12-1987

An Application of Stochastic Learning Automata to Queueing Disciplines

Tearesa Lynn Wegscheid

Western Michigan University

Follow this and additional works at: https://scholarworks.wmich.edu/masters_theses

Part of the Computer Sciences Commons

Recommended Citation
https://scholarworks.wmich.edu/masters_theses/1281

This Masters Thesis-Open Access is brought to you for free and open access by the Graduate College at ScholarWorks at WMU. It has been accepted for inclusion in Master's Theses by an authorized administrator of ScholarWorks at WMU. For more information, please contact maira.bundza@wmich.edu.
AN APPLICATION OF STOCHASTIC LEARNING AUTOMATA TO QUEUEING DISCIPLINES

by

Tearesa Lynn Wegscheid

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science
Department of Computer Science

Western Michigan University
Kalamazoo, Michigan
December 1987
AN APPLICATION OF STOCHASTIC LEARNING AUTOMATA TO QUEUEING DISCIPLINES

Tearesa Lynn Wegsheid, M.S.
Western Michigan University, 1987

There has been extensive research in the area of Stochastic Learning in both Psychology and Computer Science. This thesis examines the literature of Stochastic Learning automata with respect to operation control. It traces the development of an optimal reinforcement scheme, and examines the applications of Stochastic Learning automata in routing. The paper simulates M/M/3 and M/M/5 queueing systems similar to the system described by Glorioso and Osorio. The simulation implements \( L_{r-p} \), \( L_{i-p} \), and \( L_{r-i} \) learning models and compares their performance to Teller Window and Jockeying. A blocking factor is developed to provide more information for the \( L_{r-p} \) learning model. The results show an improvement in the performance of the learning with the blocking factor. The results also demonstrate a possible application of the learning automata in Queueing Theory to create a server device sensitive to the environment.
ACKNOWLEDGEMENTS

I would like to express my appreciation to Dr. Dionysios Kountanis for his time and encouragement throughout the course of this thesis. I have gained much insight into the problems and solution in the development of a thesis. I would also like to thank Dr. Brian Mitchell for his guidance in the literature and his assistance in the development of some elementary ideas. I would also like to thank Dr. Mohammad Meybodi for his suggestions and advice in the development of the simulation. I would like to thank Jim for his patience throughout the hard times. Finally, I would like to thank my wonderful husband, Doug, because, without his assistance and encouragement, the thesis would not have been completed.

Tearesa Lynn Wegscheid
INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of “sectioning” the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.
An application of stochastic learning automata to queueing disciplines

Wegscheid, Tearea Lynn, M.S.
Western Michigan University, 1987
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark √.

1. Glossy photographs or pages ______
2. Colored illustrations, paper or print ______
3. Photographs with dark background ______
4. Illustrations are poor copy ______
5. Pages with black marks, not original copy √
6. Print shows through as there is text on both sides of page ______
7. Indistinct, broken or small print on several pages √
8. Print exceeds margin requirements ______
9. Tightly bound copy with print lost in spine ______
10. Computer printout pages with indistinct print ______
11. Page(s) ________ lacking when material received, and not available from school or author.

12. Page(s) ________ seem to be missing in numbering only as text follows.

13. Two pages numbered ______. Text follows.

14. Curling and wrinkled pages ______

15. Dissertation contains pages with print at a slant, filmed as received ______

16. Other ____________________________

__________________________________

__________________________________

University Microfilms International

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>ii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vi</td>
</tr>
<tr>
<td><strong>CHAPTER</strong></td>
<td></td>
</tr>
<tr>
<td>I. DEFINITION OF THE PROBLEM</td>
<td>1</td>
</tr>
<tr>
<td>Stochastic Learning</td>
<td>5</td>
</tr>
<tr>
<td>Markov Chains</td>
<td>8</td>
</tr>
<tr>
<td>Definitions</td>
<td>9</td>
</tr>
<tr>
<td>Summary</td>
<td>17</td>
</tr>
<tr>
<td>II. RELATED RESEARCH</td>
<td>19</td>
</tr>
<tr>
<td>P, S, Q-Models</td>
<td>31</td>
</tr>
<tr>
<td>Non-Stationary Environments</td>
<td>34</td>
</tr>
<tr>
<td>Hierarchical Models</td>
<td>37</td>
</tr>
<tr>
<td>Application</td>
<td>45</td>
</tr>
<tr>
<td>III. STOCHASTIC LEARNING AUTOMATA IN QUEUEING THEORY</td>
<td>53</td>
</tr>
<tr>
<td>Proposed Model</td>
<td>54</td>
</tr>
<tr>
<td>Program Specification</td>
<td>61</td>
</tr>
<tr>
<td>Data Structures</td>
<td>62</td>
</tr>
<tr>
<td>Program Description</td>
<td>63</td>
</tr>
<tr>
<td>Procedure Description</td>
<td>66</td>
</tr>
<tr>
<td>IV. RESULTS</td>
<td>68</td>
</tr>
<tr>
<td>V. DISCUSSION</td>
<td>85</td>
</tr>
</tbody>
</table>
LIST OF TABLES

1. Mean Delay for 5000 Tasks Versus 1000 Tasks ... 77
2. Mean Delay for .5/.75, .5/1.0, .5/1.5 and 10/15 Second Arrival/Departure Times ......... 78
3. Mean Delays for 3 Queues Versus 5 Queues .... 79
4. Mean Delay for Blocking Versus Fixed Disciplines and Learning Models ........ 84
LIST OF FIGURES

1. The Stochastic Learning Model ..................... 10
2. Model of the Hierarchical Stochastic Learning Automata ............................................... 38
3. Model for Program Queueing Simulation ............ 55
4. Comparison of the Learning Models With Step Size .1 ...................................... 71
5. Comparison of the Learning Models With Step Size .3 ...................................... 72
6. Comparison of the Learning Models With Step Size .7 ...................................... 73
7. Comparison of Teller and Jockey .................... 74
8. Comparison of \( L_{r-p} \) to Jockey and Teller ........ 75
9. Comparison of \( L_{r-i} \) to Jockey and Teller ........ 76
10. Comparison of \( L_{i-p} \) to Jockey and Teller ........ 77
11. Comparison of the Learning models for 5 Queues .. 80
12. Comparison of Fixed Disciplines for 5 Queues ... 81
13. Comparison of the Blocking Factor With the \( L_{r-p} \) Model .............................. 82
14. Comparison of the Blocking Factor With the Fixed Disciplines ............................. 83
CHAPTER I

DEFINITION OF THE PROBLEM

What is meant by "An Application of Stochastic Learning Automata to Queueing Disciplines?" What is stochastic? What is an automaton? What is a queueing discipline? Finally, how do they relate to Computer Science? As the title implies, this thesis uses theories from several different disciplines. Issues related to Artificial Intelligence, Stochastic Learning, and Queueing Theory are addressed.

Artificial Intelligence is a very popular "buzz" term. Trade journals publish articles on a regular basis. Expert Systems, a subarea of Artificial Intelligence, is the newest technology to come out of the Computer Science field. What is Artificial Intelligence? In the simplest terms, Artificial Intelligence is the development of a computer program which exhibits some form of intelligence. Artificial Intelligence has been in various stages of development for the last 30 years. The first chess playing computer programs were attempts at Artificial Intelligence. Natural language processors and general problem solving programs were other attempts at Artificial Intelligence. These programs used techniques such as heuristic programming, inductive
inferential techniques, and automata models to develop intelligence.

Heuristics are used in game-playing, such as chess, and generally define tricks or rules of thumb that lead to short cuts in search times. They solve the problem in a "good enough" fashion. With heuristics, the program does not learn, but simply follows the predetermined rules. When the external conditions change, the rules need to be updated to account for the changes. The program cannot update the rules, thus, it does not adapt to changes in the external conditions. The heuristic techniques allow the program to exhibit intelligent behavior, but the techniques do not learn. The first objective of this thesis is to develop an Artificial Intelligence technique which resembles learning. Through the use of a learning automaton, a program can gain the capability of learning and remembering.

The performance of a learning automaton is dependent on the environment, making the program sensitive to changes in the environment and thus adaptive to a particular situation. The research conducted in the area of Stochastic Learning automata has produced models with optimal performance rather than performance which is "good enough." Thus, by the development of a learning model, a program can learn the correct response, is sensitive to its environment, and limits the number of predetermined rules. A learning automaton is a
significant improvement over the heuristics used in the past.

An interesting aspect of a learning automaton is its similarity to human learning. It has been shown in psychology that current behavior exhibited by an organism is based on the past history of the behavior and the environment's reaction to that behavior (Skinner, 1938). In some fields of psychology such as stimulus sampling, the environment influences an organism's behavior by rewarding or punishing the organism. The learning automaton model considers the basic principle of operant conditioning. The principle states that if a response to a given stimulus is reinforced, then the probability of the occurrence of the response in the presence of the stimulus is increased. The learning model develops a way of changing probability values for a set of responses depending on the past history of the responses and the environment's reactions to those responses.

In behavioral psychology, a response from an organism is either reinforced or punished by the environment. If the organism receives reinforcement or punishment from the environment as a result of its response, the probability of the occurrence of the response respectively increases or decreases (Skinner, 1938). There is a similar interaction between the learning automaton and the environment. As the automaton makes a choice, the environment either rewards or
penalizes the automaton depending on the choice made and the penalty probability value of the environment.

How does the development of a Stochastic Learning automaton relate to a queueing discipline? A queueing system consists of customers, one or more lines (queues), and a service provided by one or more servers. The objective of Queueing Theory is to develop a set of rules, called a queueing discipline, which assigns customers to a place in line in such a way that the total amount of time spent in the system is minimized. In the past, fixed queueing disciplines were developed to make the task assignments. However, the fixed disciplines made the assignments in the same way, regardless of the environmental conditions. Under one set of conditions, the fixed disciplines performed optimally, yet under a different set of conditions, they were not adequate.

The objective of this thesis is to develop a queueing discipline which is sensitive to the environment by the use of Stochastic Learning automata. The thesis uses a Stochastic Learning automaton as the tool to develop intelligence. A simulation is written to apply Stochastic Learning automata to a queueing situation. The thesis compares the performance of the learning automaton to traditional queueing disciplines and determines whether or not the learning automaton model is better than queueing disciplines used in the past.
Stochastic Learning

The discussion of Stochastic Learning models has appeared across several disciplines in the past thirty years. It first appeared in the psychology literature in the early 1950s. Psychologists attempted to explain learning by the use of statistics. Probability theory was used as a tool to describe the underlying behavioral processes referred to as learning. A number of probabilistic theories appeared in the literature. Of these theories, Estes' (1950) theory of stimulus sampling and Bush and Mosteller's (1951) theory of Stochastic Learning had the strongest impact on the research in mathematical learning models.

Estes' theory of stimulus sampling was the first attempt to statistically analyze behavior. Briefly, stimulus sampling theory states that a learning experience is a discrete change in the state of behavior as the result of an environment change. From a large sample size of stimuli, the organism responds to one or more stimuli. As the result of the response, associations are established between the stimulus situation and the organism's response. The associations provide the basis for learning a given response in a particular situation. The association between the stimulus and the response is assigned a probability value. As the associations weaken or strengthen, the
probability values decrease or increase to reflect the change. To account for generalization between two situations, common elements between two stimuli activate common associations. The degree of commonality between the situations produces a common response proportional to the amount of similarity. As the situations become more common, the responses become similar.

Bush and Mosteller (1951) developed a theory similar to stimulus sampling, but used stochastic processes (random or probabilistic processes which change through time) in the description of learning. The result of Stochastic Learning theory provided a detailed prediction of behavior for a variety of learning data which earlier theories could not explain. Stochastic processes are characterized by probability values representing the associations between responses and stimuli. The probability of the next response given the stimulus situation is completely determined by the probability value of the current state of behavior. The organism produced a response associated with the highest probability value. Markov chains or processes are used to represent the Stochastic Learning process. An example of a Markov chain in learning involved a stimulus element, a response, and a probability value representing the amount of association between the stimulus and the response. The change in the probability represented the change in the association.
Psychologists continued to elaborate on the Stochastic Learning theories and stimulus sampling theory for more than a decade. Stochastic Learning theory provided a good model for simple learning, but it could not be adapted to more complex learning models. Thus, by the mid-1960's, most of the research in Psychology shifted away from Stochastic Learning. The shift did not occur before Stochastic Learning theory was established in other disciplines. In particular, Tsetlin (1961) used finite Markov chains to describe a finite automaton in a random environment. The results of Tsetlin's paper encouraged other automata theorists to examine the use of Stochastic Learning automata to optimize operations in various systems.

The approach taken by a learning automata theorist is different than the approach of a psychologist. In learning automata theory the goal is to build algorithms exhibiting the prespecified behaviors defined by the psychological model. The learning system produces a systematic change in behavior in an optimal manner. The model may approach the optimization of learning in several ways. One approach is to reduce the learning task to an optimal set of parameters and apply some type of heuristic to solve the problem. Another approach is to view the problem as finding an optimal action out of a set of allowable actions by the use of a stochastic automaton. The second approach brings the topic of
learning automata back to stochastic processes and Markov chains, thus establishing the association of Markov chains and automata theory.

Markov Chains

In automata theory, an automaton is specified by a set of states and state transitions. A Markov chain is defined by a number of states and the probability of the system being in one of the states. The probability of the system moving from one state to the next is referred to as the transitional probability. The transitional probabilities are completely determined by the current state of the system. The current state provides the necessary information for the transition to the next state. The transitional probabilities are altered after each transition occurs. The set of transitional probabilities are independent from each other; that is, the transition probability from one particular state does not depend on how that particular state has been reached.

Initially, the probability of any one of r transitions emanating from a state is equal to 1/r. The set of r actions is exhaustive and the actions are mutually exclusive. The sum of the probabilities equals unity. A transition to a state must occur on each trial. If the state of the Markov chain is known at time $t$, the probability of any state at time $t+1$ is also known.
The model converges when knowledge of the final state probabilities is achieved. There are two types of convergence theories discussed: Ergodic and Absorbing barriers. Ergodic convergence occurs when the probability values at state $t$ and $t+1$ are made as close to one another as desired. The distribution of probability values becomes asymptotic. Absorbing barriers occur when the probability values approach an asymptote of unity. Absorbing barriers are referred to as perfect learning.

The development of a learning model is concerned with the convergence of the learning automata. Learning automata with both Absorbing and Ergodic convergence are described. Thus, Markov chains are a means of describing the automaton’s behavior and provide a means of describing the actions of the automaton.

Definitions

In the development of learning automata there are several definitions. In the discussion of Stochastic Learning automata, Narendra and Thathachar (1974), the model is represented as a "student-teacher" model. The stochastic automaton corresponds to the "student," or the learner. A random environment represents a probabilistic "teacher." The action of the stochastic automaton is the input to the environment. The environment responds to the action by producing an outcome. The response from
the environment is dependent on a penalty probability set. The outcome, generally represented by 0 or 1, serves as the input to the stochastic automaton. The outcome is used to alter the probability vector $P$, which is associated with all possible actions. The probability values are updated by a reinforcement scheme. A cycle is established: the stochastic automaton selects an action depending on the probability vector; the environment responds to the selected action; and the probability vector is updated depending on the response from the environment. Figure 1 illustrates the feedback system between the environment and the learning automaton.

Penalty Probability Set
$(c_1, c_2, \ldots, c_r)$

Environment

Action $\alpha (\alpha_1, \ldots, \alpha_r)$

Input $(0,1)$

$B(t)$

$(P,A)$

Stochastic Automaton

Probability Vector
$(p_1, p_2, \ldots, p_r)$

Figure 1. The Stochastic Learning Model
The Stochastic Learning model is defined by two components, the Stochastic Learning automaton and the environment. Formally, the Stochastic Learning automaton is defined as a sextuple \( \langle X, \Psi, \alpha, P, A, G \rangle \), where

- \( X \) is the finite input set.
- \( \Psi \) is the finite set of internal states.
- \( \alpha \) is the finite set of outputs or actions (\( \alpha_1, \alpha_2, \ldots, \alpha_r \)).
- \( P \) is the probability vector which determines the choice of states at each stage.
- \( A \) is the algorithm which generates \( P(t+1) \) from \( P(t) \).
- \( G \) is the mapping of the internal states to the actions.

The probability of an action is given by

\[
P_i(t) = \Pr(\alpha(t) = \alpha_i)
\]

that is, \( P_i(t) \) equals the probability of action \( \alpha_i \) at time \( t \), where \( \alpha(t) \) represents the action at time \( t \) (\( t=0,1,2,\ldots \)).

The sum of the probability vector is always equal to unity; that is

\[
\sum_{i=1}^{r} P_i(t) = 1 \quad \text{for all } t
\]

This condition maintains unity of the transitional probabilities since the automaton is viewed as a Markov chain. A probability vector \( P \) represents the
transitional probability of the Markov chain. Associated with each probability value is an action, \( \alpha_i \). The movement from one state to the next is determined by \( P \).

The second component, the environment, is defined as a triple \( <\alpha, C, r> \), where \( \alpha \) is the set of actions of the automaton \( (\alpha_1, \alpha_2, \ldots, \alpha_r) \), \( C \) is the penalty probability vector, and \( r \) is the total number of actions of the automaton. The penalty probability vector \( C_i \), \( i=1..r \), determines the response of the environment, or outcome, for the action \( \alpha_i \). For all \( \alpha_i \) at time \( t \), there is a corresponding penalty probability, \( C_i \). \( C_i \) is the probability that \( \alpha_i \) is rewarded or penalized by the environment. Since the objective of the learning automaton is "learning" the correct response, the actual values for \( C_i \) are unknown by the automaton. The penalty probability vector may vary with time, creating a non-stationary random environment, or they may remain constant, creating a stationary random environment.

The outcome at time \( t \) is represented by \( B(t) \). Generally, \( B \) has the values of 0 or 1, however, there are different models which can assign other values to \( B \). When \( B(t)=0 \), the outcome is called a non-penalty; when \( B(t)=1 \), the outcome is called a penalty.

Finally, as in the Markov chain, the transitional probabilities are updated after each transition occurs. The learning automaton updates the probability vector by an update algorithm. The updating of the probability
vector determines the performance of the learning automaton. The operation on the probability vector $P(t)$ considers the current probability value of $P(t)$, the actions $\alpha(t)$ and the outcome $B(t)$ at time $t$.

The updating operation is referred to as the reinforcement scheme. If the learning automaton selects an action $\alpha_i$ at instant $t$ and the environment outputs a non-penalty, the action probability $P_i(t+1)$ is increased and the other components of $P(t+1)$ are decreased. For a penalty outcome, $P_i(t+1)$ is decreased and the other components of $P(t+1)$ are increased. Schemes are developed in which the action probability $P_i(t+1)$ retains the previous value of $P_i(t)$. These schemes are known as "inaction" reinforcement schemes.

The updates of the action probability vector is conducted by the reinforcement scheme. The automaton is capable of "learning" the desired behavior by repeated updating of the probability vector. Initially, each action $\alpha_i$ has an equal chance of being the final behavior observed, however, since only one state of the automaton is the desired state, the automaton should converge to that state. By properly choosing the reinforcement scheme, the automaton converges to the desired state. It is difficult to calculate the actual values of $P_i$ since the probability values are random variables. Therefore, it is necessary to establish bounds on the probabilities rather than the actual values. For each reinforcement
scheme developed, the lower and upper bounds on the probability values were determined. Norman (1968) developed the boundaries of convergence for a general class of linear Stochastic Learning models with "distance diminishing" operators, however, many of the schemes studied do not satisfy the distance diminishing condition. Lakshmivarahan and Thathachar (1976) developed boundaries on the convergence of the probabilities for a general class of non-linear reinforcement schemes which do not satisfy the distance diminishing condition. The boundaries on other reinforcement schemes are discussed as part of the development of those schemes. The type of convergence characterizes the automaton.

The performance of the learning automaton is related to the structure of the reinforcement scheme. The learning model is classified on the basis of the property exhibited by a reinforcement scheme. There are several definitions which describe the performance of the learning automata. Generally, the reinforcement scheme is characterized by more than one of the performance measurements.

The average penalty, $M(t)$, is a basic measurement of performance. $M(t)$ is defined as the amount of penalty the automaton receives from the environment. The average penalty is equal to the sum of the probabilities of the
responses multiplied by the penalty value corresponding to the response, that is

\[ M(t) = E[B(t)/P(t)] = \sum_{i=1}^{r} P_i(t)C_i \]

Intuitively, the average penalty is low when the larger penalty values correspond to the smaller penalty probabilities. The average penalty is higher when the larger penalty values correspond to larger penalty probabilities. Initially, the automaton does not have any prior information, and each element of the probability vector is assigned an equal probability value of \(1/r\). The initial average penalty is denoted by \(M_0\). \(M_0\) is equal to the average penalty of a response chosen strictly at random.

\[ M_0 = \frac{C_1 + C_2 + \ldots + C_r}{r} \]

For every value of \(M(t)\), the learning automaton chooses an action which results in a smaller average penalty than an action chosen strictly at random. When \(M(t)\) is less than \(M_0\), the automaton is referred to as expedient. Symbolically, expediency is defined as

\[ \lim_{t \to \infty} E[M(t)] < M_0 \]

When the learning automaton obtains the desirable actions by decreasing \(E[M(t)]\) in a monotonic fashion, the learning automaton is considered to be absolutely
A learning automaton is absolutely expedient if \( E[M(t+1)|P(t)] < M(t) \) for all \( t \), all \( P(t) \), and all values of \( C_i \). Absolute expediency implies the expected value of \( M(t+1) \), given the probability value of \( P(t) \), is less than the average penalty \( M(t) \) at time \( t \). If the average penalty \( M(t+1) \) is always less than the average penalty \( M(t) \), then the learning automaton is absolutely expedient. Therefore, the absolutely expedient automaton chooses the action associated with the smallest degree of penalty.

A desirable feature of the learning automaton is to reduce the average penalty by selecting the proper action. When the learning automaton reduces the penalty, it is called optimal. Formally, optimality is defined as

\[
\lim_{t \to \infty} E[M(t)] = \min_i [C_i]
\]

or the expected value of \( M(t) \) is equal to the minimum average penalty. In the case of optimality, the learning automaton produces the action associated with the minimum penalty probability.

Even though optimality is desired, certain conditions may prevent it. In such a case, a suboptimal performance of the learning automaton is considered \( e \)-optimal. A learning automaton is called \( e \)-optimal if

\[
\lim_{t \to \infty} E[M(t)] = \min_i [C_i] + e
\]

The expected penalty value for a response is less than the minimum average penalty plus \( e \). \( e \) is an arbitrary...
variable accounting for the conditions which do not satisfy the optimal definition. When an automaton is e-optimal, the performance of the automaton can be close to optimal by choosing suitable parameters for the reinforcement scheme. The effects of the reinforcement scheme on the performance is discussed later.

In summary, a learning model is considered to be expedient if the action chosen at time t results in a smaller amount of penalty than an action simply chosen at random. If the learning model is expedient and the amount of penalty decreases monotonically, then the learning automaton is considered to be absolutely expedient. The automaton is optimal if the minimum $C_i$ is chosen by the automata. If conditions exist within the learning model which prevents the model from being optimal, then the learning model is considered as e-optimal. Optimality implies e-optimality, which implies expediency. If a class of reinforcement schemes are absolutely expedient, then e-optimality is implied.

Summary

Most of the work in the area of Stochastic Learning automata is done by the automata theorists. The present thesis examines the development of an optimal reinforcement scheme and the application of Stochastic Learning automata in Queueing Theory.
In chapter II, an in-depth literature review is presented. The development of reinforcement schemes, S-models, Q-models, P-models, non-stationary random environments, and hierarchical models are discussed. Current applications of Stochastic Learning automata in traffic routing are presented.

In chapter III, a brief explanation of Queueing Theory is presented. The development of the application of a Stochastic Learning automaton and the description of the simulation of that automaton is provided.

Chapter IV presents the results of the simulation. A discussion of the results is presented in chapter V, and the conclusion is presented in chapter VI.
CHAPTER II

RELATED RESEARCH

Since the reinforcement scheme is the critical factor of the learning model, extensive research has been conducted in the development of an optimal scheme. The research in the 1960's designed reinforcement schemes classified by only one of the performance measurements already discussed. Later, researchers developed reinforcement schemes which were characterized by several performance measurements, making them more general. The following section provides a historical development of the reinforcement schemes. For each reinforcement scheme the necessary and sufficient conditions for convergence are stated.

Tsetlin (1961) was the first to introduce behavior as a finite automaton. His model did not use a reinforcement scheme; it was a constant-structure model. The environment penalized the automaton's action according to a probability assigned to each action of the automaton. Tsetlin provided an example of an automaton which demonstrated expediency. The mathematical expectation of penalty $M(t)$ served as the measure of expediency. When the expected penalty of the automaton approached the minimum expected penalty value, $M_{\text{min}}$, the
automaton demonstrated expedient behavior. Tsetlin showed that an automaton in a stationary random environment followed the properties of a Markov chain. He assumed the Markov chain was Ergodic; thus a final set of probabilities existed for the automaton in the specified environment. The results showed that if the chain was Ergodic, the final probabilities of the actions were established.

Varshavskii and Vorontsova (1963) extended Tsetlin's finite automata to a Stochastic Learning automata model. The model was a variable-structure model since the probability values for the actions varied with time. The stochastic automaton was specified by two inputs and two internal states. For each pair of input states, there existed a probability value which determined the transfer of the automaton from one state to the other state. The Stochastic Learning model was tested in stationary and non-stationary random environments. The reinforcement scheme increased or decreased the probability of the action if the action received a non-penalty or penalty input. The reinforcement scheme determined the probability value at time \( (t+1) \) from the current probability value at time \( t \). The stationary scheme used the probability of being in the current state, the probabilities of being in state 1 or state 2, and the penalty value from the environment. The non-stationary scheme was developed which decreased or increased the
corresponding transfer probability by a set amount depending on a penalty or non-penalty from the environment.

Varshavskii and Vorontsova showed the functioning process of a stochastic automaton with a variable structure. The results showed the variable-structure automaton was equivalent to an automaton with a linear reinforcement function and an infinite number of states. The models exhibited expediency specified under the criteria expressed by Tsetlin. The results demonstrated optimal performance of the automaton in a stationary environment. For a Stochastic Learning automaton in a non-stationary environment, the automaton approached the optimal structure according to the mathematical expectations of the penalty. For a sufficient number of original states, the average value of penalties for a stochastic automaton in the non-stationary environment coincided with the mathematical expectation of the penalty received by a finite automaton.

Vorontsova (1965) continued the search for an optimal reinforcement scheme by introducing a non-linear model. The model was restricted to an automaton with two states, two inputs, and two outputs. The degree of change in the action probabilities was given by the magnitude of the increment over time. The magnitude of the increment was measured in both discrete and
continuous intervals. The reinforcement scheme was given by

\[
P_i(t+1) = \begin{cases} 
  f+(P_i(t)) & \text{with probability of } P_i(t)p_1 \\
  f-(P_i(t)) & \text{with probability of } P_i(t)(1-p_1) \\
  f+(1-P_i(t)) & \text{with probability of } (1-P_i(t))p_2 \\
  f-(1-P_i(t)) & \text{with probability of } (1-P_i(t))(1-p_2) 
\end{cases}
\]

where \( p_1 \) and \( p_2 \) are the probabilities associated with \( C_1 \) and \( C_2 \). \( f+(P_i(t)) \) represented the increase of \( P_i \) if the action \( A_1 \) was chosen and a reward was the outcome to the system. \( f-(P_i(t)) \) was the decrease in \( P_i \) if the action resulted in a penalty. In a similar matter, \( f+(1-P_i(t)) \) and \( f-(1-P_i(t)) \) represented the increase or decrease in \( P_i \) if the action \( A_2 \) was chosen and a reward or penalty was produced. In order to preserve the normalization condition, the same quantity with the opposite sign was added to the other element of \( P(t) \).

The boundaries of convergence for the reinforcement scheme were mathematically derived for both the discrete and the continuous case. Convergence depended on the expectation of the increment and the square of the increment. Conditions were imposed to insure Absorption at both boundaries. A computer simulation was conducted for both cases. The results showed, for both models, that the optimal reinforcement scheme depended on the values for the penalty probabilities \( p_1 \) and \( p_2 \). In addition, the results did not demonstrate a significant difference in the linear versus non-linear case.
Shapiro and Narendra (1969) considered the use of Stochastic Learning automata in a parameter optimization problem. They attempted to relate the concepts of automata theory and Psychology Learning Theory. Two models were developed, one based on a linear system of response sets of reward and inaction, and the other based on a linear system of reward and penalty responses. The reinforcement scheme considered the probability of the action \( A_i(t) \), the probability value \( P(t) \), an identity matrix, and a step size variable. Inaction was defined, given that \( A(t)=A_i \), as \( P_i(t)=1 \) and \( P_j(t)=0 \) (\( j \neq i \)) if the environment's response was a non-penalty, and \( P(t+1)=P(t) \) if \( A(t) \) was followed by penalty response from the environment. The reward changed the probability values, but the penalty did not affect the probability values. Under the conditions, the schemes were optimal rather than expedient. They noted problems associated with the large parameter adjustments in the automaton. In 1972, Viswanathan and Narendra made a note to the work of Shapiro and Narendra. Viswanathan and Narendra noted the reinforcement scheme did not converge optimally. By selecting a sufficiently small step-size factor, the reinforcement scheme simply approached e-optimal convergence. Sawaragi and Baba (1974) showed the learning performance of the two reinforcement schemes were e-optimal under certain conditions. If there was little a priori information about \( C_i \), the first scheme
was not expected to be e-optimal. The reward and inaction scheme gave a higher rate of learning than the reward and penalty scheme when certain conditions were satisfied.

Chandrasekaran and Shen (1968) extended the reinforcement scheme from a 2-state model to a m-state model. They looked at 2 distinct reinforcement schemes. One scheme was based on penalty probabilities and the other was based on penalty strengths. The reinforcement schemes introduced the idea of P-models and S-models, discussed later in the thesis. They evaluated the performance of the learning automaton by the rate of learning. The rate of learning was determined by the convergence of the learning automaton and the amount of learning determined by the expediency of the model. The scheme was a modification of a linear model for the experimental subject control event described in Mathematical Psychology.

The scheme was expedient since the reinforcement scheme produced a monotonically non-increasing function. Conditions were imposed to insure the model exhibited expedient, optimal and sub-optimal performance, based on the learning parameter $A$, step size. Convergence of learning was determined by the rate of expediency. The rate of convergence was reduced by a non-linear transformation of the penalty strengths. In the case of
expediency and convergence, results for a 2-state model were generalized to the m-state model.

Lakshmivarahan and Thathachar (1972) described two non-linear schemes which were optimal irrespective of the environment. The schemes were Non-Linear Reward-Inaction (NR-i) and Non-Linear Penalty-Inaction (NP-i).

The NR-i scheme was defined as

\[
P_i(t+1) = \begin{cases} 
P_i(t) & \text{if } B=1, \text{ penalty} \\
1-f[P_j(t)] & \text{if } B=0, \text{ non-penalty} \end{cases}
\]

where \( j \neq i \), \( f(x) \) was continuous, \( f(x) < x \) for all \( x \) in \((0,1)\), and \( f(x) = x \) possibly at 0 and 1. The reinforcement scheme increased \( P_i(t+1) \) if the outcome was a non-penalty, \( B=0 \), but did not change \( P_i(t+1) \) if the same action resulted in a penalty outcome, \( B=1 \).

The NP-i scheme was defined as

\[
P_i(t+1) = \begin{cases} 
g[P_i(t)] & \text{if } B=1, \text{ penalty} \\
P_i(t) & \text{if } B=0, \text{ non-penalty} \end{cases}
\]

where \( g(x) \) satisfied the conditions given for NR-i. The scheme decreased \( P_i(t+1) \) if the outcome was a penalty, \( B=1 \), but did not change \( P_i(t+1) \) if the same action resulted in a non-penalty outcome, \( B=0 \).

Lakshmivarahan and Thathachar (1972) compared non-linear against linear models. They used the average
rate of learning, defined as the expected change in the average penalty, and the average variance as the basis for their comparisons rather than the speed of convergence. The results showed the variance and the rate of learning were dependent on the non-linear function. It was not possible to minimize variance and maximize rate of learning simultaneously. In addition, the linear $L_{r-i}$ model did not differ significantly from the non-linear $N_{r-i}$ model when the rate of learning was compared. Comparing the $L_{r-i}$ and $N_{r-i}$ schemes, the average rate of learning was the same but the average variance for $N_{r-i}$ was less than the $L_{r-i}$. Both models were optimal. Lakshmivarahan and Thathachar, using the simple condition of symmetry, proved the necessary and sufficient conditions to insure the scheme was optimal. The convergence of the reinforcement scheme was shown by invoking the Martingale Theory.

Lakshmivarahan and Thathachar (1973) described a general class of non-linear reinforcement schemes for a multistate stochastic automaton in a stationary random environment. The reinforcement scheme suggested that two updating schemes were needed. One scheme updated the probability of the action selected by the automaton at time $t$. A second scheme updated the probability of the actions not selected. Lakshmivarahan and Thathachar's technique depended on the reactions and the dichotomy of actions: the selected or non-selected. They showed the
condition of symmetry of the non-linear functions in the reinforcement scheme was the necessary and sufficient condition for absolute expediency. The conditions were

\[
\frac{f_1(P)}{P_1(t)} = \frac{f_2(P)}{P_2(t)} = \ldots = \frac{f_r(P)}{P_r(t)} = \lambda(P)
\]

\[
\frac{g_1(P)}{P_1(t)} = \frac{g_2(P)}{P_2(t)} = \ldots = \frac{g_r(P)}{P_r(t)} = \mu(P)
\]

where \(\lambda\) and \(\mu\) were arbitrary continuous functions. The conditions stated that in order to obtain absolute expediency one type of updating was needed for the probability of the action selected by the automaton at time \(t\) and a different type of updating was needed for all the other action probabilities. A distinction was not made between the actions not selected by the automaton in the sense that the ratio \(P_j(t+1)/P_j(t)\) was the same for all the actions. This produced a simple condition of symmetry of the non-linear function. They used the speed of convergence in comparing the different schemes. The comparison between models showed the hybrid schemes with linear rewards and time varying schemes with non-linear penalty converged faster than the other schemes.

Lakshmivarahan and Thathachar (1976) studied the convergence properties of a general class of non-linear reinforcement schemes. One reinforcement scheme was a non-linear scheme of the reward-penalty type \((N_{r-p})\). By properly choosing the functions \(\lambda\) and \(\mu\), the scheme
represented a variety of algorithms. In particular, the reward-inaction ($N_{r-i}$) scheme was obtained by setting $\mu=0$ and the inaction-penalty scheme ($N_{i-p}$) was obtained by setting $\lambda=0$.

Lakshmivarahan and Thathachar (1976) derived properties for convergence which allowed the learning automaton to converge only to the desired state. Their approach was similar to that of Norman (1968), but their schemes were non-linear and did not necessarily satisfy Norman's "distance diminishing" requirements. They showed that absolutely expedient algorithms were e-optimal, hence sufficient conditions for designing e-optimal schemes were made available.

Lakshmivarahan and Thathachar (1976) showed the absolutely expedient reinforcement schemes lead to optimality in all stationary random environments. They showed the two important factors in the design of a reinforcement scheme were the speed of convergence and the probability of convergence to the desired state. There existed a tradeoff between these two factors. On the basis of the results of the computer simulation, the non-linear schemes designed exhibited superior performance.

Aso and Kimura (1979) significantly extended the class of automata which were absolutely expedient. The results of their work implied that all the results of Lakshmivarahan and Thathachar (1976) were a special case.
Aso and Kimura (1979) made the reinforcement scheme dependent on the action selected. This extension led to a useful reinforcement scheme.

Aso and Kimura (1979) introduced a new term: stochastic vector automaton (SVA), to emphasize that the stochastic vector itself was treated as the state of a stochastic automaton. A stochastic vector automaton was defined by \(<R, A, S, V, T, G>\). R, A, and S were sets of inputs, outputs, and discrete states. V was the set of stochastic vectors on S. T was a mapping from \(V \times S \times R\) to V, and G was an \(S \times A\) stochastic matrix. They defined the probability vector as a stationary Markov process. The vector valued Markov process was called a semi-martingale if for each value either

\[
E(P(t+1)/P(t)) \leq P(t)
\]

or

\[
E(P(t+1)/P(t)) \geq P(t)
\]

for all \(t\) that is, \(P(t)\) forms either a super-martingale or a sub-martingale. If the Markov process was a semi-martingale then the limit of \(P(t)\) existed.

Aso and Kimura (1979) showed that the new set of reinforcement schemes were absolutely expedient. The results held for \(P\)-models, \(Q\)-models, and \(S\)-models. The concept of Stochastic Vector Automata clarifies relations among stochastic automata, and was easily extended to stochastic matrix automata, which gave the general form
of many kinds of extended learning automata in a non-stationary random environment.

Meybodi and Lakshmivarahan (1982) studied a class of absorbing barrier reward-penalty schemes which displayed identical behavior under reward and penalty. They referred to the new class of schemes as strong absorbed expediency, and provided the necessary and sufficient conditions for the new class of schemes. For the scheme to be absorbing, there must be at least one absorbing state. When the scheme followed the conditions held by absolute expedient schemes plus the conditions held true for all of the states which were not absorbing, then the scheme was strongly absolutely expedient.

The first step in showing e-optimality was to derive conditions on the reinforcement scheme such that the Markov process converges with a probability of unity. The two basic rules that controlled the convergence of the model were the choice of the function such that the amount added or subtracted from $P_j$ was made proportional to $P_j$, and to find a quantity for the distribution of $P$. They found the necessary and sufficient conditions for e-optimality of the general class of learning algorithm by establishing the conditions for the learning algorithm to have only one absorbing state.
P, S, Q-Models

Another variable in the Stochastic Learning automaton is the input value entering the automaton. When the automaton receives information from the environment in the form of a penalty, the information is processed by the reinforcement scheme to alter the state probability distributions. Three types of learning models are described which explore the effects of the different values from the environment. The models are distinguished in terms of the output values from the environment. Most of the work on the effects of the input to the automaton was done between 1965 and 1973.

The models described up to this point are characterized by an output of 0 or 1; these models are called P-models. In other cases, the environment's output may take on a finite number of values or even a continuum of values. In particular, if the environment's response takes a finite number of values in \([0,1]\), the environment is called a Q-model. In the case where the response from the environment lies in the interval \([0,1]\), the model is called an S-model. Throughout the literature, attempts were made to extend the class of reinforcement schemes for P-models to S-models.

The use of S-models in multi-model search problems was first suggested by McLaren (1966). He suggested a number of reinforcement schemes with the penalty value in
the interval \([0,1]\). He incorporated an evaluation section into the learning model, which measured the amount of penalty. A transformation was used to change the outcome to a more meaningful value, considering the averages, the weights of the values, and the normalized value. By taking into account the amount of variation in the value, the transformation was neither specified by the designer nor heuristically selected according to some desirability scale. Two problems arose as a result of the approach. While optimality and expediency were proven for the P-models, similar results were not obtained for S-models. In addition, the approach required the a priori knowledge of the lower and upper bounds on the performance for the P-models.

Chandrasekaran and Shen (1968) considered both P-models and S-models in the development of an optimal model. The penalty strengths were derived from the performance index which considered the penalty strength of each state. The transformation of the penalty strength resulted in producing smaller values as the asymptotic state probabilities increased. Thus, the speed of convergence was less, even though the transformations improved the expediency of the probability values. They concluded that none of the P-models investigated were optimal but that the P-models were useful in the design of a control when the
compromise between learning rate and ability of the automaton was considered.

Viswanathan and Narendra (1973) described a procedure which extended P-model reinforcement schemes into the S-model reinforcement schemes. Since the S-model schemes required a priori knowledge of the upper and lower bounds on the performance indices, they developed a procedure to estimate the bounds for the P-models. The performance indices were represented by the output values from the environment. For the P-models, the environment action was the set \( x = \{1, 0\} \). Then \( x(t)|\alpha_i = 1 \) (a penalty) with the probability \( c_i \) and \( x(t)|\alpha_i = 0 \) (a reward) with the probability of \( 1 - c_i \), where the environment is defined by \( \{c_1, c_2, \ldots, c_r\} \). For the S-model, \( x = \{0, 1\} \). \( S_i \) estimated the value of \( x(t)|a_i \).

\[
S_i \triangleq E[x(t)|a_i] \quad i = 1, 2, \ldots, r
\]

then the expected value from the environment to the automaton \( M(n) \) was

\[
M(n) \triangleq E[x(t)] = \sum_{i=1}^{r} E[p_i(t)] S_i
\]

By letting \( M \) equal the limit of \( M(t) \) as \( t \) approached infinity, then

\[
M = \frac{1}{r} \sum_{i=1}^{r} S_i
\]
and the learning automaton was expedient, and if 
\( M = \min[S_i] \) then the learning automaton was optimal.

Viswanathan and Narendra used a computer simulation
to demonstrate the convergence of the reinforcement
schemes and the success of the proposed S-model automaton
in a multimodal search problem.

### Non-Stationary Environments

Up to this point, the thesis deals with learning
models in a stationary random environment, that is, the
random environment \( C \) characterized by \( C(p_1, p_2, \ldots, p_n) \).
Each \( C_i \) is unknown but fixed. However, it can be shown
that the output of an environment depends not only on the
input at time \( t \) but also on preceding input. The
environment may be designed to have an internal state
memorizing past behavior, that is, the penalty values of
the environment change, depending on the past
experiences, thus creating a non-stationary environment.
There are three different types of non-stationary
environments: switching, periodic, and slowly varying.

A switching environment is one which switches
between a number of stationary environments according to
a Markov chain. Tsetlin (1961) established the existence
of an optimal memory capacity for the automaton in this
environment. The performance of a variable-structure
stochastic automaton in such non-stationary environments
was investigated. He concluded that it was difficult to prove the automaton reached a minimum state for the values of the environment. There was not an advantage in comparing the expediency of an automaton in a stationary environment to one in a non-stationary environment.

A slowly varying environment is one that "slowly" varies the penalty probabilities of the environment in time. In this case, an e-optimal scheme tends to lock on to a certain action, thereby losing its ability to change. If the frequency of variation is sufficiently small, then the $L_{r-p}$ scheme, which is expedient, seems to function satisfactorily.

A periodic environment is one which the penalty probabilities of the environment periodically vary in time with a common period $t$. The period $t$ is divided into $n$ intervals. A system of $n$ automata is used so that one automaton is in operation at an instant of time and each automaton operates once in every period. The environment is characterized by the penalty probability set $[c_{11}, c_{12}, \ldots, c_{1r}, c_{21}, \ldots, c_{2r}, \ldots, c_{n1}, c_{n2}, \ldots, c_{nr}]$, where $n$ is the period of time for $c_i$ and $r$ is the number of possible responses from the environment. This is equivalent to each automaton operating in a stationary environment. Over many cycles of operation the automata converge to the desired actions. The expected output of the environment over one period $M(n)$ is defined and
conditions for expedient and optimal behavior are specified.

Narendra and Viswanathan (1972a) considered a two-level periodic system of variable-structure stochastic automata. The first level made a decision on the value of the unknown period, the second level arranged itself to operate in the random environment for one cycle. The average output of the environment on the second cycle gave the relative worth of the decision made by the first level. It was used as the input to the first level to make the next decision on the value of the period. The purpose behind the simulation was to demonstrate the optimal performance in periodic random environments of the two-level system of variable-structure stochastic automaton. The results showed the simulation was successful in the sense that it ended with the automaton at the first level choosing the true period with a probability larger than 0.98. The optimal performance of the two-level system in unknown periodic random environments with a known upper limit on its period was demonstrated though computer simulation.

Tsuji et al. (1973) considered the behavior of a finite automaton in a non-stationary random environment. The probabilistic automaton was introduced as a random environment in order to generalize the stationary random environment. The interaction between the probabilistic automaton and the two-state deterministic automaton was
considered in the case where the probabilistic automaton had two inputs and two states and was completely isolated by the zeroth approximation. The limiting state probability distribution of the automaton was obtained. Moreover, it was shown that, if the probabilistic automaton was completely isolated by the (0,k)th approximation and satisfied some conditions, then the finite automaton behaved expediently against the probabilistic automaton.

Hierarchical Models

A number of optimal reinforcement schemes were developed. The schemes are useful only when the number of actions is relatively small, on the order of five to ten. For a larger number of actions, convergence becomes extremely slow. One reason for the slow convergence is the large number of action probabilities which are updated at every instant. When all the initial probabilities are chosen to be equal, each probability starts from a low value. Hence it takes a long time for the optimal action probability to reach unity.

To overcome the problem of slow convergence for a large number of possible actions, a multilevel learning automaton was introduced. The system consisted of several levels. Each level consisted of several automata. Each automaton at a specific level was associated with an automaton directly above it. The
structure was viewed as a hierarchical tree structure. Figure 2 illustrates a small hierarchical structure.

Penalty Probability Set
\((C_1,C_2,\ldots,C_r)\)

Environment

Action \(\alpha\) Input \((0,1)\)

Stochastic Automata

\[
\begin{array}{c}
0 \quad 0 \\
\alpha_1 \quad \alpha_2 \\
0 \quad 0 \\
\alpha_{11} \quad \alpha_{12} \\
0 \quad 0 \\
\alpha_{21} \quad \alpha_{22} \\
0 \quad 0 \\
0 \quad 0 \\
0 \quad 0 \\
0 \quad 0 \\
A_{11} \quad A_{12} \\
A_{21} \quad A_{22}
\end{array}
\]

Probability Vector
\((P_1,P_2,\ldots,P_r)\)

Figure 2. Model of the Hierarchical Stochastic Learning Automata

The hierarchy consisted of a single automaton at the first level, two automata in the second level, and four automata on the third level. The action of the automaton at level one activated an automaton at the next level and
so on down to the lowest level. Associated with each level was an action, probability of the action, and a penalty vector. The action chosen at each level interacted with that level's probability vector. The actions chosen at the lowest level interacted with the environment. In a general way, the hierarchical model was described with the following notation:

A=first level automaton

N=number of levels

1st level:

\((\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_r)\) = actions of A

\((P_1(t), P_2(t), \ldots, P_r(t))\) = action probabilities of A at the instant \(t\),

\(t = 0, 1, 2, \ldots\).
nth level (n=2,3,...,N)

\[ A_{i_1\ldots i_{n-1}} = \text{automaton connected} \]

\[ \text{to action } \alpha_{i_1},\alpha_{i_2},\ldots,\alpha_{i_{n-1}} \]

\[ \text{of the } n-1 \text{ level} \]

\[ \alpha_{i_1i_2\ldots i_n} = \text{action of the automaton} \]

\[ A_{i_1i_2\ldots i_{n-1}} \]

\[ p_{i_1i_2\ldots i_n}(t) = \text{action probability of} \]

\[ \alpha_{i_1},\alpha_{i_1i_2},\ldots,\alpha_{i_1i_2\ldots i_n} \]

\[ \text{at time } t. \]

Each action at the last level had a unique path from the first level. The product of the action probability lying on the path was called the path probability. The action probability of the learning automaton corresponded to the probability of the hierarchy. The notion of absolute expediency was extended to the hierarchical model by considering the update scheme of each level of the model.

If each automaton had \( r \) actions, every action probability was set to \( 1/r \). The first level automaton chose, at random, an action from the action probability distribution. This activated the automaton \( A_i \) in the second level. The second level automaton chose an action which activated the corresponding automaton at the next level. At the last level, the final action selected
interacted with the environment to produce a response from the environment.

The reaction from the environment updated the action probability at the associated level and a new set of probabilities were established for each automaton chosen. The cycle was repeated until the action probability in one path, from the top level down to the lowest level, became unified.

Thathachar and Ramakrishnan (1981) developed a learning automata for the hierarchical system. The objective was to speed up the slow convergence of a simple automaton with a large number of actions. They adapted the reinforcement scheme established by Lakshmivarahan and Thathachar (1976). A general set of reinforcement schemes for automata at various levels were obtained by a modification of absolutely expedient algorithms. The general reinforcement scheme for the various levels is denoted by

\[ A_{j_1}A_{j_1j_2}\ldots A_{j_1j_2\ldots j_n} \]

where \( j_1, j_2, \ldots, j_n \) were integers representing the \( n \) levels within the hierarchy.
For First level Automaton:

\[ a_{j_1} = \text{action selected at instant } t \]
\[ A = \text{automaton concerned} \]

then

\[
P_{j_1}(t+1) = \begin{bmatrix} P_{j_1}(t) + \lambda(t)(1-P_{j_1}(t)) & B(t)=0 \\ P_{j_1}(t) & B(t)=1 \end{bmatrix}
\]

\[
P_{i_1}(t+1) = \begin{bmatrix} P_{i_1}(t)(1-\lambda(t)) & B(t)=0 \\ P_{i_1}(t) & B(t)=1 \end{bmatrix}
\]

\[(i_1 \neq j_1)\]

where \( \lambda(t) \) was the abbreviation for \( \lambda(P(t)) \). \( P(t) \) was the vector of all action probabilities of the first level and \( \lambda(P) \) satisfies the inequalities \( 0 < \lambda(P) < 1 \) and \( \lambda(P(t)) > 0 \), for all \( P(t) \).

For nth level Automaton:

\[ a_{j_n} = \text{action selected at } t \]
\[ A_{j_{n-1}} = \text{automaton concerned} \]

then

\[
P_{j_n}(t+1) = \begin{bmatrix} P_{j_n}(t) + \lambda_{j_{n-1}}(t)(1-P_{j_n}(t)) & B(t)=0 \\ P_{j_n}(t) & B(t)=1 \end{bmatrix}
\]

\[
P_{j_{n-1}}^{i_n}(t+1) = \begin{bmatrix} P_{j_{n-1}}^{i_n}(t)(1-\lambda_{j_{n-1}}(t)) & B(t)=0 \\ P_{j_{n-1}}^{i_n}(t) & B(t)=1 \end{bmatrix}
\]

for all other actions,

\[
P_{i_n}(t+1) = P_{i_n}(t) \quad B(t)=0,1
\]

\[(i_n \neq j_n)\]
Each automaton in the hierarchy was associated with a $\lambda$ function.

In any one cycle of operations, one automaton from each level and one action from each automaton was involved. If the outcome was a non-penalty, the probability values of the selected actions were increased. The other action probabilities of the selected automata were decreased to maintain the sum of the action probabilities at unity. The action probabilities of the automaton not involved in the cycle were unchanged.

Along with the update of the action probabilities, the organization of the automata was important in the development of the hierarchical model. Thathachar and Ramakrishnan considered the minimum number of total updates per cycle when determining the optimal number of actions associated with each automata.

Thathachar and Ramakrishnan (1981) showed the total number of actions $K$ was increased by $K^*$. Some dummy actions were included at each level such that $K^*$ was expressed as a product of two's and three's. By expressing the number of actions at each level in products of two or three, the number of updates per cycle was reduced to a minimum. The simulation results showed a considerable savings in the convergence time. By adapting the reinforcement schemes developed early, the model took on expedient characteristics.
Mitchell (1984) modified the hierarchical learning system by reducing some of the inefficiencies. He introduced a hierarchical learning automata which used unbalanced and dynamic structures rather than the balanced and static structures described by Thathachar and Ramakrishnan. The hierarchical stochastic automata structure initialization technique suggested by Thathachar and Ramakrishnan (1981) resulted in a uniform state-path probability vector only if the hierarchy was balanced. Mitchell introduced a method to eliminate the restriction of a balanced model. A hierarchical learning automata with an arbitrary structure was introduced. He showed the existence of an optimal hierarchical stochastic automata structure with order $R \geq 2$ and all subautomata having 2 or 3 states (optimal 2-3 hierarchical stochastic automata structure). In addition, there existed an optimal 2-3 hierarchical stochastic automata structure with order $R \geq 3$ and all root automata with 3 states. The reorganization of the learning automata occurred periodically. The state-path probability vector changed so that the expected total number of updates required for the hierarchical model was reduced.

These results provided the basis for a recursive algorithm which created optimal hierarchical stochastic automata structures of arbitrary order. Not only did the algorithm produce hierarchical stochastic automata structures with arbitrary order, it also reduced the expected total number of updates required for learning.
structures which minimized updating, but it also eliminated the current need for dummy actions for certain orders. Using these results, a cost formula for optimal hierarchical stochastic automata structure was derived.

Application

Most of the research conducted in the area of Stochastic Learning automata was theoretical. Theoretical papers were written describing the different types of learning models such as P-models, Q-models, and S-models. The general class of reinforcement schemes were described and the necessary and sufficient conditions for convergence were proven. Linear and non-linear reinforcement schemes were established for Penalty-Reward, Penalty-Inaction, and Inaction-Reward. Learning models were introduced with non-stationary random environments. Throughout all the theoretical work, there were only a few papers that used learning models in a practical application. Learning automata were first applied to telephone routing and control systems. Yet, with the increase sophistication of the technology, it became necessary to examine new methods for improving a control system.

Narendra, Wright and Mason (1977) discussed the use of learning automaton in telephone traffic routing. They compared the performance of a learning automata routing scheme to a fixed rule alternate routing scheme. The
simulation studied a simple network with realistic loading and constraints. The model consisted of a set of terminals generating arriving calls and a connecting network providing the physical paths or trunks over which communication took place. The arriving calls generated at the terminals were modeled by a Poisson process. The results showed for less than engineered loads, the performances of the two schemes were identical. At constant engineered loads and under general overload conditions, the performances of the two schemes were comparable. The most effective use of learning automata routing occurred when selected overload conditions prevailed but additional capacity was available elsewhere in the network.

Narendra and Thathachar (1980) applied the learning model in the traffic routing system. They used two learning models in a non-stationary environment. The corresponding penalty probability $C_i$ represented the load. As the load changed, the probability vector $C_i$ changed. For the first model, when the load lessened, $C_i$ increased. The reinforcement scheme used was a Linear Reward-Inaction ($L_{r-i}$) scheme. In the second model, the penalty probabilities $C_i$ were monotonically increasing functions. The second model used a Linear Reward-Penalty ($L_{r-p}$) reinforcement scheme.

The network used in the simulation contained four nodes connected by four trunk groups. The intervals
between calls were modeled by the Poisson distribution. When a call arrived at node 1 a choice of trunk group 1 or trunk group 2 was made, depending on some rule. If the trunk group was busy, the call attempted another trunk group. If one of the trunks in the trunk group 1 was free, the call reached node 2. Then the call made a bid for a trunk in trunk group 3, if the group was available, the call reaches node 3. A complete call was held for a period of time. The time was a random variable with an exponential distribution.

Three routing rules were used in the simulation study. The first one was a fixed rule in which trunk group 1 was always tried first. Trunk group 2 was attempted if all the trunks in group 1 were busy. The second rule used the estimates of the blocking probabilities along each route. The blocking probabilities were updated after each call. The third routing rule used the $L_{r-p}$ scheme.

The results of the learning automaton showed the schemes were optimal if the actions of the automaton were chosen successively. The simulation demonstrated that the learning automaton models were suitable for modeling traffic routing in telephone networks.

Glorioso and Osorio (1980) suggested the use of learning models in resource allocation in a distributive computer system. They used a $L_{r-p}$ model as the task scheduler in a multiple processor system. The learning
automaton was different from the earlier queueing disciplines because it functioned in the presence of a changing environment and without complete information about the state of the system.

When a task entered the system, the Network Controller (NC) for the processors selected, based on $P(t)$ (the penalty probability value at time $t$), the processor to run the task. After selecting the processor, the NC updated $P(n)$ according to the update scheme and depending on either the success or failure of scheduling the task. If the selected processor was busy, the task was put in a queue and it waited for servicing.

The deterministic scheduler, DFS, consists of a queueing system in which the arriving task had complete knowledge of the processors' relative speeds. The NC allocated the tasks to the processors in a fixed order. For example, if the speed of the processors were $\mu_1 > \mu_2 > \mu_3$, the NC assigned the arriving tasks to the first free processor, that is, the processor with the minimum expected delay.

Glorioso and Osorio (1980) discussed the results of the simulation of the two disciplines in terms of mean turn-around time and system recovery in the case of a processor failure. The results showed the turn-around time of the $L_{r-p}$ discipline was greater than that of the DFS. The results were consistent with the authors' expectations, since the fixed rule had complete
information on the system parameters, and the $L_{r-p}$ automaton was allowed to gain the information on the basis on the processor being either busy or free. In addition, the results showed both the $L_{r-p}$ model and the DFS arrived at their corresponding steady states, and the mean turn-around time was reduced as the speed of each of the processor increased. The results showed when a processor failed, the $L_{r-p}$ model was able to adapt to the changes in the environment, whereas the DFS discipline was severely affected.

Kumar (1981) used learning automata in the design of a large communication network. The communication network was defined as a set of nodes connected by links. Messages were routed from their source over the links to their ultimate destination nodes. The routing problem consisted of determining the choice of nodes so that a network performance index was optimal.

The network system was modeled by nodes numbered 1, 2, ..., N. The link $(i, k)$ was the link from node $i$ to the node $k$, with the capacity $C_{ik}$. When a message arrived at node $i$ for destination $j$, it was routed by the controller $A_{ij}$ located at node $i$. If the chosen link was free, the message was immediately transmitted to the next node, else the message waited in a link buffer. Every message at each link had a waiting delay and transmission delay. The sum of these two delays was the link delay. The learning model was applied to the routing controller.
The results showed that the queue stability of the system was dependent on several of the parameters used in the reinforcement scheme. The value of the step-size was related to the stability of the system. System stability was also related to the issue of statistical equilibrium. The reinforcement scheme required a non-linear updating scheme and an on-line estimation of the gradients. The results showed that the non-linear algorithms provided a faster convergence time. Finally, the results pointed out the need for further research into understanding the relationship between a priori knowledge, algorithm design, and adaptation.

Nedzelitsky and Narendra (1982) applied the learning model to the routing problem in data communication networks. They showed a Markovian model of the network together with its automaton converged for the overall system. The communication network was defined by a set of nodes $N$ and a set of links $L$. The links provided communication between the nodes, and a link capacity corresponded to each link. When a message entered a node, it was placed in a queue for one of the outgoing links. The operation of a learning automaton in the environment consists of the selection of one automaton action from a finite set of such actions. The environment responded with a signal that rewarded or penalized the action of the automaton. Based on the input, the automaton modified its strategy for selecting
an action, thus improving the expectation of receiving a favorable response. The learning automaton reacted to queue buildup and correctly adjusted the routing strategy when changes in the source rates or network configuration occurred. The schemes considered in the network were \( L_{r-p} \), \( L_{i-p} \), and \( L_{r-i} \). The network's queue delay was used to modify the environment. The standard routing strategies used were either message switching, where a message was routed through the network on one link at a time and the routing decision was made at a node when the message arrived at the node, or packet switching, where the message was subdivided and transmitted in packets of fixed equal length but the operation of the network was otherwise the same as with message switching.

The different models were tested by simulation. The results showed that learning automaton model provided a significant improvement in performance of the system. The effects of different limitations in the network on relative performance required further investigation.

Meybodi (1983) applied the learning model to the priority assignment in a queueing system with unknown characteristics. Meybodi developed a system with \( n \) distinct priority classes. A queue was established for each class and jobs arrived into the system based on \( n \) independent Poisson processes. A Decision Device (DD) was used to choose the next job from the \( n \)th queue to be serviced. For example, if there were 2 priority classes,
then there were 2 queues, and the DD would choose the job from either queue 1 or queue 2. The DD would use the probability vector \( P(t) = (P_1(t), P_2(t)) \) to make the choice. The reinforcement scheme would update \( P(t) \) depending on the average service time, \( T(t) \). \( P_n(t) \) was incremented if the job selected had a service time less than \( T(t) \). \( P_n(t) \) was decremented if the job selected had a service time greater than \( T(t) \). \( T(t) \) was updated when the job currently in the server was completed.

The results showed the system asymptotically converged to the classical priority assignment with a probability as close to unity as desired. That is, the proposed system asymptotically assigned the highest priority to the class with the shortest service time. The learning scheme was modified to minimize total average waiting cost of the system. The learning algorithm for assigning priority was also applied to the three class \( M/M/1 \) priority queueing system.
Most of the applications using Stochastic Learning automata involve some form of Queueing Theory. Queueing Theory describes a system in which a customer arrives for a service, waits in line to be served, is served, and exits the system. It can be applied to a situation where there is a reservoir, an input, an output, and the assumption that what enters the queue must exit the queue. For example, Queueing Theory is applied to production processing and inventory control. Rules are developed which describe the method of distributing the arriving inputs (customers) to the processors (servers). Queueing disciplines are the rules which assign the arriving customers to the servers. The objective of the queueing disciplines are to minimize the wait time of the customer.

A queueing system can be mathematically modeled. Generally, the arrival of the customers occurs on a random basis. After a large number of repetitions, the observed arrival time is approximately reproducible. A probability function can give the probability of K customers arriving before time t. When an assumption is made that customers arrive independently, the arrival of
customers follows a binomial distribution, since each customer either arrives or does not arrive before time \( t \). If the arrival times of the customers are ordered, the multi-nominal distribution is applied. As the number of potential customers increases and the number of customers arriving before time \( t \) decreases, the distribution resembles a Poisson distribution. Thus, the arrival of customer or input is based on the Poisson distribution.

The interval between the arrival and the departure of a customer is referred to as the inter-arrival time. The exponential probability distribution is used to approximate the inter-arrival time of the customer.

The present thesis examines the problem of scheduling tasks to a processor in such a way that the performance measure is optimal. The thesis looks at the performance of two queueing disciplines: the Teller Window and the Teller Window with Simultaneous Jockeying.

**Proposed Model**

The thesis explores the application of a Stochastic Learning automata in a single-server queueing system. The model is a \( m \)-queue, \( m \)-processor system (Figure 3).

The processors are homogeneous, each with the average processing rate determined by the exponential distribution, \( \mu_1, \mu_2, \ldots, \mu_m \), where \( \mu_1=\mu_2=\ldots=\mu_m \). Associated with each processor is a first-come-first-serve queue. Tasks arrive at the
Network Controller (NC) according to an independent Poisson process with a constant rate of \( \lambda_i (> 0) \) \( i=1,2,...,k \). The Network Controller (NC) determines which processor is scheduled for the arriving task, and places the task in the queue for that processor. The scheduling of the task to the chosen processor occurs instantaneously.

The simulation uses next-event time advancement. The simulation clock is initialized to zero and the time of the occurrence of a future event is determined. The simulation clock is advanced to the time of the occurrence of the most imminent event. At that point, the state of the system is updated to account for the fact that an event has occurred. The simulation clock is again advanced to the time of most imminent event, the
state of the system is updated, and the future event time is determined.

Two performance measures are used to compare and evaluate different schedulers, the mean delay in the queue and the mean number of tasks in the queues.

Five queueing disciplines are used and discussed: a Teller Window fixed discipline; a Teller Window with Instantaneous Jockeying; and the $L_{r-p}$, $L_{r-i}$, and $L_{i-p}$ learning models.

The $L_{r-p}$ scheduling discipline is described in terms of the Reward-Penalty learning automaton operating in a P-Model environment. The automaton $<X,\psi,\alpha,P,T,G>$ is described as:

$X$ - input set from the environment, considered to be binary $[0,1]$. 0 is associated with non-penalty, when the processor is free. 1 is associated with penalty, when the processor is busy.

$\psi$ - a set of internal states.

$\alpha$ - a set of outputs that corresponds to the selection of a processor for execution of arriving tasks.

$P$ - probability function which determines the assignment of incoming tasks to the processors, where $P_i(t)$ is the probability that processor $i$ is selected to execute the arriving task at time $t$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
T - the Linear Reward-Penalty described later.

G - the mapping of the internal states to the actions.

T is described as the reinforcement scheme which updates the probability value of $P(n)$. The scheme is represented by

$$
P_{i}(t+1) = \begin{cases} 
P_i(t)+\sum aP_j(t) & \text{if } B=0 \\
P_i(t)-\sum b/r-1-bP_j(t) & \text{if } B=1 
\end{cases}$$

$$
P_{j}(t+1) = \begin{cases} 
P_j(t)-aP_j(t) & \text{if } B=0 \\
P_j(t)+b/r-1-bP_j(t) & \text{if } B=1 
\end{cases}$$

where $j \neq i$ and $0 < a, b < 1$ for some arbitrary value chosen for $a$ and $b$.

The reinforcement scheme used in the present application is similar to the reinforcement scheme discussed by Lakshmivarahan and Thathachar (1976). The reinforcement schemes followed the conditions to assure the model is optimal. Thus, the reinforcement scheme used is optimal and the details concerning the optimality are not discussed. The scheme maintained unity by adjusting the probabilities to keep the total probability equal to one.
In the simulation, a Linear Reward-Inaction \((L_{r-i})\) reinforcement scheme is implemented by setting \(b=0\).

\[
P_i(t+1) = \begin{cases} 
P_i(t) + \sum aP_j(t) & \text{if } B=0 \\ P_i(t) & \text{if } B=1 \end{cases}
\]

\[
P_j(t+1) = \begin{cases} 
P_j(t) - aP_j(t) & \text{if } B=0 \\ P_j(t) & \text{if } B=1 \end{cases}
\]

A Linear Inaction-Penalty \((L_{i-p})\) reinforcement scheme is implemented by setting \(a=0\).

\[
P_i(t+1) = \begin{cases} 
P_i(t) & \text{if } B=0 \\ P_i(t) - \sum b/r-1-bP_j(t) & \text{if } B=1 \end{cases}
\]

\[
P_j(t+1) = \begin{cases} 
P_j(t) & \text{if } B=0 \\ P_j(t) + b/r-1-bP_j(t) & \text{if } B=1 \end{cases}
\]

For the cases of \(L_{i-p}\) and \(L_{r-i}\), the reinforcement schemes used are similar to the schemes developed by Lakshmivarahan and Thathachar (1976) which are shown to be e-optimal.

To complete the description of the learning automaton as a task scheduler, a variable definition of the penalty \((B=1)\) and non-penalty \((B=0)\) response from the environment is needed. When the task is assigned to the processor and the processor is free, then \(B=0\). When the
task is assigned to the processor and the processor is busy, then \( B = 1 \).

The simulation incorporates a blocking factor. The blocking factor represents the probability of the processor or server being free. The value of the blocking factor at time \( t \), along with \( P(t) \), is used by the controller to determine the queue assignment of the incoming task. The blocking factor is defined by \( i \) random variables corresponding to the number of tasks waiting in queue \( i \), \( i = 1 \ldots n \), at time \( t \). For a three processor system, let \( x(t) \), \( y(t) \), and \( z(t) \) be random variables corresponding to the number of tasks waiting in the queues for the three processors. The blocking factor is defined by

\[
Q_{1mn} = P[x(t) = 1, y(t) = m, z(t) = n]
\]

The probability that processor \( i \) is free is then equal to the sum of the probabilities that the other 2 processors are busy when processor \( i \) is free. This is expressed by

\[
Pr[\text{proc}_i \text{ is free at time } t] = \sum_{m} \sum_{n} Pr[x(t) = 0, y(t) = m, z(t) = n]
\]

The probability that the processor is busy is equal to one minus the probability that the processor is free.

The scheduling discipline for the \( L_{r-p} \) automaton is described by the following process. When a task enters the system, the network controller selects a processor to run the task based on the probability distribution
function, \( P(t) \). After making the selection, the network controller updates \( P(t) \) in accordance with the scheme and based on the success or failure of the scheduling the arriving task. If the selected processor is busy; the task joins the queue and waits for service.

The traditional queueing disciplines are Teller Window and Teller Window with Instantaneous Jockeying. Teller Window assigns the tasks to a queue based on the length of the queue at the time the task enters the system. The network controller (NC) performs the actual assignments. In the situation that all the processors are busy, the controller assigns the task to the processor with the shortest queue. If all queues are of equal length, the controller randomly assigns the task to a processor. If one or more processors are free, the controller randomly assigns the task to one of the free processors.

The Teller’s Window with Instantaneous Jockeying consists of a job arriving in the system to find the same conditions as the Teller’s window with one additional rule: if at any time, the difference in the length of the queues exceeds 1, the last job in the longer line is transferred to the end of the shorter line instantaneously.

At the beginning of each simulation, the model is initialized. The arrival time of the first task is determined and the task is assigned to one of the m
queues according to one of five queueing disciplines. When a task arrives, it joins the end of the queue if the processor is busy. If the processor is free, the task enters the processor. At the point the task enters the processor, the departure time is determined for the task. The clock is advanced to the next event, either the next arrival time or the next departure time. When the task departs a processor, the processor checks to determine if the queue is empty. If the queue is not empty, then the next task is processed. If the queue is empty, then the processor is freed and waits for the next task. With each processor, there is a queue with infinite length. Incoming tasks are assigned to the processor by the queueing discipline. The clock is advanced to the next event, either a departure or an arrival.

Program Specification

The queueing system is a M/M/m simulation. It is written in PASCAL and is implemented on a DEC-System 10 computer maintained by the Academic Computer Center at Western Michigan University. The simulation of the M/M/m queuing is similar to the M/M/1 described by Law and Kelton (1982).

The program specifications are divided into 3 parts. The first part describes the data structures used. The second part is a general description of the operations of the queueing simulation. The third section specifies the
operations of the procedures used by the program. A complete program listing is found in the Appendix.

Data Structures

The general data structures used in the program are as follows:

PROCESSORS: The processors are represented by an array TNE dimensioned by the number of processors, NUM_PROC. TNE contains the processing times for the current tasks. An array STATUS is used to determine if the processors are free (0) or busy (1). It is dimensioned according to NUM_PROC. The mean service time is an exponential probability function. The exit times of the tasks in the processors are determined by M_SEV_T and is stored in TNE.

QUEUE: A two dimensional array is used to represent the queues. The maximum length of the queue is set by QUE_SIZE and there are NUM_PROC queues. T_ARRVL stores the arrival time of each task. NIQ is a counter indicating the number of tasks in the queues.

JOB STREAM: A single variable ARRVL represents the job stream entering the system. The mean arrival time is a Poisson probability function with a mean of $\lambda_1 + \lambda_m$ represented by M_ARR_T.

CLOCK: The clock is a single variable representing a next event time advancement. NEXT determines the next event.
DISCIPLINES: Procedures LRP, FQD, and JOCKEYE represent the queue disciplines. DISPL determines which discipline is used. The procedures assign the incoming tasks to the processor queues.

ENVIRONMENT: C_VECT is a one dimensional array representing the probabilities that the processors are busy. ENV is a one dimensional array of records which is used in determining the probability the processor is busy.

REINFORCEMENT: PROB_VECT is a one dimensional array which contains the probabilities of selecting each processor. It is used by the learning models.

STATISTICS: ANIQ determines the average number of tasks in the queues. TOT_DEL accumulates the total delays of tasks in the queues.

Program Description

A general algorithm describes the program

SET SIMULATION CLOCK=0
INITIALIZE SYSTEM STATE AND STATISTICAL COUNTERS
INITIALIZE EVENT LIST
REPEAT
CALL THE TIMING ROUTINE
DETERMINE THE NEXT EVENT TYPE I
ADVANCE THE SIMULATION CLOCK
CALL EVENT ROUTINE I
EVENT MAY BE ARRIVAL OR DEPARTURE
UPDATE SYSTEM STATE
UPDATE STATISTICAL COUNTERS
GENERATE FUTURE EVENTS AND ADD TO THE EVENT LIST
UNTIL SIMULATION IS OVER
COMPUTE ESTIMATES OF INTEREST
PRINT REPORT
The mainline inputs the values for the mean service time, mean arrival time, and the number of tasks to be processed. It sets up the necessary assignments for each of the disciplines and calls SIMULATION.

SIMULATION initializes the clock, system state, statistical counters and the event list in INITIALIZE. The timing routine TIME_INC is called to determine the event type NEXT of the next event to occur and to advance the simulation CLOCK to the time of the occurrence of the next event. An IF-statement, based on NEXT, is used to pass the control to the appropriate event routine. If NEXT=1, then event routine ARRIVE is called to process the arrival of the task. If NEXT>1, then event routine DEPART is called passing the processor number of the departing task, and processes the departure of the task. After the control returns to the SIMULATION, a check is made to determine whether the number of task NUM_CUST is equal to TOT_CUST. If NUM_CUST < TOT_CUST, then TIME_INC is called to continue the simulation. If the specific number of tasks is observed, then the report, REPORT, is generated. It prints the estimates of the desired measures of performance. In addition, REPORT is called after the completion of every 50 tasks. The mainline then sets up the next simulation and continues.

ARIVAL begins by selecting the next arrival event. ARRAL is the time of the arrival of the task that arrived. DISCIPLINE is called, passing the value J. J
returns the number of the processor to which the job is to be assigned. ARIVAL determines whether the processor is idle (STATUS=0) or busy (STATUS=1). If the processor is busy, then ANIQ and T_LT_EVT are updated, NIQ is incremented and the task is assigned to the queue. If the processor is idle, then there was not a DELAY, TOT_DEL is updated, and the task is assigned to the processor. The amount of time it takes to process the task is determined.

DEPART begins by determining if there are other tasks waiting in the queue. If the NIQ is greater than zero, then ANIQ is updated, T_LT_EVT is updated, the delay for the current task is determined, and TOT_DEL is updated. The task exits, updates the TOT_CUST, brings in the next task to be processed, determines the processing time TNE and moves all the other tasks in the queue up. At this point, if the JOCKEYE is in use, then JOCKEYE is called. If there are no other tasks waiting to be processed, the task exits, STATUS is set to 0 and TNE is initialized to 10000000.

DISCIPLINE is the last major procedure. DISCIPLINE basically transfers the control of the program to one of the 3 disciplines. DISPL is set in the mainline prior to calling SIMULATE. If DISPL equals 1, then the learning model is used. If DISPL equals 2, then the learning model with the blocking factor is used. If DISPL equals 3, then the Teller Window is used. If DISPL equals 4
then Teller Window with Jockeying is used. The learning model, LRP, and learning model with blocking, NSLRP, use the value TYP to determine the type of learning model used. If TYP is equal to 1, then LRP is used; if TYP is equal to 2, then LRI is used; and if TYP is equal to 3, then LIP is used.

A further description of the procedures is provided in the next section.

Procedure Description

The general algorithms of the procedure are:

FUNCTION POISSON
initialize A=1/exp(mean arrival time)
while the random variable < A
    generate steps
    set function=number of steps needed

FUNCTION EXPO
generate random number
set function=-(mean service time)*LN(random number)

PROCEDURE LRP
Randomly choose one of the actions, based on the current probability values.
if the chosen queue is empty
    the environment produces 0
else a 1 is emitted.
the environment is updated.
if the processor is busy then
    update the probability and determine GF
else
    update the probability and determine GJ
PROCEDURE NSLRP
Randomly chooses one of the actions, based on the current probability value and a blocking factor.
if the chosen queue is empty, the environment produces 0
else a 1 is emitted.
the environment is updated.
if the processor is busy then
update the probability and determine GF
else
update the probability and determine GJ

PROCEDURE FQD
initialize internal variables
check for empty processor
if all the processors are busy then
look for queue with shortest time
store the queue number with equal or shortest line
else
randomly choose processor with equal queue lines

PROCEDURE JOCKEYED
initialize variables
check if quei exceeds quej by more than one
if true then
move job from quei to quej

PROCEDURE REPORT
prints heading and input parameters
compute and print estimates of the desired measures of performance
CHAPTER IV

RESULTS

The analysis of the results of the simulation is conducted using MINITAB version 81.1. Two performance measures are used in the analysis of the simulation: the average delay in the queue and the average number of tasks in the queue. The results use a one way analysis of variance with $\alpha = .05$. The analysis of variance tests for significant difference in the means of each parameter tested. The plots illustrate the delay in the queue versus the number of tasks. The number of tasks is measured in blocks of 50 tasks.

The simulation varies several different parameters of the learning. The results describe the effects the different parameters had on the performance of the models. The simulation looks at the performance of the three learning models by varying the step size of the reward and/or penalty by .1, .3, and .7. An analysis is conducted to determine if there is a significant difference in the performance of the learning models when the step sizes varied.

The simulation uses two task sizes, 1000 tasks and 5000 tasks, with .5/.75 mean arrival/service times. The two task sizes are used to test the performance of the
disciplines over a longer period of time. An analysis is conducted to determine if there is a significant difference in the overall performance of the discipline when the task sizes vary.

The simulation uses 4 different values for the mean arrival time and the mean service time. The times are .5/1.0, .5/1.5 and 10/15 seconds. An analysis is conducted to determine if there is a significant difference in the overall performance of the disciplines when the work load is varied.

The simulation tests the performance of the 5 disciplines in a 3 queue system and a 5 queue system with a .5/.75 mean arrival/service time with 1000 tasks. An analysis is conducted to determine if there is a significant difference in the overall performance of the disciplines when the number of queues in the system is varied.

Finally, an adaptation of the learning model is implemented. The blocking factor uses the probabilities that the queues are busy in making queueing decisions. The blocking factor is implemented for the $L_r-p$ model with a .5/.75 mean arrival/service time and 1000 customers. The analysis determines if there is a significant difference between the learning model that is given the information about the queue size and the learning model which gains with information from the actual run.
In the initial analysis, the two performance measurements across each of the models do not vary significantly. The results obtained for the average number of tasks in the queues and the average delay are consistent across all the models. Thus, to simplify the results, the results are discussed in terms of the average delay in the queues.

The results of comparing the three learning models are shown in Figure 4. The results show that the delay in the queue for $L_{r-p}$ is less than the delay in the queue for $L_{r-i}$ which is less than the delay in the queue for $L_{r-i}$. The mean differences between the three models are .1129, .1775, and .2273 respectively. The mean differences are significant. It is interesting to note that the $L_{r-p}$ and $L_{i-p}$ schemes improve their performance as the number of tasks increases, where the $L_{r-i}$ performance degenerates. The initial comparison is conducted with a step size of .1. Generally, the learning models need at least 500 tasks for the "learning" to occur. That is, by 500 tasks, the $L_{r-p}$ and $L_{i-p}$ models reach some type of stability, converging to the desired state.

The results obtained by increasing the step size from .1 to .3 are shown in Figure 5. The results show an increase in the overall delay from the means of .1129, .1775, and .2273 to the means of .1186, .219, and .395. The increase in the performance of the $L_{r-i}$ and $L_{r-p}$ is
significant, but the increase in the $L_{i-p}$ is not. The performance reaches stability by 350 tasks in all three models. There is less variation in the delay times once the learning automaton reached stability for the step size of .3 than for the step size of .1.

When the step size is increased to .7 (Figure 6), there is an improvement in the performance of the models over the performance at .3 but not over the performance at .1. The mean delays are .181 for $L_{r-p}$, .395 for $L_{r-i}$, and...
and .0721 for \( L_{i-p} \). There is not a difference in the \( L_{r-i} \) performance. There is a significant improvement in performance from .1 and .3 for the \( L_{r-p} \) and the \( L_{i-p} \). The performance level still reaches stability by 350 tasks.

In the comparison of the two fixed disciplines (Figure 7), it is interesting to note that the Teller has an overall performance mean of .0369 as opposed to the delay of .0435 of the Jockey. The mean difference is slightly significant. The performance of the both models
increases until task 500 and then reaches a stable performance level after 500 tasks.

In the comparison of the learning models to the fixed disciplines (Figures 8, 9, 10), the performance of the fixed disciplines are significantly better than the learning models. In each case, the Jockey and the Teller are significantly better than the learning models. This is constant with the findings of Glorioso and Osorio (1980).

Glorioso and Osorio (1980) showed that the performance of the fixed Teller and Jockey schemes were
better since they have all knowledge of the queue sizes, that is, the relative rates of the processors. The same is true of this simulation; the fixed disciplines have prior knowledge of the queue sizes, whereas the learning models acquire the knowledge as they operate.

The comparison of the number of tasks (Table 1) shows a significant improvement in the $L_{i-p}$ model and a significant degeneration in the $L_{i-r}$ model as the number of tasks is increased. This is expected since the $L_{i-p}$
performance improves with time and $L_{r-p}$ performance degenerates with time. There is a significant difference in the performance of the fixed model; the performance degenerates when given more tasks. There is not a major change in the relative performance of the five models; the Teller has the best performance with the mean delay of .0512 and the $L_{i-r}$ has the worst performance with .3128. It would be interesting to examine the performance of the models for several other task sizes to determine if the order of performance would change. Even though there is a significant improvement in the $L_{i-p}$ and

---

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 9. Comparison of $L_{r-i}$ to Jockey and Teller

A significant degeneration in the Teller, the difference between their means is not significant.

The results of varying the mean arrival time and the mean processing time are given in the Table 2. The times used were .5/.75, .5/1.5, .5/1.0, and 10/15 seconds respectively.

In the comparison of the task loads, there is one significant change. It occurs at 10/15 sec arrival/service between the $L_{r-p}$ and the $L_{i-p}$. Up to this point, the $L_{i-p}$ performance is better than the $L_{r-p}$ model, but as the processing time and the arrival time
Figure 10. Comparison of $L_{i-p}$ to Jockey and Teller

Table 1

Mean Delay for 5000 Tasks Versus 1000 Tasks

<table>
<thead>
<tr>
<th>DISCIPLINE</th>
<th>1000 TASKS</th>
<th>5000 TASKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_{r-p}$</td>
<td>0.1775</td>
<td>0.1732</td>
</tr>
<tr>
<td>$L_{r-i}$</td>
<td>0.2273</td>
<td>0.3128</td>
</tr>
<tr>
<td>$L_{i-p}$</td>
<td>0.1129</td>
<td>0.0870</td>
</tr>
<tr>
<td>TELLER</td>
<td>0.0319</td>
<td>0.0449</td>
</tr>
<tr>
<td>JOCKEY</td>
<td>0.0435</td>
<td>0.0512</td>
</tr>
</tbody>
</table>
Table 2
Mean Delay for .5/.75, .5/1.0, .5/1.5 and 10/15 Second Arrival/Departure Times

<table>
<thead>
<tr>
<th>DISCIPLINE</th>
<th>.5/.75</th>
<th>.5/1.0</th>
<th>.5/1.5</th>
<th>10/15</th>
</tr>
</thead>
<tbody>
<tr>
<td>L&lt;sub&gt;r-p&lt;/sub&gt;</td>
<td>.1775</td>
<td>.345</td>
<td>1.19</td>
<td>11.5</td>
</tr>
<tr>
<td>L&lt;sub&gt;r-i&lt;/sub&gt;</td>
<td>.2273</td>
<td>.667</td>
<td>1.205</td>
<td>35.5</td>
</tr>
<tr>
<td>L&lt;sub&gt;i-p&lt;/sub&gt;</td>
<td>.1129</td>
<td>.2384</td>
<td>.7637</td>
<td>17.06</td>
</tr>
<tr>
<td>TELLER</td>
<td>.0369</td>
<td>.0987</td>
<td>.2669</td>
<td>3.515</td>
</tr>
<tr>
<td>JOCKEY</td>
<td>.0435</td>
<td>.0919</td>
<td>.2555</td>
<td>3.930</td>
</tr>
</tbody>
</table>

increases, the performance of the L<sub>r-p</sub> is significantly better than the L<sub>i-p</sub>. The remainder of the results are consistent with the previous results except for the .5/1.0 sec for the Teller and the Jockey. The performance is not significantly different for the .5/1.0 sec. The mean difference between the two models decreases as the task load increases. However, there is not a significant difference between the Jockey and the Teller disciplines as the task load increases. The fixed models still outperform the learning models significantly and the performance measured increases as the task load increases.

The mean delay for 3 queues and 5 queues are given in Table 3.
There is a marked increase in the performance of the learning models versus the fixed disciplines in the five queue simulation. The fixed disciplines are significantly better than the learning models. The difference appears when looking at the individual learning models (Figure 11) and the individual fixed disciplines (Figure 12). In the learning models, the performance of the $L_{r-p}$ increases significantly from the 3 queue system to the 5 queue system, where as the $L_{r-i}$ and the $L_{i-p}$ remain about the same. The results indicate that $L_{r-i}$ has a better performance in the 5 queue system than the $L_{r-p}$ model. This is different than the 3 queue system.

In the fixed schemes, the Jockey discipline is significantly improved in the performance from the 3
Figure 11. Comparison of the Learning models for 5 Queues

queue system to the 5 queue system. The Teller also improves its performance significantly, yet not to the degree of the Jockey. As a result, the overall performance of the Jockey is significant better than the Teller.

The blocking factor is an attempt to provide the learning model with some knowledge of the queue sizes. The knowledge is acquired over time as the simulation runs. Below are the graphs comparing the blocking factor to the $L_{r-p}$ (Figure 13), Jockey and Teller disciplines (Figure 14) with $a=b=.1$, $.5/.75$ sec arrival/service times, and 1000 tasks. The blocking factor is tested

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 12. Comparison of Fixed Disciplines for 5 Queues

with three other variables, with 5000 tasks, 10/15 sec
arrival/service times, and for the 5 queue case. Table 4
shows the mean values for these tests.

In the 1000 task environment with .5/.75 and a=b=.1,
the performance of the \( L_{r-p} \) model is significantly better
than that of the blocking model. However, as the task
size increases to 5000, the performance of the blocking
model improves over the \( L_{r-p} \) model significantly. Yet,
the performance of the blocking model is not better than
the performance of the fixed disciplines. When the
arrival/service time is increased to increase the load,
the Blocking model is significant better than the \( L_{r-p} \)
model. The results are shown for the increase in the number of queues from 3 to 5. The Blocking model improves the performance of the learning model significantly. However, as in all cases, the performance of the fixed models is significantly better than the learning model with blocking. Since there is an improvement in the performance of the learning model with the blocking factor, the results show that increasing the amount of information about the queue sizes is important.
to the performance of the learning model. The improved performance of the blocking factor from the 1000 task environment and the 5000 task environment may be the result of acquiring more information since the blocking factor depends on the number of tasks in a queue.
<table>
<thead>
<tr>
<th>DISCIPLINE</th>
<th>1000 TASKS 3 QUEUES</th>
<th>5000 TASKS 3 QUEUES</th>
<th>1000 TASKS 3 QUEUES</th>
<th>1000 TASKS 5 QUEUES</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_{r-p}$</td>
<td>.1775</td>
<td>.1732</td>
<td>11.5</td>
<td>.2530</td>
</tr>
<tr>
<td>BLOCKING</td>
<td>.1861</td>
<td>.12522</td>
<td>6.583</td>
<td>.1736</td>
</tr>
<tr>
<td>TELLER</td>
<td>.0369</td>
<td>.0449</td>
<td>3.515</td>
<td>.0326</td>
</tr>
<tr>
<td>JOCKEY</td>
<td>.0435</td>
<td>.0512</td>
<td>3.930</td>
<td>.0308</td>
</tr>
</tbody>
</table>
CHAPTER V

DISCUSSION

This chapter summarizes the results of the simulations, makes suggestions for further research in the area of learning automata, and explores additional applications.

The objective of the thesis is to provide a theoretical background into Stochastic Learning automata by looking at its development. Stochastic Learning began in Psychology, then later developed in the area of Operations Control. The convergence of the learning automaton was proven by the use of Markov chains and the application of Martingale theory. The development of the reinforcement scheme was traced back to Tsetlin (1961) and a historical overview of the reinforcement scheme was presented. A brief explanation of Queueing Theory was provided and a discussion of the application of Stochastic Learning automata to Queueing Theory was covered. Finally, a simulation was designed to further look at the usefulness of the learning automaton in Queueing Theory.

The results generally confirm the research previously conducted. In the analysis of the learning models, the performance of the three models is consistent
with Lakshmivarahan and Thathachar (1974). It is shown by Viswanathan and Narendra (1970) that the $L_{r-i}$ scheme totally changes the performance of the model. Generally, the $L_{r-i}$ and $L_{i-p}$ models converge faster to a single response, but a faster convergence may result in a convergence to an incorrect response. It is also shown that the $L_{i-p}$ scheme is inferior to the $L_{r-i}$ and $L_{r-p}$ schemes. This is inconsistent with the finding of the current paper. The evaluation of the $L_{i-p}$ and $L_{r-i}$ may depend on the application. Even though $L_{i-p}$ is proven to be theoretically inferior, in actual application the inaction component may improve the behavior of the learning model.

There is an interesting correlation between the behavior of the learning model with the $L_{i-p}$, $L_{r-i}$, and $L_{r-p}$ reinforcement schemes and the behavior of an organism. It is shown by Ferester and Skinner (1957) that learning behavior of an organism occurs faster and more accurately when either a reinforcement scheme or a punishment schedule is used as opposed to a combination of reinforcement/punishment. This result is consistent with the behavior demonstrated by the learning models.

The second factor of the reinforcement schemes is the convergence rate of learning models given the three different step sizes. The results are consistent with Lakshmivarahan and Thathachar (1973) and Lakshmivarahan and Thathachar (1976). They showed that the speed of
convergence is increased by changing the step size in the reinforcement scheme. Yet, by changing the step size, the probability that the model converges to the undesirable state is increased. Again, the results of the current thesis is consistent with previous results. It is seen that as the step size is increased from .1 to .3, the learning model converges after 350 tasks rather than 500 tasks. Another interesting comparison is made between the behavior of the learning model actual behavior of an organism. It was shown by Ferester and Skinner (1957) in the psychology literature that as the amount of reinforcement increased, the rate of learning increased. But, for large amounts of reinforcement, the probability of the organism learning an incorrect response is increased. These results seem to be consistent between theory and actual behavior.

The final factor to look at is the stability of the learning models as the step size is varied, the arrival/Departure times are varied, and the task sizes are varied. Generally, the performance of the $L_{r-p}$ model is more stable than the performance of the $L_{r-i}$ and $L_{i-p}$ models. When the task size is increased, the performance of the $L_{r-p}$ model remains constant; when the queue size is increased, the performance remains constant. With the different parameters tested, the performance of the $L_{r-p}$ is less affected by the changes. The stability of the learning model is an important factor. If the
application requires the model to be stable as the environmental conditions vary, then the $L_{r-p}$ model may be better than the $L_{r-i}$ or $L_{i-p}$ models. If the application requires a model to be more sensitive to a changing environment, then the $L_{r-i}$ or $L_{i-p}$ models may be better. This would depend on the application of the model.

In the comparison of the Teller and the Jockey models with the learning automata, the results are consistent with the results obtained by Glorioso and Osorio (1980). The Teller and Jockey disciplines have complete knowledge of the number of tasks in the queue. It is expected that the performance of the Teller and the Jockey is better than the learning models since the learning models gain this information based on the likelihood that the processors are either busy or free. When the task size is increased from 1000 to 5000, the performance of the learning models improves in contrast to the degeneration in the performance of the fixed rules. This is the result of the learning models gathering more information about the queues, thus improving the performance.

The learning model using the blocking factor confirms the importance of having some knowledge about the environment. The learning with the blocking factor model is tested on the cases where the other learning models demonstrated overall changes in their performance. The learning model with the blocking factor demonstrated
an overall improvement in performance when compared to the learning model without the blocking factor. The incorporation of a blocking factor into a learning model is important in queueing. It enables the queueing discipline to be sensitive to environmental changes while the discipline operates at the same level as the fixed disciplines. The results of the simulation indicate that the Learning Model with Blocking Factor is a step towards developing a discipline which is sensitive to the environment.
CHAPTER VI

CONCLUSION

Since the area of Stochastic Learning automata is so open, it is hard to draw a conclusion. The present thesis provides a history of learning automata and explores one possible application. The application raises many questions as to the use and direction of Stochastic Learning automata. The conclusion of the thesis and the question left unanswered will be presented together. Some suggestions for future research will be addressed as solutions to the open questions.

The first set of open problems presented in the thesis comes from the actual results of the thesis. The results only examine part of the capability of the learning model. Other parameters can be altered to test the performance of the learning model in a non-stable environment. Since the main advantage of the learning automata is the fact that it is sensitive to changes in the environment, it would be interesting to explore the effects of varying parameters, unequal $\mu$ values, larger numbers of tasks, and unequal $\lambda$ values. The performance of the learning models may surpass the performance of the fixed disciplines if several environmental changes occur.
A second set of open problems left unanswered is the application of the Stochastic Learning automata to other areas. Currently, it has been applied in the area of operation control. Problems such as networking, telecommunications, and routing have explored the possible of the learning automata. But can the learning model be applied in the area of Expert Systems? In some systems, the inference engine is designed in a hierarchical structure with probability value being updated by a Bayesian formula. Since the structure of the inference engine closely resembles a hierarchical Stochastic Learning automata, can the learning model be applied? If a Stochastic Learning automata is applied, can it improve the performance of the Expert System?

Problems that arose in the mid-1960s with Stochastic Learning automata in Psychology may again surface if their application becomes complex. These are questions that should be examined since there is an increasing emphasis placed on the development of Expert Systems by industry.

As mentioned, there are several open problems and more research needs to be conducted in the area of Stochastic Learning automata application. This research would not only benefit the area of Artificial Intelligence but also any area of Computer Science that uses Artificial Intelligence techniques.
APPENDIX

Program listing

PROGRAM LEARN_QUE(INPUT: /, OUTPUT, REPRT);

CONST
QUESIZE = 150; (* the size of each queue *)
NUM_PROC = 5; (* the number of processors *)

TYPE
ENV_REC = RECORD (* contains the prob that the queue is busy *)
  OCCUR : INTEGER;
  PROB : REAL;
END;

CONST_ACTION (* a subrange used in the array declarations *)
= 1..NUM_PROC;

PROB_TYPE (* array to contain the prob values *)
= ARRAY [CONST_ACTION] OF REAL;

VAR_TYPE = ARRAY [CONST_ACTION] OF INTEGER;

ENV_TYPE = ARRAY [CONST_ACTION, 1..QUESIZE] OF ENV_REC;
VAR
  REPRT : TEXT; (* the report for the statistics *)
  NEXT, (* the next event to occur *)
  NUM_CUST, (* the current number of customers *)
  TOT_CUST, (* the total number of customers to be processed *)
  SEED, (* random number seed for pascal random function *)
  TYP, (* the type of learning model used *)
  DISPL : INTEGER; (* the queueing discipline in use *)
  CLOCK, (* the clock which controls the simulation *)
  M_ARR_T, (* the mean arrival time *)
  M_SEV_T, (* the mean service time *)
  TLT_EVT, (* time to the next event *)
  TOT_DEL, (* the total amount of delay time *)
  ARRAL, (* the time for the next arrival *)
  ANIQ, (* area under the number in the queue *)
  DELAY: REAL; (* the delay for the current task *)
  C_VECT, (* prob that determines if processor is busy *)
  PROB_VECT, (* the probability vector used in the lrp *)
  TNE : PROB_TYPE; (* the time for the next task to depart *)
  NIQ, (* the pointers for the queues *)
  STATUS : VAR_TYPE; (* the status for the processors *)
  T_ARRVL : ARRAY [CONST_ACTION, 1..QUE_SIZE] OF REAL;
  ENV : ENV_TYPE; (* env record for occurrence of processor busy *)
FUNCTION POISSON (X : REAL) : REAL;

VAR
   A, B, I : REAL; (* temporary variables *)

BEGIN
   A := 1 / EXP(X); (* set A=E-LAMDA *)
   B := 1; (* initialize variables *)
   I := 0;
   WHILE A < B DO BEGIN (* generates random variable in steps of one *)
      B := B * RANDOM (SEED);
      I := I + 1;
   END;

   POISSON := I; (* assign POISSON the final step level *)
END;

FUNCTION EXPON (X : REAL) : REAL;

VAR
   U : REAL; (* random variable *)

BEGIN
   U := RANDOM (SEED);
   EXPON := -X * LN(U);
END;
PROCEDURE DISCIPLINE (VAR PROC_C : INTEGER);

PROCEDURE NSLRP (VAR PROC_C : INTEGER);

PROCEDURE ENV_UPDATE;

IF NIQ[PROC_C] > 0 THEN BEGIN
   (* increment the actual number in que *)
   ENV[PROC_C,NIQ[PROC_C]].OCCUR :=
      ENV[PROC_C,NIQ[PROC_C]].OCCUR + 1;
   FOR I := 1 TO NUM_PROC DO BEGIN
      (* compute prob the processor is busy *)
      C_VECT[I] := 0;
      FOR J := 1 TO QUE_SIZE DO BEGIN
         (* determines the prob of 1,2,3.. tasks in que *)
         ENV[I,J].PROB := ENV[I,J].OCCUR/NUM_CUST;
         (* adds up the over all prob *)
         C_VECT[I] := C_VECT[I] + ENV[I,J].PROB
      END
   END
END;
(*****************************************************************)
(* the reinforcement scheme to update the prob_vect               *)
(*****************************************************************)
PROCEDURE SUPDATE;

VAR
  FJ, GJ, SUMFJ, SUMGJ : REAL;

BEGIN
  SUMFJ := 0; (* initialize the sum accumulators *)
  SUMGJ := 0;
  IF B = 0 THEN BEGIN (* if response was a nonpenalty *)
    FOR J := 1 TO NUM_PROC DO BEGIN
      IF J <> PROC_C THEN BEGIN
        (* for every response not chosen *)
        FJ := SS1 * PROB_VECT[J];
        SUMFJ := SUMFJ + FJ;
        (* subtract an amount to decrease the prob *)
      END;
    END;
    (* increase the correct prob *)
  END;
  ELSE BEGIN (* for a penalty *)
    FOR J := 1 TO NUM_PROC DO BEGIN
      IF J <> PROC_C THEN BEGIN
        (* for each response not chosen *)
        GJ := SS2 / (NUM_PROC - 1) - SS2 * PROB_VECT[J];
        SUMGJ := SUMGJ + GJ;
        (* add an amount to increase the prob *)
      END;
    END;
    (* decrease the incorrect prob *)
  END;

BEGIN (* the beginning of the blocking learning model *)
  (* choose processor considering prob and block *)
  (* prob value and the prob of number task in que *)
  FOR J := 1 TO NUM_PROC - 1 DO BEGIN
    T1 := PROB_VECT[J] * RANDOM(SEED) * (1 - C_VECT[J]);
    T2 := PROB_VECT[J+1] * RANDOM(SEED) * (1 - C_VECT[J+1]);
    IF T1 > T2
      THEN PROC_C := J
    ELSE PROC_C := J + 1;
  END;
  IF STATUS[PROC_C] = 0 (* if the processor is busy *)
THEN \( B := 0 \)
ELSE \( B := 1; \)

ENV\_UPDATE;

CASE TYP OF

1: BEGIN
   \( SS1 := 0.7; \)
   \( SS2 := 0.7 \)
END;

2: BEGIN
   \( SS1 := 0.7; \)
   \( SS2 := 0 \)
END;

3: BEGIN
   \( SS1 := 0; \)
   \( SS2 := 0.7 \)
END;

END;

SUP\_UPDATE;

END;
(* procedure which simulates the learning model *)

PROCEDURE LRP (VAR PROC_C : INTEGER);

VAR
  B, J : INTEGER;
  SS1, SS2, T1, T2 : REAL;

  (* index variable *)
PROCEDURE LUPDATE;

VAR
   FJ, GJ, SUMFJ, SUMGJ
   : REAL;

BEGIN
  SUMFJ := 0; (* initializes the sums *)
  SUMGJ := 0;
  IF B=0 THEN BEGIN (* if the response was a nonpenalty *)
     FOR J:=1 TO NUM_PROC DO BEGIN (* for every response not chosen *)
        IF (J <> PROC_C) AND (PROB_VECT[PROC_C] < 0.9999999)
        THEN BEGIN
           FJ := SS1 * PROB_VECT[J];
           SUMFJ := SUMFJ + FJ;
           (* subtract an amount to decrease the prob *)
        END
      END;
  END
  ELSE BEGIN (* for a penalty *)
     FOR J:=1 TO NUM_PROC DO BEGIN (* for each response not chosen *)
        IF J<>PROC_C THEN BEGIN
           GJ := SS2 / (NUM_PROC-1) - SS2 * PROB_VECT[J];
           SUMGJ := SUMGJ + GJ;
           (* add an amount to increase the prob *)
        END
      END;

  END

END;
BEGIN (* the beginning of the learning model *)

(* choose the que according to the prob value *)
(* and the prob of x number of tasks in the que *)
FOR J:=1 TO NUM_PROC-1 DO BEGIN
  T1 := PROB_VECT(J) * RANDOM(SEED);
  T2 := PROB_VECT(J+1) * RANDOM(SEED);
  IF T1 > T2 THEN PROC_C := J
    ELSE PROC_C := J + 1;

  IF STATUS[PROC_C] = 0 (* if the processor is free *)
    THEN B := 0 (* emit a nonpenalty *)
    ELSE B := 1; (* emit a penalty *)

  CASE TYP OF
  1: BEGIN
      SS1 := 0.7;
      SS2 := 0.7
      END;
  2: BEGIN
      SS1 := 0.7;
      SS2 := 0;
      END;
  3: BEGIN
      SS1 := 0;
      SS2 := 0.7
      END;
  END;
LUPDATE;
END;
PROCEDURE FQD (VAR PROC_C : INTEGER);
VAR
  J, SIZE_PTR, I
: INTEGER;
SIZE_QUE
  : VAR_TYPE;
BEGIN
  (* initialize variables *)
  FOR J:=1 TO NUM_PROC DO SIZE_QUE[J] := 0;
  SIZE_PTR:=0;
  FOR J:=1 TO NUM_PROC DO (* checking for empty queues *)
    IF NIQ[J]=0 THEN
      BEGIN
        SIZE_PTR := SIZE_PTR+1;
        SIZE_QUE[SIZE_PTR] := J;
      END;
  IF SIZE_PTR = 0 THEN BEGIN(* checking for the shortest queues *)
    SIZE_PTR := 1;
    SIZE_QUE[SIZE_PTR] := 1;
    FOR J := 2 TO NUM_PROC DO BEGIN
      IF NIQ[J] = NIQ[SIZE_PTR] THEN BEGIN
        SIZE_PTR := SIZE_PTR + 1;
        SIZE_QUE[SIZE_PTR] := J;
      END
    END;
    IF NIQ[J] > NIQ[SIZE_PTR] THEN BEGIN
      SIZE_PTR:=1;
      SIZE_QUE[SIZE_PTR]:=J;
    END;
  END;
  IF (SIZE_PTR = 0) OR (SIZE_PTR = NUM_PROC) THEN (* randomly choosing from all queues *)
    BEGIN
      PROC_C:=TRUNC(NUM_PROC*RANDOM(SEED)) + 1
    ELSE
      FOR I:=1 TO NUM_PROC-1 DO
        IF SIZE_PTR=1 THEN BEGIN
          (* randomly choosing from the shortest queues *)
          J := TRUNC(I*RANDOM(SEED)) + 1;
          PROC_C := SIZE_QUE[J];
        END;
  END;
END;
END;
BEGIN
CASE DISPL OF
1: LRP (PROC_C); (* the normal learning model discipline *)
2: NSLRP (PROC_C); (* the learning model with the blocking *)
3: FQD (PROC_C); (* the teller window *)
4: FQD (PROC_C); (* the jockey discipline *)
END;
END;
(* procedure which simulations teller window with jockeying *)

PROCEDURE JOCKEYED;

VAR
    J, I  :  INTEGER;
    DONE :  BOOLEAN;

BEGIN
    DONE := FALSE; (* initialize variables *)
    I := 1;
    J := I + 1;

    WHILE NOT DONE DO BEGIN
        (* checking for queue exceeded by 1 *)
        IF NIQ[I] > NIQ[J]+1 THEN BEGIN
            NIQ[J] := NIQ[J]+1; (* jockey longer que to shorter que *)
            T_ARRVL[J,NIQ[I]] := T_ARRVL[I,NIQ[I]];
            NIQ[I]:=NIQ[I]-1;
        END
        ELSE BEGIN
            J := I; (* increment pointers *)
            I := I+1
        END;

        IF I > NUM_PROC THEN  (* checking for end condition *)
            DONE:=TRUE;

    END;
END;
PROCEDURE INITIALIZE;

VAR
  I : INTEGER; (* index variable *)
BEGIN
  CLOCK := 0; (* initialize the simulation clock *)
  ARRIVAL := CLOCK + POISSON(M_ARR_T);
  FOR I := 1 TO NUM_PROC DO BEGIN (* initialize the array elements *)
    NIQ[I] := 1;
    TNE[I] := 100000000;
    ANIQ := 0.0;
    NIQ[I] := 0;
    STATUS[I] := 0;
    PROB_VECT[I] := 1 / NUM_PROC;
  END;

  TOT_DEL := 0.0; (* initialize the statistical variables *)
  NUM_CUST := 0;
  T_LT_EVT := 0;
END;
PROCEDURE TIME_INC;

VAR
   RMIN : REAL; (* the minimum amount of time *)
   I   : INTEGER; (* the index variable *)
BEGIN
   RMIN := ARRAL; (* initialize RMIN to the arrival time *)
   NEXT := 1;
   FOR I := 1 TO NUM_PROC DO BEGIN
      (* check to determine the smallest time *)
      IF (TNE[I] < RMIN) THEN BEGIN
         RMIN := TNE[I]; (* set RMIN to the smaller value *)
         NEXT := I+1
      END;
   END;
   IF (NEXT = 1) THEN CLOCK := ARRAL
      ELSE CLOCK := TNE[NEXT-1];
END;
PROCEDURE ARRIVAL;

VAR
  J : INTEGER;
BEGIN
  (* determine the arrival time of the next task *)
  ARRVAL := CLOCK + POISSON(M_ARR_T);
  DISCIPLINE(J); (* determine the processor to handle the task *)
  IF STATUS(J)=1 THEN BEGIN (* if the processor is busy *)
    (* compute the statistics *)
    ANIQ := ANIQ + NIQ[J]*(CLOCK-T_LT_EVT);
    T_LT_EVT := CLOCK;
    NIQ[J] := NIQ[J] + 1; (* assign the task to the que *)
    IF NIQ[J] > QUE_SIZE (* check for overflow *)
      THEN WRITELN('QUE OVERFLOW')
    ELSE T_ARRVL[J,NIQ[J]]:=CLOCK
  END ELSE BEGIN (* que is empty task is assigned to processor *)
    DELAY := 0.0;
    TOT_DEL := TOT_DEL + DELAY;
    STATUS[J] := 1; (* determine the processing time for the task *)
    TNE[J] := CLOCK + EXPON(M_SEV_T)
  END
END;
PROCEDURE DEPART (I : INTEGER);

VAR
  II : INTEGER;

BEGIN
  IF NIQ[I] > 0 THEN BEGIN (* if a task is waiting in the que *)
    (* compute the statistics *)
    ANIQ := ANIQ + NIQ[I]*(CLOCK-T_LT_EVT);
    T_LT_EVT := CLOCK;
    DELAY := CLOCK - T_ARRVL[I,1];
    TOT_DEL := TOT_DEL + DELAY;
    NUM_CUST := NUM_CUST+1; (* count the departing task *)
    (* compute the processing time of the next task *)
    TNE[I] := CLOCK + EXPON(M_SEV_T);
    NIQ[I] := NIQ[I]-1;
    IF NIQ[I] > 0 THEN (* move the remaining task up in the que *)
      FOR II := 1 TO NIQ[I] DO
        T_ARRVL[I,II] := T_ARRVL[I,II+1];

    IF DISPL=4 THEN JOCKEYED; (* if jockey, rearrange the que *)
  END
  ELSE BEGIN
    STATUS[I] := 0; (* set the processor to free *)
    TNE[I] := 10000000;
    NUM_CUST := NUM_CUST+1
  END;
END;
(* procedure printout system parameters and final statistics *)

PROCEDURE REPORT;

VAR
   AVGDEL, AVGNIQ : REAL;

BEGIN
   IF NUM_CUST < 51 THEN
      BEGIN
         (* print out the state of the system *)
         WRITELN (REPRT,'MEAN INTERARRIVAL TIME ',
         M_ARR T:10:5,' MINUTES');
         WRITELN (REPRT,'MEAN SERVICE TIME ',M_SEV T:10:5,' MINUTES');
         WRITELN (REPRT,'NUMBER OF TASKS',NUM_CUST:4);
         WRITELN (REPRT);
      END;
   
   AVGDEL := TOT_DEL/NUM_CUST;
   AVGNIQ := ANIQ/CLOCK;

   WRITELN (REPRT,AVGDEL:10:5,' ',AVGNIQ:10:5);

END;

(* determines the type of discipline used in the simulation *)

PROCEDURE SIMULATION;

BEGIN
   INITIALIZE; (* initializes the system variables *)
   REPEAT
      TIME_INC; (* increments clocks and determines next event *)
      IF NEXT=1
         THEN ARRIVAL
      ELSE DEPART (NEXT-1);

      (* prints out the statistics after every 50 tasks *)
      IF ((NUM_CUST MOD 50) = 0) AND (NEXT<>1) THEN REPORT;

   UNTIL NUM_CUST=TOT_CUST;

END;
(* the mainline *)

BEGIN

   REWRITE (REPRT);

   (* initialize the constants used in the program *)

   WRITELN ('ENTER THE NUMBER OF CUSTOMERS');
   READ (TOT_CUST);

   WRITELN ('ENTER THE MEAN ARRIVAL TIME');
   READ (M_ARR_T);

   WRITELN ('ENTER THE MEAN SERVICE TIME');
   READ (M_SEV_T);

   WRITELN (REPRT,'THE LEARNING MODEL');
   WRITELN (REPRT); WRITELN (REPRT);
   DISPL 1;
   TYP := 1;
   SIMULATION;

   WRITELN (REPRT,'THE LEARNING LRI MODEL');
   WRITELN (REPRT); WRITELN (REPRT);
   DISPL := 1;
   TYP := 2;
   SIMULATION;

   WRITELN (REPRT,'THE LEARNING LIP MODEL');
   WRITELN (REPRT); WRITELN (REPRT);
   DISPL := 1;
   TYP := 3;
   SIMULATION;

   WRITELN (REPRT,'THE BLOCKING MODEL');
   WRITELN (REPRT); WRITELN (REPRT);
   DISPL := 2;
   TYP := 1;
   SIMULATION;

   WRITELN (REPRT,'TELLER WINDOW'); WRITELN (REPRT); WRITELN (REPRT);
   DISPL := 3;
   TYP := 3;
   SIMULATION;

   WRITELN (REPRT,'THE JOCKEYING'); WRITELN (REPRT); WRITELN (REPRT);
   DISPL := 4;
   TYP := 1;
   SIMULATION;

END.
BIBLIOGRAPHY


