



Distributed Learning Automaton

by

*Aleksander Mølsæther Stensby
and
Ole-Alexander Moy*

IKT 404 - Distributed Systems

Agder University College

Grimstad, May 14, 2007

Abstract

The field of Learning Automata (LA) is growing rapidly. Complex environments and large scale problems call for robust and scalable solutions that is efficient and exact. Centralized solutions with action space growing exponentially is not desirable. A decentralized decision making process may even be required in some distributed environments.

In this report we present a Distributed Learning Automaton (DLA) based on a novel and previously unpublished scheme. The DLA presented forms an effective scheme for distributed learning, and we claim that the DLA allows the best action to be found in a decentralized manner. We also claim that the DLA scales better than the L_{R-I} scheme known from the literature[1] with respect to the number of actions available to the automaton. The solution involves a team of distributed automata, each representing one possible action, operating in a stochastic environment. Experimental results shows that the DLA schemes performance converge to the desirable optimal solution when the number of internal states is sufficiently high.

This project is perform as a student project in the course
ICT 404, Distributed Systems at Agder University College, Grimstad.

Contents

Contents	1
1 Introduction	3
1.1 Acknowledgments	3
1.2 Target audience	4
1.3 Report outline	4
2 Problem statement	5
3 Background	6
3.1 Learning Automata	6
3.1.1 Stochastic Automata	6
3.1.2 Random Environment	7
3.1.3 Stochastic Automaton in a Random Environment	7
3.1.4 Efficiency	8
3.1.5 Tsetlin Automata	9
3.1.6 Updating schemes	9
3.2 Applications	11
3.3 Distribution	11
3.3.1 Distribution techniques	11
4 Solution	14
4.1 The Automaton	15
4.1.1 Updating scheme	16
4.2 Environment	17
4.3 Distribution	18
5 Implementation	19
5.1 EnvironmentServer	19
5.1.1 Messages	20
5.2 LAClient	21
6 Testing	22
6.1 Testing method	22

6.1.1	DLA testing parameters	23
6.1.2	L_{R-I} testing parameters	23
6.2	Tests	23
6.2.1	Test#1 - $k = 5$	23
6.2.2	Test#2 - $k = 50$	26
7	Discussion and conclusion	27
7.1	Further work	28
	Bibliography	29

Chapter 1

Introduction

Learning Automata (LA) is a growing field of research and it has numerous applications, among other pattern recognition, bandwidth allocation, QoS in networks, scheduling and routing.

In this paper we investigate the distribution of a k -action Learning Automaton (LA). The automaton's task is to identify the best action in a distributed fashion. When the number of actions available to an automaton grows, and the automaton operates in complex environments, distribution is often desired. Scalability issues and geographical location are both important motivators.

In this paper we will design and implement a Distributed Learning Automaton (DLA) based on a novel and previously unpublished scheme. We will measure the efficiency of the automaton in terms of learning speed and accuracy to prove that the DLA presented forms an effective scheme for distributed learning, and that the DLA allows the best action to be found in a decentralized manner. We also claim that the DLA scales better than the L_{R-I} scheme with respect to the number of actions available to the automaton. The DLA is similar to the centralized automaton described in [2].

1.1 Acknowledgments

We would like to thank our supervisor Ole-Christoffer Granmo for excellent guidance throughout the project period. Granmo has been a tremendous support in our report writing and has contributed with much useful input, especially concerning our testing phase and on the scheme itself.

1.2 Target audience

The target audience for this report is people with fundamental knowledge about distribution and computer networks. Learning Automata is an unknown field to many master level students, and we have therefore chosen to dedicate a large portion of our background chapter to this.

1.3 Report outline

The rest of this report is organized as follows: In chapter 2 on the following page we present the problem statement and constraints taken for the project. In chapter 3 on page 6 we give a brief overview of the field of learning automata and distribution. In chapter 4 we fully describe our DLA solution. The implementation is presented in chapter 5 on page 19. We test the solution in chapter 6 before we conclude the report in chapter 7 on page 27 with a brief discussion of the test results and our final conclusion.

Chapter 2

Problem statement

In this paper we investigate the distribution of a k -action Learning Automaton (LA) in a perfect communication network. The automaton's task is to identify the best action in a distributed fashion. When the number of actions available to an automaton grows, and the automaton operates in complex environments, distribution is often desired. Scalability issues and geographical location are both important motivators. In this paper we will design and implement a Distributed Learning Automaton (DLA) based on a novel and previously unpublished scheme. We will measure the efficiency of the automaton in terms of learning speed and accuracy, and we will compare the results to an Automaton that is using the well known L_{R-I} updating scheme. The automaton will be implemented using the Java programming language using socket level communication.

Chapter 3

Background

In this chapter we give a brief overview of the field of Learning Automata, its theory and application, then describe some relevant distribution techniques related to the problem at hand.

3.1 Learning Automata

Learning Automata (LA) are adaptive processes that are built to make decisions in an environment. They can progressively improve their performance by the means of a learning process. In general, the concept of an automaton is a system that takes in a sequence of inputs and puts out a sequence of actions. The automaton updates its action probabilities based on response given by an environment. A learning automaton combine rapid and accurate convergence with low computational complexity. In this section we describe the Stochastic Automata then the environment before we explain the actual Learning Automata which is a Stochastic Automaton in a Random Environment.

3.1.1 Stochastic Automata

In [5], Narendra and Thathatchar define a stochastic automaton as a sextuple $\{\bar{\beta}, \bar{\Phi}, \bar{\alpha}, \bar{p}, A, G\}$ where $\bar{\beta}$ is the input set, $\bar{\Phi}$ is the set of internal states and $\bar{\alpha}$ is the output or action set. \bar{p} is the state probability vector which is used to decided the choice of the state at each time step. A is the updating function. This is also called the updating scheme or the reinforcement scheme. G represent the output function that the automaton use to map $G : \phi \rightarrow \alpha$. In other words, G is the function that decide the output α based on the automaton's current state ϕ . Often $\beta \in \{0, 1\}$ where 0 normally represent a reward, or a non-penalty, and 1 normally represent a penalty. In the general case $\bar{\beta} = \{\beta_1, \dots, \beta_m\}$. The set of internal

states are noted as $\bar{\Phi} = \{\phi_1, \dots, \phi_N\}$ and the set of actions as $\bar{\alpha} = \{\alpha_1, \dots, \alpha_k\}$. The probability vector at time n is noted as $p(n) = [p_1(n), p_2(n), \dots, p_s(n)]^T$. The updating function is used to calculate $p(n+1)$ based on $p(n)$.

3.1.2 Random Environment

[6] define an environment as

“(..)a large class of general unknown random media in which an automaton or a group of automata can operate.”

A random environment is an environment where the impact/result of the automata's actions does not necessarily produce the same response each time it is performed.

The environment has inputs $\alpha(n) = \{\alpha_1, \dots, \alpha_k\}$ and outputs $\bar{\beta} = \{\beta_1, \dots, \beta_m\}$. Normally, $\beta \in \{0, 1\}$ where 0 normally is called a reward (or a non-penalty response) and 1 being called a penalty response. To decide the response, the environment also has a penalty probability vector $\bar{c} = \{c_1, \dots, c_r\}$.

We also differentiate between two environments, namely a stationary and a non-stationary one. We say that the environment is stationary if \bar{c} does not depend on the time step n . Otherwise it is non-stationary, i.e. the penalty probabilities change. A stationary environment is also known as a STATIC environment, and a non-stationary environment is known as a DYNAMIC environment.

3.1.3 Stochastic Automaton in a Random Environment

The environment's input corresponds to the automaton's output, and the environment's output corresponds to the automaton's input. Together, the Stochastic Automaton and the Random Environment constitute the Learning Automaton shown in figure 3.1 on the following page. Narendra and Thathachar define the learning automaton as follows[5]:

“A learning automaton is a stochastic automaton that operates in a random environment and updates its action probabilities in accordance with the inputs received from the environment so as to improve its performance in some specified sense.”

By interacting with the environment and receiving response to its action, we say that the automaton is learning. More correctly we say that the automaton is learning if it reduces the number of penalties received as a result of interaction with the environment. The number of automaton states determines both the accuracy and the speed of the learning process.

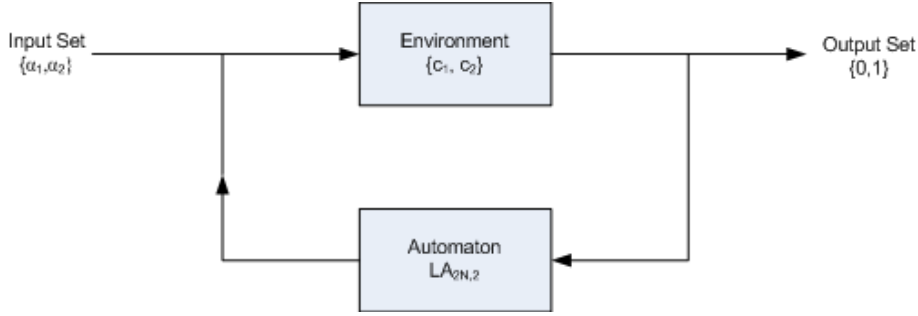


Figure 3.1: Learning Automaton (Stochastic Automaton in a Random Environment)

3.1.4 Efficiency

Narendra and Thathachar points out four behaviour definitions in [5] useful to judging the behaviour and efficiency of learning automata. The average penalty received by the automaton is a very useful quantity in this evaluation. Note that the four definitions in this section are paraphrased and shortened on the background of those defined in [5]. The first behaviour defined is expediency [8].

Definition 1. A learning automaton is called *expedient* if

$$\lim_{n \rightarrow \infty} E[M(n)] < M_0 \quad (3.1)$$

where $E[M(n)]$ is the estimated value of the average penalty conditioned on $p(n)$ at a certain time step n and M_0 is the initial average penalty when actions are chosen at random. In other words, the learning automaton is expedient if it does better than one that choose actions purely at random.

If one can minimize the average penalty by properly selecting the actions we call the automaton optimal.

Definition 2. A learning automaton is called *optimal* if

$$\lim_{n \rightarrow \infty} E[M(n)] = c_l \quad (3.2)$$

where c_l is the minimal penalty probability c_i .

Optimality is desirable, but can seldomly be achieved. Often, one would rather aim at a suboptimal performance. This is given by what we call ε -optimality [9].

Definition 3. A learning automaton is called ε -*optimal* if

$$\lim_{n \rightarrow \infty} E[M(n)] < c_l + \varepsilon \quad (3.3)$$

can be obtained from any arbitrary $\varepsilon > 0$ by a suitable choice of the parameters of the reinforcement scheme. This implies a performance as close to the optimal as desired.

The last definition is called absolute expediency [4].

Definition 4. A learning automaton is called *absolutely expedient* if

$$E[M(n+1)|p(n)] < M(n) \quad (3.4)$$

for all n , all $p_k(n) \in (0, 1)$ ($k = 1, \dots, r$) and all possible values of c_i ($i = 1, \dots, r$).

From this definition we see that absolute expediency means that the expected average penalty $E[M(n)]$ is strictly monotonically decreasing with n .

3.1.5 Tsetlin Automata

Tsetlin was a pioneer in the field of automata theory and his work in the early 1960's has made the foundation for the further development of the field [8]. Tsetlin studied the behaviour of deterministic automata in random environment. He presented an automaton with two actions and two internal states called the Two-State Automaton $L_{2,2}$ or a Two-State Tsetlin Automaton. The state updating rule is very simple, yet powerful; Upon receiving a penalty the state switches and upon receiving a reward, the automaton remains in the same state. The environment in this case consist of the penalty probabilities $\{c_1, c_2\}$. The transition matrices are shown below in 3.5.

$$F(0) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad F(1) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad (3.5)$$

A Tsetlin-automaton can learn the optimal action if the lowest penalty probability is less than 0.5. Also, extensions of the $L_{2,2}$ automaton have been made. The Two Action Automaton with memory, called $L_{2N,2}$ is an automaton with $2N$ states and 2 actions.

3.1.6 Updating schemes

The transition function of an automaton determines the state at the instant $(n+1)$ in terms of the state and input at the instant n . In section 3.1.1 on page 6 we used the variable A to represent the updating function. In this section we investigate

two common updating or reinforcement schemes, namely the linear reward-penalty scheme and the linear reward inaction scheme. Narendra and Thathachar outlines an updating scheme in the following manner [5]:

“If the learning automaton selects an action α_i at instant n and a non penalty input occurs, the action probability $p_i(n)$ is increased, and all the other components of $p(n)$ is decreased. For a penalty input, $p_i(n)$ is decreased, and the other components are increased.(...) Occasionally the action probabilities may be retained at the previous values, in which case the status quo is known as “inaction”.”

The Linear Reward-Penalty Scheme (L_{R-P})

The linear reward-penalty scheme (L_{R-P}) is one of the earliest schemes in mathematical psychology, proposed as early as in 1958 by Bush and Mosteller [1]. Automata using the L_{R-P} scheme is *expedient* in all stationary random environments. The L_{R-P} scheme can be explained through the following updating algorithm:

$$p_i(n+1) = \begin{cases} p_i(n) + a(1 - p_i(n)) & \alpha(n) = \alpha_i, \beta(n) = 0 \\ (1 - b)p_i(n) & \alpha(n) = \alpha_i, \beta(n) = 1 \end{cases} \text{ for all } j \neq i. \quad (3.6)$$

$$p_j(n+1) = \begin{cases} (1 - a)p_j(n) & \alpha(n) = \alpha_i, \beta(n) = 0 \\ p_j(n) + b(1 - p_j(n)) & \alpha(n) = \alpha_i, \beta(n) = 1 \end{cases} \text{ for all } j \neq i. \quad (3.7)$$

In the algorithm above, a and b represent reward and penalty parameters respectively where $0 < a, b < 1$.

The Linear Reward-Inaction Scheme (L_{R-I})

The linear reward-inaction scheme is derived from the L_{R-P} scheme by not changing the probabilities when a penalty is given. (We set $b = 0$). The L_{R-I} scheme is considered to be ϵ -optimal[9]. The L_{R-I} scheme is explained in the following algorithm:

$$p_i(n+1) = \begin{cases} p_i(n) + a(1 - p_i(n)) & \alpha(n) = \alpha_i, \beta(n) = 0 \\ p_i(n) & \alpha(n) = \alpha_i, \beta(n) = 1 \end{cases} \text{ for all } j \neq i. \quad (3.8)$$

$$p_j(n+1) = \begin{cases} (1 - a)p_j(n) & \alpha(n) = \alpha_i, \beta(n) = 0 \\ p_j(n) & \alpha(n) = \alpha_i, \beta(n) = 1 \end{cases} \text{ for all } j \neq i. \quad (3.9)$$

3.2 Applications

The field of Machine Learning and Learning Automata has been applied to many real-life problems. Pattern recognition, scheduling, image processing, routing and bandwidth allocation in computer networks are only a few examples. Quality of Service control in Sensor Networks[3] is also a scenario where learning automata has been applied and shown great results. Scenarios like the sensor networks also brings on new challenges, and decentralization occurs as a necessary feature. In the next section we explain distribution and different techniques used in distribution.

3.3 Distribution

The need for distribution often arise when the scenario involves complex natural and man-made systems. In reality, the complete information exchange needed for centralized decision making may not be feasible. In this section we investigate the benefits of distributed learning, and we also look into some of the problems distribution brings with it. [2] points out the weakness of a centralized LA with a large number of actions in this fashion:

“(...) It is conceivable that this problem can be resolved with a single LA possessing an extended number of actions. But we do not recommend it for scalability reasons - the action space would grow exponentially. Furthermore, the benefit of having a decentralized decision making process would be lost (e.g., for distributed environment such as wireless sensor networks).”

We round up this section by describing briefly some of the most common distribution techniques.

3.3.1 Distribution techniques

The server or coordinator can be one fixed entity in the network, leaving all the nodes or clients with a single point of failure that can easily be traced and fixed or the coordinator can be a node selected with an election algorithm and that is replaced if it is unresponsive for some time by the remaining clients.

An algorithm that could be used to the later example is the Bully Election Algorithm that could have a score based upon the nodes average response time to a number of the other nodes combined with current workload and capable bandwidth.

Client / Server Architecture

A server can be one computer or a cluster of computers that have one or more clients or nodes connected to it. The client can either have a persistent connection or just send or receive information and then disconnect from the server.

An example from our daily lives are web servers where the web browser requests a page on the server and receives this before it disconnects from the web server until you click a link that takes you to another page on the server or you open a stream.

Sockets

Socket is a peer to peer communication scheme on an IP based network. Stream Socket is commonly used and utilizes TCP/IP which provides a reliable link between the client and server. For instance, Stream Socket is used with Telnet to ensure that all packages that are sent from the client are received on the server side in the same order as they were typed. TCP/IP ensures that packages are not corrupted with a checksum of the package content. The sender of the package either knows that the target has received the package or a timeout occurs and actions can be taken according to this information, though they might be arbitrary.

Web Services

A Web Service is a collective term for a resource or functional component that can be requested via the web and that returns information that can be used in applications. The resource can have one or more functions that can be used.

Most Web Services communicate via SOAP, return data in an XML structure and has a WSDL document to give developers that access the Web Service an overview of what capabilities it has. The WSDL document can be parsed by IDE one uses and the Web Service is referred to as a class with the methods defined in the WSDL.

An example of a Web Service can be the Amazon Web Service that allows applications to send in an ISBN number and the Web Services does a database call and tries to find the book requested by its ISBN number. If the book is found, it is returned in an XML structure.

RPC / RMI

Remote Procedure Call(RPC) is a method that allows a computer program to access and execute functions on a remote computer via a network. RPC calls are writ-

ten with object-orientation in mind and is accessed much in the same way as the programmer would access functions in another class. This leaves the developers above the socket stage and with simpler programming.

Remote Method Invocation[10] or RMI which it's usually mentioned as, is Java's version of remote procedure calls[7] and is a way to make distributed applications on the Java platform. The RMI API gives full object-oriented polymorphisms between remote applications by applying object serialization. There are two ways to write RMI applications, one with callbacks and one without callbacks.

Using callbacks means that one client can call a function on another client remotely, for this to work all clients connected is working both as a server and a client. Doing it like this you don't need a centralized server that everyone needs to connect to, like you would using RMI without callbacks.

There are three layers in RMI:

- **Stub/Skeleton**
The stub is the program running on the client side and the skeleton is the program on the server.
- **Remote Reference Layer**
This layer finds out the state of the call from the client to the server, for example if the call a single request or a multi cast.
- **Transport Connection Layer**
Sets up and maintains the requests.

The request traverses down the layers on the side sending the request and the other way on the receivers end.

Perfect Communication

We will assume that the server and the clients are running on a perfect network with perfect service. Perfect service means that there can be no arbitrary faults or server/client deaths that indicate imperfect communications.

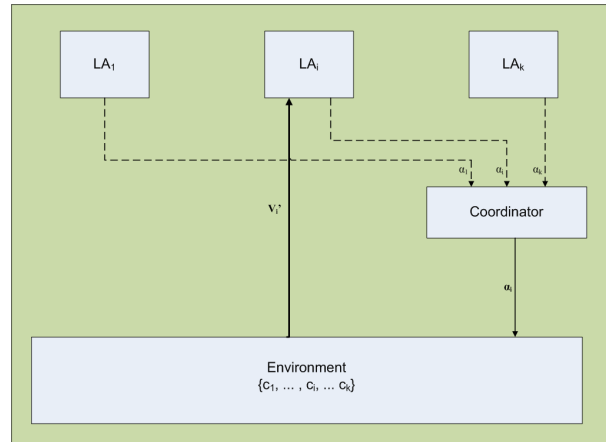


Figure 4.2: A centralized team of cooperating LA.

4.1 The Automaton

The DLA presented in this paper is a modified version of the team of centralized LA described previously in figure 4.2. The main difference is that the coordinator and environment is merged into one single unit, and that each individual LA in the system is distributed in addition to the environment. The system architecture of the DLA is shown in figure 4.3.

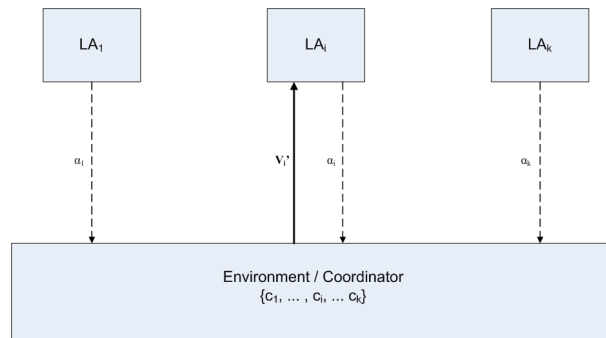


Figure 4.3: Distributed Learning Automaton (DLA)

Each automaton LA_i represents a single action α_i . Since the DLA can be viewed as a distributed version of the k -action automaton, we have k LAs in the system, each representing the actions $\bar{\alpha} = \{\alpha_1, \dots, \alpha_k\}$. Corresponding to these actions, the environment consist of the penalty probabilities $\bar{c} = \{c_1, \dots, c_k\}$. We note the response given by the environment corresponding to an action α_i as V_i^1 .

We assume that each automaton has the same number of internal states $\bar{\Sigma}$. We

¹The response given by the environment is also commonly noted as β

denote the number of internal states as N thus giving us $\bar{\Sigma} = \{\sigma_1, \dots, \sigma_N\}$. Since each automaton only has one action α_i , the choice available to the automaton is to either take the action α_i or to ignore (i.e. to take no action). Because of this each automaton only keep track of the one probability $p(n)$ given fully by the automaton's current state ϕ .

Each individual LA_i does not know about the other automata in the system. It is only the environment that has knowledge of the automata that the system consist of, and we make the assumption that the DLA operates in a perfect communication network. As we will describe later in this chapter, we have designed our solution using socket communication, but the solution can easily be redesigned to fit communication based on for instance web services or remote procedure calls / remote method invocation.

The main idea is that the clients (the individual LA_i) communicate solely with the server (the environment / coordinator) through message passing. At each time step, the automata will make a decision to either take the action or not. Upon taking the action the automaton sends a message to the environment signaling that they decided to perform the action. The environment will only choose the first message or action received in that time step and discard all other messages. A response is given to the automaton that took the action, consisting of either a penalty or a reward. The environment can also choose to fully omit responding, which will be explained later. The automaton that receives the response will in turn update their state based on the response and move on to the next time step. The automata that does not receive any response will wait for a time-out before advancing to the next time step.

4.1.1 Updating scheme

The individual LAs updates their current state σ based on the response V_i' given by the environment. Our solution make use of an updating scheme we have chosen to call the DLA Reward-Penalty Scheme (DLA_{R-P}). The scheme is in essence the same scheme as used by the Tsetlin Automaton $LA_{2N,2}$ described in section 3.1.5 on page 9.

The state of the automaton LA_i at the time t is expressed as $s_i(t)$. If each automaton has N states, the probability of choosing action α_i is given by $\frac{s_i(t)}{N}$.

The state of the automaton is updated as

$$s_i(t+1) := s_i(t) + 1, \text{ if } v_i'(t) = 0 \text{ and } 1 \leq s_i(t) < N \quad (4.1)$$

$$:= s_i(t) - 1, \text{ if } v_i'(t) = 1 \text{ and } 1 < s_i(t) \leq N \quad (4.2)$$

$$:= s_i(t), \quad \text{otherwise} \quad (4.3)$$

4.2 Environment

The environment consist of the penalty probabilities $\bar{c} = \{c_1, \dots, c_k\}$, as stated earlier. In addition to \bar{c} , the environment also keep track of the last response it gave, V . The environment keeps track of this in order to balance the total probability of the whole DLA. By never giving the same response twice in a row, the total action probability of the DLA totals to approximately 1 over time. It is still important to note that since the internal automata does not know about each other or communicate with each other in any way, there is no way for them to synchronize their probabilities. We also want to keep the amount of data transfered between the automata and the environment to a minimum. By letting the environment suppress repeated penalties, or repeated rewards, we prevent automata from increasing their action probability more rapidly than other automata decrease theirs. (From a theoretical point of view, we really want to decrease all other action probabilities by the amount we increase the rewarded action, keeping the total probability summed to 1 at all time.) By making use of response suppression in this way we achieve approximately the same², only without the enormous information flow it would require to keep all automata synchronized, and the solution will scale in a decent fashion.

As a scenario, the action received α_i has the corresponding penalty probability c_i . If the last response V was a penalty, and the environment draws a penalty for α_i based on the probability c_i , the response will be discarded. On the other hand if the environment draws a reward, automaton LA_i will receive its reward from the environment. The last response V is global for the whole DLA, thus it does not matter if LA_1 received the previous penalty and LA_2 is supposed to receive a penalty for its next action. The response will be suppressed until a correct action is taken by any of the automata.

As mentioned above, the environment will only choose the first message or action received in that time step and discard all other messages. This is what we call “First Come, First Served” (FCFS) with discarding. We make the assumption that the same LA_i will not be the first to communicate with the environment at each time step. This would make the automaton get stuck in an endless loop, which is undesirable.

²As long as the total probability converges to 1, it is sufficient for our solution.

4.3 Distribution

The server and clients communicate via stream socket over TCP/IP. The TCP/IP transport layer leaves the server and clients with a relatively reliable connection where the desired action send from the client to the server can not easily be corrupted because of the TCP checksum mechanism. Both client and server have advantages of TCP/IP because the client is informed when the server receives a message and the server is informed when the answer is received by the client.

Chapter 5

Implementation

This chapter is dedicated to the implementation of our solution. The implementation is done using the Java programming language and sockets for network communication.

The DLA consist of one environment server implemented as *EnvironmentServer* and multiple instances of an LA client implemented as *LAClient*.

5.1 EnvironmentServer

The *EnvironmentServer* holds the penalty probabilities *penaltyProbs* in addition to a map containing one *ClientHandler* for each client. The *ClientHandler* is constantly listening for messages from the client assigned to it and upon arrival calls the action on the server. The use of a *ClientHandler* is just one way of organizing the communication in a socket-to-socket system. The last part of the *EnvironmentServer* is an instance of a *ConnectionListener*.

The *ConnectionListener* listens for incoming connections on the server socket, and adds them as clients upon connection. The clients are also assigned a unique identifier *ID* by the *ConnectionListener*.

In our implementation we have chosen to accept a predefined number of clients (LAs). After the desired number of clients has connected to the server, the server will broadcast a *START* message to all clients, letting them know that the environment is ready and set up. We also specify the number of time steps for testing purposes. This is explained further in section 5.2 on page 21. The *EnvironmentServer* keep track of the last response given through the variable *lastResponseGiven*. The environment response algorithm is outlined in the following pseudo code.

Listing 5.1: Environment Response Algorithm

```

IF ( rand < pProb )
THEN:
//PENALTY
  IF ( lastResponse != PENALTY )
    THEN: GivePenalty
    lastResponse := PENALTY
  ELSE: NoResponse
ELSE:
//REWARD
  IF ( lastResponse != REWARD )
    THEN: GiveReward
    lastResponse := REWARD
  ELSE: NoResponse

```

In the previous algorithm, the penalty probability $pProb$ corresponds to the incoming action from one client and $rand$ is a random double $0 \leq rand \leq 1$.

5.1.1 Messages

The *EnvironmentServer* operates with the following messages:

Listing 5.2: Messages

```

public final static int DOACTION = 2;
public final static int REWARD = 3;
public final static int PENALTY = 4;
public final static int START = 5;

```

The *DOACTION* message is sent from the *LAClient* when it choose to perform its action. *REWARD* and *PENALTY* is given by the *EnvironmentServer* as response to a *DOACTION* message. *START*, as mentioned earlier, activates the DLA.

In addition to these messages, we choose to add a message for *INACTION* and one for *NORESPONSE*. This was done to make the testing process easier and to leave out the time-out aspect of the solution, presented in the previous chapter. It should be noted that we decided to not implement the time-out feature since this would only cause the testing phase to take longer time. This aspect can easily be added in future implementations by implementing a simple wait clause to the flow of the automaton and environment. Time-out is merely a trivial mechanism, and we don't consider this an important aspect in our paper.

Listing 5.3: Additional Messages

```

public static final int INACTION = 6;
public static final int NORESPONSE = 7;

```

5.2 LAClient

The *LAClient* consist of N internal states. It keeps track of its current state through the variable *state*. The initial state is decided by the number of states and the number of actions in the DLA. For our implementation, we have chosen to predefine this, but this can also be initialized by the environment at the beginning of the automata life cycle. For simplicity we choose an even number of states corresponding to the number of actions. By the formula $\phi(0) = (1/k) * N$ we get the initial state of each automaton. In the case of 5 actions and $N = 10$ states, the initial state will be 2. Then each action will have 20% probability of being chosen.osen to predefine this, but this can also be initialized by the environment at the beginning of the automata life cycle. For simplicity we choose an even number of states corresponding to the number of actions. By the formula $\phi(0) = (1/k) * N$ we get the initial state of each automaton. In the case of 5 actions and $N = 10$ states, the initial state will be 2. Then each action will have 20% probability of being chosen.

The actual automaton time step cycle is implemented in the *step()*-method of the *LAClient*. The automaton thread loops until the number of time cycles specified has been reach. The automaton will decide to perform an action at each time step using their current state as the action probability ($P = state/N$). The number of time steps for each run is given as input to the DLA for testing purposes.

Upon selecting an action and receiving a response, the automaton updates it's state according to the state updating algorithm 4.1 on page 16 presented in chapter 4. The algorithm is quite simple and can be outlined as the following pseudo code:

Listing 5.4: State Updating Algorithm

```

IF ( PENALTY )
THEN:
    IF ( state > 1)
    THEN: state := state - 1
ELSE:
    IF ( state < N )
    THEN: state := state + 1

```

The first special case occurs when the current state is the first state, e.g. state number one, the state will remain the same upon receiving a penalty. The other special case occurs when the current state is the last state, e.g. state number N, the state will remain the same upon receiving a reward.

In the next chapter we describe the testing of the DLA and compare it to the L_{R-I} scheme presented previously in section 3.1.6 on page 10. For further information on our Java specific implementation we refer the reader to the source code.

Chapter 6

Testing

In this chapter we present our testing results of the DLA and compare these result to a decentralized L_{R-I} automaton with k -actions. We have conducted these tests to prove the superiority of the DLA scheme over the L_{R-I} scheme when it comes to scalability with respect to the number of actions available to the automaton. We also demonstrate how the DLA is able to identify the best action in a decentralized manner.

6.1 Testing method

The testing method we have chosen to use is that of an estimated reward probability. This method is equivalent to the estimated penalty probability method explained in chapter 3 in section 3.1.4 on page 8 about efficiency of learning automata. Instead of using the penalty probability we have chosen to estimate the reward probability for the simple reason that we want to compare efficiency with the L_{R-I} scheme, and since the L_{R-I} scheme only update the action probabilities upon reward, we found that this suited our test scenario better.

The method counts the number of rewards given by the environment in an array $Rewards[n]$, with length T representing the number of time steps in the simulation. At each time step n we add to the reward-counter for the specific action. The simulation is then repeated Z times, and the estimated reward probability is calculated by the following equation:

$$E[Q(n)] = \left(\frac{Rewards[n]}{Z} \right) \quad (6.1)$$

6.1.1 DLA testing parameters

The test suite we use for the DLA take four parameters as input; the number of automata states to use (N), the number of actions (k), the number of time steps (T) and the the number of tests to run (Z). In addition to these parameters we also have to specify the penalty probabilities (representing the correct action).

6.1.2 L_{R-I} testing parameters

The L_{R-I} test suite takes four parameters as input. The only difference from the DLA parameters is that the (N) parameter is replaced by a learning parameter or reward parameter a . In addition to these parameters we also have to specify the penalty probabilities (representing the correct action). The rate of convergence is determined both by the the learning parameter a and the penalty probabilities [6].

6.2 Tests

We have performed two tests, where the first test has three sub-tests. The two tests consist of one test with 5 actions available to the automaton, and a second test with 50 actions available to the automaton. The three sub tests consisted of varying number of states in the DLA case and varying learning parameter in the L_{R-I} case. This section describe these tests, and their corresponding results. We have used the value $Z = 1000$ in all the following tests.

6.2.1 Test#1 - $k = 5$

The first testing phase performed was on a DLA with 5 actions. In each of these tests, action 2 was the correct action and had a reward probability of 80%. The rest of the actions had a reward probability of 10%.

DLA: $N = 1000, T = 15000$

Figure 6.1 on the next page shows the result of the 5-action DLA with 1000 internal states and 15000 time steps. It shows that the DLA reaches an estimated reward probability of 50% after 3900 time steps and that it converges to the desired reward probability of 80% after 8100 time steps.

Figure 6.2 on the following page shows the L_{R-I} automaton with 5 actions and the learning parameter set to .01. The L_{R-I} solution converges rapidly.

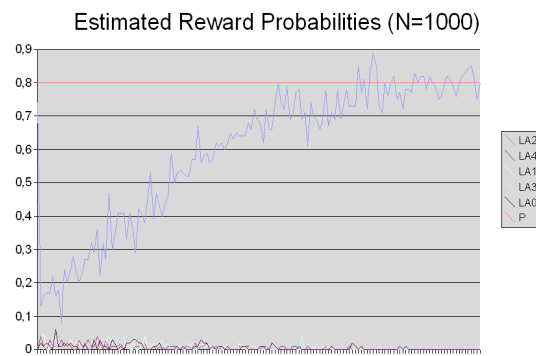


Figure 6.1: DLA, 5 actions, 1000 states (x-axis range from 0 to 15 000)

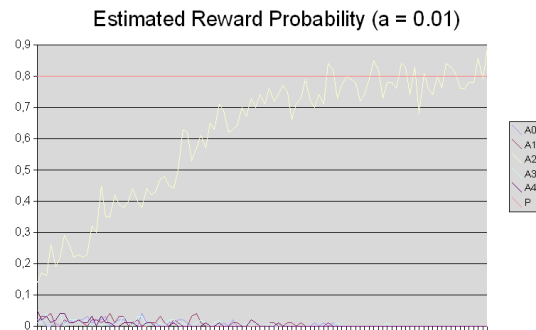


Figure 6.2: L_{R-I} , 5 actions, $a = 0.01$ (x-axis range from 0 to 1 000)

By changing the learning parameter to $a = 0.1$ we see that the L_{R-I} converges much faster, but with more noise. This is shown in figure 6.3.

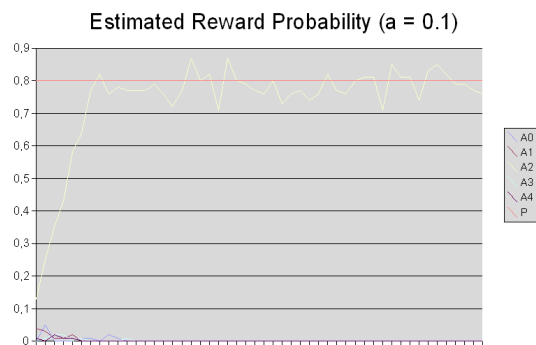


Figure 6.3: L_{R-I} , 5 actions, $a = 0.1$ (x-axis range from 0 to 1 000)

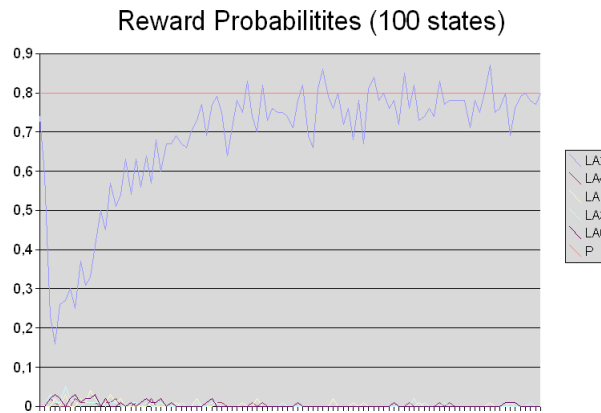


Figure 6.4: DLA, 5 actions, 100 states (x-axis range from 0 to 4000)

DLA: $N = 100, T = 4000$

In this test, shown in figure 6.4, the DLA reaches an estimated reward probability of 50% after 560 time steps. It converges to the desired reward probability of 80% after only 1400 time steps.

DLA: $N = 25, T = 2000$

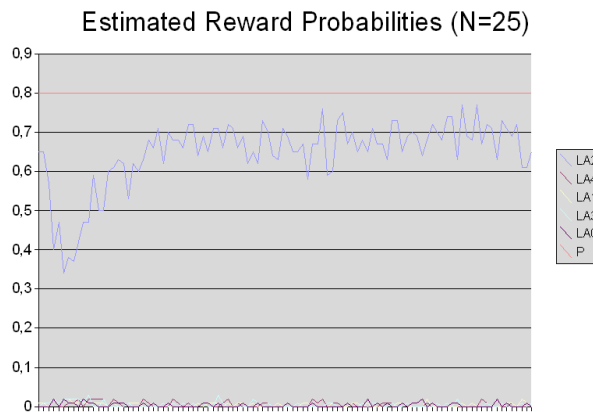


Figure 6.5: DLA, 5 actions, 25 states (x-axis range from 0 to 2000)

Here, the DLA reaches an estimated reward probability of 50% after 160 time steps. With so few internal states, the DLA does not reach its desired reward probability of 80%, but converges to an approximate reward probability of 70% after 450 time steps. This is shown in figure 6.5.

6.2.2 Test#2 - $k = 50$

In the final test we used 50 actions. The results were as shown in figure 6.6 and figure 6.7, each representing the DLA with 1000 states, and the L_{R-I} with learning parameter set to .1 respectively. It clearly shows that the L_{R-I} is not nearly as good as the DLA when it comes to scalability. We see this by comparing the rate of convergence in figure 6.3 on page 24 with the same learning parameter as in figure 6.7.

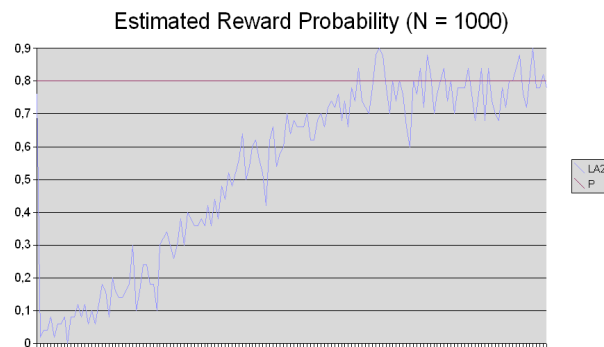


Figure 6.6: DLA, 50 actions, 1000 states (x-axis range from 0 to 15 000)

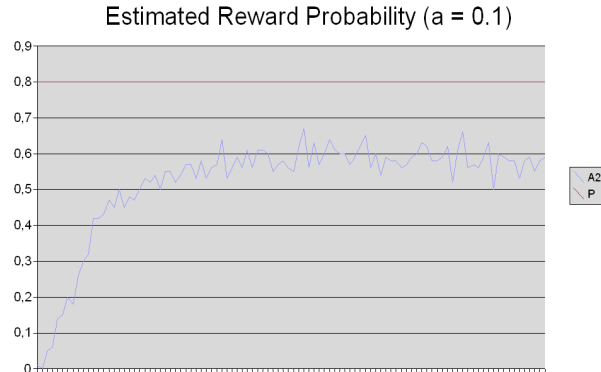


Figure 6.7: L_{R-I} , 50 actions, $a = 0.1$ (x-axis range from 0 to 1000)

Chapter 7

Discussion and conclusion

The Distributed Learning Automaton presented in this report utilize a simple, yet powerful state updating scheme based on the Tsetlin $L_{2N,2}$ scheme. The test results show that the DLA scheme presented in this report is superior to the L_{R-I} scheme when it comes to scaling with respect to the number of actions available to the automaton. We have also proved that the DLA scheme is able to find the best action in a decentralized manner, which can be desirable in large, complex environments such as wireless sensor networks.

The experiments done on the DLA scheme demonstrate that the automaton is able to find the best action and converge to the optimal solution when using a sufficiently high number of internal states. One weakness to the scheme is still that a high number of internal states demand a rather high number of time steps for the estimated reward probability to converge to the optimal value. A low number of internal states clearly demonstrate rapid learning, but is not nearly as exact as in the case of a large number of internal states. Still, the last test shows that the DLA scheme scales better than the L_{R-I} scheme, compared to the tests performed on automata with fewer actions.

Despite this obvious weakness, we conclude that the DLA presented in this report forms an effective scheme for distributed learning. It is able to find the desirable action, and it scales better than the L_{R-I} scheme with respect to the number of actions available to the automaton, even though the L_{R-I} is more efficient in its learning with a significantly smaller amount of actions.

7.1 Further work

For further work with the DLA scheme we strongly suggest improvements to the one clear weakness identified in this project, namely the requirement for a rather large number of internal states and corresponding required number of time steps.

Bibliography

- [1] R. R. Bush and F. Mosteller, *Stochastic Models for Learning*. John Wiley & Sons, New York, 1958.
- [2] O.-C. Granmo, B. J. Oommen, S. A. Myrer, and M. G. Olsen, “Learning automata-based solutions to the nonlinear fractional knapsack problem with applications to optimal resource allocation,” *IEEE Transactions on Systems Man and Cybernetics*, vol. 37, pp. 166–175, 2007.
- [3] R. Iyer and L. Kleinrock, “Qos control for sensor networks,” in *Proc. of the IEEE International Conference on Communications, 2003.*, 2003.
- [4] S. Lakshmivarahan and M. Thathachar, “Absolutely expedient learning algorithms for stochastic automata,” *IEEE Transactions on Systems Man and Cybernetics*, vol. SMC-3, pp. 281–286, 1973.
- [5] K. Narendra and M. Thathachar, “Learning automata — a survey,” *IEEE Transactions on Systems Man and Cybernetics*, vol. SMC-4, pp. 323–334, 1974.
- [6] ———, *Learning Automata*. Prentice-Hall, 1989.
- [7] Remote method invocation. Sun Microsystems, Inc. [Online]. Available: <http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp>
- [8] M. Tsetlin, “On the behaviour of finite automata in random media.” *Avtomat. Telemekh.*, vol. 22, pp. 1345–1354, 1961.
- [9] R. Viswanathan and K. Narendra, “A note on the linear reinforcement scheme for variable-structure stochastic automata.” *IEEE Transactions on Systems Man and Cybernetics*, vol. SMC-2, pp. 292–294, 1972.
- [10] Java remote method invocation. Wikipedia. [Online]. Available: http://en.wikipedia.org/wiki/Java_remote_method_invocation