

AN APPLICATION OF N-GRAM
SELF-ORGANIZING MAPS

By

LEE YONG TAN

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma

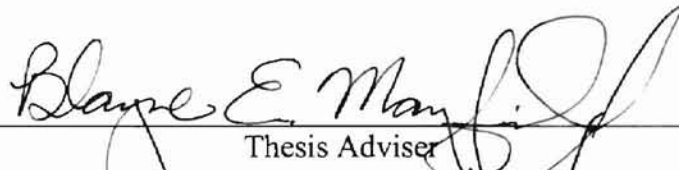
1995

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1999

OKLAHOMA STATE UNIVERSITY

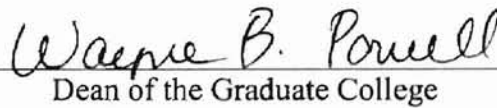
AN APPLICATION OF N-GRAM
SELF-ORGANIZING MAPS

Thesis Approved:


Thesis Adviser






Dean of the Graduate College

ACKNOWLEDGEMENTS

I would like to express my sincerely appreciation to Dr. Blayne Mayfield for being my thesis advisor. I thank him for his intelligent supervision, constructive guidance and friendship. I would also like to express my gratitude to my thesis committee members, Dr. John Chandler and Dr. George Hedrick for assisting with my research. And special thanks go to Dr. Mansur Samadzadeh for his kindness assistance for my thesis research.

In addition, I would like to specially thank my girlfriend, Wan Fong Estelle Kew, for her strong encouragement and support through the difficult times, generous assistant, immeasurable love and understanding through out the whole process. I would also like to specially thank my parent for their financial support, encouragement and understanding that giving me the opportunity to further my education at Oklahoma State University.

Finally, I thank the Computer Science Department for giving me the opportunity to accomplish the undergraduate and the graduate programs. Special acknowledgement also goes to Mr. Mike Bradley for his friendship and assistance on proof reading my thesis report and not forgetting my best friend, Mr. Kuang Ming Yoo for his generous support and kind assistance in my thesis printing. My thanks also go to the librarians and many others, who have in one way or other, make my completion of thesis possible.

TABLE OF CONTENTS

1. INTRODUCTION.....	1
1.1. Perspective of Artificial Neural Networks	1
1.1.1. Self-Organizing Map	2
1.1.2. Sequential Self-Organizing Map.....	3
1.1.3. Self-Organizing Map Using N-gram Method	4
2. LITERATURE REVIEW	5
2.1. Biological Neural Network	5
2.2. Artificial Neural Networks	6
2.3. Self-organizing Map	8
2.3.1. SOM Architecture	9
2.3.2. Input and Output Layers	10
2.3.3. Weights	11
2.3.4. The Competitive Process	11
2.3.5. Training.....	13
2.3.6. Kohonen Rule	14
2.3.7. Neighborhoods.....	14
2.3.8. Learning Rate.....	18
2.4. Neural Filtering.....	19
2.4.1. N-gram	20
2.4.2. Neural Filtering Model	20
2.5. Sequential Self-Organizing Maps.....	21
2.5.1. Architecture.....	22
3. METHODS	24
3.1. Introduction.....	24
3.1.1. An Application of N-gramSOM	25
3.1.2. N-gramSOM Architecture	25
3.1.3. The N-gramSOM Algorithm.....	27
4. IMPLEMENTATION AND TESTING.....	30
4.1. Introduction.....	30
4.2. Implementation	31
4.2.1. String Generator.....	31
4.2.2. N-gram Generator	34

4.2.3. N-gram Translator.....	35
4.2.4. N-gramSOM	37
4.2.5. Build FSA	38
4.2.6. Nondeterministic FSA to Deterministic FSA (NftoDf)	42
4.2.7. Minimizing FSA	45
4.2.8. Equivalence Check.....	49
4.2.9. Determining String Length	49
4.2.10. N-gramSOM System.....	52
4.3. Testing.....	54
4.3.1. Selection of FSA.....	54
4.3.2. Experimental Results	55
4.3.3. Failures from the Testing.....	57
4.3.4. Compare the N-gramSOM to SeqSOM	58
5. CONCLUSIONS AND FUTURE WORK.....	60
5.1. Conclusions.....	60
5.2 Future Work	61
REFERENCES.....	63
APPENDICES.....	66
APPENDIX A <i>GLOSSARY</i>	67
APPENDIX B <i>ACRONYMS</i>	69
APPENDIX C FINITE STATE AUTOMATA USED FOR EXPERIMENTING N-GRAMSOM.....	71
APPENDIX D <i>TRAINING SET</i>	83

LIST OF TABLES

Table 1. Calculate the size of the binary vector.....	27
Table 2. The size of training sets	55
Table 3. The final results	56
Table 4. The parameters that N-gramSOM used to learn a language.....	56

LIST OF FIGURES

Figure 1. Biological Neurons [Hagan96].....	6
Figure 2. An simple architecture of a feedforward ANN	8
Figure 3. SOM architecture [Caudill93]	10
Figure 4. The “Mexican-hat function” of lateral interaction [Kohonen89].....	15
Figure 5. Neighborhood function for rectangular grid [Fausett94]	17
Figure 6. “Bubble” neighborhood function [Hollmen96].....	17
Figure 7a. Learning with a fixed learning rate over time	19
Figure 7b. Learning with a decreasing learning rate over time	19
Figure 8. A window of size 3 scrolls over the input text “wind”.....	21
Figure 9. Illustration of the SeqSOM Architeture	23
Figure 10. Illustration of the N-gramSOM Architecture.	26
Figure 11. Algorithm of Kohonen’s SOM.....	28
Figure 12. Algorithm of N-gramSOM System.	29
Figure 13. Input formats for the String Generator.....	32
Figure 14. An example of input formats with real data.....	33
Figure 15. The graphical illustration of FSA1.....	33
Figure 16. Sample strings generated from FSA1.....	34
Figure 17. Sample of tri-gram.....	35
Figure 18. A lookup table.	36

Figure 19. Sample training set.	37
Figure 20. Sample map of N-gramSOM.....	38
Figure 21. Sample of network responses	40
Figure 22. An FSA generated from N-gramSOM.	40
Figure 23. A diagram that illustrates the FSA in Figure 22.....	41
Figure 24. An algorithm that convert NDFSA to DFSA [Linz96].	43
Figure 25. A DFSA generated from the algorithm in Figure 24.....	44
Figure 26. A diagram that illustrates the DFSA in Figure 25.....	45
Figure 27. The minimization algorithm by Martin.	47
Figure 28. A minimized DFSA.....	48
Figure 29. A diagram that illustrates the minimized DFSA in Figure 28.....	48
Figure 30. The definition of indistinguishable state.	50
Figure 31. A lemma to determine two states in an FSA, which are indistinguishable.	50
Figure 32. An algorithm to decide the equivalence of two FSAs.....	50
Figure 33. An union of two machines.....	51
Figure 34. Illustration of N-gramSOM System.	53

CHAPTER 1

INTRODUCTION

1.1 Perspective of Artificial Neural Networks

Artificial Intelligence (AI) is one of the emerging fields in computer science that has interested many scientists and researchers. It is about the study of machines that can understand and make judgements, in the way that humans do. Animal and human intelligence is the inspiration to the development of AI. These machines with replicated intelligence traits have been applied in many fields including aerospace, banking and finance, medical, manufacturing, and telecommunication. In addition, many researchers' keen interests have led further to the rapid growth and development of these machines. Artificial Neural Networks (ANNs) are a sub-field of AI and are designed to simulate the intelligence aspects of biological neural networks. Warren McCulloch and Walter Pitts first proposed ANNs that could compute any arithmetic or logical function in 1943 [McCulloch 43]. The first practical application of ANNs appeared in the late 1950's, with work done by Frank Rosenblatt on a device called the perceptron [Freeman92]. During the same period, Bernard Widrow and Ted Hoff [Widrow60] introduced another ANN application called Adaline [Freeman92]. Research in ANN dropped off between 1969 and the early 1980's as a result of a publication by Marvin Minsky and

Seymour Papert, who convinced others that ANNs were a dead end [Freeman92; Minsky69]. ANN research was revitalized during the mid-1980s when the backpropagation algorithm for training multi-layered networks became more widespread [Hagan96]. With many researchers concentrating on the backpropagation algorithm, Teuvo Kohonen pursued research in associative and topology preserving neural networks during the same period [Maren90]. The Self-organizing Map (SOM), developed by Kohonen, is a significant achievement in ANN research [Caudill93].

1.1.1 Self-Organizing Map

SOM is a winner-take-all, competition type of neural network. It does not require complicated mathematical calculations as compared to the widely accepted feedforward networks, which commonly use the backpropagation learning technique. In addition, SOM is an unsupervised learning network that also provides topological preservation mapping from higher dimensional space to one- or two-dimensional space [Hiotis93; Kohonen89].

The architecture of SOM is a two-layered network with an input-layer and an output-layer [Caudill93; Kohonen89]. The input-layer is a one-dimensional array of neurodes. However, the output-layer can be arranged as a two-dimensional array of neurodes. The neurodes in the output-layer are not interconnected. Each neurode in the output-layer is connected to each neurode in the input layer by a weight vector. The individual weights of connections between the input neurodes and the output neurodes can be described as a strength or capacity. During the training process, the weight

vectors of the winning neurode and its neighbors are updated over time to more closely match those in the input vector. The neurode with a weight vector that most closely matches to the input vector, is the winner. Once the training is complete, the SOM is able to classify new input data to the best-matching neurode.

1.1.2 Sequential Self-Organizing Map

SOM was designed to process only fixed lengths of input vectors. It cannot effectively process a collection of input vectors of various lengths, nor recognize the relationships within the input data. Therefore, a neural network referred to as the Sequential Self-Organizing Map (SeqSOM) was proposed by Boydston and Mayfield in 1995 [Boydston95]. The SeqSOM can deal effectively with data of variable input lengths. The SeqSOM uses a feedback method to build relationships between subsequent input vectors. SeqSOM has been trained and tested using strings from languages accepted by Finite State Automata (FSA). The network was shown to be able to both learn and capture the relationships of the strings. After the network was trained, a new FSA was created, which could recognize the strings from the language accepted by the original FSA.

1.1.3 Self-Organizing Map using N-gram Method

A method called “n-gram” also has been used to deal with data of variable input lengths, which can be either characters or words. The n-gram method uses a fixed length of scrolling, overlapping windows on the input text to produce a sequence of input vectors [Scholtes92]. An example of the window is shown in Figure 8 in CHAPTER 2. The derived input vectors can then be used to train a neural network.

The study of the SOM, SeqSOM, and n-gram methods leads to the possibility of applying the n-gram method to SOM. Thereafter, the network is able to both learn and capture the relationship between a sequence of input strings and produce an FSA equivalent to the original FSA that produced the strings. The network, named N-gramSOM, is used initially as a neural filter for contextual data by Scholtes [Scholtes92]. After modifying the original N-gramSOM training method to use the SeqSOM training method, N-gramSOM is able to construct an FSA that recognizes the language of the input contextual data and that performs better than SeqSOM.

CHAPTER 2

LITERATURE REVIEW

2.1 Biological Neural Network

The development of Artificial Neural Networks (ANNs) was first inspired by the characteristics of brain function and their relation to biological counterparts in the brain. In the human brain, there are large numbers of elements called neurons that are highly interconnected so as to facilitate thinking, movement, and autonomic responses, etc. These neurons have four main components: dendrites, a cell body, an axon, and synapses, which are shown in Figure 1. Each of these components has their specific function. The dendrites are tree-like receptive networks of nerve fibers that receive electro-chemical stimuli from other neurons, and then send a signal into the cell body [Anderson95]. The cell body performs a processing function that is characterized as summing up all the dendrite input signals; if the sum exceeds a threshold, a signal will be sent out via the axon, a single long fiber, to other neurons. A small gap known as a synapse exists between the contact point of the axon of one cell and a dendrite of another cell. The arrangement of the neurons and the strength of any individual synapse establish the

function of the neural network [Hagan96]. The study of biological neural networks has set a foundation for the development of ANNs.

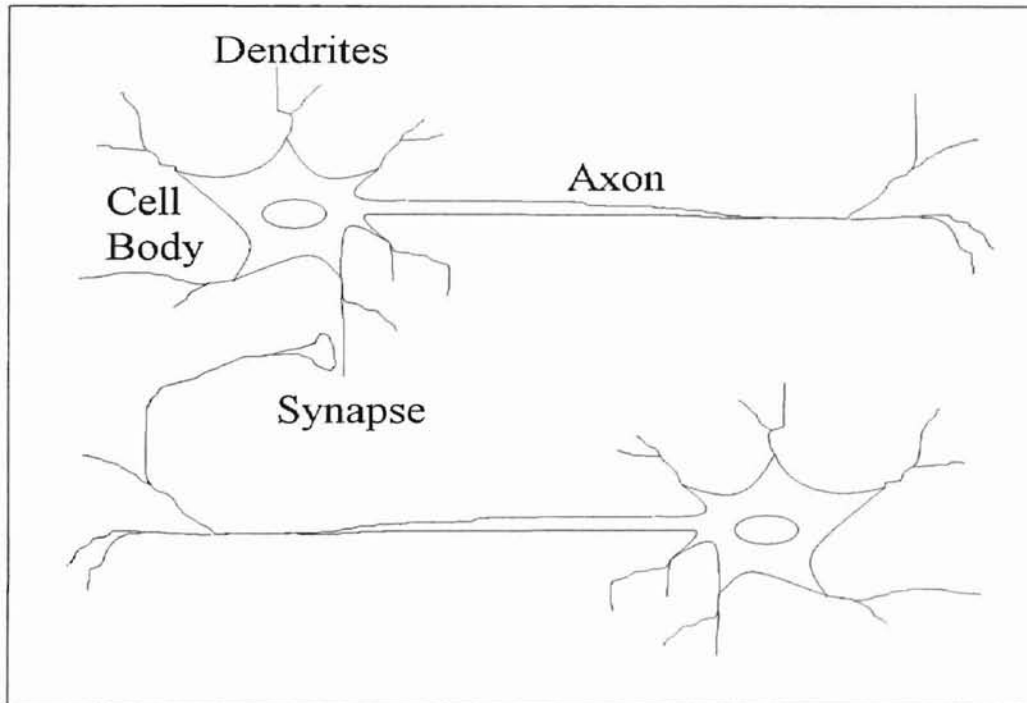


Figure 1. An example of two biological neurons [Hagan96]

2.2 Artificial Neural Networks

An ANN is composed of a group of artificial neurons that are similar in function to the neurons found in biological neural networks even though an ANN does not approach the complexity of the brain. The structure of ANNs is similar to biological

neural networks only in terms of their simple computational building block and connection of neurons [Hagan96]. An artificial neuron also is called a “processing element” [Ritter91] or a “neurode [Kohonen89].”

The ANN architecture is comprised of neurodes that are interconnected to one another and are usually arranged into layers of neurodes. An example of an ANN architecture, referred to as a feedforward ANN, is illustrated in Figure 2. It consists of three or more layers: an input-layer, one or more hidden-layers, and an output-layer. The function of the input-layer is to distribute the input signals to each of the neurodes in the hidden-layer. The hidden-layer is used to calculate activation values, which are forwarded to the output-layer. The activation values of neurodes in the output-layer are the network’s response to a given input signal. An error value also is computed by comparing the activation values of the output-layer with its target values. A target value is the correct output of the network expected from the input signal. Thereafter, the error values will be distributed back to the hidden-layer for updating the internal weights, which serves to draw closer resemblance to the target values.

The feedforward ANN is just one of many types of ANNs. The main ANN for this thesis is the Self-Organizing Map (SOM).

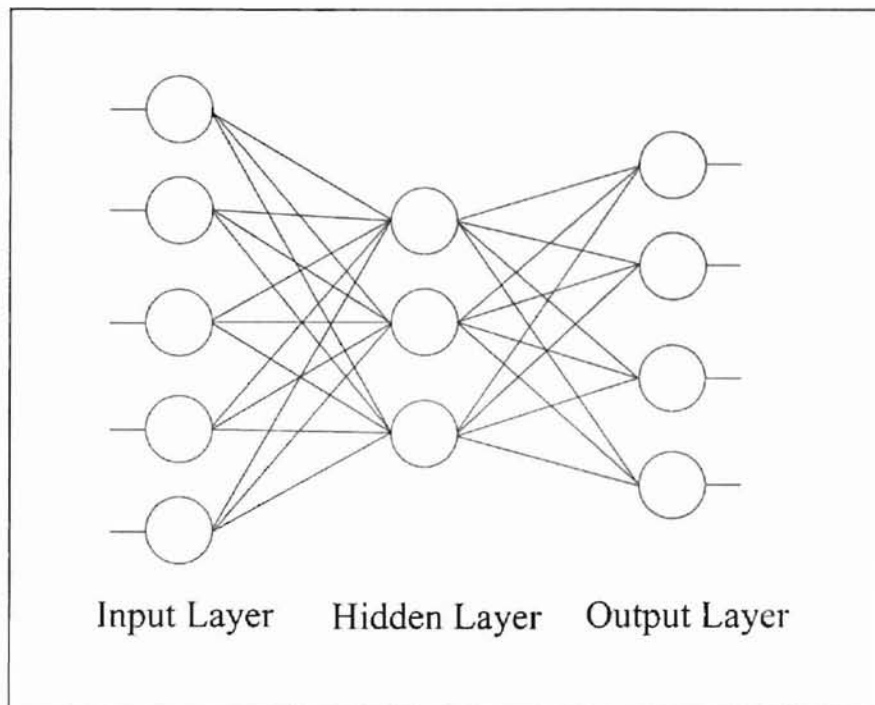


Figure 2. A simple structure of a feedforward ANN.

2.3 Self-Organizing Map

One type of ANNs, called the Self-Organizing map (SOM), was proposed by Kohonen in the early 1980's [Kohonen89]. SOM is a competitive neural network based on the idea of competition and neighborhood updates. The concept of competition and neighborhoods will be explained in the 2.3.4 and 2.3.7. SOM is an unsupervised learning network that does not need human assistance during its learning period

SOM also is a "topology-preserving map" [Fausett94]. Topology is the mathematical study of the properties of objects that are preserved through deformations,

twisting, and stretching [Weiss98A]. The various maps formed in SOMs are able to describe topological relations of input signals using a one- or two-dimensional medium for representation [Kohonen89]. Since the output layer of an SOM usually is a planar array, it can preserve topological relationships while performing dimensionality reduction from n dimensions into a one- or two-dimensional representation space [Hiotis93; Kohonen89].

2.3.1 SOM Architecture

The basic architecture of the SOM, shown in Figure 3, is a two-layered network with an input-layer and an output-layer [Caudill93; Kohonen89]. The input-layer is a one-dimensional array of neurodes. However, the output-layer normally is arranged as a two-dimensional array of neurodes. The neurodes in the output-layer are not interconnected [Kohonen89]. Each neurode in the output-layer is fully connected to the neurodes in the input-layer. Like the input-layer of the feedforward network described in section 2.2, the SOM input-layer distributes its input signal to each of the neurodes in the output-layer across a set of adaptive, weighted connections. The neurodes of the output-layer then make up a competitive assembly [Caudill93]. During the self-organizing process, the neurode whose weight vector matches an input vector most closely is chosen as the winner for that input vector. The selection of a winner will be explained in section 2.3.4. The winning neurode and its neighboring neurodes update their weights by using the Kohonen rule [Fausett94; Hagan96]. The Kohonen rule will be explained in section

2.3.6. By updating the weights (w_{ij}) at each pass, the winning neurode and its neighboring neurodes will be made to match the input data more closely.

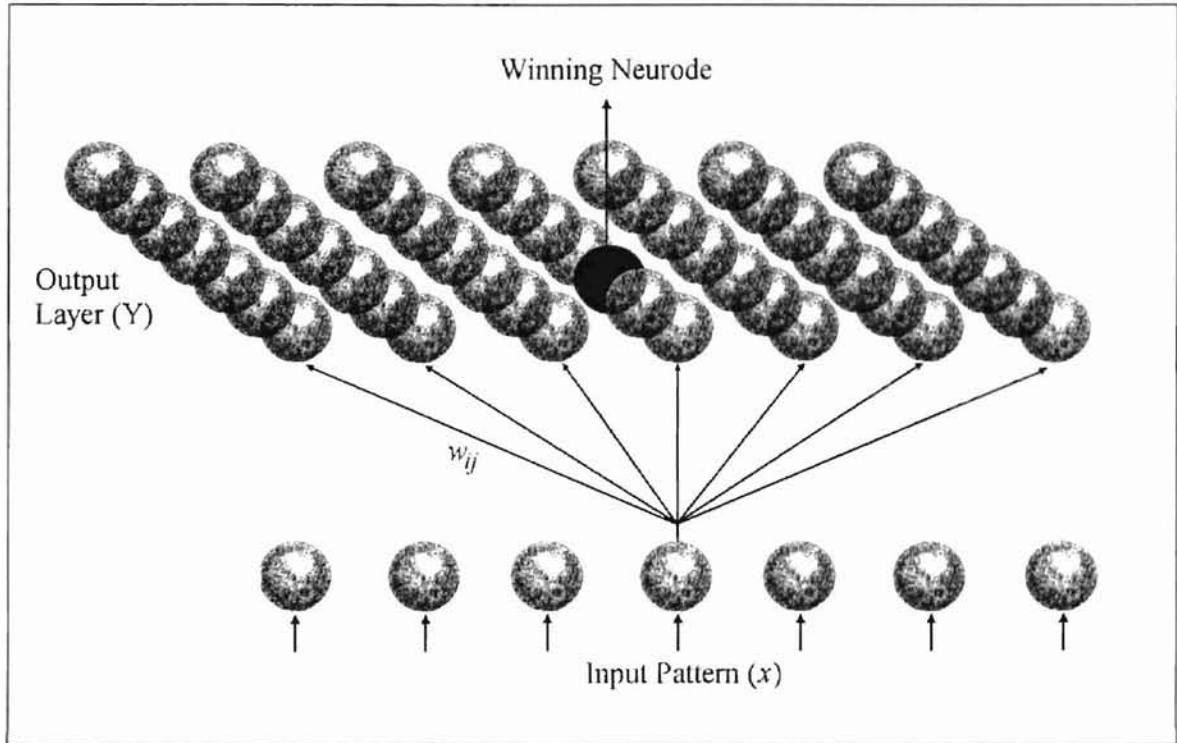


Figure 3. SOM architecture [Caudill93].

2.3.2 Input and Output Layers

The input-layer is fully connected to the output-layer and it delivers its signal to each neurode in the output-layer [Hiotis93]. An input signal is a vector such as:

$$\mathbf{x} = (x_1, x_2, \dots, x_n) \quad (1)$$

The output-layer (Y) is a two-dimensional array consisting of m neurodes, connected to the input-layer by a weight vector that is illustrated in Figure 3 such as:

$$\mathbf{w}_j = (w_{1j}, w_{2j}, \dots, w_{nj}) \quad (2)$$

where $1 \leq j \leq m$ and n is the length of \mathbf{x} . Note that the weight vector is the same size as the input vector. w_{ij} refers to the incoming weight from input-layer neurode x_i to the output-layer neurode Y_j .

2.3.3 Weights

The individual weights of the connections between the input neurodes and the output neurodes can be described as a strength or capacity. They are similar to the synapses in the biological network as shown in Figure 1. The weights are used to compute the response of a neurode to a given input signal. During training, the weights may be changed based on a training function.

2.3.4 The Competitive Process

In SOM, the neurodes of the output layer are competing with one another to represent each input vector. The winning neurode is chosen based on the minimal distance between the weight vector of the neurode and the input vector. The weight

vector of the winning neurode is then adjusted to bring the weight vector closer to the input vector. There are two methods to find the winning neurode.

The first method uses the dot product (I_j) between the input vector and weight vector, as expressed below:

$$I_j = \sum_{i=1}^n w_{ij} x_i \quad (3)$$

where $w_{ij} \in$ the weight vector \mathbf{w}_j for neurode j , and $x_i \in$ the input vector \mathbf{x} [Caudill93].

The neurode that has the largest value of I_j is the winning neurode. Therefore, the winning neurode is the one that has the largest dot product, which implies that the angle between the input vector and the winner's weight vector is smaller than that of any other neurode's weight vector.

The second method uses the square of the Euclidean distance (D_j) between the input vector and the weight vector, as expressed below:

$$D_j = \sum_{i=1}^n (w_{ij} - x_i)^2 \quad (4)$$

where $w_{ij} \in$ the weight vector \mathbf{w}_j for neurode j , and $x_i \in$ the input vector \mathbf{x} [Fausett94]. D_j is the sum of the square of the differences between the weight vector and the input vector.

The neurode with the smallest value of the D_j is the winning neurode. The Euclidean distance is used to calculate the distance between the input vector and a weight vector. Therefore, the smaller the value of D_j , the closer the distance between the input vector and the weight vector.

After determining the winning neurode by either of the above methods, the weight vectors for all neurodes within a specified neighborhood (i.e., radius) of the winning neurode are updated using the Kohonen rule [Kohonen89]. The weight-updating process is called training.

2.3.5 Training

A collection of input vectors that is used to train an ANN is called the training set. Each member of the training set is presented to the ANN during the training process; as a result, the weights within the network may be updated using the learning rules. An interval during which each member of the training set is presented to the network once is called an epoch. Therefore, the network can be trained for many epochs with the training set. There are two types of methods used to train an ANN: supervised methods and unsupervised methods. In a supervised method, the training sets and the target values are known ahead of time. During the training process, the learning algorithm uses the prior knowledge about the target values to adjust the weights to more closely map the input vectors to the target values.

The difference between unsupervised training and supervised training is that there is no prior knowledge of the output vector for an unsupervised method. In other words, when a training set is presented to the network, no target values will be known ahead of time. SOM is an ANN that uses unsupervised training.

Initially, a training set is chosen and the weight vectors are initialized with random values. Weight vectors may also be initialized based on prior knowledge [Boydston95]. In addition, the value of the neighborhood size and learning rate must also be initialized (the learning rate is a parameter that controls the weight adjustments.) During the training process, the neighborhood (see section 2.3.7) and the learning rate (see section 2.3.8) size are reduced based on elapsed time τ . The purpose of reducing the learning rate and the neighborhood size over time is to fine-tune the map.

2.3.6 Kohonen Rule

The weights of the winning neurode and its neighbors are updated using the Kohonen rule [Fausett94],

$$w_{ij}(new) = w_{ij}(old) + \alpha [x_i - w_{ij}(old)] \quad (5)$$

where $w_{ij}(new)$ is the new weight vector, $w_{ij}(old)$ is the previous weight vector, x_i is the current input vector, and α is the current learning rate.

2.3.7 Neighborhoods

The neurons of the brain are densely interconnected and each neuron can have thousand of lateral interconnections with neighboring neurons [Kohonen89]. Kohonen

says that there are both anatomical and physiological evidences from the mammalian brains to suggest that the degree of lateral interaction are related to the distance at which the excitation occurs [Kohonen89]. Neurons that are closest to the active cells have more positive lateral feedback center of excitation. A region of negative lateral feedback is formed after the positive lateral feedback is diminished. A minimal positive lateral feedback will be formed after the negative lateral feedback region that is farther from the center of the excitation [Kohonen89]. The degree of lateral interaction is usually described as having the form of a Mexican hat [Kohonen89]. The “Mexican-hat function” is illustrated in Figure 4.

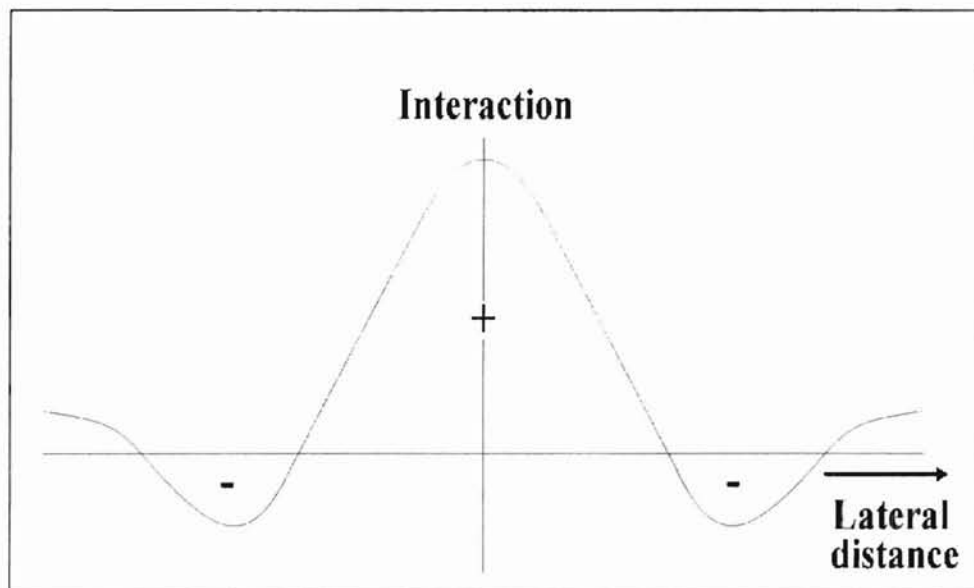


Figure 4. The “Mexican-hat function” of lateral interaction [Kohonen89].

The SOM model does not directly implement lateral feedback. As the neurodes in SOM’s output-layer are not interconnected, there is no lateral connection. Instead, a

neighborhood is defined that includes all the neurodes within a given radius of a winning neurode. An example of a neighborhood is illustrated in Figure 5, where the neighborhood represents the lateral distance between the neurodes [Kohonen89]. Only the weight vectors of the neurodes within the neighborhood can be updated [Hagan96; Kohonen89]. This not only corresponds to the concept of the center of positive lateral feedback of the “Mexican-hat function”, but also simplifies the “Mexican-hat function” to a “bubble” neighborhood function. The bubble neighborhood function is illustrated in Figure 6. The bubble neighborhood function is a constant function within the defined neighborhood of the winning neurode; that is, weight vector of each neurode in the neighborhood is updated with the same proportion of the difference between its weight vector and the input vector [Hollmen96]. The neighborhood radius is decreased gradually during the training process. The neighborhood radius can be reduced using a linearly decreasing function such as:

$$\eta_{new} = \left[\eta_{old} \left(1 - \frac{\tau}{\tau_{max}} \right) \right] \quad (6)$$

where η is the neighborhood radius, τ is elapsed training time, and τ_{max} is the maximum allowable training time [Boydston97; Fausett94; Kohonen89]. The purpose of decreasing the neighborhood radius during the training process is to sharpen the response of the neurodes within the neighborhoods and form clusters. If the neighborhood radius is zero, it means that the neighborhood only contains the winning neurode, thus it will not form

any cluster. A cluster is a group of neurodes that are adjacent to one another and that match similar input vectors.

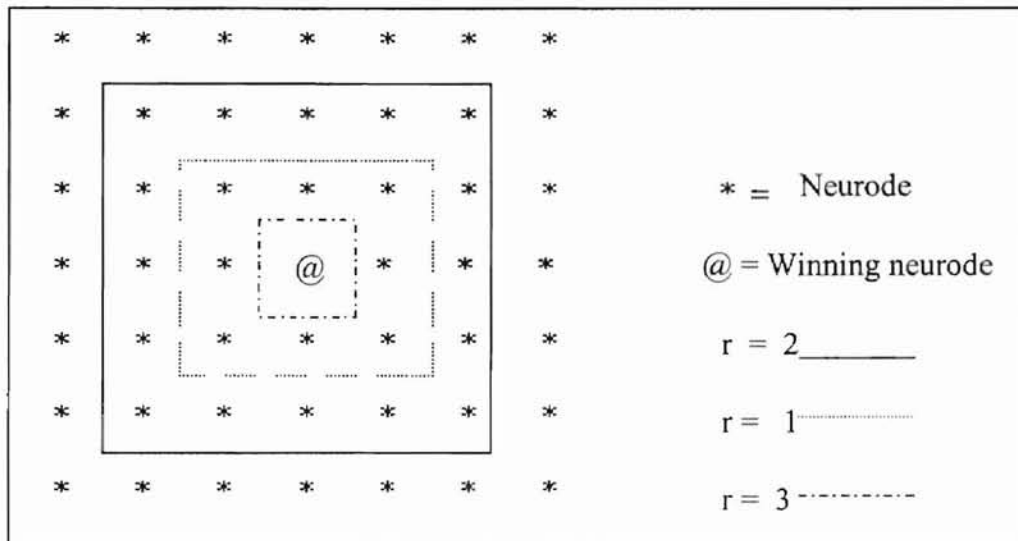


Figure 5. Neighborhood function for rectangular grid [Fausett94].

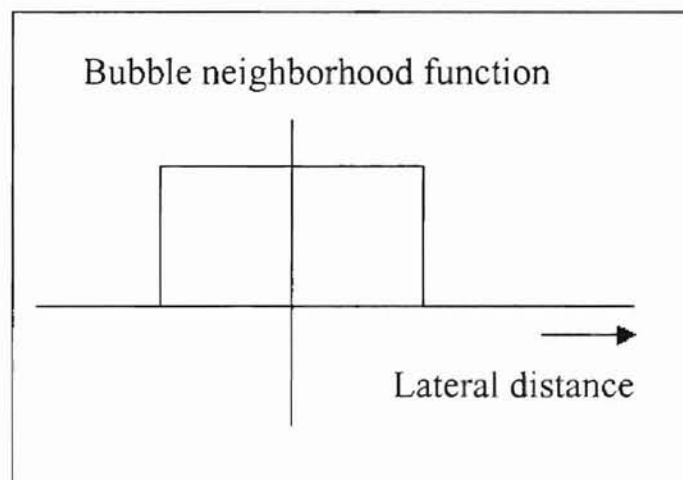


Figure 6. "Bubble" neighborhood function [Hollmen96].

2.3.8 Learning Rate

The learning rate controls how much the weight vectors can change in one pass and is a slowly decreasing function of time during the training process [Fausett94]. Kohonen indicates that both a linearly and geometrically decreasing function of time for the learning rate will produce similar results [Fausett94; Kohonen89]. The following linearly decreasing function is used in this research:

$$\alpha_{new} = \alpha_{old} \left(1 - \frac{\tau}{\tau_{max}} \right) \quad (7)$$

where the α_{new} is the new learning rate, α_{old} is the previous learning rate, τ is the elapsed time, and τ_{max} is the maximum allowable training time [Boydston97; Fausett94; Hollmen96; Kohonen89].

Figure 7a shows that the weight vector could oscillate between points “a” and “b” but can never get any closer to the input vector with a constant learning rate. However, Figure 7b shows that by reducing the learning rate over a period of time, the values in the weight vector will gradually adjust themselves closer to the input vector from points “a” to “b,” “b” to “c,” and then “c” to “d.” The result of reducing the learning rate and the neighborhood size during the training process over a period of time will yield the formation of clusters [Kohonen89].

Learning is usually performed in two phases. In the first phase, a relatively large initial learning rate is used ($\alpha=0.3, \dots, 0.99$), whereas in the second phase a smaller initial

learning rate is used ($\alpha=0.01, \dots, 0.1$) [Hollmen96]. The first phase is used to create an initial formation of the correct order [Fausett94]. The second phase is used to fine-tune the map and yield the final convergence [Fausett94; Hollmen96].

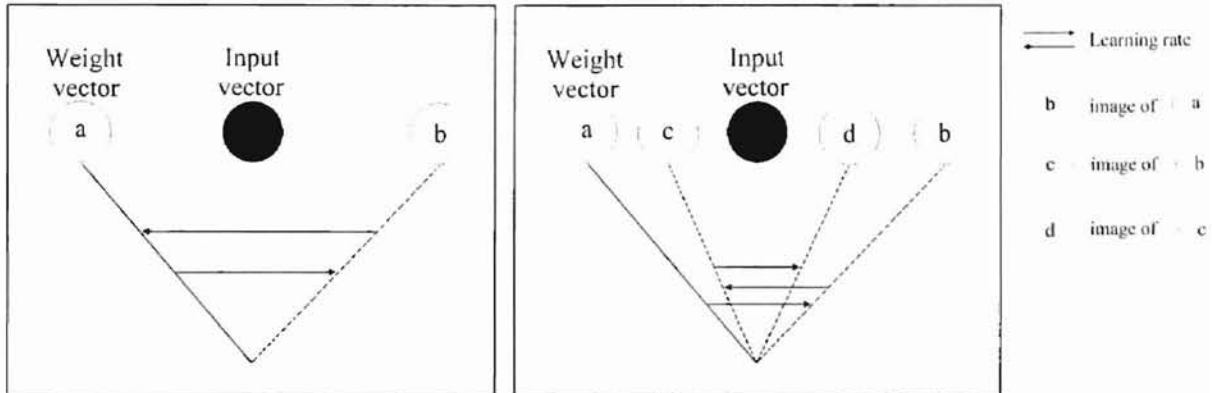


Figure 7a. Learning with a fixed learning rate.

Figure 7b. Learning with a decreasing learning rate over time.

2.4 Neural Filtering

The Neural Filtering method implemented by Scholtes is used for free-text information filtering [Scholtes92]. Free-text is a keyword that is used to retrieve any related information from databases [Lebanon95]. A neural filtering model that combines the n-gram method with the SOM is proposed by Scholtes. The proposed model is able to act as a neural filter to retrieve associated subjects from the dynamic free-text database [Scholtes92].

2.4.1 N-Gram

An n-gram is a vector that contains a sequence of characters from a word or token; where each n-gram must contain at least one non-blank character, and at most n characters. An example of the 3-grams or tri-grams of a word, “windows” is given below to illustrate the meaning of n-gram:

--w, -wi, win, ind, ndo, dow, ows, ws-, s--,

where the “-” represents a space [Scholtes92].

2.4.2 Neural Filtering Model

The neural filtering model consists of two parts: the preprocessing section and the SOM. The preprocessing section is used to prepare input vectors using the n-gram method. A “window” of size 3 is used to represent the 3-gram or tri-gram shown in Figure 8. The window will scroll over the elements in the input text and a program will translate them into input vectors with the assistance of a lookup table. An element of the window can be either a character or a word. All unique elements in the input text are assigned randomly to specific codes in the lookup table. Each input vector to the SOM holds exactly one n-gram.

The architecture of the SOM in the second part is the same as the original SOM architecture, which consists of an input-layer and an output-layer. Each neurode in the

output-layer competes to represent an n-gram. SOM will form clusters on the map for the most frequent n-grams occurring in the input text. However, the less frequent n-grams will be overridden when the number of neurons in the output-layer is less than the number of the n-grams generated in the input text [Scholtes92].

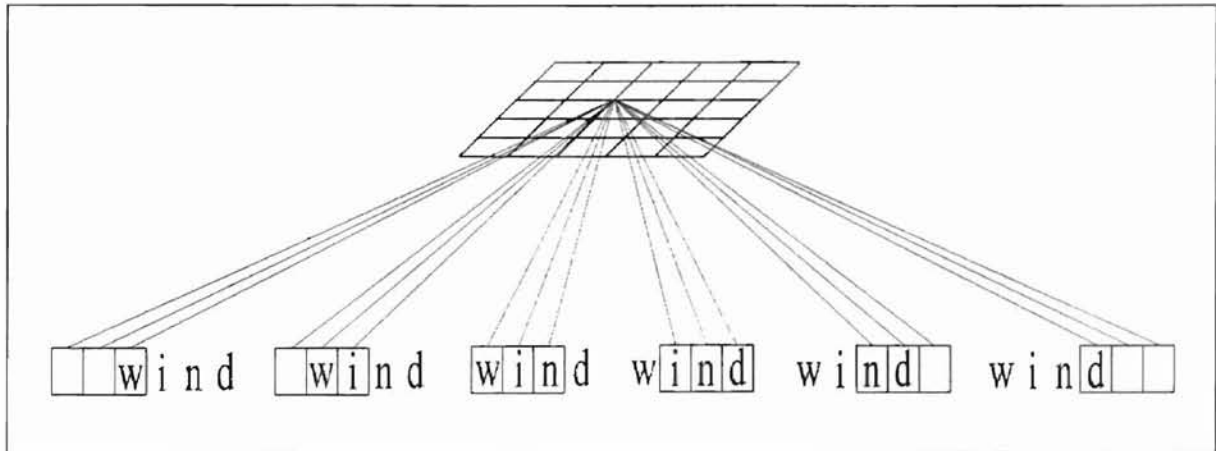


Figure 8. A window of size 3 scrolls over the input text "wind" [Scholtes92].

2.5 Sequential Self-Organizing Maps

SOM is not designed to process sequential input vectors and is not able to recognize the relationships between the sequential input vectors. Therefore, the Sequential Self-Organizing Map (SeqSOM) was proposed by Boydston and Mayfield, which uses a feedback method to build the relationship within the sequence of the input vectors [Boydston95]. A set of strings from a language accepted by an FSA is used to train the network. The network is able to learn and capture the relationship of the strings

and produce a new representation of the FSA. The new FSA is able to accept any strings from a language accepted by the original FSA.

2.5.1 SeqSOM Architecture

The architecture of the SeqSOM is illustrated in Figure 9, which basically is the same as the original SOM architecture, consisting of an input-layer and an output-layer.

The difference between the SeqSOM and the original SOM architecture is that the SeqSOM uses a feedback method. Unlike the input vector in original SOM, the input vector in the SeqSOM is an “input bundle”. The input bundle is a concatenation of an input vector from the input vector sequence with three feedback coordinate values, which consists of a row, column, and plane (as illustrates in Figure 9). The first input bundle is different from others because it contains no feedback values. When an input bundle is distributed to each neurode in the output-layer, a winning neurode will be chosen. The output-layer coordinates of the winning neurode are used as the feedback value and are concatenated with the next input vector to form a new input bundle. The new input bundle is used as the next input to the network. This procedure will continue until all the input vectors from the input vector sequence are consumed [Boydston97]. Three feedback values must be added to the weight vector of each neurode to accommodate the feedback values.

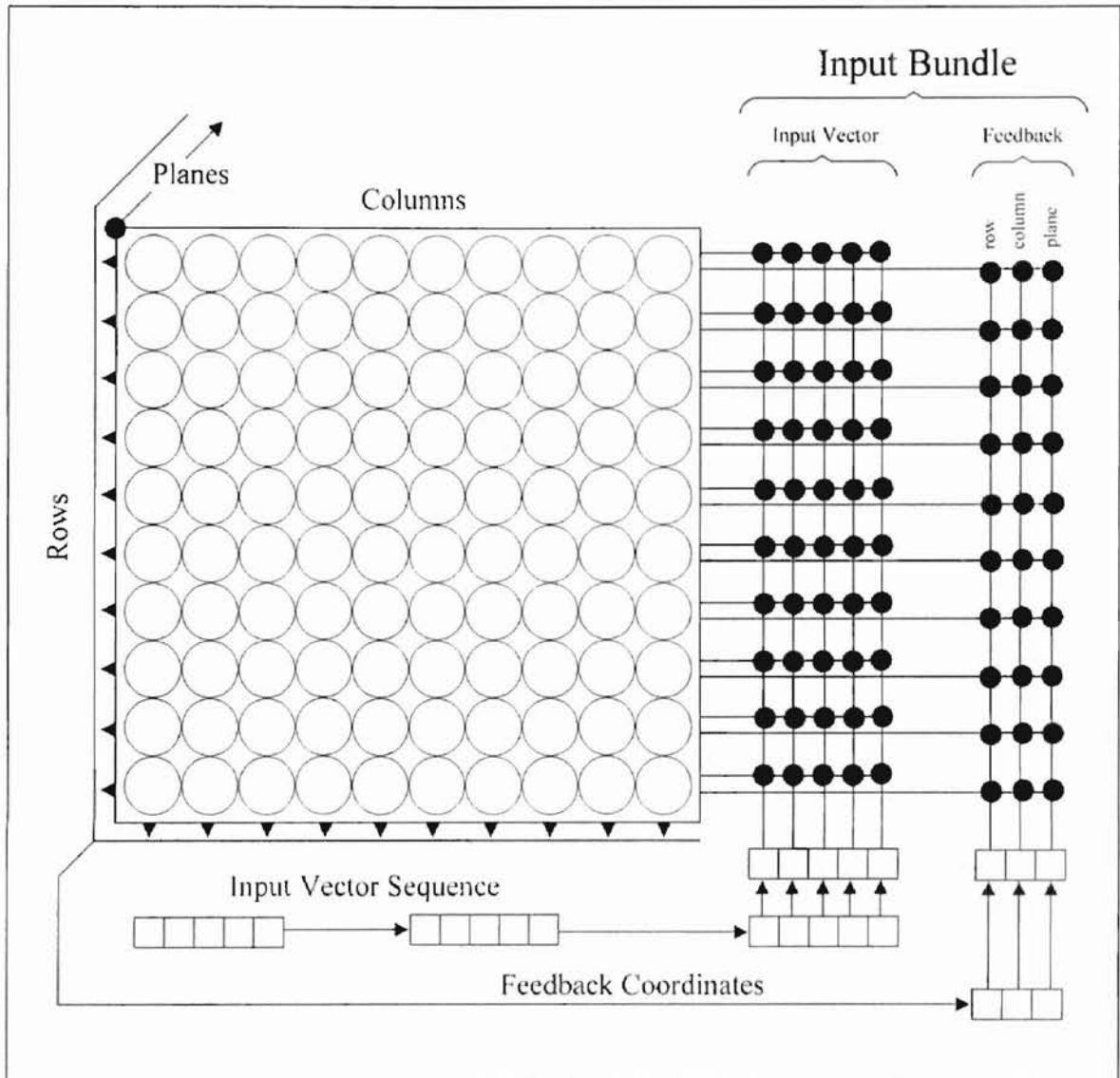


Figure 9. Illustration of the SeqSOM Architecture [Boystun95].

CHAPTER 3

METHODS

3.1 Introduction

The Kohonen SOM is not designed to process input vectors of varying sizes. It also not able to capture the sequential relationships between a sequence of input vectors. To overcome the problem, a network called SeqSOM designed by Boydston is able to process input vectors with varying sizes [Boydston95; Boydston97]. The SeqSOM breaks an input vector into a sequence of fixed size new input vectors and feed this sequence of input vectors into the network sequentially. SeqSOM uses a feedback method to relate the input vectors in a sequence to one another. As a result, the network is able to capture the sequential relationships between the input vectors in a sequence. The architecture of SeqSOM is illustrated in CHAPTER 2.

Another model is proposed by Scholtes using Kohonen SOM to deal with the contextual data [Scholtes92]. Scholtes uses the n-gram method to generate a sequence of input vectors from a strings and feed the n-grams into the network sequentially. His research shows that the network is able to capture the most frequent occurrence of n-grams. This model is used as a neural filter to retrieve associated subjects from the dynamic free-text database. Scholtes did not give a name to this model; nevertheless, in

this thesis, it will be referred to as N-gramSOM. This model is described in detail
CHAPTER 2.

3.1.1 An Application of N-gramSOM

With the study of the SeqSOM and the N-gramSOM methods, there is a possibility that the N-gramSOM method may be able to capture the relationships between a sequence of input vectors, as the SeqSOM does. For this research purpose, a sequence of n-grams will be generated from a language that is accepted by an FSA. The sequence of n-grams will be translated to a sequence of input vectors using a lookup table and feed the input vectors into the network sequentially. The network will capture the relationships between the sequence of input vectors and produce an equivalent FSA compared to the original FSA.

3.1.2 N-gramSOM Architecture

The architecture of the N-gramSOM, as illustrated in Figure 10, is the same as the original SOM that consists of an input-layer and a two-dimensional output-layer. When a winning neurode is chosen, the weight vectors of the winning neurode and its neighborhood are updated with the Kohonen Rule [Kohonen89].

In addition, the N-gramSOM also shares the same underlying architecture as the SeqSOM and both use a sequential input method. A difference between these two architectures is that the N-gramSOM does not use a feedback method.

The size of an input vector that represents an n-gram is n times larger than the length of a binary representation of a symbol. For example, if a 3-gram or tri-gram is used to generate a sequence of input vectors and the length of the binary representation of a symbol is four digits, then the size of the input vector will be twelve digits. The example is explained in more detail in Table 1.

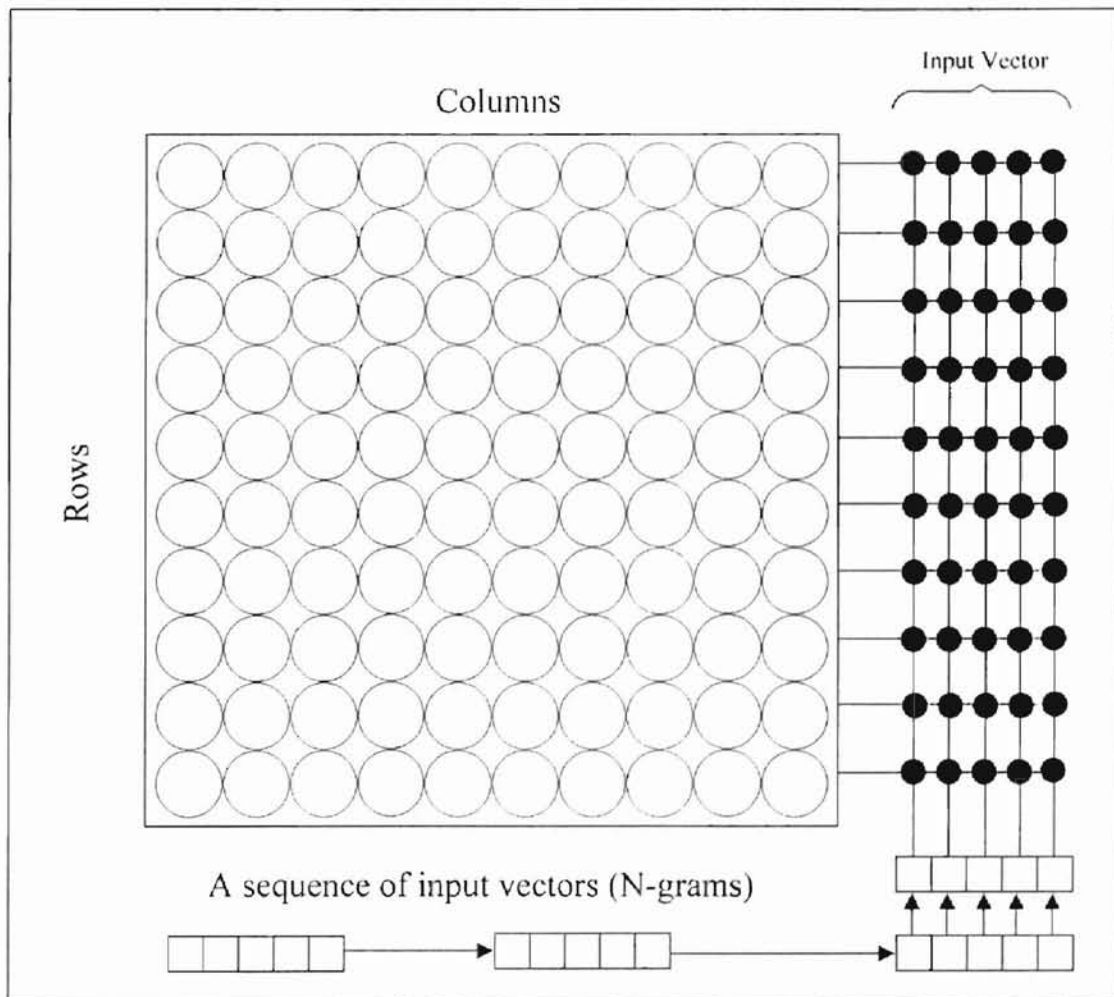


Figure 10. Illustration of the N-gramSOM Architecture. This architecture is a modification of Figure 9.

	Representation	Size
3-gram or Tri-gram	abc	3
Binary representation	a – 0000 b – 0010 c – 0100	4
Input Vector	000000100100	4 x 3 =12

Table 1. The size of the input vector is calculated by multiplying the size of the tri-gram with the size of the binary representation.

3.1.3 The N-gramSOM Algorithm

The algorithm for the SOM and the N-gramSOM are mostly identical except for the training method. The N-gramSOM algorithm serves as an extension of the SOM algorithm. The algorithms are shown in Figures 11 and 12. The training method for Kohonen's SOM selects input vectors in a random order to feed into the network. On the other hand, the training method for the N-gramSOM feeds input vectors from a sequence into the network sequentially to maintain the order and relationship of the n-grams. The differences of these two algorithms are printed in bold in Figure 11 and Figure 12.

```

BEGIN
  Initialize all neurode's weight vectors to random values;
  Set neighborhood radius;
  Set learning rate;
  While stopping condition is false
  Begin
    For all input vectors in the training set(each vector is picked once and
in random order)
      Begin
        Calculate the square of Euclidean distance for each neurode to
the input vector;
        Find the neurode that has minimum Euclidean distance as the
winning neurode;
        Update the weights of the winning neurode and its neighbors
using the Kohonen rule;
      End;
      Update learning rate at a specified time;
      Update neighborhood radius at a specified time;
    End;
  End;
END.

```

Figure 11. Algorithm of Kohonen's SOM [Fausett94]

```

BEGIN
  Initialize all neurode's weight vectors to random values;
  Set neighborhood radius;
  Set learning rate;
  While stopping condition is false;
  Begin
    For all input sequences in the training set
    Begin
      For all input vectors in the sequence(all input vectors are picked
      sequentially)
      Begin
        Calculate the square of Euclidean distance for each neurode to
        the input vector;
        Find the neurode that has minimum Euclidean distance as the
        winning neurode;
        Update the weights of the winning neurode and its neighbors
        using the Kohonen rule;
      End;
    End;
  End;
  Update learning rate at a specified time;
  Update neighborhood radius at a specified time;
End;
END.

```

Figure 12. Algorithm of the N-gramSOM

CHAPTER 4

IMPLEMENTATION AND TESTING

4.1 Introduction

In the early stages of N-gramSOM software development for this research, Kohonen's SOM algorithm was implemented using C language to understand in more detail how the network works; it was tested on the UNIX operating system. In the later stages, a visual aid tool was needed to validate the network learning activities and their final convergence. Without a visual aid tool, the validation process would have been more difficult. Therefore, the network was implemented again by using Microsoft Visual Basic 4.0, and tested in the Microsoft Windows 95 platform. However, the execution of the network using Microsoft Visual Basic 4.0 was very slow. As a result, the network was implemented again using Microsoft Visual C++ 5.0 and was tested again on the Microsoft Windows 95 platform. The reasons for choosing Microsoft Visual C++ 5.0 are that it executes much faster than Microsoft Visual Basic 4.0 and that it also provides visual aid. After the Kohonen's SOM was validated, the algorithm was used to implement put into use by N-gramSOM. After the N-gramSOM algorithm was validated, visual aid was no longer needed. Therefore, the software then was moved to a SUN server to do testing, since the SUN server has multiple CPUs that are faster than the CPU

of PC, and it is able to handle multiple tasks much better. As a result, several tests could be run at the same time, saving a lot of time on testing.

4.2 Implementation

For the purpose of this research, a set of tools was programmed using C++ language. The tools consist of eight process stages: String Generator, N-gram Generator, N-gram Translator, N-gramSOM, Build FSA, Determinize FSA, Minimize FSA, and Check for Equivalent FSA. Each of these tools is designed to work independently. Therefore, they can be used to do independent testing for every stage or to build a system to do all the testing automatically. The effort to program these tools also may benefit future research. Each of the tools will be explained and discussed respectively in the following sections. An N-gramSOM system is built using the tools mentioned above and also will be explained in the last section of this chapter.

4.2.1 String Generator

For the purpose of this research, a set of strings from the language accepted by an FSA is chosen to train the N-gramSOM network. Therefore, String Generator is programmed to prepare strings for the network.

String Generator needs three parameters: an input file, an output file and the maximum length of the strings to be generated. The input file contains a given FSA in

specified format, illustrated in Figure 13. Figure 14 shows the input file representing the FSA illustrated in Figure 15. The output file contains the strings generated by String Generator. The String Generator will generate all strings from the given FSA with the maximum length specified by the third parameter. When the first output file is created by String Generator, it contains a lot of duplicate strings. Duplicate strings are unnecessary for training the network; in addition, these strings increase the network training time. Therefore, all duplicate strings should be removed. A Mergesort program by Kruse, Leung and Tondo is used to sort the strings [KruseLT91]. The worst-case analysis running time for the Mergesort program is $O(n \log n)$ [KruseLT91]. Kruse, Leung and Tondo define this $O(n \log n)$ as the Mergesort algorithm does no more than $n \log n$ basic operations and the size of its input is n [KruseLT91]. The Mergesort algorithm is one of the most efficient sorting algorithms and is easy to implement.

After the file is sorted, all the duplicate strings are grouped together and are removed easily by a simple program. As a result, the size of the file is greatly reduced. An example of the strings generated by the String Generator using the FSA from Figures 14 and 15 is illustrated in Figure 16 (with duplicate strings removed.)

```
# the total number of final state(s)
start_state final_state(s)
&
start_state state2 output_symbol
...
state1 state2 output_symbol
...
...
$
```

Figure 13. The input formats for the String Generator. The symbols “#” means the beginning of the input format, “&” means the beginning of the state transition format, and “\$” means the end of both the input and the state transition format.


```

# 1
0 5
&
0 1 t
0 2 p
1 1 s
1 3 x
2 2 t
2 4 v
3 2 x
3 5 s
4 3 p
4 5 v
$

```

Figure 14. An example of the input formats with real data. The diagram of this FSA is illustrated in Figure 15.

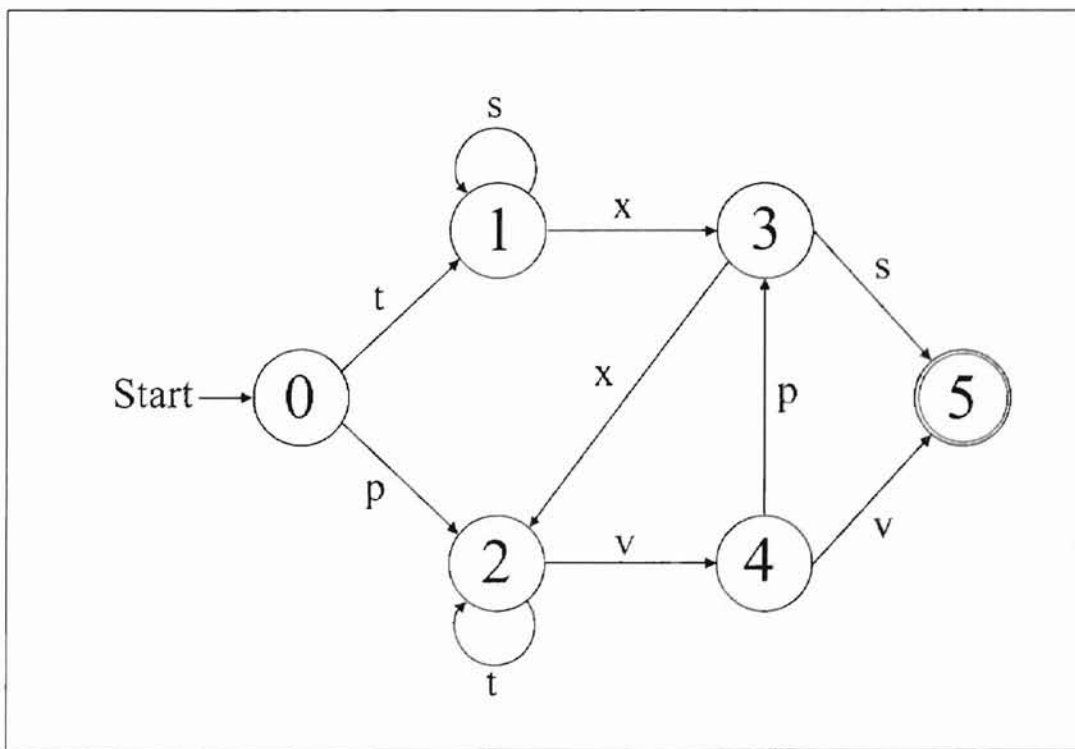


Figure 15. The graphical illustration of the FSA of Figure 14. Each rim with a number represents a state. The “Start” followed with an arrow indicates that the state 0 is the start state. The double rim in state 5 indicates that the state is the final state. The arrows represent the transitions from one state to another state.

pttttttv	Pttvpxtvps
ptttttvps	pttvpxtvv
ptttttvv	pttvpxvps
ptttttvps	pttvpxvv
ptttttv	pttvv
pttttvps	ptvps
pttttvpxvv	ptvpxtttv
pttttv	ptvpxtvps
ptttvps	ptvpxttv
ptttvpxtvv	ptvpxtvps
ptttvpxvps	ptvpxtvv
ptttvpxvv	ptvpxvps
ptttv	ptvpxvpxvv
pttvps	ptvpxvv
pttvpxttvv	ptvv

Figure 16. A sample of the string file generated from the FSA shown in Figure 15 with string length of 10 or less. The complete set of the 103 strings is included in APPENDIX B.

4.2.2 N-gram Generator

The n-gram is explained in CHAPTER II. After the String Generator prepares a string file, all strings in the string file must be translated to n-grams. Therefore, a program named the N-gram Generator was written to generate n-grams from a file containing strings. All generated n-grams are saved to another file. N-grams are generated based on the method discussed in CHAPTER II and illustrated in Figure 8.

The N-gram Generator needs three parameters: an input file, an output file and the size of the n-gram. The input file contains a set of strings generated from String Generator. The output file contains n-grams with the predetermined n-gram size. The n-

grams that contain spaces before the first character occurs, as discussed in CHAPTER 2, will not be used in this research. The N-gram Generator therefore eliminates that type of n-grams. Two extra symbols, a “B” and an “E”, are added to the n-grams file and indicate the beginning and the ending of a word, respectively. These two symbols will be used to represent the start state and the final state respectively while analyzing the output of the N-gramSOM network. An example of tri-gram or a 3-gram of a string is shown at Figure 17.

B	B
ptt	ptt
ttt	ttt
ttt	ttt
ttt	ttt
ttt	ttt
ttt	ttv
ttv	tvp
tvv	vps
vv	ps
v	s
E	E

Figure 17. Two examples of a tri-gram or a 3-gram generated from the word “pttttttvv” and “ptttttvps”.

4.2.3 N-gram Translator

N-gramSOM is designed to train using input vectors of binary values only.

Therefore, before the n-grams can be used to train N-gramSOM, they must be translated

to binary representation. Because of this, a program called the N-gram Translator was programmed to translate n-grams to vector of binary values.

The N-gram Translator collects all the unique symbols from a file of n-grams and creates a binary representation for each symbol. The binary representations are orthogonal to one another. Based on these symbols and their binary representations, a lookup table is built. An example of the lookup table is illustrated in Figure 18.

The N-gram Translator needs two parameters: an input file and an output file. The input file contains n-grams that are generated from a string file; the output file contains the binary vectors that represent each n-gram in the input file. The output file is used to train the N-gramSOM network. An example of the input file is illustrated in Figure 19.

B	0000001
E	0000010
p	0000100
s	0001000
t	0010000
v	0100000
x	1000000

Figure 18. An example of a lookup table. This table is created from the strings in Figure 16. The “0000001” represents an “B”, “0000010” represents an “E”, and so on. In addition, “0000000” represents an empty space.

00000010000000000000	00000010000000000000
00001000010000001000	00001000010000001000
00100000010000001000	00100000010000001000
00100000010000001000	00100000010000001000
00100000010000001000	00100000010000001000
00100000010000001000	00100000010000001000
00100000010000001000	00100000010000001000
00100000010000001000	00100000010000001000
00100000010000001000	0010000001000000000100
00100000010000001000	010000000001000001000
01000000100000000000	00001000001000000000
01000000000000000000	00010000000000000000
00000100000000000000	00000100000000000000

Figure 19. The binary numbers in this figure are directly translated from the tri-grams in Figure 17. The “00000010000000000000” represents “B”, the “00001000010000001000” represents “ptt”, and so on.

4.2.4 N-GramSOM

N-gramSOM is a modified SOM network, which is trained using input vectors generated by the N-gram Translator. The network model is illustrated in Figure 10, and its training process is shown in the algorithm in Figure 12. During the training process, the program feeds the same training set into the network for many epochs.

The training process will be slow if the training set is read from the file for each epoch. To speed up the training process, the training set is read in at the beginning of execution and stored into internal memory, so that it can be reused over all epochs. The length of training time can range from a few epochs to a few hundred epochs, depending on needs of the network. After the network is trained, the weights of the network are saved into a file; therefore, they can be loaded to an untrained network later, which can be

put into application immediately. An example of a map of the weights of a network, after being trained, is illustrated in Figure 20.

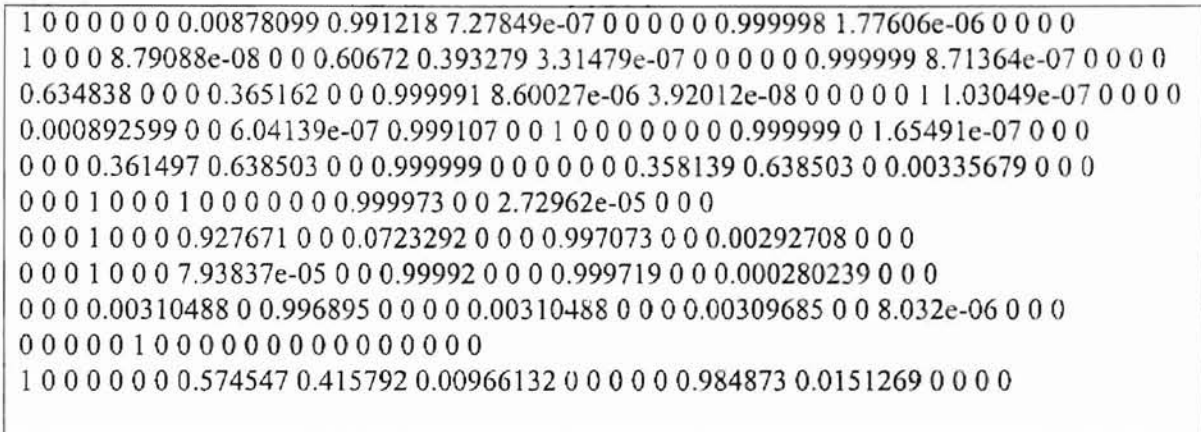


Figure 20. A partial map that consists of 11 out of 100 weights of a 10x10 N-gramSOM networks.

4.2.5 Build FSA

After the network is trained, the same training set is reused to test its response by feeding the training set into the network again. The response of the network is determined by the winning neurode for each input vector from the training set. The response of the network is then recorded. A new FSA will be analyzed and translated from the recorded data. An example of the network response is shown in Figure 21. In this figure, each line of data represents the network responses for each input sequence from a training set, and also represents a transition from start state to final state. For

example, “89 B 74 ptt 72 ttt 72 ttt 72 ttt 72 ttt 52 ttv 50 tvv 93 vv 95 v 9 E” is the network response for the input sequence “B ptt ttt ttt ttt ttt ttv tvv vv v E.” The first number, “89,” is the index of the winning neurode for the letter “B.” The start state is the index of the winning neurode, “89,” of the letter “B.” Therefore, the first number always serves as the start state. The last number before “E,” in this case “9,” is the network response of the letter “E” and will not be used. The letter “E” indicates that the number, “95,” before the last number, “9,” represents the actual final state. A transition from one state to another state can be analyzed from the last example in which state 89 to state 74 produces symbol “p”; state 74 to state 72 produce symbol “t”, and so on. The symbol is the first letter of each word, such that “p” is for “ptt” and “t” is for “ttt”. The full translation of the transitions from the above example is illustrated at below:

89 \xrightarrow{p} 74 \xrightarrow{t} 72 \xrightarrow{t} 72 \xrightarrow{t} 72 \xrightarrow{t} 72 \xrightarrow{t} 52 \xrightarrow{t} 50 \xrightarrow{v} 93 \xrightarrow{v} 95 ,

where 89 \xrightarrow{p} 74 means the transition from state 89 to state 74, and produces output symbol “p,” An example of an FSA translated from the network response data in Figure 21, is illustrated in Figure 22; this FSA is shown graphically in Figure 23.

```

89 B 74 ptt 72 ttt 72 ttt 72 ttt 72 ttt 72 ttt 52 ttv 50 tvv 93 vv 95 v 9 E
89 B 74 ptt 72 ttt 72 ttt 72 ttt 72 ttt 52 ttv 70 tvp 97 vps 47 ps 25 s 9 E
89 B 74 ptt 72 ttt 72 ttt 72 ttt 72 ttt 52 ttv 50 tvv 93 vv 95 v 9 E
89 B 74 ptt 72 ttt 72 ttt 72 ttt 52 ttv 70 tvp 97 vps 47 ps 25 s 9 E
89 B 74 ptt 72 ttt 72 ttt 72 ttt 52 ttv 50 tvv 93 vv 95 v 9 E
89 B 74 ptt 72 ttt 72 ttt 52 ttv 70 tvp 97 vps 47 ps 25 s 9 E
89 B 74 ptt 72 ttt 72 ttt 52 ttv 70 tvp 76 vpx 3 pxv 0 xv v 93 vv 95 v 9 E
89 B 74 ptt 72 ttt 72 ttt 52 ttv 50 tvv 93 vv 95 v 9 E
89 B 74 ptt 72 ttt 52 ttv 70 tvp 97 vps 47 ps 25 s 9 E
89 B 74 ptt 72 ttt 52 ttv 70 tvp 76 vpx 23 pxt 30 xtv 50 tvv 93 vv 95 v 9 E
89 B 74 ptt 72 ttt 52 ttv 70 tvp 76 vpx 3 pxv 90 xvp 97 vps 47 ps 25 s 9 E
89 B 74 ptt 72 ttt 52 ttv 70 tvp 76 vpx 3 pxv 0 xv v 93 vv 95 v 9 E
89 B 74 ptt 72 ttt 52 ttv 50 tvv 93 vv 95 v 9 E

```

Figure 21. Part the network response from the strings at the left column in Figure 17.

# 2	58 15 s	32 44 x	76 3 p
89 25 95	58 5 s	7 15 s	97 47 p
&	64 50 t	7 5 s	0 93 v
89 29 t	64 70 t	93 95 v	90 76 v
89 49 t	69 21 x	15 56 x	90 97 v
89 50 p	69 32 x	5 21 x	30 50 t
89 58 t	74 52 t	5 32 x	30 70 t
89 64 p	74 72 t	70 76 v	44 52 t
89 69 t	80 76 v	70 97 v	44 72 t
89 74 p	80 97 v	21 0 x	23 30 x
89 80 p	56 25 s	52 50 t	23 44 x
29 56 x	27 27 s	52 70 t	3 0 x
49 27 s	27 7 s	72 52 t	3 90 x
49 7 s	21 90 x	72 72 t	47 25 s
50 93 v	32 30 x	76 23 p	\$

Figure 22. The FSA translated from the responses of the network. The diagram of this FSA is illustrated in Figure 23.

4.2.6 Nondeterministic FSA to Deterministic FSA (NftoDf)

If an FSA is nondeterministic, it means that in each situation there is a finite set of possible moves that the FSA can make, rather than just assigning a unique move [Linz96]. A nondeterministic FSA (NDFSA) is defined as the following:

$$(Q, \Sigma, \delta_N, q_0, F_N),$$

where

Q is a finite set of internal states,

Σ is a finite set of symbols called the input alphabet,

$\delta_N: Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$ is a total function called the transition function,

$q_0 \in Q$ is the initial state,

$F_N \subseteq Q$ is a set of final states,

λ is an empty string [Linz96].

If an FSA is deterministic, it means that in each situation there is a unique move in the FSA. A deterministic FSA (DFSA) is defined as the following:

$$(Q, \Sigma, \delta_D, q_0, F_D),$$

where Q, Σ, q_0 and F_D are defined as the NDFSA above, and δ_D is defined as

$$\delta_D: Q \times \Sigma \rightarrow Q.$$

The new FSA constructed by N-gramSOM normally has many states and is

complicated; in addition, the new FSA may be a NDFSA. To reduce the size and complication of the new FSA, it needs to be minimized. John C. Martin developed a minimization algorithm that only works on minimizing a DFSA [Martin91]. This algorithm will be discussed in the next section. To use the minimization algorithm, the new NDFSA must be converted to an equivalent DFSA. An algorithm by Peter Linz is used to convert the NDFSA to an equivalent DFSA, illustrated in Figure 24 [Linz96].

```

BEGIN
  Create a graph  $G_D$  with vertex  $\{q_0\}$  and mark the vertex as the start state;
  Repeat until no more edges are missing
  begin
    Take any vertex of  $G_D$   $\{q_i, q_j, \dots, q_k\}$  that has no outgoing edge for some  $a \in \Sigma$ ;
    Compute  $\delta_N(q_i, a)$ ,  $\delta_N(q_j, a)$  ...,  $\delta_N(q_k, a)$ ;
    Then form the union of all these  $\delta$ , yielding the set  $\{q_i, q_m, \dots, q_n\}$ ;
    Create a vertex for  $G_D$  labeled  $\{q_i, q_m, \dots, q_n\}$  if it does not exist;
    Add to  $G_D$  an edge from  $\{q_i, q_j, \dots, q_k\}$  to  $\{q_i, q_m, \dots, q_n\}$  and label it with  $a$ ;
  end;
  Any vertex of  $G_D$  whose label contains any  $q_f \in F_N$  is identified as final vertex;
  for all vertex
    for all  $a \in \Sigma$ 
      if there is no outgoing edge labeled  $a$ 
        Add to  $G_D$  an edge from the vertex to a vertex  $\{trap\}$ ;
END.

```

Figure 24. An algorithm that converts the NDFSA to an equivalent DFSA [Linz96].

A tool named Determinize FSA was developed based on the algorithm in Figure 24. The program takes two parameters: an input file and an output file. The input file contains an NDFSA and the output file contains an equivalent DFSA, which is converted from the aforementioned NDFSA. An example of the input file NDFSA is shown in

Figure 22 and Figure 23. The output file that is constructed by the Determinize FSA from the input data in Figure 22 is illustrated in Figure 25, and the graphical look of the FSA is illustrated in Figure 26.

# 2	213256 trap v
89 25 95	50527072 50527072 t
&	50527072 769397 v
89 29495869 t	50527072 trap p
89 50647480 p	50527072 trap s
89 trap s	50527072 trap x
89 trap v	769397 23347 p
89 trap x	769397 95 v
29495869 152757 s	769397 trap s
29495869 213256 x	769397 trap t
29495869 trap p	769397 trap x
29495869 trap t	0304490 50527072 t
29495869 trap v	0304490 769397 v
50647480 50527072 t	0304490 trap p
50647480 769397 v	0304490 trap s
50647480 trap p	0304490 trap x
50647480 trap s	25 trap p
50647480 trap x	25 trap s
trap trap p	25 trap t
trap trap s	25 trap v
trap trap t	25 trap x
trap trap v	23347 0304490 x
trap trap x	23347 25 s
152757 152757 s	23347 trap p
152757 213256 x	23347 trap t
152757 trap p	23347 trap v
152757 trap t	95 trap p
152757 trap v	95 trap s
213256 0304490 x	95 trap t
213256 25 s	95 trap v
213256 trap p	95 trap x
213256 trap t	\$

Figure 25. An example of an output file constructed by the Determinized FSA program, from the input file in Figure 22.

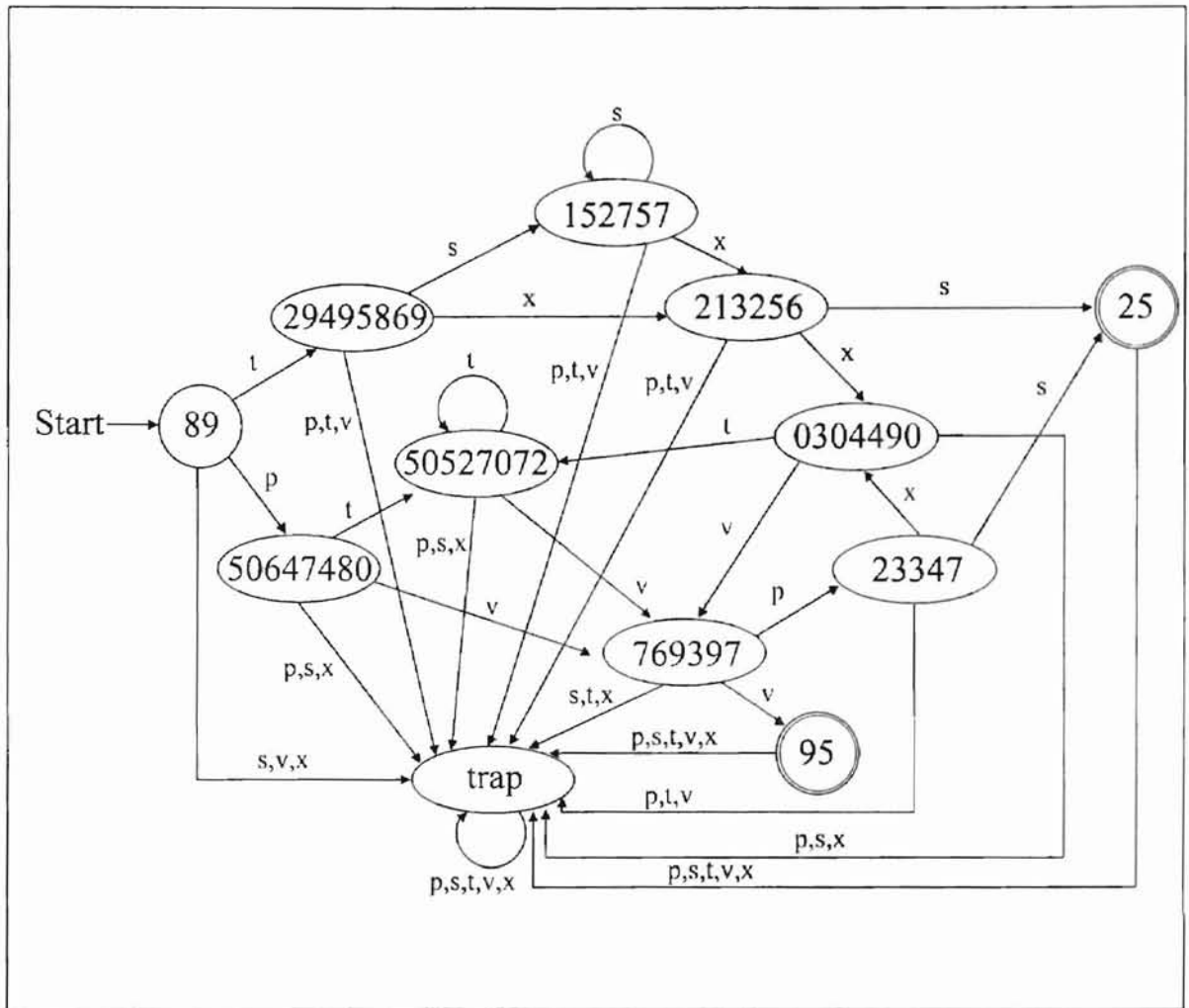


Figure 26. A graphical looks of the DFSA from Figure 25.

4.2.7 Minimizing FSA

Many DFSAs accept the same language; those DFSAs may have a different number of states. According to Martin's theory of the uniqueness of the minimum DFSA, if there are two equivalent DFSAs and both DFSAs are reduced to minimum number of states, they will have the same number of states and will look the same

[Martin91]. If a DFSA does not have the minimum number of states, it must have some redundant parts. It is therefore always advisable to reduce a DFSA to the equivalent DFSA that has the fewest number of states.

The new FSA created by N-GramSOM usually is an NDFSA, has many states, and is complicated. After the new FSA is constructed, it must be compared to the original FSA to check their equivalence. Therefore, the new FSA should be minimized to reduce the process time for checking the equivalence process. The method to check equivalence of both FSAs will be discussed in detail in the section 4.2.8. Martin developed an efficient algorithm that minimizes the number of states of a DFSA [Martin91], as shown in Figure 27. The algorithm has been proved to be reliable by the author; but it only works with a DFSA. Since the new FSA is an NDFSA, it must be converted from an NDFSA to an equivalent DFSA to use the algorithm. The method and the algorithm that convert the NDFSA to an equivalent DFSA are discussed in the section 4.2.6.

The Minimized FSA tool is implemented based on the algorithm shown in Figure 27. The tool takes two parameters: a DFSA file-as-input file and a minimized DFSA file-as-output file. The input file is demonstrated in Figure 25 and the output file is demonstrated in Figure 28. The diagram of the minimized DFSA in Figure 28 is illustrated in Figure 29.

```

BEGIN
    Create a  $(N-1) \times (N-1)$  matrix  $M$  where  $N$  is number of states and label the
        columns with the state name from the first state to  $N-1$  state and label the rows
        with the state name from the second state to  $N$  state;
    Create a set  $S(i, j)$  where  $i$  is the row number and  $j$  is the column number in  $M$ 
        and  $S(i, j)$  is going to store a set of state pairs;
    List all unordered pairs  $(p, q)$  with  $p \neq q$ ;
    for each pair  $(p, q)$  with  $p \neq q$ ;
    begin
        if exactly one of  $p, q$  is in  $F$  then
            MARK  $(p, q)$ ;
        else
            Initialize the set  $S(p, q)$  to be empty;
    end;
    for each pair  $(p, q)$  with  $p \neq q$ 
    begin
        if  $(p, q)$  is not marked then
            for each  $a \in \Sigma$ 
            begin
                 $r = \delta(p, a)$ ;
                 $s = \delta(q, a)$ ;

                if  $r \neq s$  then
                    if  $(r, s)$  is not marked then
                        Insert  $(p, q)$  into  $S(r, s)$ ;
                    else
                        MARK  $(p, q)$ ;
            end;
        end;
    end;
END.

Procedure MARK  $(p, q)$ 
BEGIN
    Mark  $(p, q)$ ;
    for each pair  $(r, s)$  in  $S(p, q)$  ;
        MARK  $(r, s)$ ;
END.

```

Figure 27. An minimization algorithm by Martin [Martin91].

# 1	trap trap s
89 25	trap trap t
&	trap trap v
89 29495869 t	trap trap x
89 50647480 p	213256 25 s
89 trap s	213256 50647480 x
89 trap v	213256 trap p
89 trap x	213256 trap t
29495869 213256 x	213256 trap v
29495869 29495869 s	769397 213256 p
29495869 trap p	769397 25 v
29495869 trap t	769397 trap s
29495869 trap v	769397 trap t
50647480 50647480 t	769397 trap x
50647480 769397 v	25 trap p
50647480 trap p	25 trap s
50647480 trap s	25 trap t
0647480 trap x	25 trap v
trap trap p	25 trap x
	\$

Figure 28. An example of minimized DFSA from Figure 16.

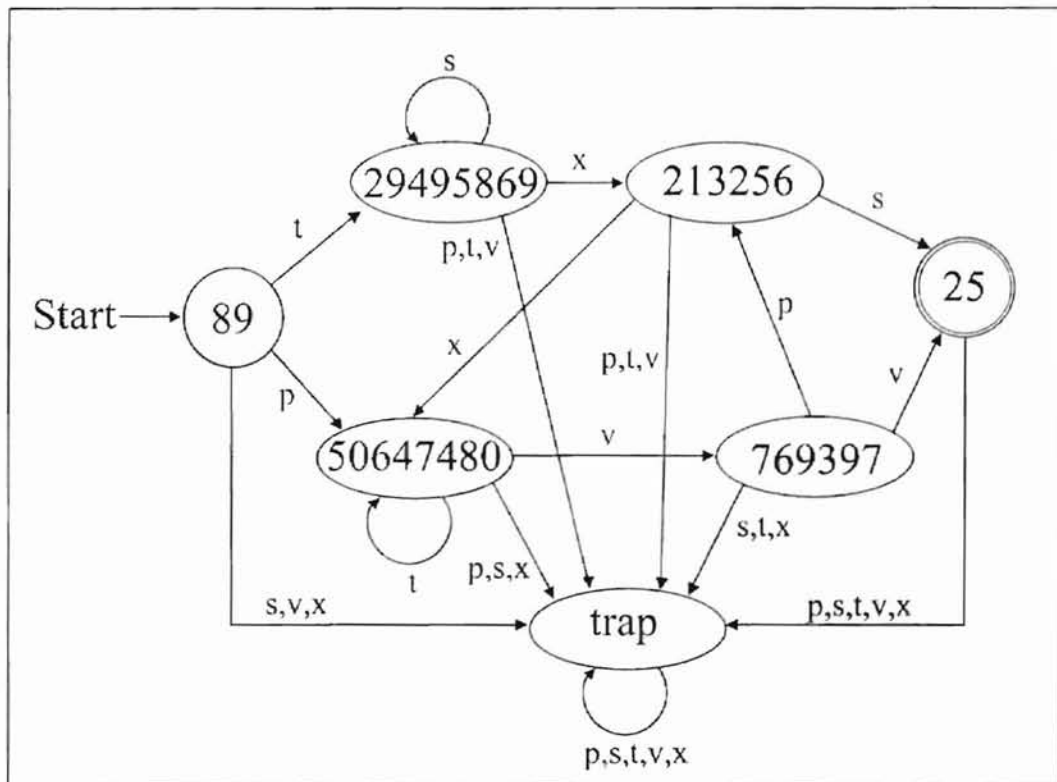


Figure 29. The diagram of the minimized DFSA in Figure 19.

4.2.8 Equivalence Check

After the trained N-gramSOM creates a new FSA, the equivalence of the original FSA and the new FSA is questioned. Two methods can be used to check for the equivalence of both FSAs: visual comparison and algorithm comparison. As mentioned in the Minimizing FSA section, according to the uniqueness of the minimum DFSA, both FSAs are converted to DFSAs and minimized. If both minimized FSAs look exactly the same then they are equivalent.

If an algorithm is used to check for the equivalency, the strings from both FSAs are generated, then compared. A problem is encountered in that the number of the strings in the language of an FSA can be infinite. However, an algorithm developed by Aho and Ulman shows a finite, maximum length of strings needed to test the equivalence of two FSAs [AhoUll72]. The selection of the maximum length of strings will be discussed in the section 4.2.9.

4.2.9 Determining String Length

As discussed in the previous section, the maximum length of strings needed to check the equivalence of two FSAs must be predetermined. Aho and Ullman's definition (Figure 30), lemma (Figure 30) and algorithm (Figure 31) are used to support this discussion [AhoUll72].

DEFINITION 1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automata, and let q_1 and q_2 be distinct states. We say that x in Σ^* distinguishes q_1 from q_2 if $(q_1, x) \xrightarrow{*} (q_3, e)$, $(q_2, x) \xrightarrow{*} (q_4, e)$, and exactly one of q_3 and q_4 is in F . We say that q_1 and q_2 are k -indistinguishable, written $q_1 \stackrel{k}{\equiv} q_2$, if and only if there is no x , with $|x| \leq k$, which distinguishes q_1 and q_2 . We say that the two states q_1 and q_2 are indistinguishable, written $q_1 \equiv q_2$, if and only if they are k -indistinguishable for all $k \geq 0$.

A state $q \in Q$ is said to be inaccessible if there is no input string x such that $(q_0, x) \xrightarrow{*} (q, e)$.

Figure 30. The definition of indistinguishable state [AhoUll72].

LEMMA 1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton with n states. States q_1 and q_2 are indistinguishable if and only if they are $(n-2)$ -indistinguishable.

Figure 31. A lemma to determine two states in an FSA, which are indistinguishable [AhoUll72].

ALGORITHM 1

Input: two finite automata $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2)$ such that $Q_1 \cap Q_2 = \emptyset$.

Output. "YES" if $L(M_1) = L(M_2)$, "NO" otherwise.

Method. Construct the finite automaton

$$M = (Q_1 \cup Q_2, \Sigma_1 \cup \Sigma_2, \delta_1 \cup \delta_2, q_1, F_1 \cup F_2)$$

Using Lemma 1 determine whether $q_1 \stackrel{?}{\equiv} q_2$. If so, say "YES"; otherwise, say "NO".

Figure 32. An algorithm to decide the equivalence of two FSAs. [AhoUll72].

The algorithm in Figure 32 can be explained as follows. Assume that there exist two machines $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2)$ that share no states; the combination of both machines, $M = (Q_1 \cup Q_2, \Sigma_1 \cup \Sigma_2, \delta_1 \cup \delta_2, q_0, F_1 \cup F_2)$, is used to perform equivalence check using Lemma 1. The union of the machines M_1 and M_2 is shown in Figure 33. Further assume that M_1 and M_2 use the same alphabet. Therefore, the alphabet of M is equal to the alphabet of M_1 and M_2 ; i.e. $\Sigma = \Sigma_1 \cup \Sigma_2 = \Sigma_1 = \Sigma_2$. The subset of all strings, Σ^* , with length $n-2$ or less are used by the algorithm, where $n = |Q_1 \cup Q_2| = |Q_1| + |Q_2| + 1$; and the additional one in the last part of the equation is the start state of M as shown in Figure 33. According to Definition 1, the two start states q_0 and q_1 are $(n-2)$ -indistinguishable, if and only if no strings with length $n-2$ in Σ^* are distinguishable—thus, by Lemma 1, q_0 and q_1 are $(n-2)$ -indistinguishable. If q_0 and q_1 are $(n-2)$ -indistinguishable, then M_1 and M_2 are equivalent according to Algorithm 1.

In conclusion, the maximum length of the string that needs to be used to compare strings from two FSAs is the sum of the number of states of the both FSAs minus 1; i.e., $n = |Q_1 \cup Q_2| - 2 = (|Q_1| + |Q_2| + 1) - 2 = |Q_1| + |Q_2| - 1$.

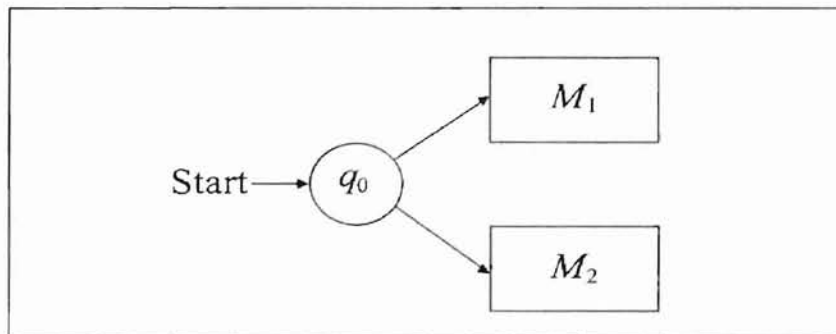


Figure 33. $M = (Q_1 \cup Q_2, \Sigma_1 \cup \Sigma_2, \delta_1 \cup \delta_2, q_0, F_1 \cup F_2)$ is the union of two machines $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2)$.

Given two FSA machines, M_0 and M_1 , their equivalence is tested using their respective languages (L_0 and L_1). If L_0 is accepted by M_1 and L_1 is accepted by M_0 , then M_0 and M_1 are equivalent; otherwise M_0 and M_1 are not equivalent. A program named Check Equivalence was programmed based on this concept.

Check Equivalence needs three parameters: two input files contain two FSAs and an output file contains the string "YES" or "NO". The program will call the String Generator to generate strings up to the length of $n-1$, where n is the total number of states in both FSAs. If the strings generated from the first FSA are accepted by the second FSA, and the vice versa, then the program will output the string "YES" to indicate that the two FSAs are equivalent; otherwise the program will output the string "NO" indicate that the two FSAs are not equivalent.

4.2.10 N-gramSOM System

For the purpose of research, a system is built using all the tools discussed above. The system is named the N-gramSOM System. The system is illustrated in Figure 34. The system is designed to handle three different options: train the N-gramSOM, put the N-gramSOM into applications, or both. Therefore, the system can be used to train the N-gramSOM network only, with many different languages and save all the trained data maps to different directories. The maps will be loaded for application into the N-gramSOM again. During the application phase, the system will construct a new FSA based on the knowledge that the network has learned. The newly created FSA will be

compared to the original FSA to check their equiv. The results and statistics will be output to a file. The system also can do the both jobs described above at once. Thus, during the testing phase, the three different options are open to experimentation. The complete system is illustrated in Figure 33. The complete source code of the N-gramSOM System is not included in the thesis. Future researchers may obtain the N-gramSOM source code by contacting the author.

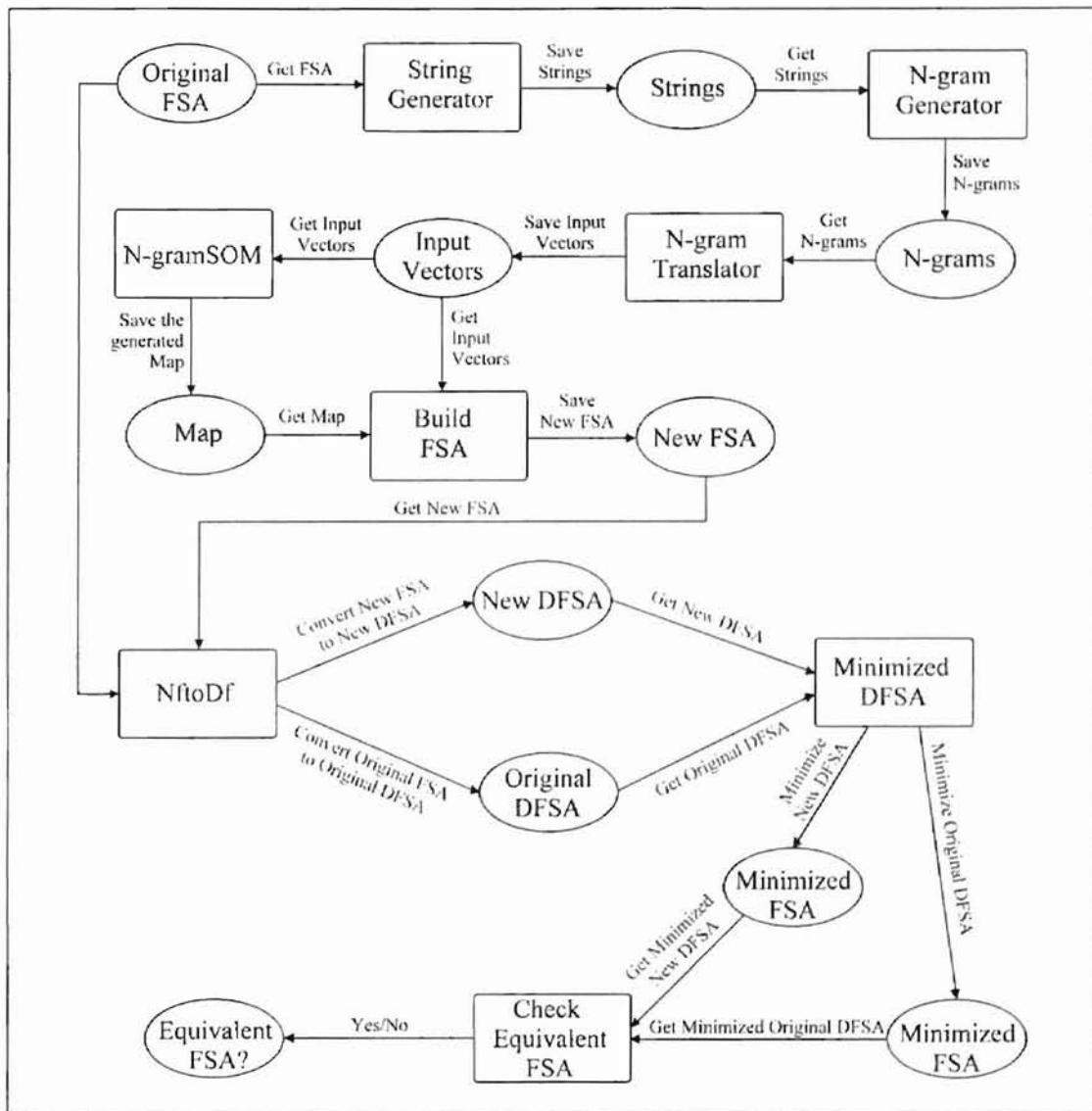


Figure 34. Illustration of the N-gramSOM System. The oval box represents a file, the rounded rectangular box represents a process, and the arrow represents a process transition.

4.3 Testing

N-gramSOM System was developed on the SunOS 5.5.1 platform with the UNIX System V Release 4.0 operating system. All the testing processes also were done in the same platform. After the implementation of N-gramSOM System was completed, the next problem encountered was the selection of testing FSA.

4.3.1 Selection of FSA

Eleven FSAs were chosen to test the N-gramSOM. Of these, Boydston has used four to test the SeqSOM, all of which are among the eleven test cases used for the current research. The other seven FSAs were obtained from internet sites [Giovan98; Pelts98]. One of the FSAs from Boydston is illustrated in Figure 16; the rest of the FSAs are included in APPENDIX A.

The data chosen to train N-gramSOM are all strings with length of 10 or less, generated from the FSA in Figure 16, and the FSAs from APPENDIX A. The total number of strings generated from each FSA is shown in Table 2. The larger the size of the training set is the longer the time it will take to train the network.

Name of FSA	Total number of strings with length 10 or less	Name of FSA	Total number of strings with length 10 or less
FSA1	103	FSA7	1024
FSA2	30	FSA8	683
FSA3	9	FSA9	4
FSA4	133	FSA10	2026
FSA5	1233	FSA11	511
FSA6	511		

Table 2. The total numbers of strings generated by each FSA (training set), with a string length of 10 or less.

4.3.2 Experimental Results

The number of states in the eleven FSAs used in the test ranges from three to six, and their alphabet sizes ranges from two to five. Each FSA was used to train an N-gramSOM network. A new FSA was constructed from the trained N-gramSOM with the same set of strings. After the new FSA was constructed, it was converted to a DFSA and then minimized. The minimized FSA was compared to the original FSA for equivalence. The results of six out of eleven test cases are shown in Table 3. All the FSAs shown in Table 3 are equivalent, showing that N-gramSOM is able to learn languages and construct equivalent FSAs. The parameters that were used are shown in Table 3. The results of the other five FSAs are inconclusive, because no specific data shows that N-gramSOM is not able to learn their languages. Those FSAs are left for future research, and their diagrams are shown in APPENDIX A.

In Table 3, the first column contains the names of the FSAs used to test N-gramSOM; the total number of states is shown in the second column. The third column

shows the total numbers of state of the DFSA that is converted from the original FSA. From the fourth column to the sixth column are the new FSAs that are generated by N-gramSOM. The last column demonstrates that the original DFSA is equivalent to the minimized DFSA, constructed by N-gramSOM.

In Table 4, the first column is the name of the FSA that is used to test N-gramSOM. The second column is the size of an n-gram used to generate input data. The third column is the rate that controls the weight adjustment in the network. The fourth column is the size of the neighborhood used to fine-tune the map. The offset is used to limit the smallest weight value in the network—for example, if the offset is 1.0e-08, any weight in a network that is less than 1.0e-08 will be set to 0.

Test Cases	Original FSA	Original DFSA	New FSA	New DFSA	Minimized DFSA	Equivalence
FSA1	6	7	31	12	7	YES
FSA2	3	3	12	4	3	YES
FSA3	5	6	11	7	6	YES
FSA4	5	5	14	7	5	YES
FSA6	3	4	12	4	4	YES
FSA11	3	4	12	4	4	YES

Table 3. The final results after 300 tests run for each test cases.

Test Cases	N-gram size	Epoch	Learning rate	Neighborhood size	Offset
FSA1	3	60	0.3	3	1.0e-08
FSA2	3	4	0.3	3	1.0e-08
FSA3	4	200	0.1	3	1.0e-08
FSA4	3	5	0.3	3	1.0e-08
FSA6	3	3	0.3	3	1.0e-08
FSA11	3	2	0.3	3	1.0e-08

Table 4. The parameters that the N-gramSOM use to learn the languages.

4.3.3 Failures from the Testing

The N-gramSOM sometimes fails to generate an equivalent FSA. From the observations during testing N-gramSOM, the parameters shown in Table 3 are the those that can affect the output of the network. After many tests, two forms of failure were observed: N-gramSOM inconsistently produced equivalent FSA, or the network never produced equivalent FSA. Another observation is that when one parameter is changed, the other parameters may need to be changed, or the performance of the network could be affected.

Increasing the size of the n-gram increases the likelihood that the network can produce an equivalent FSA. However, the size of the training set and the training time also are increased, the reason being that the larger the size of the n-gram, the more unique the patterns that can be produced. The more input data provided for the network, the more knowledge it can learn. Increasing the training time also allows increases the likelihood that a network will generate an equivalent FSA. If the network does not have enough training time, it will not have enough time to learn. Therefore, the longer the network is trained, the more the neurodes of the network will be adjusted and settled down gradually. Once the neurodes are settled down, the increase of the training time will not make any difference. As a result, other parameters, like the learning rate and neighborhood size, can be adjusted to improve the network performance. Reducing the learning rate or the neighborhood size, or both, will increase the likelihood that the network can produce an equivalent FSA. However, reducing the learning rate requires that the training time must also be increased. Reducing the learning rate and

neighborhood size will sharpen the edge of the clusters formed, and the states will be more well defined. Therefore, the performance of the network can be improved by reducing the learning rate and neighborhood size, as well as increasing the learning time. Increasing the value of the offset shows improvement of the network performance, up to a point. The offset is a control parameter that is used to limit the smallest value of the weights of the network. The offset value of $1.0e-08$ appears to be the optimal value that works for every test case, but any offset value greater than $1.0e-08$ may cause the network performance to drop immediately.

Another problem that may cause the network to fail is that the number of the strings that are provided to train the network is not enough. Nonetheless, for the eleven test cases used in this research, the string length of 10 or less appears to be more than enough. As a result, only one size of string was used in this research.

4.3.4 Compare the N-gramSOM to SeqSOM

The purpose of this research is to show that N-gramSOM, like SeqSOM, is another technique to process contextual data. During the training and analyzing of the network, a few differences between the results of N-gramSOM and the report of SeqSOM were observed.

Boydston [Boydston97] tested using different string lengths, ranging from nine to twenty. However, the only string length used in the N-gramSOM was ten. SeqSOM does not use an offset value, whereas N-gramSOM uses an offset value to control the

weights of the network, and it shows improvement of performance.

SeqSOM is able to learn and produce two equivalent FSAs out of the four test cases used by Boystun. However, N-gramSOM is able to learn and produce four equivalent FSAs out of the four test cases. From this point of view, N-gramSOM had about the same success rate: ~50% of the test cases.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

The purpose of the research in this thesis is to show that N-gramSOM network is able to learn and capture the contextual nature of input data. The experiments that have been done show that N-gramSOM can successfully learn and construct an FSA that accepts the language represented by the input data. Like the SeqSOM, N-gramSOM is trained with strings generated from a given FSA. After the network is trained, a new FSA is constructed. The new FSA is compared to the original FSA for equivalence. For the purpose of this research, a set of tools is programmed. The tools are:String Generator, N-gram Generator, Build Table, N-gram Translator, N-gramSOM network, Build FSA, Determinize FSA, Minimized FSA and Check Equivalence. All of these tools are formed together to be an N-gramSOM System, which is used to perform all tests.

The results of this research show that N-gramSOM is able to learn the language from a given FSA and construct an equivalent FSA. Eleven FSAs are tested and N-gramSOM is able to produce six equivalent FSAs. By contrast, while N-gramSOM is able to learn and capture the behavior of the four FSAs that have been used to the SeqSOM, SeqSOM is able to construct only two equivalent FSAs out the four test cases.

From these observations, adjusting the parameters of n-gram size, learning time, learning rate, neighborhood size and offset can improve the network performance regarding learning contextual data.

5.2 Future Work

This research shows that N-gramSOM has the potential to learn a language from an FSA, and to construct an equivalent FSA. However, the research cannot show that N-gramSOM is able to work on all cases. The primary problem of whether N-gramSOM will be able to learn a language from contextual data and produce an FSA that accepts the language is solved.

Since the research shows that the parameters of n-gram size, training time, learning rate, neighborhood size and offset are factors that can affect the performance of N-gramSOM, the parameters are still not limited. More investigation should be done concerning other possible parameters that can improve the network performance. The parameter values shown in Table 3 in CHAPTER V are varied for different test cases, except the offset and neighborhood size. Therefore, the optimal value for the parameters is open to discussion.

In addition, another optimal value that needs to be investigated is the size of network. The FSAs used for this research have a small training set. For future research, more complicated FSAs need to be tested. The network size used for this research is 10×10 , because only small FSAs have been used. Therefore, for more complicated

FSAs, the network size should be increased. The optimal value of the network size remains unanswered.

In this research, N-gramSOM did not produce equivalent FSAs from the test case FSA5, FSA7, FSA8, FSA9 and FSA10. Those FSAs are shown in Appendix A. In conclusion, more investigation is needed on these FSAs, including the utilization of the above-mentioned and, possibly other parameters.

REFERENCES

- [AhoUll72] Aho and Ullman. The Theory of Parsing, Transition and Compiling. Volume 1: Parsing. Prentice-Hall, Englewood Cliffs, NJ, 1972.
- [Anderson95] Anderson, J. An Introduction to Neural Networks. Cambridge, MA: The MIT Press, 1995.
- [Boydstun95] Boydstun, R. and Mayfield, B. "Investigation of Sequential Self-Organizing Maps." Computer Science Department, Stillwater, Oklahoma State University, 1995.
- [Boydstun97] Boydstun, R. Investigation of Sequential Self-Organizing Maps. Computer Science Department, Stillwater, OK: Master Thesis, 1997.
- [Caudill93] Caudill, M. "A Little Knowledge is a Dangerous Thing." AI Expert, 8(6), 16-22, 1993.
- [Fausett94] Fausett, L. Fundamental of Neural Networks: Architectures, Algorithms, and Applications. Englewood Cliffs, NJ: Prentice Hall, 1994
- [Freeman92] Freeman, J. and Skapura, D. Neural Networks, Algorithms, Applications, and Programming Techniques. Reading, MA: Addison-Wesley Publishing Company, 1992.
- [Giovan98] Giovannetti, R. Finite Automata: Examples. URL: <http://twilight.dsi.unimi.it/~colors/AUTOMATAHT/MANUAL/AUTOMATA-MAN.html>, Date accessed: 6/1998.
- [Hagan96] Hagan M., Demuth, H., and Beale, M. Neural Network Design. Boston, MA: PWS Publishing Company, 1996.
- [Hiotis93] Hiotis, A. "Inside a Self-Organizing Map." AI Expert. 8(4), 38-42, 1993.
- [Kohonen89] Kohonen, T. Self-organizing and Associative Memory Third Edition. New York, NY: Springer-Verlag, 1989.

- [KruseLT91] Kruse, R., Leung, B. and Tondo, C. Data Structure and Program Design in C. Englewood Cliffs, New Jersey:Prentice Hall, 1991
- [Lebanon95] Lebanon Valley College. Free Text vs. Controlled Vocabulary. URL: <http://www.lvc.edu/www/library/free.html>, Date accessed: 5/1998
- [Linz96] Linz, P. An Introduction to Formal Languages and Automata Second Edition. Lexington, Massachusetts: D.C. Heath and Company, 1996.
- [Maren90] Maren, A., Harston, C., and Par, R. Handbook of Neural Computing Applications. San Diego: Academic Press, 1990.
- [Martin91] Martin, J. Introduction to Languages and the Theory of Computation. New York, NY: McGraw-Hill, Inc., 1991.
- [McCulloch43] McCulloch, W. and Pitts, W. "A Logical Calculus of the Ideas Immanent in Nervous Activity." Bulletin of Mathematical Biophysics. 5, 115-133, 1943.
- [Minsky69] Minsky, M. and Papert, S. Perceptrons. Cambridge, MA: MIT Press, 1969.
- [Pelts98] Peltsverger, B. Finite-State Automata. URL: <http://soda.gsw.peacenet.edu/DM/intro.html>. Date accessed: 6/1998.
- [Ritter91] Ritter, Hedge, Martinetz, Thomas, and Schulten, Kaus. Neural Computation and Self-Organizing Maps: An Introduction. Reading, Massachusetts: Addison-Wesley, 1991.
- [Scholtes92] Scholtes, J. "Neural Nets in Information Retrieval." A Case Study of the 1987 Pravda. Science of Artificial Neural Networks. 1710, 631-640, 1982.
- [Weisstein98A] Weisstein E. Topology. URL: <http://www.astro.virginia.edu/~eww6n/math/Topology.html>, Date accessed: 2/98.
- [Weisstein98B] Weisstein E. (1998). Markov Chain. URL: <http://www.astro.virginia.edu/~eww6n/math/MarkovChain.html>, Date accessed on 5/98.
- [Widrow60] Widrow, B. and Hoff, M. "Adaptive Switching Circuits." New York, NY: IRE WESCON Convention Record, IRE Part 4, 96-104, 1960.

[Winston92]

Winston, P. Artificial Intelligence Third Edition. New York, NY:
Addison-Wesley Publishing Company, 1992.

APPENDICES

APPENDIX A

GLOSSARY

GLOSSARY

Artificial Intelligence (AI): the study of machines that can understand, make judgements, etc., in the way that humans do.

Artificial Neural Network (ANN): software that simulates the intelligent aspects of biological neural networks.

Deterministic Finite State Automata (DFSA): if an FSA is deterministic, it means that in every state for each distinctive alphabet there is a unique move to another state in the FSA [Linz96].

Epoch: the process of feeding each vector of a training set into the network once.

Learning rate: a parameter that controls weight adjustments in an ANN.

Neurode: an artificial neuron, also is called a “processing element [Kohonen89].”

N-gram: a vector that contains a sequence of characters from a word or token; each n-gram must contain at least one non-blank character, and at most n characters.

Nondeterministic Finite State Automata (NDFSA): if an FSA is nondeterministic, it means that in every state for each distinctive alphabet there is a finite set of possible moves that the FSA can make, rather than just assigning a unique move [Linz96].

Offset: the smallest permitted weight value in the network; if any weight in a network that is less than the offset, it will be set to 0.

Target value: the expected output of the neural network.

Topology: the mathematical study of the properties of objects that are preserved through deformations, twisting, and stretching [Weiss98A].

Training set: a collection of input vectors that is used to train an ANN.

Weight: the strength or capacity of an individual connection between the input neurodes and output neurodes.

APPENDIX B
ACRONYMS

ACRONYMS

AI: Artificial Intelligence

ANN: Artificial Neural Network

SOM: Self-Organizing Map

SeqSOM: Sequential Self-Organizing Map

N-gramSOM: N-gram Self-Organizing Map

FSA: Finite State Automata

DFSA: Deterministic Finite State Automata

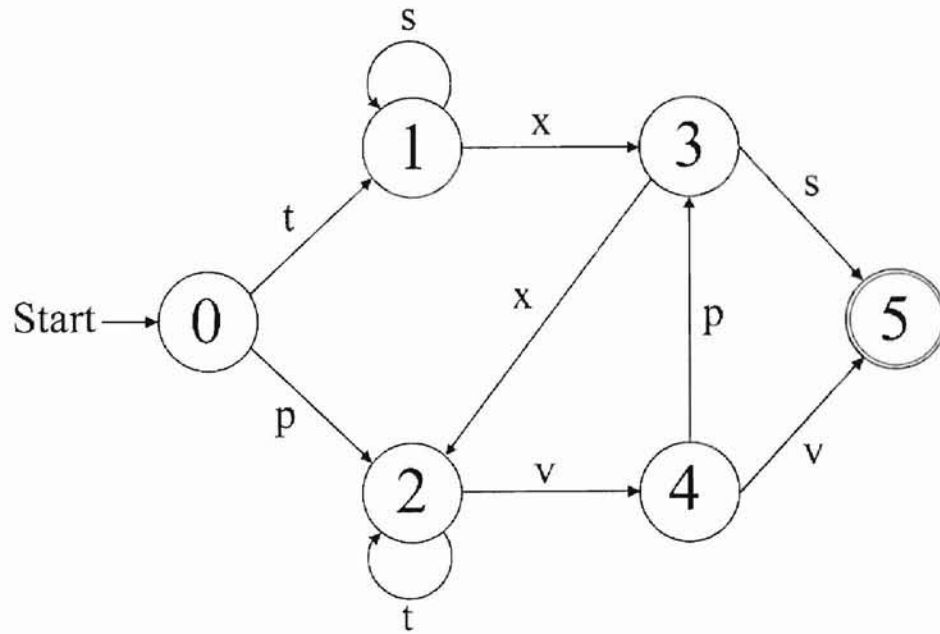
NDFSA: Nondeterministic Finite State Automata

NftoDf: Nondeterministic Finite State Automata to Deterministic Finite State Automata

APPENDIX C

FINITE STATE AUTOMATA USED FOR EXPERIMENTING N-GRAMSOM

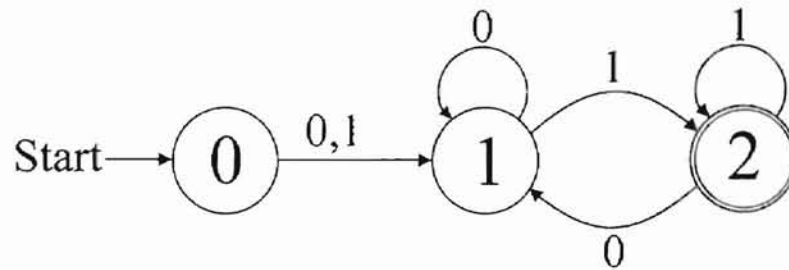
Test Case FSA1



Input Format

```
# 1 10  
0 5  
&  
0 1 t  
0 2 p  
1 1 s  
1 3 x  
2 2 t  
2 4 v  
3 2 x  
3 5 s  
4 3 p  
4 5 v  
$
```

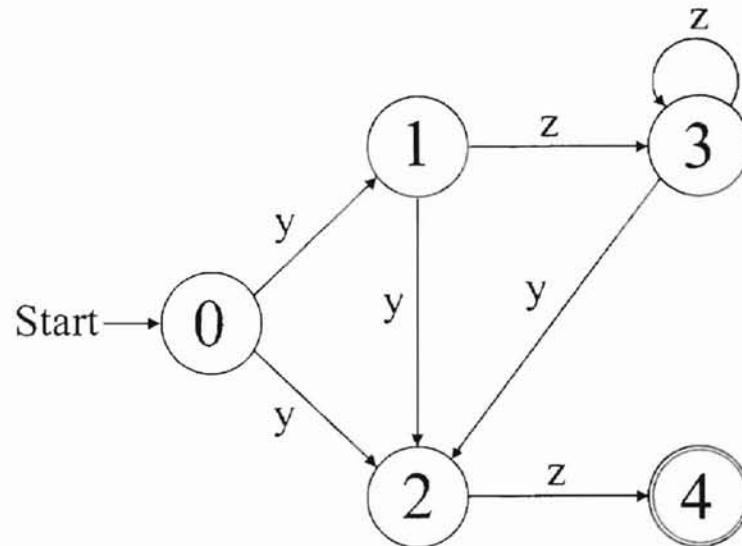

Test Case FSA2



Input Format

1 10
0 2
&
0 1 0
0 1 1
1 1 0
1 2 1
2 1 0
2 2 1
\$

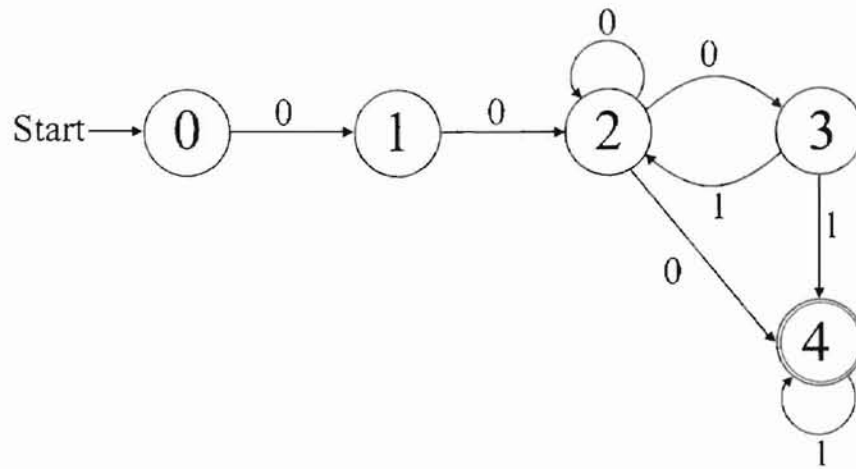
Test Case FSA3



Input Format

1 10
0 4
&
0 1 y
0 2 y
1 3 z
1 2 y
3 3 z
3 2 y
2 4 z
\$

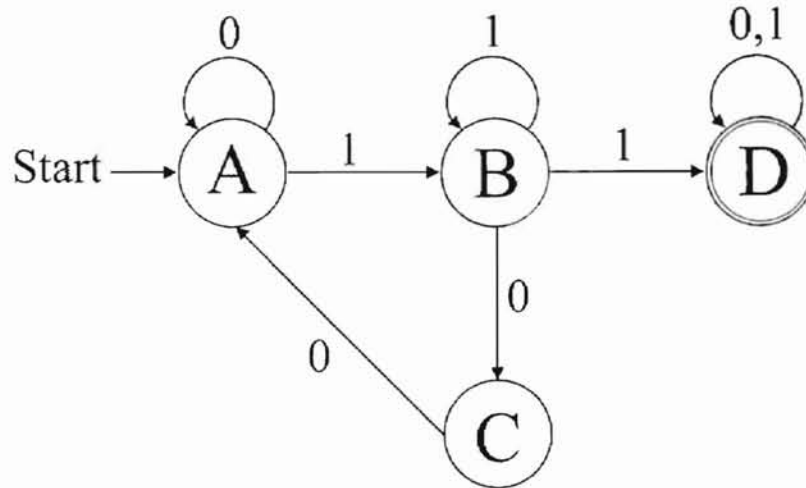
Test Case FSA4



Input Format

```
# 1 10  
0 4  
&  
0 1 0  
1 2 0  
2 2 0  
2 3 0  
2 4 0  
3 2 1  
3 4 1  
4 4 1  
$
```

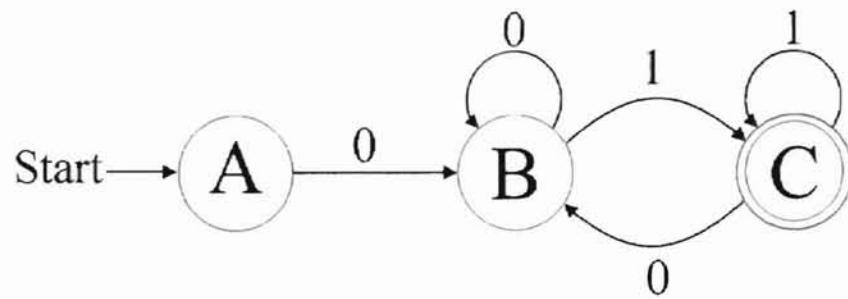
Test Case FSA5



Input Format

1 10
A D
&
A A 0
A B 1
B B 1
B C 0
C A 0
C D 1
D D 0
D D 1
\$

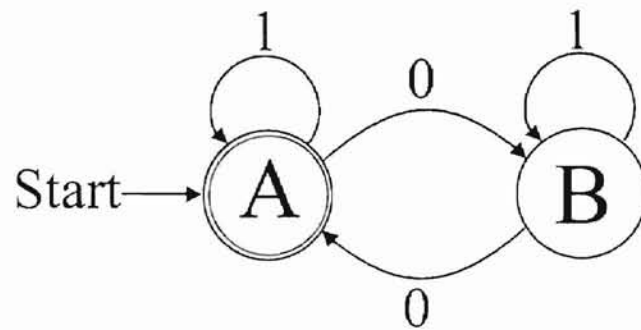
Test Case FSA6



Input Format

1 10
A C
&
A B 0
B B 0
B C 1
C B 0
C C 1
\$

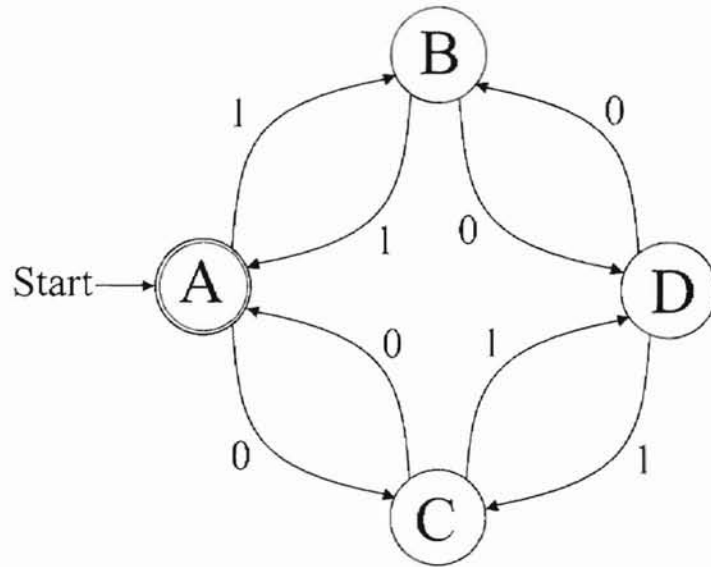
Test Case FSA7



Input Format

1 10
A A
&
A A 1
A B 0
B A 0
B B 1
S

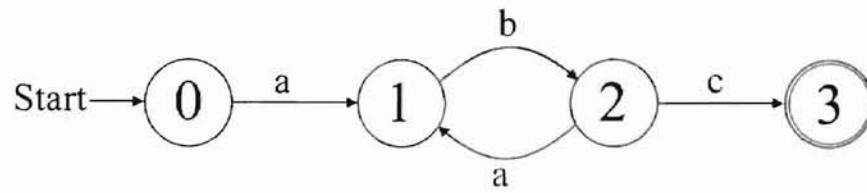
Test Case FSA8



Input Format

1 10
A A
&
A B 1
A C 0
B A 1
B D 0
C A 0
C D 1
D B 0
D C 1
\$

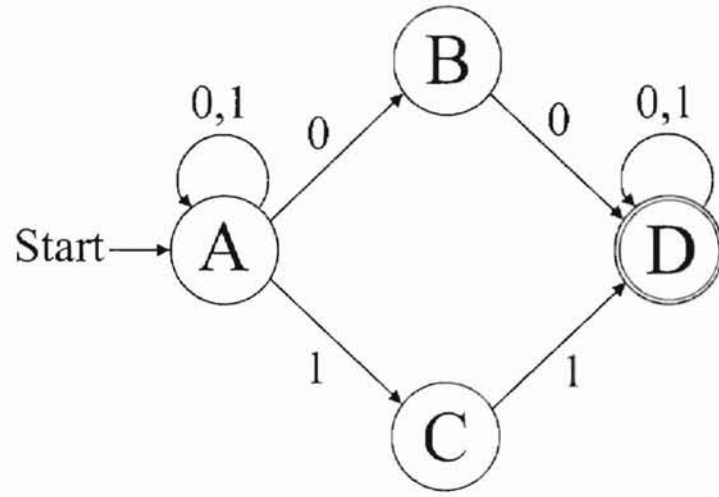
Test Case FSA9



Input Format

```
# 1 10  
0 3  
&  
0 1 a  
1 2 b  
2 1 a  
2 3 c  
$
```

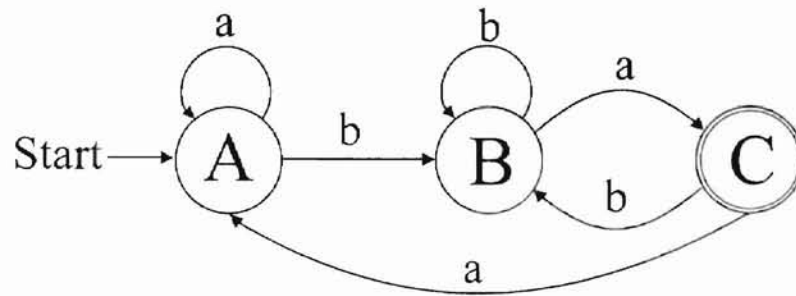

Test Case 10



Input Format

2 10
A C E
&
A A 0
A A 1
A B 0
A C 1
B D 0
C D 1
D D 0
D D 1
\$

Test Case 11



Input Format

1 10
A C
&
A A a
A B b
B B b
B C a
C B b
C A a
\$

APPENDIX D
TRAINING SET

The Complete Training Set from FSA1 with String Length of 10

pttttttv	pvpxttv	tsxxttvps
ptttttvps	pvpxtvps	tsxxttvv
ptttttv	pvpxtvp xv	tsxxtvps
ptttttvps	pvpxtv	tsxxttv
pttttv	pvp xvps	tsxxtvps
ptttvps	pvp xvpxtv	tsxxtvp xv
ptttvp xv	pvp xvpxvps	tsxxtv
ptttv	pvp xvpxv	tsxxvps
ptttvps	pvp xv	tsxxvp xt v
ptttvp xt v	pvv	tsxxvp xvps
ptttvp xvps	tssssssxs	tsxxvp xv
ptttvp xv	tssssssxs	tsxxv
ptttv	tssssssxs	txs
pttvps	tsssssx xv	txxttttv
pttvpxttv	tssssxs	txxtttvps
pttvpxtvps	tssssx tv	txxtttv
pttvpxtv	tssssx xvps	txxttvps
pttvpxvps	tssssx xv	txxttv
pttvpxv	tssssxs	txxtvps
pttv	tsssxxtv	txxtvp xv
ptvps	tsssxxtvps	txxtv
ptvpxttv	tsssxxtv	txxtvps
ptvpxttvps	tsssx xvps	txxtvp xt v
ptvpxttv	tsssx xv	txxtvp xvps
ptvpxtvps	tssxs	txxtvp xv
ptvpxtv	tssxxttv	txxtv
ptvpxvps	tssxxtvps	txxvps
ptvpxvpxv	tssxxtv	lxxvp xt v
ptvpxv	tssxxtvps	lxxvp xt vps
ptv	tssxxtv	lxxvp xt vps
pvps	tssxxvps	lxxvp xv
pvpxtttv	tssxxvp xv	lxxvps
pvpxttvps	tssxxv	lxxvp xt v
pvpxttv	tsxs	lxxvp xt vps
pvpxtvps	tsxxtttv	lxxvp xv
		lxxv

VITA²

Lee Yong Tan

Candidate for the Degree of

Master of Science

Thesis: AN APPLICATION OF N-GRAM SELF-ORGANIZING MAPS

Major Field: Computer Science

Biographical:

Personal Data: Born in Kuala Ketil, Kedah, Malaysia, January 25, 1972, the son of Sow Hee and Geek Keaw Tan.

Educational: Graduated from Jit Sin High School, Bukit Mertajam, Penang, Malaysia in December 1991; received Bachelor of Science degree in Computer Science from Oklahoma State University, Stillwater, Oklahoma in December 1994; completed requirements for the Master of Science degree at Oklahoma State University in May 1999.

Experience: Employed by Oklahoma State University, Computer Information Services as a computer lab assistant, September 1996 to May 1997.