated at a simulation pause. The maximum speed of a node specified above (i.e., $20m/s$) is needed to calculate how much a node can move in a second. This is because, the position of a node at the i^{th} second is calculated as:

$$Xpos(i) = Xpos(i-1) + randomNumber \tag{7.6}$$

$$Ypos(i) = Ypos(i-1) + randomNumber \tag{7.7}$$

In the above, $Xpos(i-1)$ denotes the abscissa of a node in the previous second, and $Xpos(i)$ denotes the abscissa of a node in the current second. Similarly, $Ypos(i-1)$ denotes the ordinate of a node in the previous second, and $Ypos(i)$ denotes the ordinate of a node in the current second. If the maximum speed is $20m/s$, the *randomNumber* shown above will be a random number generated between -20 and $+20$. The maximum distance two nodes can have for which they are connected (they can deliver packets to each other) is directly dependent on the simulation parameter "sparsity". Sparsity of

the network is an attribute that signifies how the nodes connect with one another. It denotes a coefficient whose value ranges between 0 and 1: a value of 1 signifies that very few edges (100% sparse coefficient) connect with one another, whereas a value of 0 signifies that the maximum possible number of nodes connect with one another (0% sparse coefficient). The reader should observe that in the simulation there is no fixed number of links in the networks. The links are recalculated at each simulation pause. This is because two nodes are considered to have a link if they are within a certain distance of each other. What the sparsity does is that it directly influences this distance. Another parameter that is used in the simulations is called the pause time. It signifies how the algorithms accommodate to node mobility. Each of the simulations were run for 500 seconds. During the simulation period, traffic is generated between a certain pair of nodes with a rate of $10KB/s$.

We determine how far a node is from the center of the square, by measuring its Euclidean distance from the center.

7.4.3.2 Benchmark Algorithms

In order to establish how our algorithm performs when compared to existing algorithms, we selected three algorithms, all of which we ran along with our proposed algorithm in the simulated environment. The three algorithms we chose as benchmark algorithms are:

1. DSR Algorithm
2. Multipath Routing Algorithm
3. E2FT Algorithm[10]

[10]Here we consider only the optimized version of E2FT that provides an optimization methodology taking mobility of nodes into account.

Of all these three algorithms, E2FT represents the state-of-the-art in the area of fault tolerant routing in MANETs. So, we reckon that the performance comparison between our algorithm and E2FT is crucial. However, since DSR and the multipath routing algorithms are currently widely used in deployed MANETs, they were also considered. Although DSR is a simple routing algorithm, it is weak when it concerns routing information in the presence of malfunctioning nodes. On the other hand, multipath routing is perhaps a very strong routing algorithm in terms of routing information when there are misbehaving nodes. But as we mentioned earlier, the greatest limitation of multipath routing is that it brings with it a large network overhead, as it loads all different routes between a pair of source-destination nodes with redundant packets, just to ensure that the destination node receives at least one correct copy of the packet sent from the source.

7.4.3.3 Performance Metrics

Two metrics were used for evaluating the performance of the algorithms invoked in the experiments:

1. *Percentage of packets delivered*: This represents the rate of successful delivery of packets to the destination. This is calculated as follows. At each second, the packet delivery probability of all the paths in use is calculated. Then, for each packet sent at that time unit, a random number between 0 and 1 is generated. If the number is lower than the packet delivery probability, the packet is considered as delivered. After all the iterations, the percentage of delivered packets is calculated as follows:

$$\text{percentage delivered packets} = \frac{\text{total number of delivered packets}}{\text{total number of sent packets}}$$

2. *Overhead*: This represents the overall number of packets sent. The overhead is calculated as the product of the total length of all paths in use, and the number of packets sent per second.

7.4.3.4 Experimental Results

Several experiments were conducted to assess the performance of WEFTR (the proposed algorithm) with respect to the benchmark algorithms. The results of the following three sets of experiments are presented below:

- Variation in pause time
- Variation in sparsity
- Variation in faultiness of nodes

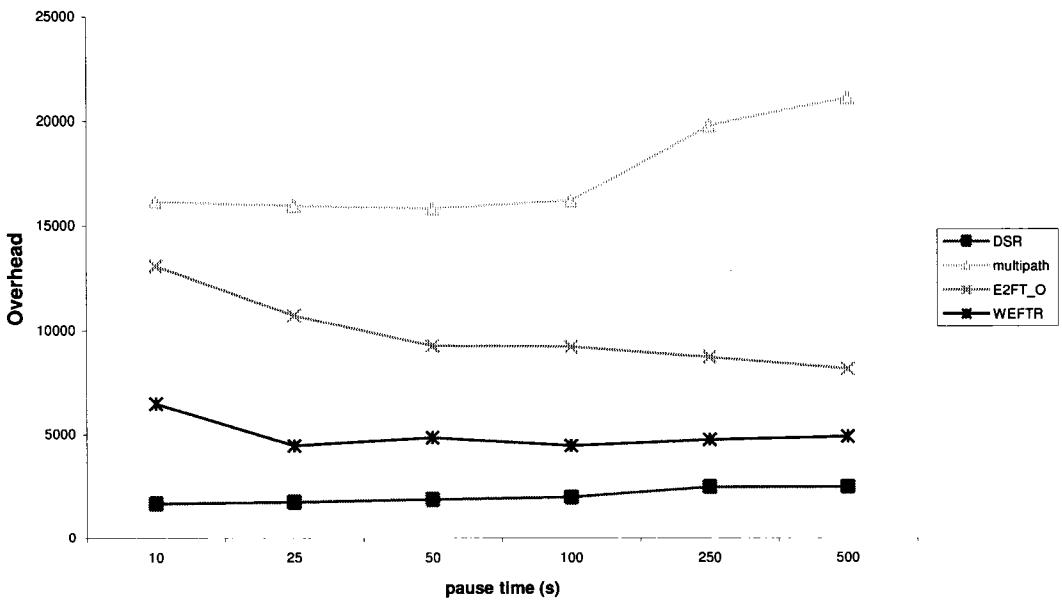


Figure 7.1: Plot of overhead versus pause time for the various algorithms tested.

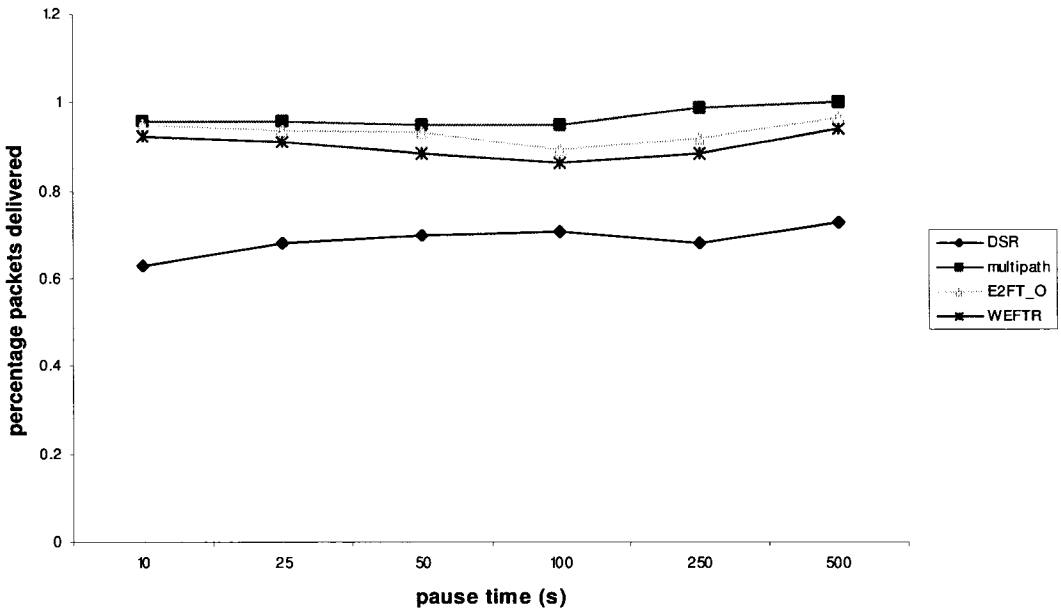


Figure 7.2: Plot of percentage delivered packets versus pause time for the various algorithms tested.

Variation in pause time: As noted earlier, pause time is a parameter specific to the simulation, which indicates how much an algorithm is capable of accommodating with the mobility of the nodes. As seen from Figure 7.1, we notice that just with respect to the overhead, blind multipath routing is the worst, DSR the best, and E2FT somewhere in between the DSR and multipath curves. Our proposed algorithm further improves on the performance of E2FT by decreasing the overhead by $25 - 50\%$. For example, when the pause time is 250 seconds, the overhead for multipath routing is $19,790$, that for E2FT is $8,740$, and for WEFTR it is $7,225$. On the other hand, from Figure 7.2, we observe that WEFTR achieves an almost similar order of performance (slightly inferior, to be precise) as compared to E2FT. By taking Figures 7.1 and 7.2 together, we can infer that our proposed algorithm (WEFTR) is capable of significantly reducing the overhead of the currently best available fault tolerant routing algorithm (E2FT), while

achieving a performance guarantee of at least 80% delivery of packets. Our algorithm always performs much better than either DSR or blind multipath routing.

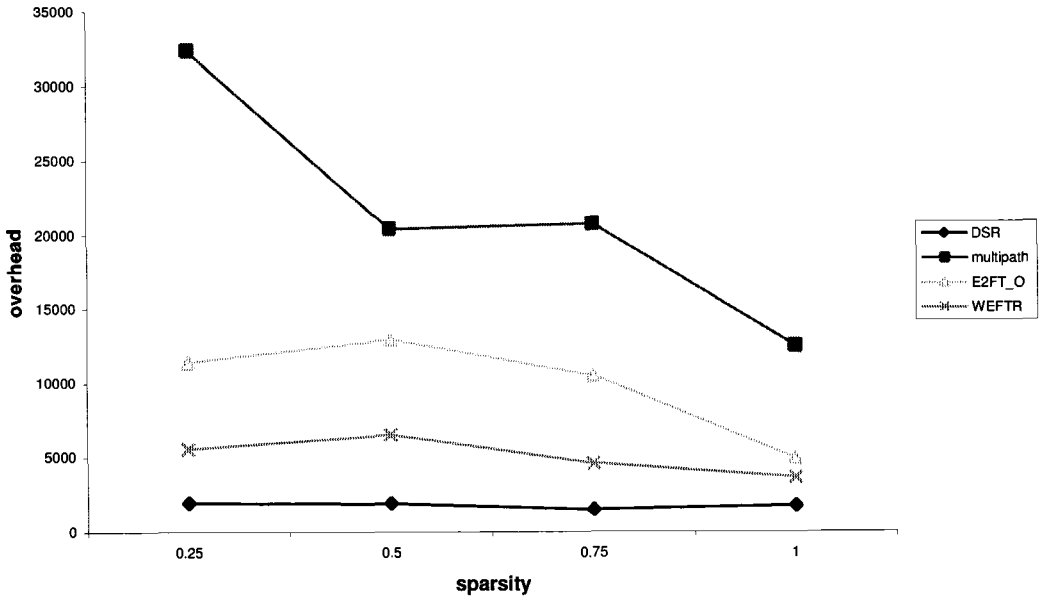


Figure 7.3: Plot of overhead versus sparsity for the various algorithms tested.

Variation in sparsity: In the second set of experiments, we intended to study how the algorithms compare with respect to one another with the variation in the sparsity of the nodes in a network. The value of sparsity ranges between 0 and 1, where "0" represents the least percentage of sparsity, and "1" represents the greatest percentage of sparsity. Since the nodes are mobile, how often they connect depends on how close they can get to one another, and is thus directly related to the sparsity. The different sparsity values used in our experiments indicate the relative number of edges between the nodes in a network.

Figures 7.3 and 7.4 show the performance comparison of all the studied algorithms with respect to the overall overhead, and the percentage of packets successfully routed by the algorithms. From Figure 7.3, we can clearly observe that even at different

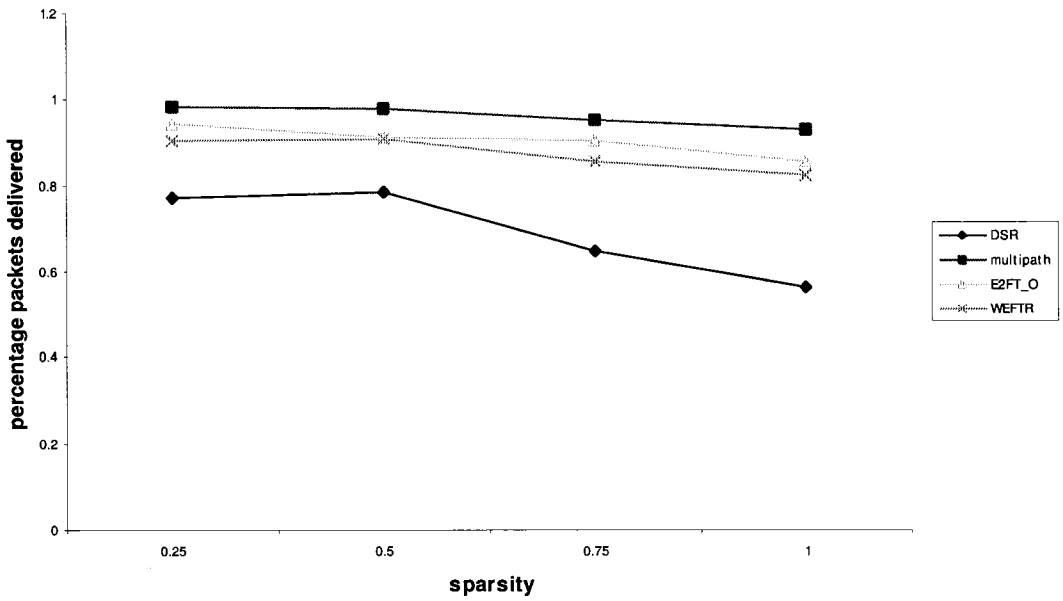


Figure 7.4: Plot of percentage of delivered packets versus sparsity for the various algorithms tested.

values of sparsity, E2FT is capable of significantly reducing the overall overhead. For example, when the value of sparsity is 0.25, the overhead for multipath routing is 32,320, that of E2FT is 11,410, whereas the overhead for our proposed algorithm is 5,570. Also, it can be observed that the performance of E2FT is much better at lower sparsity values than at the higher ones. On the other hand, if we look at Figure 7.4, we can observe that, in general, the percentage of packets delivered by both E2FT and WEFTR are similar. Thus, for this set of experiments as well, we observe that WEFTR significantly reduces the overhead as compared to E2FT or blind multipath routing algorithms. This is done while achieving similar performance to that of E2FT or multipath (and much better performance than DSR algorithm) with respect to the number packets successfully routed.

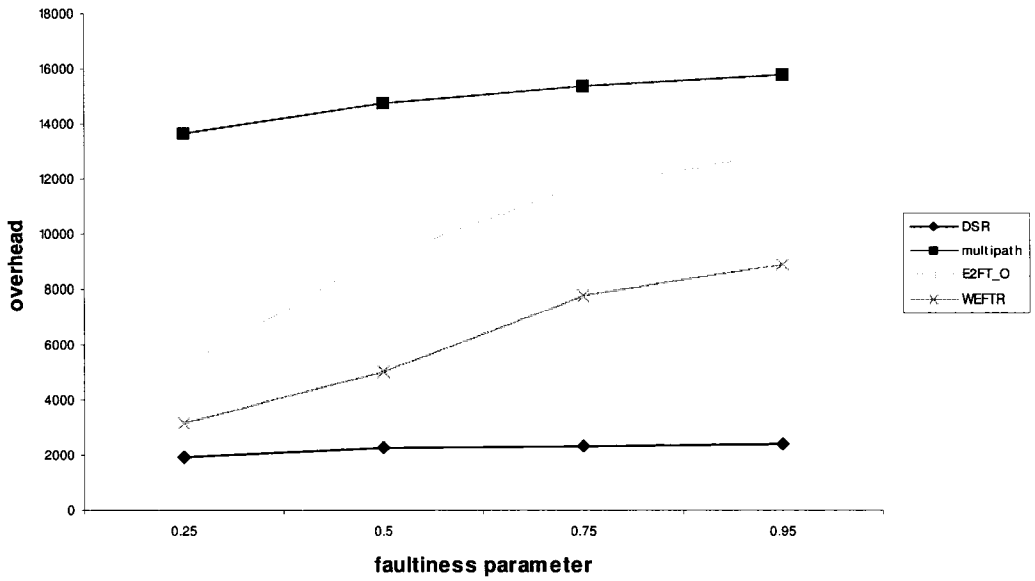


Figure 7.5: Plot of overhead versus faultiness parameter for the various algorithms tested

Variation in faultiness: Faultiness is an internal simulation parameter that indicates how many nodes will be faulty in a given environment. It influences the faultiness behavior of the nodes, given their distance from the center of the region of operation of the nodes. Figures 7.5 and 7.6 show the variation in overhead, and the percentage of delivered packets, with the variation in the faultiness parameter. In our experiments, we have used the faultiness parameter to vary in a scale from a very low value to a very high value on a scale of 0 to 1. We observe that, even in this set of experiments, our proposed algorithm shows much better performance as compared to the other algorithms. For example, at the faultiness parameter value of 0.25, the overhead for blind multipath routing is 13,690, for E2FT it is 5,240, whereas that in the case of WEFTR it is 3,150. Thus, in this case, our algorithm shows an improvement of about 62% over multipath routing, and an improvement of about 40% over E2FT algorithm. All of these algorithms, however, in general, show similar performance with respect to

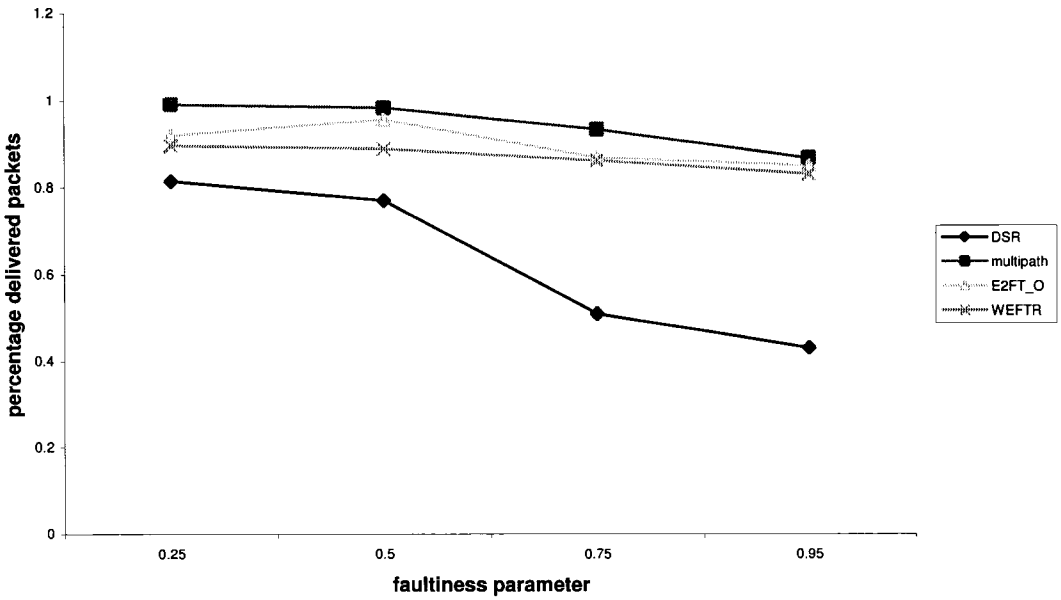


Figure 7.6: Plot of percentage of delivered packets versus faultiness parameter for the various algorithms tested.

the percentage of successfully delivered packets.

7.5 Conclusions

In this Chapter, we have studied an interesting, yet challenging, problem of fault tolerant routing in MANETs. The problem is that of efficiently routing packets in MANETs, in adversarial environments particularly, when there are misbehaving nodes present in the network. We have attempted to devise an algorithm, which would be able to successfully route packets by tolerating faults in the network. There are two principal metrics that characterize any fault tolerant routing algorithms designed for MANETs: (1) overhead, and (2) percentage of successfully delivered packets. The traditional algorithms, DSR and multipath routing, have the potential to attain two extremes of the goal. While multipath routing is a very strong algorithm for maximizing the number

of successfully delivered packets, it introduces an extremely large overhead on the network. On the other hand, DSR has a low overhead, but at the same time, has a very poor fault tolerant routing algorithm, because it will drop packets if there are problems in the route identified by the algorithm. E2FT algorithm was proposed by Xue and Nahrstedt [110], which is capable of minimizing the overhead as compared to multipath routing algorithm, while achieving a similar order of performance (slightly inferior, to be precise) with respect to the number of packets successfully delivered. In this Chapter, we proposed a powerful algorithm which significantly reduces the overhead over the E2FT algorithm, while achieving similar order of performance as far as the number of successfully delivered packets are concerned, and simultaneously minimizing the overhead. Our algorithm was experimentally established to have achieved the above goal.

Chapter 8

Conclusions

In this Thesis we studied four important, and challenging problems in the area of *network routing and traffic engineering.* These are:

1. **Dynamic Single-Source Shortest Path Routing Problem**: This problem (referred to as the DSSSP problem) involves maintaining shortest paths in a network from one node to all the other nodes in the network in which the costs of the links change in a stochastic manner.

2. **Dynamic All-Pairs Shortest Path Routing Problem**: This problem (referred to as the DAPSP problem) involves maintaining shortest paths dynamically between *all* pairs of nodes in a stochastic network in which the costs of the links change in a stochastic manner.

3. **Quality-of-Service (QoS) and Traffic Engineering (TE) Routing Problem**: This problem concerns computing bandwidth guaranteed paths (LSPs), and dynamically routing LSP setup requests (arriving one at a time) between a pair of ingress-egress routers by satisfying the requested bandwidth. This work focuses on MPLS Traffic Engineering.

4. **Fault Tolerant Routing Problem for Mobile Ad Hoc Networks**: This problem concerns efficient routing in adversarial environments in a MANET network, i.e., one that contains malfunctioning nodes.

Our solutions[1], which utilize *Learning Automata* [47, 66], *Random Races* [64], and *Weak-Estimator Learning* [72, 73], are demonstrated to be superior to the state-of-the-art. Refereed Publications that record our contributions have been mentioned in the respective Chapters.

Our contributions and the directions of future research in each area are listed below.

8.1 Dynamic Single-Source Shortest Path Routing Problem

8.1.1 Contributions

In Chapter 4, we proposed two LA-based dynamic adaptive algorithms for the DSSSP problem (using linear updating and Pursuit-based updating), and experimentally evaluated their performance with respect to three important performance criteria - the *number of scanned links* in the network, the *number of processed nodes*, and the *time per update operation.* Our solutions [55, 58] were shown to be superior to the state-of-the-art on all these counts. They converge to the "statistical shortest path tree in the "average network topology irrespective of whether there are new changes in link-weights taking place or not. Also, our solutions do not probe all the links, and even those that are probed are not probed equally often.

Apart from telecommunications, we believe that our solutions are also applicable to transportation, spatial databases, and information visualization.

[1]The reader should note that we have restricted our investigation to only a few principal issues in each of these problems. We do not claim to have investigated all the possible issues. This is neither feasible nor practical within the limited time frame of our Doctoral research.

8.1.2 Future Research Directions

The future research in this area could involve:

- Testing the proposed algorithms on large (on topologies with 10,000 to 100,000 nodes) simulated networks.
- Testing the performance of the algorithms in real-life networks.
- Designing a hybrid learning algorithm combining our simpler learning scheme with that of the enhanced RR scheme [83] (which considers multiple edge-weight updates).
- Assessing the suitability of our solutions to similar problems in application domains other than telecommunications.
- Achieving a formal analysis for the *game* models of our solutions.

8.2 Dynamic All-Pairs Shortest Path Routing Problem

8.2.1 Contributions

As in Chapter 4, in Chapter 5, we proposed two LA-based dynamic adaptive algorithms for the DAPSP problem (using linear updating and Pursuit-based updating), and experimentally evaluated their performance with respect to three relevant important performance criteria – the *number of scanned links* in the network, the *number of processed nodes*, and the *time per update operation.* This problem, though similar to the DSSSP problem, is much more complex because we seek an efficient solution that does not merely apply the DSSSP algorithm for all the pairs of nodes in the network.

As in the DSSSP case, our solutions [57] were shown to be superior to the competitive methods, and converging to the set of "statistical" shortest paths in the "average" network topology irrespective of whether there are new changes in link-weights taking place or not. Again, our solutions do not probe all the links, and even those that are probed, are not probed equally often.

8.2.2 Future Research Directions

All the future work mentioned in Section 7.1.2 for the DSSSP problem, are also applicable for the DAPSP problem.

8.3 Quality-of-Service & Traffic Engineering Routing Problem

8.3.1 Contributions

In Chapter 6, we presented an efficient adaptive online routing algorithm for the computation of bandwidth-guaranteed paths in MPLS-based networks, using random-race-based scheme that computes an optimal *ordering* of the routes. The proposed solution was shown to have a better performance than the important algorithms in the literature.

8.3.2 Future Research Directions

The future research work in this area could involve the investigation of the following issues:

- Testing the performance of our proposed algorithm on actual, typical, atypical, and more complex networks.

- Testing the algorithms on real networks, under real traffic conditions.
- Although the three performance criteria chosen in our work are fundamental for evaluating the performance of any TE, it would be interesting to test the performance of our algorithm with respect to other performance criteria.
- Finally, in our study we considered idealistic situations where there were no network link failures. With regard to future work, it would be interesting to consider situations were there are realistic link-ups and downs occurring in a network. It would also be interesting to see how our algorithm can perform "rerouting" of paths if a desired path fails.

8.4 Fault Tolerant Routing Problem for Mobile Ad Hoc Networks

8.4.1 Contributions

In Chapter 7, we proposed a solution for fault-tolerant routing in MANETs in adversarial environments. Our solution is based on the paradigm of using weak-estimator learning [72, 73], and is capable of reducing the high overhead of the existing algorithms, while achieving a similar order of packet delivery rate guarantee as the others.

8.4.2 Future Research Directions

The future work in this area can be extended as follows:

- It would be beneficial to study the effectiveness of our proposed algorithm for different kinds of faulty MANET networks (for example, those that have security flaws).

- Finally, due to the generic nature of the estimation scheme we used, it would be interesting to investigate the applicability of our proposed algorithms in environments *other* than MANETs.

8.5 Summary

In this Thesis, we have reported the results of four interesting problems studied in the area of routing and traffic engineering in networks. In all cases, we were able to propose algorithms superior to the existing ones with respect to certain pertinent performance considerations. It is our sincere hope that this research would motivate further research in each of these areas, and that our solutions could be improved academically and used in the industry.

Bibliography

[1] C. R. Atkinson, G. H. Bower and E. J. Crowthers, *An Introduction to Mathematical Learning Theory.* John Wiley & Sons, New York, 1965.

[2] M. Agache, *Estimator Based Learning Algorithms.* M.C.S. Thesis, School of Computer Science, Carleton University, Ottawa, Ontario, Canada, 2000.

[3] G. Ausiello, G. Italiano, A. Marchetti-Spaccamela and U. Nanni, *Incremental Algorithms for minimal length paths.* Journal of Algorithms, Vol. 12, No. 4, 1991, pp. 615-638.

[4] R. K. Ahuja, T. L. Magnanti and J. B. Orlin, *Network Flows: Theory Algorithms, and Applications.* Prentice Hall, New Jersey, 1993.

[5] M. Agache and B. J. Oommen, *Generalized Pursuit Learning Schemes: New Families of Continuous and Discretized Learning Automata.* IEEE Trans. on Systems, Man, and Cybernetics – Part B, Vol. 32, No. 6, December 2002, pp. 738-749.

[6] A. F. Atlasis, M. P. Saltouros and A. V. Vasilakos, *On the Use of a Stochastic Estimator Learning Algorithm to the ATM Routing Problem: A Methodology.* Proceedings of IEEE GLOBECOM, December 1998.

[7] A. F. Atlasis and A. V. Vasilakos, *LB-SELA: Rate-Based Access Control in ATM Networks.* Proceedings of IEEE INFOCOM, March 1994, pp. 1552-1569.

[8] *Avici*^{TM} Systems Inc. *Traffic Engineering with Multi-Protocol Label Switching.* 2000. A white paper that can be accessed online at http://www.avici.com/technology/whitepapers/mpls_wp.pdf (Last accessed: March 2, 2004)

[9] A. F. Atlasis, A. V. Vasilakos and N. H. Loukas, *The Use of Learning Algorithms in ATM Networks Call Admission Control Problem: A Methodology.* Computer Networks, Vol. 34, No. 3, September 2000.

[10] R. Bellman, *On a Routing Problem.* Quart. Appl. Math., Vol. 16, 1958, pp. 87-90.

[11] M. Baron and N. Grannot, *Consistent Estimation of Early and Frequent Change Points.* Foundation of Statistical Inference, Springer, Heidelberg, 2003.

[12] P. Bickel and K. Doksum, *Mathematical Statistics: Basic Ideas and Selected Topics.* Vol. 1, Prentice Hall, 2nd Edition, 2000.

[13] R. R. Bush and F. Mosteller, *Stochastic Models for Learning.* John Wiley & Sons, New York, 1958.

[14] R. Boutaba, W. Szeto, Y. Iraqi, *DORA: Efficient Routing for MPLS Traffic Engineering.* Journal of Network and Systems Management, Vol. 10, No. 3, 2002, pp. 309-325.

[15] G. Casella and R. Berger, *Statistical Inference.* Brooks/Cole Publishing Co., 2nd Edn., 2001.

[16] G. D. Caro, F. Ducatelle and L. M. Gambardella, *AntHocNet: An Ant-Based Hybrid Routing Algorithm for Mobile Ad Hoc Networks.* Technical Report No. IDSIA-25-04-2004, Dalle Molle Institute for Artificial Intelligence, Switzerland, August 2004. (Also appeared in the Proceedings of Parallel Problem Solving from Nature VIII, LNCS 3242, Springer-Verlag, 2004, pp. 461-470).

[17] Cisco Systems, Inc. http://www.cisco.com/univercd/cc/td/doc/cisintwk /ito_doc/qos.htm (Last accessed: March 1, 2004).

[18] T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms.* MIT Press, Massachusetts, 1990.

[19] E. W. Dijkstra, *A Note on Two Problems in Connection with Graphs.* Numerische Mathematik, Vol. 1, 1959, pp. 269-271.

[20] C. Demetrescu and G. Italiano, *Fully Dynamic All-Pairs Shortest Paths with Real Weights.* Proceedings of the 42nd IEEE Annual Symposium on Foundations of Computer Science (FOCS'01), Las Vegas, Nevada, 2001, pp. 260-267.

[21] C. Demetrescu and G. F. Italiano, *A New Approach to Dynamic All Pairs Shortest Paths.* Proceedings of the 35th Annual ACM Symposium on the Theory of Computing, San Diego, CA, 2003, pp. 159-166.

[22] C. Demetrescu, D. Frigioni, A. Marchetti-Spaccamela and U. Nanni, *Maintaining Shortest Paths in Digraphs with Arbitrary Arc Weights: An Experimental Study.* Lecture Notes in Computer Science, Vol. 1982, 2001, pp. 218-229.

[23] R. Duda, P. Hart and D. Stork, *Pattern Classification.* John Wiley & Sons, New York, 2nd Edition, 2000.

[24] S. Even and H. Gazit, *Updating Distances in Dynamic Graphs.* Methods of Operations Research, Vol. 49, 1985, pp. 371-387.

[25] K. Fukunaga, *Introduction to Statistical Pattern Recognitio.* Academic Press, 1990.

[26] P. G. Franciosa, D. Frigioni and R. Giaccio, *Semi-Dynamic Shortest Paths and Breadth First Search in Digraphs.* Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, Vol. 1200, 1997, pp. 33-46.

[27] D. Frigioni, M. Ioffreda and U. Nanni, *Experimental Analysis of Dynamic Algorithms for the Single Source Shortest Path Problem.* ACM Journal of Experimental Algorithmics, Vol. 3, Article 5, 1998.

[28] R. W. Floyd, *Algorithm 97 (SHORTEST PATH).* Communications of the ACM, Vol. 5, No. 6, 1962, pp. 345.

[29] K. S. Fu and R. W. McLaren, *An Application of Stochastic Automata to the Synthesis of Learning Systems.* Technical Report, TR-EE-65-17, Purdue University, USA, 1965.

[30] D. Frigioni, A. Marchetti-Spaccamela and U. Nanni, ACM-SIAM Symposium on Discrete Algorithms, 1996, pp. 212-221.

[31] D. Frigioni, A. Marchetti-Spaccamela and U. Nanni, *Fully Dynamic Algorithms for Maintaining Shortest Paths Trees.* Journal of Algorithms, Vol. 34, 2000, pp. 251-281.

[32] J. Fakcharoemphol and S. Rao, *Planar Graphs, Negative Weight Edges, Shortest Paths, and Near Linear Time.* Proceedings of the 42^{nd} IEEE Symposium on Foundations of Computer Science (FOCS'01), Las Vegas, Nevada, 2001, pp. 232-241.

[33] E. Gombay, *Sequential Change-Point Detection and estimation.* Sequential Analysis, Vol. 22, 2003, pp. 203-222.

[34] R. Guerin, A. Orda and D. Williams, *QoS Routing Mechanisms and OSPF Extensions.* Proceedings of the Global Internet Miniconference, November 1997.

[35] M. Henzinger, P. Klein, S. Rao and S. Subramanian, *Faster Shortest-Path Algorithms for Planar Graphs.* Journal of Computer and System Sciences, Vol. 55, No. 1, 1997, pp. 3-23.

[36] R. Herbich, *Learning Kernel Classifiers: Theory and Algorithms.*, MIT Press, Cambridge, MA, USA, 2001.

[37] I. Iliadis and D. Bauer, *A New Class of Online Minimum-Interference Routing Algorithms.* NETWORKING 2002, LNCS 2345, pp. 959-971.

[38] T. M. Jang, *Estimation and Prediction-Based Connection Admission Control in Broadband Satellite Systems.* ERTI Journal, Vol. 22, No. 4, 2000, pp. 40-50.

[39] D. B. Johnson and D. A. Maltz, *Dynamic Source Routing in Ad Hoc Wireless Networks.* Mobile Computing, 1996, pp. 153-181.

[40] V. King, *Fully Dynamic Algorithms for Maintaining All-Pairs Shortest Paths and Transitive Closure in Digraphs.* Proceedings of the 40th IEEE Symposium on Foundations of Computer Science (FOCS99), 1999, pp. 81-99.

[41] K. Kar, M. Kodialam and T. V. Lakshman, *Minimum Interference Routing of Bandwidth Guaranteed Tunnels with MPLS Traffic Engineering Applications.* IEEE Journal of Selected Areas in Communications, Vol. 18, No. 12, December 2000, pp. 2566-2579.

[42] M. Kodialam and T. V. Lakshman, *Minimum Interference Routing with Applications to MPLS Traffic Engineering.* IEEE INFOCOM, 2000, pp. 884-893.

[43] P. Krishnaiah and B. Miao, *Review About Estimation of Change Points.* Handbook of Statistics, Elsevier, Amsterdam, Vol. 7, 1988, pp. 375-402.

[44] V. I. Krinsky, *An Asymptotically Optimal Automaton with Exponential Convergence.* Biofizika, Vol. 9, 1964, pp. 484-487.

[45] V. Krylov, *On the Stochastic Automaton Which is Asymptotically Optimal in Random Medium.* Automation and Remote Control, Vol. 24, 1964, pp. 1114-1116.

[46] S. Lakshmivarahan, *ϵ-Optimal Learning Algorithms Non-absorbing Barrier Type.* University of Oklahoma, School of Electrical Engineering and Computing Sciences, Technical Report EECS 7901, February 1979.

[47] S. Lakshmivarahan, *Learning Algorithms Theory and Applications.* Springer-Verlag, New York, 1981.

[48] P. Loubal, *A Network Evaluation Procedure.* Highway Research Record 205, 1967, pp. 96-109.

[49] J. K. Lanctôt and B. J. Oommen, *Discretized Estimator Learning Automata.* IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-22, Nov-Dec 1992, pp. 1473-1483.

[50] S. Lakshmivarahan and M. A. L. Thathachar, *Absolutely Expedient Learning Algorithms for Stochastic Automata.* IEEE Transactions on Systems, Man, and Cybernetics, SMC-3, 1973, pp. 281-86.

[51] Q. Ma, *Quality-of-Service Routing in Integrated Services Networks.* Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, January 1998.

[52] M. K. Marina and S. R. Das, *On-Demand Multipath Distance Vector Routing in Ad Hoc Networks.* Proceedings of the 9th International Conference on Network Protocols, Riverside, California, 2001.

[53] J. Moy, *OSPF Version 2.* Internet Draft, RFC 2178, 1997.

[54] S. Misra and B. J. Oommen, *Stochastic Learning Automata-Based Dynamic Algorithms for the Single-Source Shortest Path Problem.* Proceedings of the 17th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE 2004), Ottawa, Ontario, Canada, May

17–20, 2004. Appeared in the Lecture Notes in Artificial Intelligence, Vol. 3029, pp. 239-248, Springer-Verlag, B. Orchard, C. Yang, M. Ali (Eds.)

[55] S. Misra and B. J. Oommen, *GPSPA: A New Adaptive Algorithm for Maintaining Shortest Path Routing Trees in Stochastic Networks.* International Journal of Communication Systems, Vol. 17, No. 10, pp. 963-984, 2004, John Wiley & Sons.

[56] S. Misra and B. J. Oommen, *Generalized Pursuit Learning Algorithms for Shortest Path Routing Tree Computation.* Proceedings of the 9^{th} IEEE Symposium on Computers and Communications (IEEE ISCC 2004), Alexandria, Egypt, June 29-July 1, pp. 891-896, 2004.

[57] S. Misra and B. J. Oommen, *New Algorithms for Maintaining Dynamic All-Pairs Shortest Paths.* Proceedings of the 10^{th} IEEE Symposium on Computers and Communications (IEEE ISCC 2005), Cartagena, Spain, June 27–30, 2005.

[58] S. Misra and B. J. Oommen, *Dynamic Algorithms for the Shortest Path Routing Problem: Learning Automata-Based Solutions.* IEEE Transactions on Systems, Man, and Cybernetics, Part B. (To appear in December 2005).

[59] J. McQuillan, I. Richer and E. Rosen, *The New Routing Algorithm for the ARPANET.* IEEE Transactions on Communications, Vol. COM-28, No. 5, 1980, pp. 711-719.

[60] Q. Ma, P. Steenkiste and H. Zhang, *Routing High-Bandwidth Traffic in Max-Min Fair Share Networks.* ACM SIGCOMM, Stanford, CA, August 1996, pp. 206-217.

[61] S. Mueller, R. P. Tsang and D. Ghosal, *Multipath Routing in Mobile Ad Hoc Networks: Issues and Challenges.* Invited paper in Lecture Notes in Computer Science, Maria Carla Calzarossa and Erol Gelenbe (Eds.), 2004.

[62] J. Murchland, *The Effect of Increasing or Decreasing the Length of a Single Arc on All Shortest Distances in a Graph.* Technical Report, LBS-TNT-26, London.

[63] P. Narvaez, K. -Y. Siu and H. Y. Tzeng, *New Dynamic Algorithms for Shortest Path Tree Computation.* IEEE/ACM Transactions on Networking, Vol. 8, No. 6, 2000, pp. 734-746.

[64] D. T. H. Ng, B. J. Oommen and E. R. Hansen, *Adaptive Learning Mechanisms for Ordering Actions Using Random Races.* IEEE Transactions on Systems, Man, and Cybernetics, Vol. 23, No. 5, 1993.

[65] M. F. Norman, *On Linear Models With Two Absorbing Barriers.* Journal of Mathematical Psychology, Vol. 5, 1968, pp. 225-241.

[66] K. S. Narendra and M. A. L. Thathachar, *Learning Automata.* Prentice-Hall, 1989.

[67] S. Nelakuditi and Z. L. Zhang, *On Selection of Paths for Multipath Routing.* Proceedings of the 9^{th} International Workshop on Quality of Service, LNCS, Springer-Verlag, London, Vol. 2092, 2001.

[68] B. J. Oommen and J. P. R. Christensen, *ϵ-Optimal Discretized Linear Reward-Penalty Learning Automata.* IEEE Transactions on Systems, Man, and Cybernetics, Vol. 18, No. 3, May/June 1998, pp. 451-457.

[69] B. J. Oommen and E. R. Hansen, *The Asymptotic Optimality of Discretized Linear Reward-Inaction Learning Automata.* IEEE Transactions on Systems, Man, and Cybernetics, May/June 1984, pp. 542-545.

[70] B. J. Oommen, *Absorbing and Ergodic Discretized Two Action Learning Automata.* IEEE Transactions on Systems, Man, and Cybernetics, March/April 1986, pp. 282-293.

[71] M. S. Obaidat, G. I. Papadimitriou and A. S. Pomportsis, *Learning Automata: Theory, Paradigms, and Applications.* IEEE Transactions on Systems, Man, and Cybernetics Part B, Vol. 32, No. 6, December 2002, pp. 706-709.

[72] B. J. Oommen and L. Rueda, *Stochastic Learning-Based Weak Estimation of Multinomial Random Variables and Its Applications to Pattern Recognition in Non-stationary Environments.* (a draft of the above manuscript can be obtained from the authors).

[73] B. J. Oommen and L. Rueda, *A New Family of Weak Estimators for Training in Non-Stationary Distributions.* Proceedings of the 2004 International Symposium on Structural, Syntactic, and Statistical Pattern Recognition, Lisbon, Portugal, August 2004, pp. 644-652.

[74] B. J. Oommen and T. D. Roberts, *Discretized Learning Automata Solutions to the Capacity Assignment Problem for Prioritized Networks.* IEEE Transactions on Systems, Man, and Cybernetics, Vol. 32, No. 6, December 2002, pp. 821-831.

[75] E. Osborne and A. Simha, *Traffic Engineering with MPLS.* Cisco Press, July 2002.

[76] L. Peterson and B. Davie, *Computer Networks: A Systems Approach.*, 2nd Edition, Morgan Kauffman Publishers, 2000.

[77] C. E. Perkins and E. M. Royer, *Ad-Hoc On-Demand Distance Vector Routing.* Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications, New Orleans, Louisiana, 1999.

[78] W. Pedrycz and A. Vasilakos, *Computational Intelligence in Telecommunications Networks.* CRC Press, Boca Raton, Florida, 2000.

[79] G. Ramalingam, *Bounded Incremental Computation.* Lecture Notes in Computer Science, Springer-Verlag, Berlin, Vol. 1089, 1996.

[80] V. Rodinov, *The Parametric Problem of Shortest Distances.* USSR Computational Math. and Math Phys., Vol. 8, No. 5, 1968, pp. 336-343.

[81] H. Rohnert, *A Dynamization of the All-Pairs Least Cost Problem.* Proceedings of the 2nd Annual Symposium on Theoretical Aspects of Computer Scienc (STACS85), LNCS 182, 1985, pp. 279-286.

[82] G. Ramalingam and T. Reps, *On the Computational Complexity of Dynamic Graph Problems.* Theoretical Computer Science, Vol. 158, No. 1, 1996, pp. 233-277.

[83] G. Ramalingam and T. Reps, *An Incremental Algorithm for a Generalization of the Shortest Path Problem.* Journal of Algorithms, Vol. 21, 1996, pp. 267-305.

[84] B. Ray and R. Tsay, *Bayesian Methods for Change-Point Detection in Long-Range Dependent Processes.* Journal of Time Series Analysis, Vol. 23, No. 6, 2002, pp. 687-705.

[85] P. S. Sastry, *Systems of Learning Automata: Estimator Algorithms Applications.* Ph.D. Thesis, Department of Electrical Engineering, Indian Institute of Science, Bangalore, India, June 1985.

[86] W. Szeto, R. Boutaba and Y. Iraqi, *Dynamic Online Routing Algorithm for MPLS Traffic Engineering.* Proceedings of NETWORKING 2002, LNCS 2345, pp. 936-946.

[87] V. Srinivasan, C. F. Chiasserini, P. S. Nuggehalli and R. R. Rao, *Optimal Rate Allocation for Energy-Efficient Multipath Routing in Wireless Ad Hoc Networks.* IEEE Transactions on Wireless Communications, Vol. 3, No. 3, 2004.

[88] S. Subramanian and V. Muthukumar, *Alternate Path Routing Algorithm for Traffic Engineering.* Proceedings of the 15th ICSENG, 2002, http://www.ee.unlv.edu/~venkim/opnet/icseng02iPaper.pdf

[89] I. J. Shapiro and K. S. Narendra, *Use of Stochastic Automata for Parameter Self-Optimization with Multi-Modal Performance Criteria.* IEEE Transactions on Systems Science, and Cybernetics, SSC-5, 1969, pp. 352-360.

[90] P. Spira and A. Pan, *On Finding and Updating Spanning Trees and Shortest Paths.* SIAM Journal of Computing, Vol. 4, No. 3, 1975, pp. 375-380.

[91] M. Schwartz and T. Stern, *Routing Techniques Used in Computer Communications Networks, IEEE Transactions on Communications.* Vol. 28, 1980, pp. 539-552.

[92] S. Suri, M. Waldvogel, D. Bauer and P. R. Warkhede, *Profile-Based Routing and Traffic Engineering.* Computer Communications, Vol. 26, 2003, pp. 351-365.

[93] S. Suri, M. Waldvogel, D. Bauer and P. R Warkhede, *Profile-Based Routing: A New Framework for MPLS Traffic Engineering.* In Quality of Future Internet Services (Ed. Fernando Boavida), Lecture Notes in Computer Science, Springer-Verlag, Vol. 2156, September 2001, pp. 138-157.

[94] M. A. L. Thathachar and B. J. Oommen, *Discretized Reward-Inaction Learning Automata.* Journal of Cybernetics and Information Sciences, Spring 1979, pp. 24-29.

[95] M. A. L. Thathachar and P. S. Sastry, *Pursuit Algorithm for Learning Automata.* Unpublished paper that can be available from the authors.

[96] M. A. L. Thathachar and P. S. Sastry, *Networks of Learning Automata.* Kluwer Academic Publishers, 2003.

[97] M. L. Tsetlin, *On the Behaviour of Finite Automata in Random Media.* Automation and Remote Control, Vol. 22, 1962, pp. 1210-1219. Originally in Avtomatika i Telemekhanika, Vol. 22, 1961, pp. 1345-1354.

[98] M. A. L. Thathachar and P. S. Sastry, *A New Approach to Designing Reinforcement Schemes for Learning Automata.* Presented at the IEEE International Conference on Cybernetics and Society, Bombay, India, January 1984.

[99] M. A. L. Thathachar and P. S. Sastry, *A Class of Rapidly Converging Algorithms for Learning Automata.* IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-15, January 1985, pp. 168-175.

[100] M. A. L. Thathachar and P. S. Sastry, *Estimator Algorithms for Learning Automata.* Proceedings of the Platinum Jubilee Conference on Systems and Signal Processing, Department of Electrical Engineering, Indian Institute of Science, Bangalore, India, December 1986.

[101] M. A. L. Thathachar and P. S. Sastry, *Varieties of Learning Automata: An Overview.* IEEE Transactions on Systems, Man, and Cybernetics Part B, Vol. 32, No. 6., December 2002, pp. 711-722.

[102] R. Viswanathan and K. S. Narendra, *Comparison of Expedient and Optimal Reinforcement Schemes for Learning Systems.* Journal of Cybernetics, Vol. 2, 1972, pp. 21-37.

[103] A. V. Vasilakos and G. Papadimitriou, *Ergodic Discretized Estimator Learning Automata with High Accuracy and High Adaptation Rate for Nonstationary Environments.* Neurocomputing, Vol. 4, pp. 181-196, May 1992.

[104] A. Vasilakos, M. P. Saltouros, A. F. Atlassis and W. Pedrycz, *Optimizing QoS Routing in Hierarchical ATM Networks Using Computational Intelligence Techniques.* IEEE Trans. Syst., Man, and Cybern., Part C, Vol. 33, No. 3, August 2003, pp. 297-312.

[105] V. I. Varshavskii and I. P. Vorontsova, *On the Behavior of Stochastic Automata with a Variable Structure.* Automation and Remote Control, Vol. 24, 1963, pp. 327-333.

[106] Z. Wang and J. Crowcroft, *Quality-of-Service Routing for Supporting Multimedia Applications.* IEEE Journal of Selected Areas in Communications, Vol. 14, No. 7, September 1996, pp. 1228-1234.

[107] K. Wu and J. Harms, *On-Demand Multipath Routing for Mobile Ad Hoc Networks.* Proceedings of EMPCC, Vienna, February 2001.

[108] S. W. H. Wong, *The Online and Offline Properties of Routing Algorithms in MPLS.* M.Sc. Thesis, Department of Computer Science, University of British Columbia, July 2002.

[109] B. Wang, X. Su and P. Chen, *Efficient Bandwidth Guaranteed Routing Algorithms.* Journal of Parallel and Distributed Systems and Networks, 2002.

[110] Y. Xue and K. Nahrstedt, *Fault Tolerant Routing in Mobile Ad Hoc Networks.* Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC), New Orleans, Louisiania, March 2003, pp. 1174-1179.

[111] Y. Xue and K. Nahrstedt, *Providing Fault-Tolerant Ad Hoc Routing Service in Adversarial Environments.* Wireless Personal Communications, Vol. 29, 2004, pp. 367-388.

[112] Z. Ye, S. V. Krishnamurthy and S. K. Tripathi, *A Framework for Reliable Routing in Mobile Ad Hoc Networks.* Proceedings of IEEE INFOCOM, San Francisco, 2003.

[113] *Zipf's law.*, http://www.nist.gov/dads/HTML/zipfslaw.html