T.C.
MARMARA UNIVERSITY
INSTITUTE FOR GRADUATE STUDIES IN
PURE AND APPLIED SCIENCES

CYCLE DETECTION IN NOISY SIGNALS BY CONSTRUCTIVE AUTOMATA: AN ADAPTIVE SYNTACTIC APPROACH TO PATTERN RECOGNITION

Aleksei USTIMOV

(Computer Engineering, MSc.)

THESIS

FOR THE DEGREE OF MASTER OF SCIENCE

IN

COMPUTER ENGINEERING PROGRAMME

SUPERVISOR
Yrd.Doç.Dr. M. Borahan TÜMER

İSTANBUL 2006

T.C.
MARMARA UNIVERSITY
INSTITUTE FOR GRADUATE STUDIES IN
PURE AND APPLIED SCIENCES

CYCLE DETECTION IN NOISY SIGNALS BY CONSTRUCTIVE AUTOMATA: AN ADAPTIVE SYNTACTIC APPROACH TO PATTERN RECOGNITION

Aleksei USTIMOV, MSc.
(141100320030133)

THESIS
FOR THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING PROGRAMME

SUPERVISOR
Yrd.Doç.Dr. M. Borahan TÜMER

İSTANBUL 2006

ACKNOWLEDGEMENT

First, I want to thank my advisor Asst.Prof. M. Borahan TÜMER for his guidance and support in development of thesis application and in the completion of this thesis.

Secondly, I thank all my friends for their understanding. Special thanks go to my school friend Hürkan BAHÇETEPE for his contributions to this thesis and offering to use the computational power of his computer.

Finally, it is my pleasure to thank my family and my fiancée for their endless support and belief in me and my work.

January, 2006 **Aleksei USTIMOV**

CONTENTS

ÖZET

BÜYÜYEN ÖZDEVİNİR İLE GÜRÜLTÜLÜ SINYALLERDE DÖNGÜ BULMA: SÖZDİZİMSEL ÖRÜNTÜ TANIMAYA, UYUMLU YAKLAŞIM

Yapay zekanın günümüzdeki en önemli parçalardan biri sözdizimsel örüntü tanımadır. Sonlu özdevinirin bulunmasından sonra örüntü tanıma büyük derecede gelişme göstermiştir. Örüntü tanıma sorunların çoğu için özdevinir elle kurulabilir ve ayarlanabilir; fakat teknolojideki gelişimler daha karmaşık sorunları çözebilmek için gelişmiş araçları sunar. Bu sorunlar için özdevinirin elle kuruluşu kullanışsız ve zaman alıcı bir iştir. Ayrıca, bu sorunların çoğu gürültülü ve eksik veriler ile başa çıkmayı gerektirir. Bu tip veriler için güçlü ve uyumlu analiz aracına ihtiyaç doğar. Böyle bir verilerin analizi için büyüyen özdevinir kullanışlı bir araçtır. Özdevirinin kurulması tek durumdan başlar ve, büyüme sırasında kullanılmış ham veriden öğrenilen örüntüyü tanıyabilen, son durumunu alan özdevinirle tamamlanır. Bu yaklaşım, ham verilerin büyük miktarlarda kolaylıkla elde edilebildiği, ve sadece gürültülü değil, zamanı ve yapıyı değişenler sorunlar için uygundur.

Bu çalışmada anlatılan yaklaşım tek boyutlu yapıları irdelendi. Özdevinir, sunulan veri dizisinde saklı olan örüntüyü tanımak için geliştirildi. Yaklaşımda, gürültülü dizi üreteç tarafından üretilen ilkellerin sentetik dizisi kullanıldı, çünkü bu çalışmanın amacı gerçek dünya verileri tanımakta kullanılmak üzere tasarlanmış bu yöntemin başarımını ölçmektedir.

Ocak, 2006 **Aleksei USTIMOV**

ABSTRACT

CYCLE DETECTION IN NOISY SIGNALS BY CONSTRUCTIVE AUTOMATA: AN ADAPTIVE SYNTACTIC APPROACH TO PATTERN RECOGNITION

Syntactic pattern recognition is one of the today's most important titles in artificial intelligence. Pattern recognition has developed a great deal after invention of the finite automata. Automata can be created and tuned manually for most problems in pattern recognition; but advances in technology provide powerful tools for solving more complicated problems. Manually creating an automaton for use in such problems is a cumbersome and time consuming job. Many of these problems require dealing with a diversity of noisy and imperfect structures of data. This type of data arises the need for a robust and adaptive medium of analysis. Constructive automata are useful tools for analysis of such data. Construction of an automaton starts with a single state and ends up with a full automaton that is able to recognize a pattern learned from the raw data presented during the construction. This approach suits best those problems where raw data is easily available in vast amounts, and it is not only noisy, but also subject and time-varying.

The approach discussed in this work assumed patterns with 1-D sequential structures. The automaton was constructed for recognition of a specific pattern (cycle) that was hidden in presented data sequence. In this work we used synthetic sequences of primitives, generated by a noisy sequence generator, since our goal was to determine the performance of the method which will ultimately be used in recognition of real world data in the future.

January, 2006 **Aleksei USTIMOV**

CLAIM FOR ORIGINALITY

CYCLE DETECTION IN NOISY SIGNALS BY CONSTRUCTIVE AUTOMATA: AN ADAPTIVE SYNTACTIC APPROACH TO PATTERN RECOGNITION

This work presents a novel approach to syntactic pattern recognition (SPR), called *adaptive syntactic pattern recognition* (ASPR). The novelty of the approach is the changes in the training section of the syntactic pattern recognition. This approach merges the "grammatical inference" and the "automata construction" modules into one module, which constructs an automaton and, at the same time, adapts it to the input data. Algorithm of the automaton construction is a general purpose algorithm that may be used for data of the same structure.

Ocak, 2006 **Yrd.Doç.Dr. M. Borahan TÜMER** **Aleksei USTIMOV**

LIST OF SYMBOLS

$\|C\|$	: cycle length
$\|\Sigma\|$	: alphabet length
1–D	: one dimensional
2–D	: two dimensional
3–D	: three dimensional
c	: a matrix of response probabilities
C	: cycle
D_{ij}	: any token sequence
$F(.,.)$	: a stochastic transition function
$H(.,.)$	: a stochastic output function
L	: learning algorithm
L_{R-I}	: learning algorithm based on linear reward-inaction scheme
$O(.)$	: average runtime function
$Pr(.)$	: probability of state or transition
Q	: periodic sequence
S	: number of states
α	: a set of actions
β	: a set of environment responses
$\delta_{\varphi_i\varphi_j}^{\tau_k}$	: transition (from φ_i to the φ_j with τ_k)
δ^+	: selected transition
Δ	: set of transitions
Δ^+	: added set of improbable transitions
Δ^-	: removed set of improbable transitions
φ	: state
φ^+	: added improbable state

φ^{-} : removed improbable state

$\boldsymbol{\Phi}$: set of states

$\boldsymbol{\lambda}$: constant learning parameter

$\boldsymbol{\nu}_{\varphi_i}$: state frequency

$\boldsymbol{\nu}_{\varphi_i\varphi_j}^{\tau_k}$: transition frequency

$\boldsymbol{\Sigma}$: token alphabet

$\boldsymbol{\tau}$: token

LIST OF ABBREVIATIONS

CAS : Currently Active State

CNC : Cycle Neighborhood Coefficient

ECG : Electrocardiogram

EEG : Electroencephalogram

FSLA : Fixed Structure Learning Automaton

LA : Learning Automata

MCL : Maximum Cycle Length

NAS : Next Active State

OCR : Optical Character Recognition

SNC : State Neighborhood Coefficient

SR : Speech Recognition

SPR : Syntactic Pattern Recognition

TNC : Transition Neighborhood Coefficient

VSLA : Variable Structure Learning Automaton

XML : Extensible Markup Language

LIST OF FIGURES

LIST OF TABLES

PART I

INTRODUCTION AND OBJECTIVES

I.1. INTRODUCTION

Pattern recognition is one of today's most important titles in artificial intelligence. Decision–making algorithms require as much certain information about an environment as possible to make better decisions. This information is extracted from data collected by sensor devices. In most cases extracting such information means recognizing a pattern which may be an object in a picture, structure in a sequence, etc. Earlier, when the concept of automaton was not invented yet, designing and implementing a pattern recognition system was a difficult task, requiring the involvement of high quality personnel. After the invention of the finite automata, basic pattern recognition problems, like parsing an XML or constructing a text search algorithm, became daily tasks [1]. Also, the latest advances in the computer technology made complex pattern recognition problems such as face, fingerprint, speech or handwritten letter or word recognition possible to solve within reasonable time intervals.

Syntactic approaches to pattern recognition require detection of patterns from a given relevant dataset [6-8]. The structure of the raw data varies depending upon the type of application. For example, medical recordings received from various parts of the human body (ECG, EEG recordings) come as a sequence of low level voltage values one at a time [13]. Hence, the dataset is a 1-D sequence. But if the problem is to recognize a fingerprint or human face, the raw data are the 2-D images. Converting them to 1-D arrays will result in a partial loss of information or increase in complexity. That's why a graph or a tree suits better for representing such data.

I.2. OBJECTIVES

In general, syntactic approach to attacking problems using sequential structures of data involves two phases [7]:

1. Transformation of the original structure of data into a structure of the elements of a token alphabet, called the *feature extraction phase*.
2. Search within the structure for specific patterns called the *recognition (parsing) phase*.

Before the feature extraction can take place, data have to be converted to the language an automaton can understand. A primitive representing a specific class of frequently occurring data segments with elementary structures, replaces all occurrences of such a structure that best matches with the corresponding class definition according to any error function. This process is called *quantization* or *classification* [8]. Primitives are sometimes called *tokens* and the set of tokens is called the *token alphabet*. The construction of the token alphabet (classification) may involve either some adaptive or a less complex non-adaptive procedure to define each token in the alphabet. Once the token alphabet is constructed, the sequence of raw data is decomposed into segments of data values and each segment is replaced by the token in the alphabet that provides the best match. So the output of classification is a sequence of tokens. Features may consist of one token or be elementary structures of tokens. In this work, each feature is assumed to consist of one token so the term *token* will be used instead of the term *feature* [6]. Moreover the classification procedure is beyond the scope of this work. Noisy Sequence Generator produces a token sequence which is assumed to be an output of the classification procedure.

The second phase is the search for specific patterns within the token sequence. That phase is accomplished by an automaton that is designed to accept only specific token sequences or patterns. The construction and adjustment of such an automaton is a critical step that is performed manually in many works in the literature [11, 14, 18]. In most cases, patterns may be partially existent due to the noise in data. This makes a manual construction of an automaton an even more difficult task. The objective of this work is to

develop an algorithm for automatic construction of an automaton that is able to recognize a repeating pattern in the given sequence and distinguish between noise and real data.

This work presents a novel approach to syntactic pattern recognition (SPR), called *adaptive syntactic pattern recognition.* Non-learning systems, like XML or HTML parsers, use predefined grammars for the SPR. Those systems do not require a training session. The systems that do, use specifically designed algorithms for "grammatical inference" and the "automata construction" modules in the training section of the SPR. For example, the text search is a SPR problem that requires a training section to construct an automaton for detecting short strings in a large text [1].

The novelty of the approach is the changes in the training section of the syntactic pattern recognition. This approach merges the "grammatical inference" and the "automata construction" modules into one module, which constructs an automaton while the adapting it to the data presented. The algorithm of the automaton construction is a general purpose algorithm that may be used for data of the same structure. This work uses sequential structure of data to obtain the *variable structure learning automaton* (VSLA) [9, 10]. Because of the noise present in the input data constructed automaton cannot be directly used in the parsing phase. The automaton is presented to the Noise Remover, which detects and removes noise and makes an automaton ready for the parsing phase. Cycle detector checks the results of the Noise Remover. It uses a second pass through the same sequence and tries to detect clean cycle using an automaton that was a result of the noise removing. Cycle detecting process is a recognition section of the SPR.

PART II

GENERAL BACKGROUND

II.1. LEARNING AUTOMATA

Today's technology provides ways to store, collect and exchange vast amounts of raw data. Digital sensors of all types help us collect the data. Networks are responsible for transferring that data to a storage place and filling data storage devices with enormous storage capacity. If it was known exactly what rules generated that data – no data would be needed and it would be more appropriate to directly use the formulas and logic of the source to make predictions.

The raw data is not generated randomly. The source system uses some unknown logic to create events that are recorded as data. Sometimes identifying the complete logic of that system may not be possible, still certain patterns or regularities of acceptable accuracy can be detected [2].

Machine learning principles are applied in many fields. In finance, banks analyze their past data to build models to use in credit applications, fraud detection and the stock market. In manufacturing, learning principles are used for optimization, control and troubleshooting. In medicine, learning programs are used for medical diagnosis. In telecommunications, call patterns are analyzed for network optimization and maximizing the quality of service. In science, large amounts of data in physics, astronomy and biology can only be analyzed fast enough by computers. In internet, searching for relevant information cannot be done manually [2].

Machine learning is a part of artificial intelligence. In a changing environment a system, built using learning rules, should have the ability to learn and adapt to changes.

The main advantage of such a system is that the designer does not have to worry about each possible case that environment can produce.

The concept of Learning Automata (LA) operating in an unknown environment came out of combined work of psychologists (modeling observed behavior of being known to be intelligent like humans and animals), statisticians (modeling the choice of possible actions based on past observations and tryouts), operations researchers (implementing optimal strategies) and system theorists (finding rational decisions in random environments) [4].

The concrete, analytical concept was initially introduced by Tsetlin. He considered the learning behaviors of finite deterministic automata in a stationary environment and proved that they give better results than actions based on a random guess. The study of learning behaviors and abilities of automata was continued by Varshavskii and Vorontsova, and has been done extensively by many researchers [3]. To understand more clearly what a LA operating in an unknown environment is, some definitions have to be introduced.

The term *environment* refers to a combination of all external conditions that may affect the automaton. Mathematically environment can be defined by a triple $\{\alpha, c, \beta\}$ where α represents a set of inputs, c a set of penalty probabilities and β a set of outputs (Fig II.1).

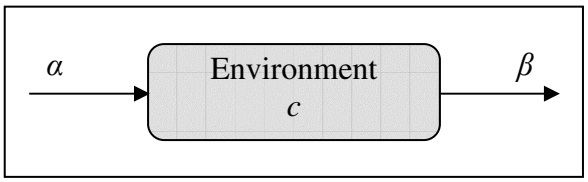

Figure II.1 Mathematical definition of the environment.

Input $\alpha(n) = \alpha_j$ is applied to the environment at discrete time n ($n = 0, 1, 2, \ldots$). The environment produces an output $\beta(n) = \beta_i$ using a value $c(\alpha, \beta) = c_{ij}$ from a set c in a following way:

$$\beta(n) = \max_{\beta_i} \bigcup_{i=1}^{r} \Pr\left(\beta(n) = \beta_i \middle| \alpha(n) = \alpha_j, c(n) = c_{ij}\right)$$

where r denotes the number of different output values in set β.

The concept of an *automaton* used in automata theory is a very general one applicable to a variety of abstract systems. Automaton is defined by the quintuple:

$<\Phi, \alpha, \beta, F(.,.), H(.,.)>$ where

- $\Phi = \{\varphi_1, \varphi_2, \ldots, \varphi_S\}$ is a set of states of size S,
- $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_r\}$ is a set of output actions of size r,
- $\beta = \{ \beta_1, \beta_2, \ldots, \beta_q\}$ is a set of inputs of size q,
- $F(.,.) : \Phi \times \beta \rightarrow \Phi$ a *transition function* that maps current state and the current input to the next state,
- $H(.,.) : \Phi \times \beta \rightarrow \alpha$ an *output function* that maps current state and the current input to the current output.

In this automaton the input and current state determine next state and output action. Such an automaton is called finite if sets Φ, α and β are finite.

Parameters of both functions $F(.,.)$, $H(.,.)$ are state and automaton input, and the outputs state and action respectively. Those functions determine their outputs using 3–dimensional matrices where first two dimensions are states and inputs, and last dimensions are states and output actions respectively. All entries of those matrices are values in [0, 1] and stand for probabilities. For example, to calculate an output function $F(.,.)$ selects an action that corresponds to the maximum value in the array $F[\varphi_i, \beta_j]$ in the matrix $F[\Phi, \beta, \Phi]$ where φ_i is the current state and β_j is the current input. This is same with function $H(.,.)$.

If the entries of transition and/or output function matrices change with time to improve performance, this automaton called *learning automaton* because the result of mentioned functions may not always be the same.

The connection of a learning automaton to the environment is shown in Fig. II.2. The output of an automaton at the time n is an action $\alpha(n)$ which at the same time is an input to the environment. In its turn, the response of an environment $\beta(n)$ is an input to an automaton.

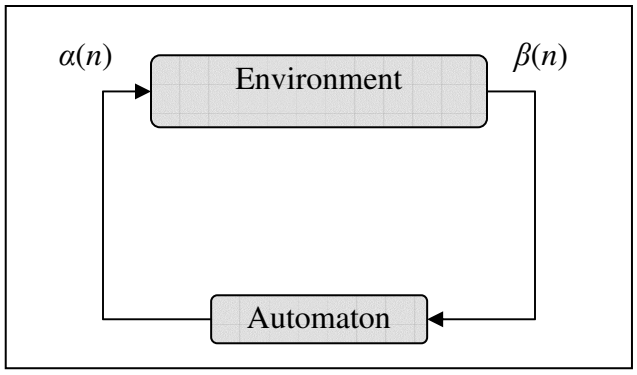

Figure II.2 Interconnection between a learning automaton and environment.

II.2. REINFORCEMENT LEARNING

There are three types of learning: *supervised*, *unsupervised* and *reinforcement*.

The aim of the supervised learning is to learn a mapping from the input to an output whose correct values are known [2]. This is the simplest and the most common used learning technique. Learning a mapping means to determine all entries of $F(.,.)$ and $H(.,.)$ function matrixes.

In unsupervised learning, correct output values for inputs are not known. The aim is to find regularities in the input where certain patterns occur more frequently than others. In statistics this is called *frequency estimation* [2]. The best example of unsupervised learning is the compression algorithms.

Reinforcement learning is learning what to do to maximize a reward. The agent (automaton) does not know which actions to take, instead must discover which actions yield the most reward by trying them. In some cases choosing an action may affect not only the immediate reward, but all the subsequent rewards as well. Agent targets the problem and develops ways to solve it trying different actions. Clearly, such an agent has to consider a state of the environment and take the actions that affect that state in a necessary way. The agent also must have a goal(s) relating the state of the environment. All reinforcement learning agents have explicit goals and can choose actions to influence their environments.

One of the dilemmas specific only to the reinforcement learning is a trade-off between *exploration* and *exploitation*. To receive maximum possible reward learning

agent must take actions that it tried in the past and found to be effective. On the other hand there are untried actions that may provide much better result. So the agent has to exploit already known actions and it also has to explore in order to find better actions in future.

Reinforcement learning contains following important terms: an *agent*, an *environment*, a *policy*, a *reward function*, a *value function* and a *model* [5]. Agent, which is an automaton, and environment were discussed before.

A policy is an agent's way of behaving. Given current state and the response of an environment agent must be able to determine next state and next action using a policy. In the other words a policy is $F(.,.)$ and $H(.,.)$ functions together. Policy is usually a set of matrixes of stochastic values, but sometimes it may involve extensive search and computation processes [2, 5].

A reward function is the goal of the reinforcement learning problem. An agent's objective is to maximize the received reward. A reward function defines what states, transitions and actions are good and bad for the agent according to the response received from the environment. For example if an action selected by the agent is resulted in low response (punishment) then the function updates the policy so that the policy selects other actions when in the same situation again. This also means that the function is unalterable by the agent.

While a reward function shows what is currently favorable, a value function shows what is favorable in a long time. A value function calculates a total amount of reward that can be expected starting from given state. For example a state might always receive low immediate reward, but still have high value because it is followed by states receiving high rewards. A value function does not affect the policy of an agent. It is used to evaluate a quality of the policy. However, it is much harder to determine values than rewards. Rewards are provided directly by the environment, but values must be calculated from he sequences of agent's observations.

Final element of reinforcement learning system is model of the environment. Model represents the behavior of the environment. For example, the model may predict the next state, reward and action given current state and the response value. Model is used for planning, meaning to decide which action to choose considering umber of possible

situations before they actually experienced. Planning and models is new addition to reinforcement learning paradigm. Earlier, the reinforcement learning systems worked only with trial-and-error methods what was almost the opposite of planning.

II.3. SYNTACTIC PATTERN RECOGNITION

Syntactic pattern recognition gained attention in 1980s and since then widely applied to many life recognition problems such as optical character recognition [18], fingerprint recognition [19], speech recognition [20], remote sensing data analysis, biomedical data analysis [14, 16, 21], scene analysis, texture analysis, 3-D objects recognition, 2-D mathematical symbols, chemical structures, etc.

In many pattern recognition problems structural information that describes the pattern is important so the syntactic methods have to be used. A pattern can be decomposed into simpler subpatterns. Each subpattern, in its turn, can be decomposed into even simpler subpatterns, and so on. The simplest subpatterns are called *primitives* or *terminals*. So a pattern can be represented as a structure of primitives. This structure can be list, graph, tree, matrix etc. After parsing the representation it becomes possible to assign a pattern to the correct class [6].

The block diagram showing the SPR system is shown in Fig. II.3. diagram consists of two parts: training and recognition [7]. Training part uses ready patterns as an input. First module of the training part is primitive (and relation) selection. This module decomposes the given digital pattern into primitives. Second module, grammatical inference, tries to determine the type of the pattern's structure. Usually this module's job is done manually if the complexity of the grammar is analytically tractable. This increases the quality of the recognition system and removes the need for training part. Last module of the training part is the automata construction. The main objective of those automata is to be able to parse (recognize) a pattern.

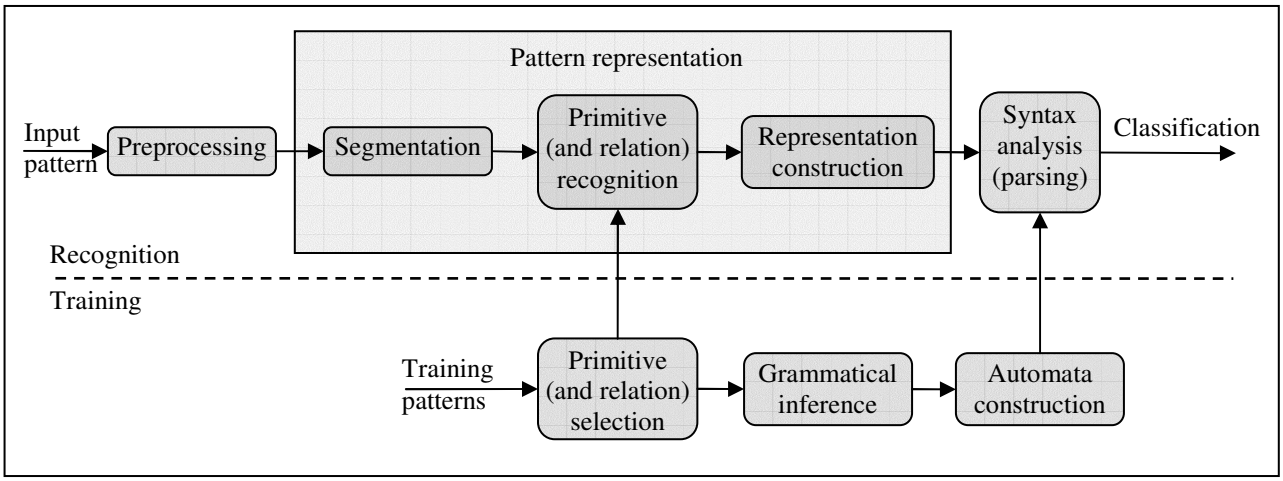

Figure II.3 Block diagram of SPR system.

Recognition part of the diagram includes three main steps. Preprocessing step is necessary because the input pattern is not in the ready-to-recognize format and has to be converted to before recognizing. Pattern representation step is responsible for extraction of primitives (and relations) from an input data and collecting them into a pattern representation using predefined structure (list, tree, graph, matrix, etc.). To achieve this raw input data must be divided into segments first (segmentation). Than, primitive recognition module extracts primitives (and relations) from each segment and passes them to the representation constructor, which constructs a digital representation of a possible pattern. Last step is the syntax analysis. Its main task is to parse (recognize) received digital structure and check if it belongs to a grammar constructed during training part by grammatical inference module.

PART III

CYCLE DETECTION

III.1. METHODOLOGY

The flow diagram containing main components is provided in Fig. III.1 to illustrate primary tasks accomplished in the method. Right after the start there is a "Noisy Sequence Generator" which produces noisy sequences of tokens. To create a sequence, the generator uses a token alphabet, specifies a cycle length and noise amount parameters which gives the user control over the generated sequence. Noisy sequences are used to simulate the environment with different behaviors. Once the "Noisy Sequence Generator" produces the periodic sequence (Q) with randomly injected noise of desired amount, it passes that sequence to the "Input Channel". "Input Channel" is a simple module that is responsible for feeding tokens of the sequence ($Q(t)$) one at a time to "VSLA Constructor" and "Cycle Detector". Beginning receiving tokens VSLA starts to be progressively constructed from a single state with no transitions to complete automaton adapted to the presented input. The construction process results in **V**ariable **S**tructure **L**earning **A**utomaton (VSLA). VSLA is a stochastic learning automaton of variable topology [4] with the capability of adapting itself to the presented token sequence via reinforcement learning. During its construction, VSLA also incorporates those states and transitions, among others in its structure, that originate from the noise in the sequence. To discover the potential cycle of the sequence, VSLA undergoes a noise removal process in the "Noise Remover" module where the states and transitions that represent noise are eliminated. "Noise Remover" is the system's most complicated module. At this point, when the construction of the VSLA is over and the structure of the automaton is not

subject to modification (i.e., not variable) anymore, VSLA changes to a fixed structure learning automaton (FSLA) [4]. Noise removal process produces one possible FSLAs (with its match score), capable of accepting its own specific potential cycle. Finally, FSLA is presented to the "Cycle Detector" to cast a cycle. "Cycle Detector" presents received tokens to FSLA and waits for the acceptance response. After receiving one "Cycle Finder" passes a short sequence of tokens that resulted in acceptance, i.e. cycle, to an "Output Channel". "Output Channel" simply passes received cycles to an output devise (screen, file etc.) in a user friendly form.

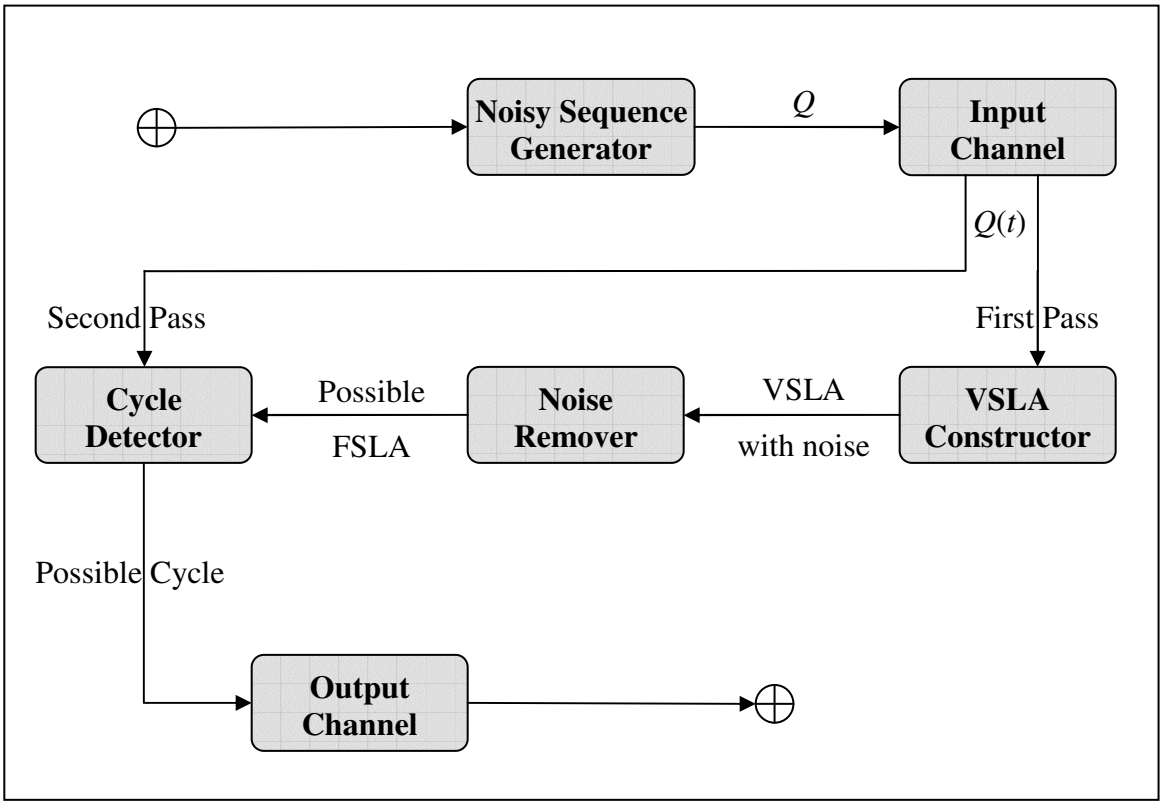

Figure III.1 Process flow diagram of the method.

III.2. NOISY SEQUENCE GENERATOR

To simulate a target environment a simple module called *Noisy Sequence Generator* is used. The generator produces noisy and periodic sequence of tokens, which is going to be used to construct VSLA.

Generator is a parametric module that allows the user to control all steps of sequence generation, starting from defining cycle and ending at the presentation of the generated sequence. During the sequence generation Noisy Sequence Generator requires following parameters:

1. the token alphabet,
2. the length of the cycle,
3. the length of the sequence in terms of the numbers of cycles,
4. the ratio in percent of noisy tokens to all tokens in the sequence,
5. the percent distribution of the three types of noise listed above.
    a. the ratio in percent of *replaced* tokens to all noisy tokens in the sequence,
    b. the ratio in percent of *removed* tokens to all noisy tokens in the sequence,
    c. the ratio in percent of *injected* tokens to all noisy tokens in the sequence.

First parameter specifies the token alphabet. The token alphabet is a set consisting of numbers, letters or any symbols. Alphabet is like a pool of tokens from which the generator selects tokens for a cycle or noise injection. Second and third parameters specify, in respective order, the number of tokens in the cycle and how many cycles the sequence should be consist of. First three parameters are enough to produce a noise-free sequence. The last two parameters describe the strategy of noise insertion. Fourth parameter specifies the percent amount of noise within the entire sequence that is to be randomly injected. The last parameter specifies the distribution of three noise types during the random injection.

First of all, the generator creates a cycle of desired length consisting of randomly selected tokens from the alphabet. After that, cycle is repeated a predefined number of times to generate a noise-free sequence. Finally, the generator randomly injects noise in the sequence. There are 3 types of noise:

a token or a sequence of tokens that

1. are missing to form a correct cycle – *replacement*,
2. replace a correct token within a cycle – *removal*,
3. the existence of which distort any cycle – *injection*.

First type is the replacement of token with the different one randomly selected from the alphabet. Second type is simply deleting a token from the sequence, and third type of noise is inserting a randomly selected token.

Token is denoted by τ. So an *alphabet* is defined as:

$$\Sigma = \{ \tau_k \mid k \in [1, K] \wedge \tau_i \neq \tau_j \text{, for } i \neq j \},$$

and the *cycle*, which is a short sequence of tokens, is:

$$C = \tau_1 \ldots \tau_P, \text{ where } \tau_p \in \Sigma \text{ and } p \in [1, P].$$

Further, *noise* is defined to be any token or sequence of tokens that distort any cycle within the generated random sequence. A noisy sequence may be defined as

$$Q = \bigcup_i D_{1,i} C D_{2,i} ,$$

where $i = 1, 2, \ldots$; $\cup$ standing for the concatenation operation, $D_{1,i}$ and $D_{2,i}$ denote any two sequences (including the empty sequence) of alphabet tokens other than C before and after the i^{th} cycle in the sequence, respectively.

In order for the "Cycle Detector" to be possible to detect a cycle in a periodic sequence correctly, that sequence must contain non-distorted cycle(s). The generator randomly chooses places for noise injection, which makes it possible to leave some cycles undistorted. Fig. III.2 provides the statistics of the sequence generator and illustrates the behavior of the percent count of non-distorted cycles as the cycle length and the amount of random noise in the sequence change. Noise-free sequence (Noise = 0)

contains “clean” cycles with probability of 1. The percent count of non-distorted cycles in the sequence tends to fall as the amount of noise rises. This tendency to fall is dramatic in longer cycles while it is relatively slower for shorter cycles. So, sequences with long cycles and high amount of noise must be large enough to have high probability of containing “clean” cycle.

The noisy sequence produced by the generator is then used in the process of constructing VSLA and discovering the possible cycle(s) after a second pass of the input sequence.

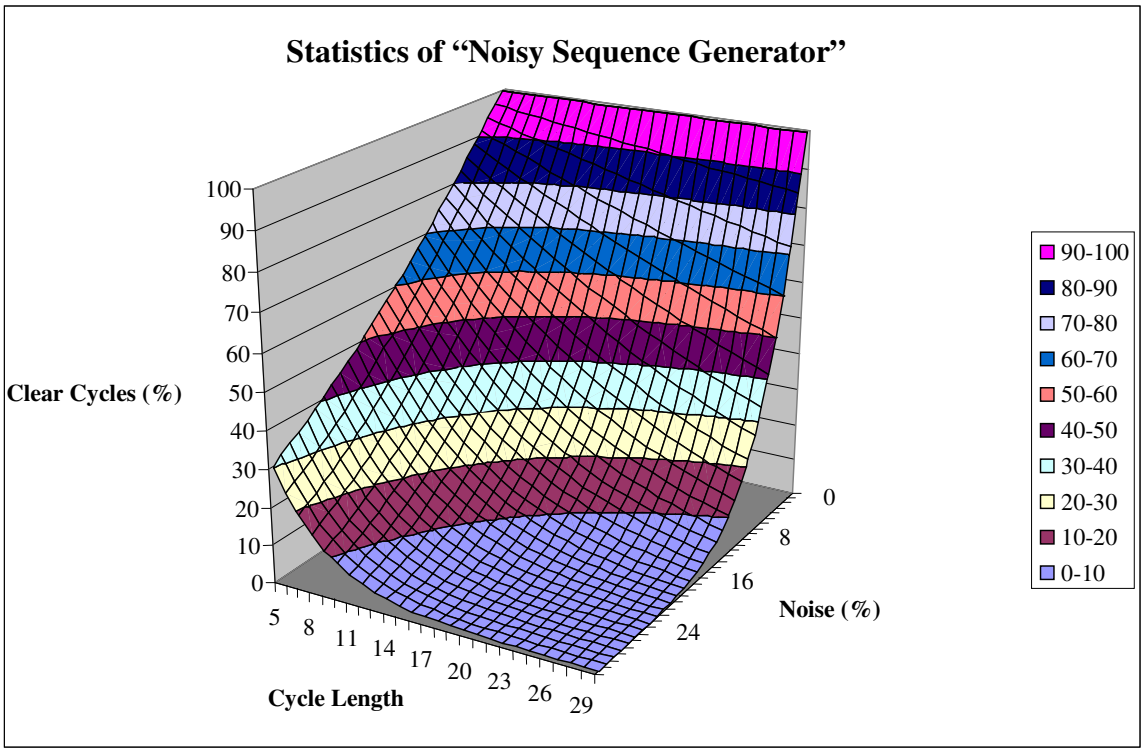

Figure III.2 Average values of non-distorted cycles in a generated sequence depending on the noise injected and cycle lengths.

III.3. VARIABLE STRUCTURE LEARNING AUTOMATON

This work is a constructive approach to building a variable structure learning automaton composed of a single state at the beginning. The construction of the variable structure learning automaton (VSLA) is based on two principles:

1. each input token presented carries VSLA from one state to another via transitions with adjustable probabilities;
2. the transitions and states, initiated by non-distorted tokens of a noisy sequence, are more frequently used than transitions initiated by noise.

Based on the first principle, structural components of VSLA (i.e., states and transitions) are inserted as necessary during the presentation of tokens from a noisy and periodic token sequence. The second principle is exploited to be able to identify and remove transitions that are initiated by noisy tokens. The concept of transition and state probabilities is used to distinguish between the transitions initiated by the real (i.e., not distorted) tokens of the sequence and those that are noise. Transitions and states that are more frequently used are rewarded by increasing the probability of the transition and the state that transition is initiated from.

Each state of VSLA can initiate transitions only with a specific token. The contribution of each token presented to VSLA is that if the token is presented to VSLA for the first time, a new state and transition are inserted. New state has no outgoing transitions and can initiate transitions only with the new token that was the reason of its insertion. New transition is initiated from the state that is associated with the previous token and uses to move VSLA from that state to the newly inserted state. If the token is not new (i.e. has been presented before) than the VSLA already contains the state associated to that token so only a new transition is inserted between the state associated with the previous token and the state associated with the new token (if a VSLA does not already contains such a transition). State that initiates inserted (or, if exists, selected) transition is called the *currently active state* (CAS). The state that the inserted (selected) transition leads to is called the *next active state* (NAS). The selected transition is rewarded, while all other transitions that initiate from the CAS are penalized. The CAS is rewarded as well, which results in penalizing all other states.

III.3.1. Definition of VSLA

The formal definition of an automaton is given above and stays that an automaton is a quintuple $<\Phi, \alpha, \beta, F(.,.), H(.,.)>$. VSLA, when moving from state to state doesn't

produce actions, which implies that α and $H(.,.)$ are not parts of the VSLA. Next, VSLA uses the inputs from an environment, which are tokens, so instead of an input set β VSLA uses token alphabet Σ. VSLA is a learning automaton and uses a learning algorithm L and learning speed constant λ so:

VSLA is a quintuple $<\Phi, \Sigma, F(.,.), L, \lambda>$

- $\Phi(n) = \{ \varphi_1, \ldots, \varphi_S \}$ is a set of states, representing temporal positions within the token sequence;
- $\Sigma = \{ \tau_1, \ldots, \tau_K \}$ is a set of tokens, the *token alphabet*;
- $F(.,.) : \Phi \times \Sigma \rightarrow \Phi$ is the stochastic transition relation mapping of the current state and token to the next set of states;
- L is the learning algorithm used to update the probability of each relevant state and transition upon receipt of each token;
- λ is a constant learning parameter.

Also, $\Delta(n) = \bigcup_{\tau_k \in \Sigma} \bigcup_{\varphi_i \in \Phi(n)} \bigcup_{\varphi_j \in \Phi(n)} \delta_{\varphi_i \varphi_j}^{\tau_k}$ is a set of transitions $\delta_{\varphi_i \varphi_j}^{\tau_k}$ where a transition $\delta_{\varphi_i \varphi_j}^{\tau_k}$ defines a move of VSLA initiated by k^{th} alphabet token τ_k from the source state φ_i to the destination state φ_j. Learning during the construction of VSLA occurs based on reinforcement learning principles. VSLA considers each token as a response produced by an environment. Each token is regarded as a reward for both the transition that have moved VSLA with the previous token to the NAS and the state that initiates the rewarded transition. The value of each transition and state is their probability. The learning algorithm L, is defined as follows:

Let φ_i, $i \in [1, S]$ be the state to be rewarded. Further, $\delta_{\varphi_i \varphi_j}^{\tau_k}$ be the transition deserving rewarding. Then the probability adjustments are accomplished based on linear reward-inaction scheme (L_{R-I}) [4] as follows:

States:

$$f_i(n+1) = f_i(n) + \lambda[1 - f_i(n)]$$
$$f_i(n+1) = [1 - \lambda] f_i(n)$$

Transitions:

$$f_{ij}^{\tau_k}(n+1) = f_{ij}^{\tau_k}(n) + \lambda\left[1 - f_{ij}^{\tau_k}(n)\right]$$
$$f_{il}^{\tau_k}(n+1) = [1 - \lambda] f_{il}^{\tau_k}(n) \qquad \forall l \neq j$$
$$f_{lr}^{\tau_m}(n+1) = f_{lr}^{\tau_m}(n) \qquad \forall r \in [1; S],\ \forall l \neq i \text{ and/or } \tau_m \neq \tau_k$$

where $f_i(n)$ is the probability of state φ_i at time n, $f_{ij}^{\tau_m}(n)$ is the probability of transition $\delta_{\varphi_i\varphi_j}^{\tau_k}$ and $\lambda \in (0; 1)$.

III.3.2. Construction of VSLA

Construction of VSLA from a noisy periodic sequence is relatively a simple and straightforward process. The algorithm of the construction of VSLA is given in Fig. III.3. VSLA is an automaton with an adjustable topology. In other words, it adapts itself to the input sequence. Starting with a single state, as its environment presents a sequence of tokens, one token at a time, VSLA undergoes an adaptation process by which VSLA extends its structure to be able to recognize a repeating pattern within the given sequence. VSLA is also a learning automaton with transition and state probabilities.

The construction of the VSLA is shown using an example sequence generated from a cycle C = (3 4 4 5 5 1). The token alphabet Σ = {1, 2, 3, 4, 5}. The initial structure of VSLA and step-by-step modifications to its structure upon the receipt of the first eight tokens are shown in Table III.1. Each line in Table III.1 illustrates the execution of the algorithm upon receipt of a token $\tau_{cur}(n)$ at the discrete time step n. Each column indicates the result of a single step in the algorithm in Fig. III.3 at the current time n. for simplicity the fifth step of the algorithm is not shown in the Table III.1. Probabilities of the generated transitions are given in the figures showing the topologies of the VSLA during the construction. Sum of the probabilities of all states and all transitions initiating from any state has to be equal to 1.

Algorithm

1. initialize
    a. VSLA by a start state a with probability 1 prior to the receipt of first token
    b. currently active state: none
2. receive current token τ_{cur}
3. identify next active state (NAS(n)) (i.e., the state from which transitions originate with τ_{cur} only, or, if not, the state with no transitions
4. if a currently active state (CAS(n)) exists, select the transition $\delta^+(n)$ between CAS(n) and NAS(n)
5. if a currently active state (CAS(n)) exists, generate reward to
    a. CAS(n) among all states
    b. selected transition $\delta^+(n)$ among transitions originating from CAS(n)
6. remove all improbable transitions $\Delta^-(n)$ from CAS(n)
7. remove the inaccessible improbable state $\varphi^-(n)$
8. mark NAS(n) as CAS(n + 1)
9. perform temporary structure modification
    a. insert a currently improbable state $\varphi_{imp}(n)$
    b. insert new currently improbable transitions $\Delta^+(n)$ from CAS(n) to all states (including the improbable state)
10. loop back to step 2 as long as there are tokens left in the sequence

Figure III.3 The algorithm for the construction of VSLA.

It is clear from the first line in Table III.1 that VSLA initially consists of a single state, the start state, a (Fig. III.4) which represents the machine at the time $n = 0$ prior to the receipt of the first token $\tau_{cur}(1)= 3 \in \Sigma$. Received by VSLA at $n = 1$ at step 2 in Fig. III.3 (2nd line of Table III.1), 3 cannot move VSLA anywhere. Therefore, a becomes NAS(1) at step 3. Steps 4 – 6 are skipped since CAS(1) has not been defined yet. Since a is not improbable, nothing is performed at step 7, either. At this point, VSLA still has a single state a. Step 8 is, where a is marked as CAS(n + 1). At steps 9(a) and 9(b), b is inserted as φ^+_{imp} and δ^3_{aa} and δ^3_{ab} are added to the structure of VSLA as $\Delta^+(1)$, respectively.

Table III.1 Values of the certain variables upon the receipt of first 8 tokens from a sequence of an example run.

Step			2	3	4	6	7	8	9a	9b
Discrete time n	Φ	CAS	τ_{cur}	NAS	δ^{+}	Δ^{-}	φ^{-}	NAS $\downarrow$ CAS	$\varphi^{+}{}_{imp}$	Δ^{+}
0	a	-	-	-	-	-	-	-	-	-
1	a	-	3	a	-	-	-	a	b	δ^{3}_{aa}, δ^{3}_{ab}
2	a	a	4	b	δ^{3}_{ab}	δ^{3}_{aa}	-	b	c	δ^{4}_{ba}, δ^{4}_{bb}, δ^{4}_{bc}
3	a, b	b	4	b	δ^{4}_{bb}	δ^{4}_{ba}, δ^{4}_{bc}	c	b	c	δ^{4}_{ba}, δ^{4}_{bc}
4	a, b	b	5	c	δ^{4}_{bc}	δ^{4}_{ba}	-	c	d	δ^{5}_{ca}, δ^{5}_{cb}, δ^{5}_{cc}, δ^{5}_{cd}
5	a, b, c	c	5	c	δ^{5}_{cc}	δ^{5}_{ca}, δ^{5}_{cb}, δ^{5}_{cd}	d	c	d	δ^{5}_{ca}, δ^{5}_{cb}, δ^{5}_{cd}
6	a, b, c	c	1	d	δ^{5}_{cd}	δ^{5}_{ca}, δ^{5}_{cb}	-	d	e	δ^{1}_{da}, δ^{1}_{db}, δ^{1}_{dc}, δ^{1}_{dd}, δ^{1}_{de}
7	a, b, c, d	d	3	a	δ^{1}_{da}	δ^{1}_{db}, δ^{1}_{dc}, δ^{1}_{dd}, δ^{1}_{de}	e	a	e	δ^{3}_{aa}, δ^{3}_{ac}, δ^{3}_{ad}, δ^{3}_{ae}
8	a, b, c, d	a	4	b	δ^{3}_{ab}	δ^{3}_{aa}, δ^{3}_{ac}, δ^{3}_{ad}, δ^{3}_{ae}	e	b	e	δ^{4}_{ba}, δ^{4}_{bd}, δ^{4}_{be}
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

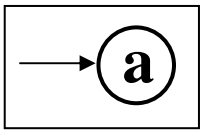

Figure III.4. The initial topology is a single state with probability 1 prior to the receipt of the first token.

Fig. III.5 illustrates the structure of VSLA at the end the processing of 9 steps at n = 1. Here, the states and transitions denoted by a dash circle and dash lines point out the improbable states and transitions that were inserted to extend the topology of the automaton. The transition that moves the VSLA to the correct state is rewarded and maintained while all other dashed transitions are removed. If the token is a duplicate of one of those already presented to the VSLA then the improbable state is also removed.

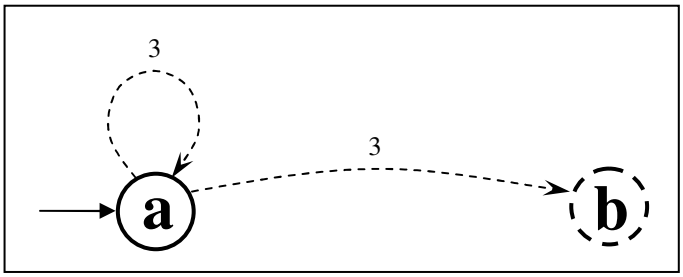

Figure III.5. After the presentation of the first token 3, VSLA still maintains its topology (the state *a* drawn with a solid line) until insertions of necessary structural components are performed.

With the next token $\tau_{cur}(2) = 4$ received at the 3rd line of Table III.1, b is identified to be NAS(2), since no other states in VSLA can move with 4. CAS(2) is a from the previous turn of the loop. At Step 4, transition δ_{ab}^{3} is selected and at Step 5, a and δ_{ab}^{3} are rewarded. Steps 6 and 7 are where δ_{aa}^{3} is removed. Next, b is marked to be the CAS(3). At Steps 9(a) and 9(b), improbable state c and a set of transitions $\{ \delta_{ba}^{4}, \delta_{bb}^{4}, \delta_{bc}^{4} \}$ are inserted as φ^{+}_{imp} and Δ^{+}, respectively. The resulting VSLA is illustrated in Fig. III.6.

The 4th line in Table III.1, upon receipt of the next token $\tau_{cur}(3) = 4$, sets NAS(3) = b and $\delta^{+} = \delta_{bb}^{4}$. Following the probability adjustment of both b and δ_{bb}^{4}, algorithm performs these assignments: CAS(4) = b, $\varphi^{+}_{imp}(3) = c$, and $\Delta^{+}(3) = \{ \delta_{ba}^{4}, \delta_{bc}^{4} \}$. The topology of VSLA at this point is shown in Fig. III.7.

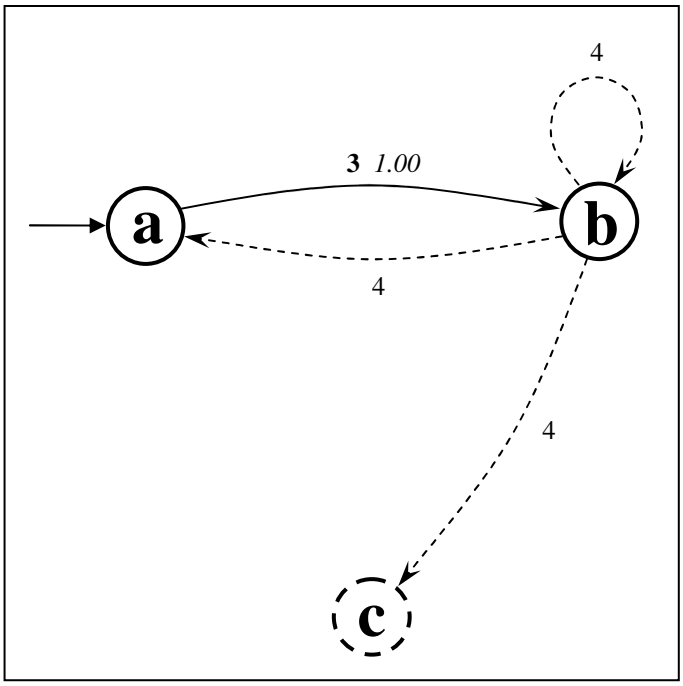

Figure III.6. Structure of VSLA after processing second token 4.

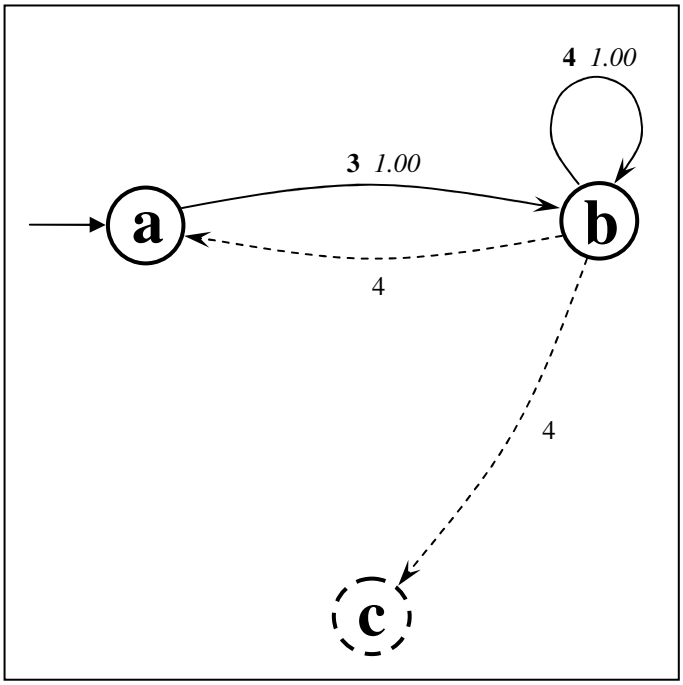

Figure III.7. VSLA structure after 4 has been processed and second 4 received.

The next token presented is $\tau_{cur}(4) = 5$ at line 5 in Table III.1. Since τ_{cur} is a new token (i.e., not a duplicate of those already presented), NAS(4) = *c* and $\delta^{+} = \delta^{4}_{bc}$. After the probability adjustment of *b* and δ^{4}_{bc} , $\Delta^{-}(4) = \{ \delta^{4}_{ba} \}$ (Step 6), CAS(5) = NAS(4) = *c*, $\varphi^{+}{}_{imp}(4) = d$ and $\Delta^{+}(4) = \{ \delta^{5}_{ca}, \delta^{5}_{cb}, \delta^{5}_{cc}, \delta^{5}_{cd} \}$. Now, VSLA is as shown in Fig. III.8.

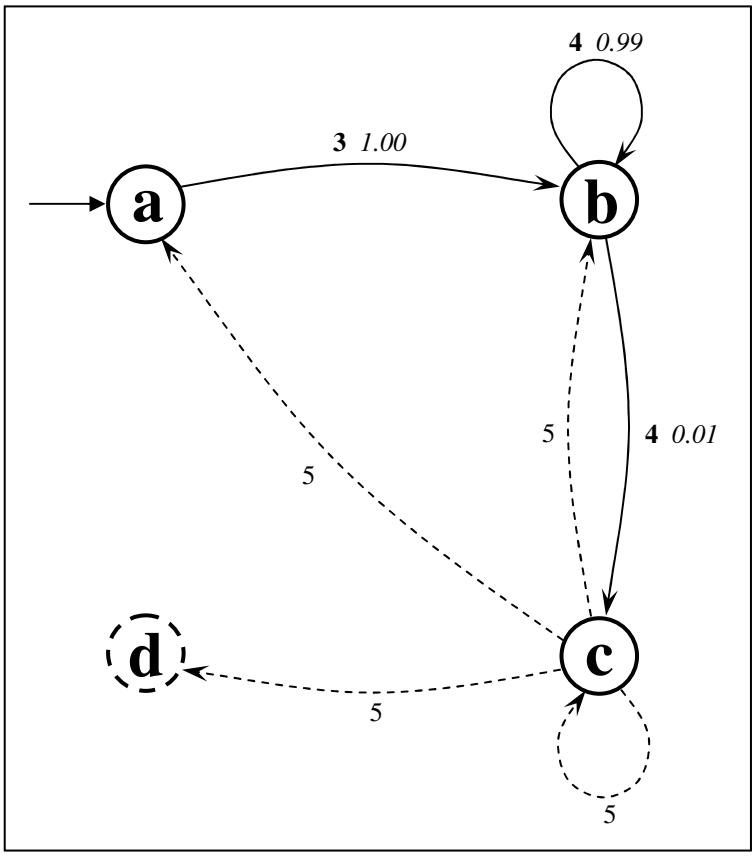

Figure III.8. VSLA topology follows after the first four tokens have been processed.

The next token $\tau_{cur}(5) = 5$ (line 6 in Table 5-2-2) is not a new one ($\tau_{cur}(5) = \tau_{cur}(4) = 5$) so NAS(5) = *c* again and s.o. VSLA structure at n = 5, 6 and 7 is shown in Fig. III.9, Fig. III.10 and Fig. III.11 respectively.

The process explained above for each of the first 6 lines in Table III.1 is the iteration repeating itself as long as there are tokens left in the noisy sequence of data. It may be seen that the construction is a simple and straightforward process. 50000-token sequence was used in the construction of the VSLA. Topology of the VSLA during the construction is illustrated in figures of Appendix B. The more tokens are presented from

the noisy sequence, the more complicated VSLA becomes. Noisy sequence generator injects noise randomly which results in presence of almost each possible combination of token pairs ($|\Sigma|^2$ = 25 different pairs or transitions), which, in its turn, creates such an effect.

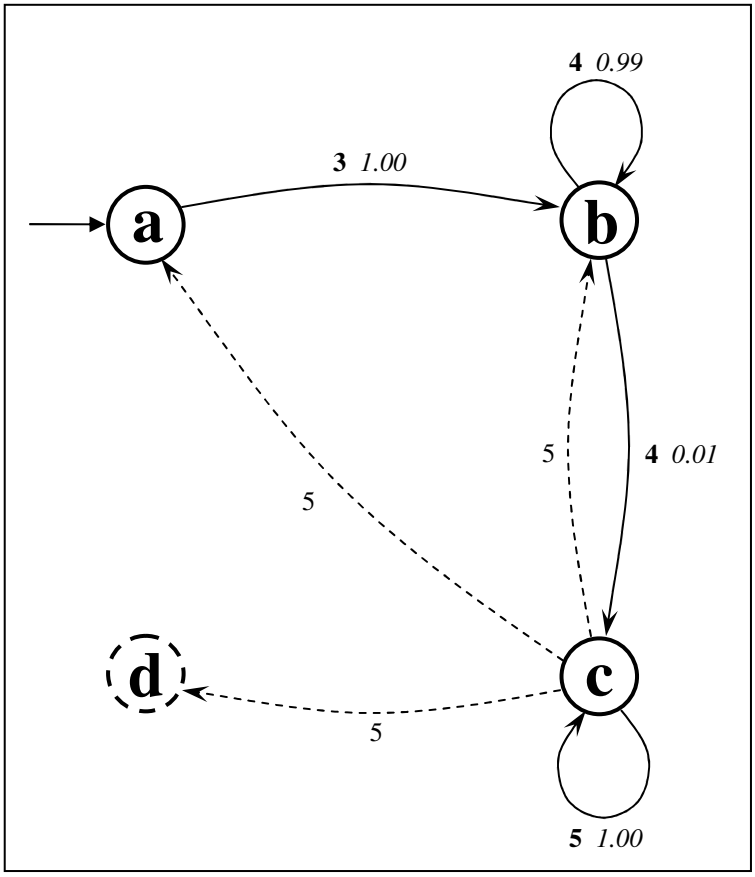

Figure III.9. Structure of VSLA following the process of the second 5.

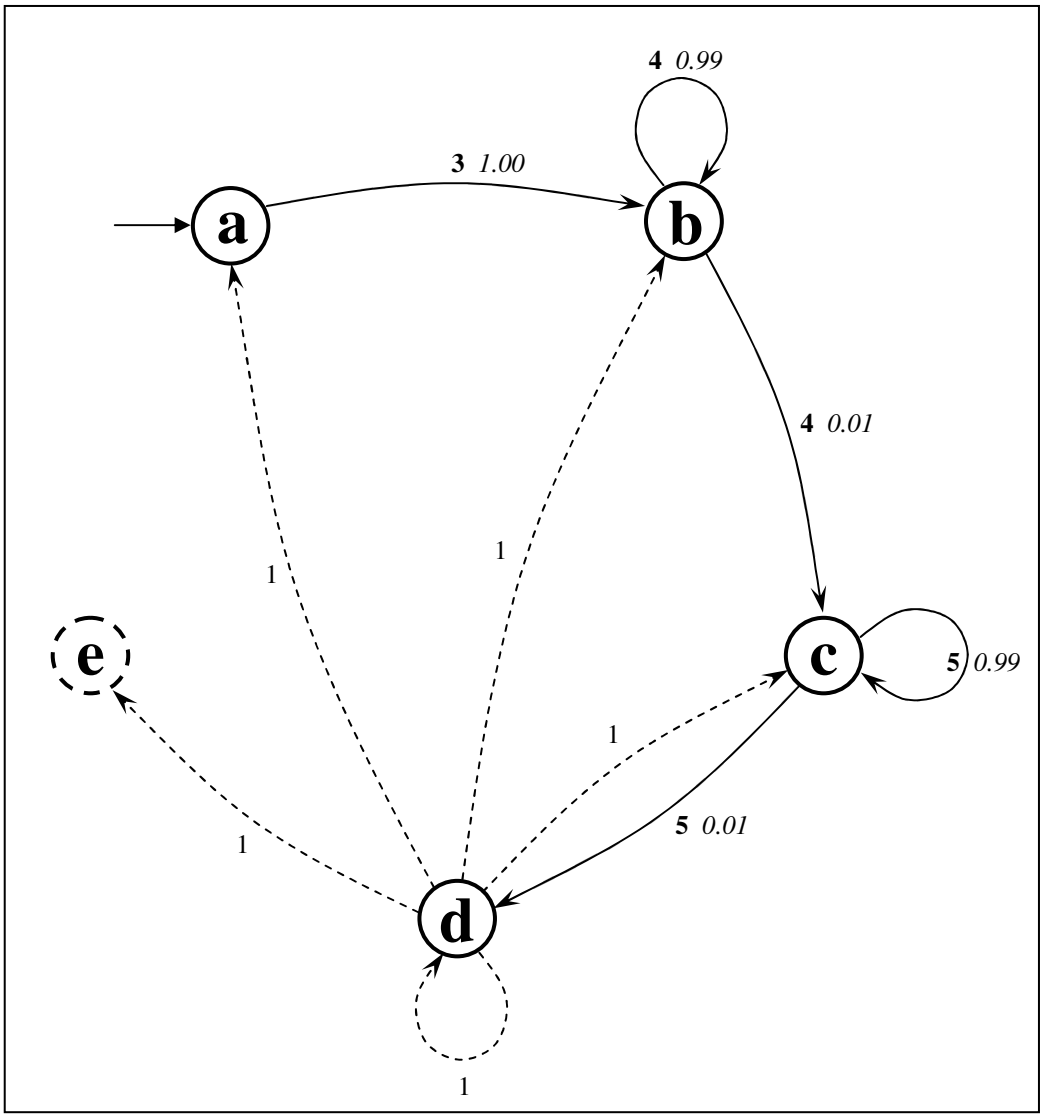

Figure III.10. VSLA topology after the last token of the first cycle presented and the necessary operations performed.

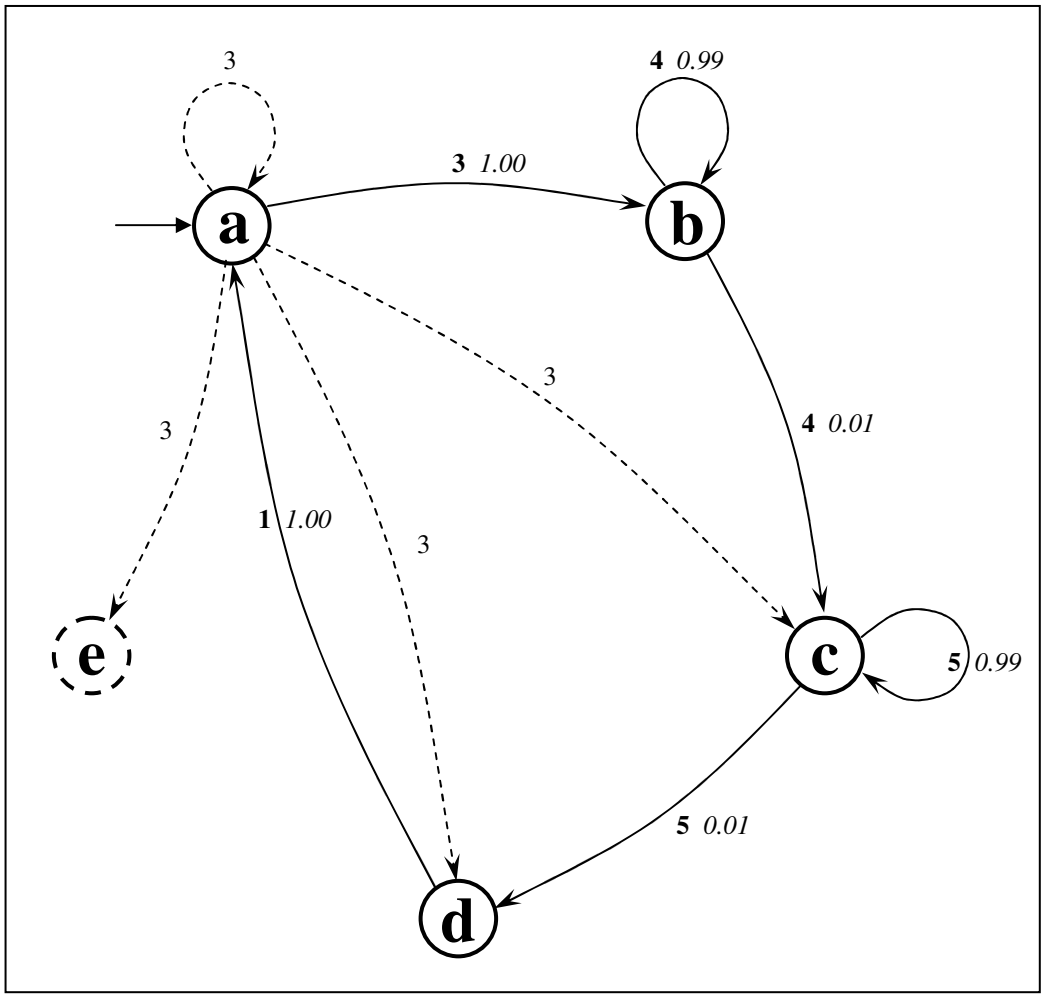

Figure III.11. Structure of VSLA following the process of the second cycle's first token 3.

III.4. NOISE REMOVER

VSLA is a stochastic automaton that results from the online construction and contains those structural components (i.e., states and transitions), among others, that represent noise within the input token sequence. For VSLA to recognize the noise-free cycle of the token sequence, the noisy states and transitions should be located and removed.

III.4.1. Frequencies

Before going any further several concepts which used in the process of how to discover and remove the noisy structural components have to be defined.

The number of times a transition $\delta_{\varphi_i\varphi_j}^{\tau_k}$ is traversed to complete a noise-free cycle of the input sequence presented is called the *transition traversal frequency* and symbolized like $\nu_{\varphi_i\varphi_j}^{\tau_k}$.

Similarly the *state traversal frequency,* which symbolized as ν_{φ_i} is defined to be the number of times a state φ_i is visited during a path traversal to complete a noise-free cycle of the input sequence presented.

These two definitions lead to some simple rules. First is that the state frequency has to be equal to sum of the frequencies of all transitions originating from that state:

$$\nu_{\varphi_i} = \sum_{j=1}^{S} \nu_{\varphi_i\varphi_j}^{\tau_k}$$

Second is that the sum of all state frequencies or all transition frequencies of VSLA provides the cycle length:

$$|C| = \sum_{i=1}^{S} \nu_{\varphi_i} = \sum_{i=1}^{S}\sum_{j=1}^{S} \nu_{\varphi_i\varphi_j}^{\tau_k}$$

III.4.2. Noise Removal Process

Main task of the Noise Remover is to find frequencies for all transitions and states. If the frequency is 0 it means that the transition or state is noisy. The frequency of a transition or state can be derived from its probability. After detecting frequencies for all states and transitions, noise remover removes all states and transitions that received the frequency of 0, normalizes resulting automaton and checks if that automaton is able to contain cycle information.

The algorithm of the Noise Remover is presented in Fig. III.12. Noise Remover is the most complicated part of the system which deals with vast amount of data. This is the reason why the operation of the Noise Remover may not be shown on a single example.

Instead of that it would be more appropriate to explain the work of each step of an algorithm on different separate examples.

Algorithm

1. remove all states (with transitions) whose probability is below the minimum value
2. calculate maximum possible value of ν_{φ_l} as
3. foreach ν_{φ_l} from 1 to calculated maximum value
    a. calculate possible frequency range for all states using given frequency of the φ_l
    b. match set of transitions, initiating from each state, to each frequency distribution of classes in the range calculated above
    c. create all possible FSLAs using the transition match results
    d. filter created FSLAs
    e. if there are FSLAs left return FSLA with maximum score and exit
4. loop back to step 3
5. remove φ_l (with transitions)
6. loop back to step 2 as long as Φ of VSLA is not empty

Figure III.12 The algorithm for the Noise Remover.

The goal of the first step of an algorithm, presented in Fig. III.12, is to eliminate the obvious noisy structure components before entering the complex calculations. Noise Remover has the constant value, which limits the area of search. That constant is called Maximum Cycle Length (MCL). Also there is another constant value, which is called Cycle Neighborhood Coefficient (CNC), broadening an area of search limited by MCL to a given extent. Given the values of MCL and CNC, the lowest probability of a state with frequency of 1 has to be 1 / (MCL * CNC). So the first step removes all states from VSLA whose probability is below the calculated threshold value. Removing a state, also results in removing all transitions that lead to or initiate from that state. After all removing operations have been done automaton is normalized. Normalization of an automaton is simply the normalization of probabilities of all states and all transitions initiating from each state to 1.

Second step is the beginning of search mechanism. The goal here is to determine the maximum possible frequency value of φ_l. The term φ_l denotes the state in VSLA with the smallest probability. For example: suppose that the MCL is 30 and CNC is 1.3 (30% neighborhood). Then the extended MCL is MCL * CNC = 30 * 1.3 = 39 and the minimum state probability is 1 / 39 ≈ 0.03. So the maximum frequency of the state φ_l, whose probability is $\Pr(\varphi_l) = 0.13$ (above the minimum state probability), is an integer value no more than 0.13 / (1 / 39) = 5.

The value calculated in step 2 provides an area of search. Frequencies of all states other than φ_l depend on the ν_{φ_i} . So, in the step 3a ranges of possible frequencies for all states are calculated. The probability of the state, say φ_i is higher than the probability of φ_l by definition. The frequencies are proportional to the probabilities so the frequency of φ_i is $\nu_{\varphi_i} = \nu_{\varphi_l} * \Pr(\varphi_i) / \Pr(\varphi_l)$. The values of all terms in this equation are known, but the result is not an integer and possibly wrong when rounded. Here algorithm uses another constant value, which is called State Neighborhood Coefficient (SNC). This constant states that the real value of ν_{φ_i} lies between ν_{φ_i} / SNC and ν_{φ_i} * SNC floored and ceiled respectively. For example: $\Pr(\varphi_i) = 0.29$, $\Pr(\varphi_l) = 0.11$, current value of ν_{φ_i} (provided in step 3 of the algorithm) is 2 and SNC = 1.3 (30% neighborhood). Then the probable value of ν_{φ_i} is 2 * 0.29 / 0.11 ≈ 5.27 and the range of frequencies for state φ_i according to the frequency of state φ_l is between $\lfloor 5.27/1.3 \rfloor$ and $\lceil 5.27*1.3 \rceil$ which is [4; 7]. Same procedure is applied to all states.

Step 3b is the key step in the Noise Removal process. This step uses the last constant value called Transition Neighborhood Coefficient (TNC). This constant is used to extend the probabilities of transitions in mathematical calculations. The exact usages of this constant are shown later. Step 3b requires two parameters. First is a set of transitions initiated from one state. Second is a range of frequency distributions for that state. Frequency distributions are the arrays of integer values sorted in the descending order. For example: 4:2:1, 2:1:1 and s.o. Sum of the elements of the frequency distribution gives the class of the distribution, as in the previous example first distribution's class is 7 (4 + 2 + 1) and second's 4 (2 + 1 + 1). First 6 classes of distributions are shown in the Table III.2. Frequency distributions are used to determine the frequencies for a set of transitions

that initiates from one state. For example: if state's frequency is 5 then the frequencies of transitions initiated from that state have to match one of the distributions in class 5. So the second parameter leads to the set of distributions, which may be matched to transitions from the first parameter. For example, suppose that the probabilities of the transition set are {0.49, 0.45, 0.03, 0.02, 0.01} and they are sorted in the descending order like frequency distributions. Also the frequency distributions range is [1, 3], which means that process will try to match each of the 6 distributions in classes 1, 2 and 3 to the probabilities set. Matching process starts with checking the size and separator ratio. First of all, if number of transitions is less than number of elements in distribution then this transition set may not be matched to such a distribution. After size check, the separator ratio check is performed. Suppose that the process tries to match transition set with probabilities given above to the distribution 1 (the only distribution in the class 1). It is obvious that the number of transitions is more than the distribution size ($5 \geq 1$), so the size check is passed. The ratio check is performed as follows: matching subalgorithm calculates 2 different values of ratios. First ratio value states that the probability of the first noisy transition must be less than the probability of the last valid transition divided to the corresponding frequency value in distribution and extended up by TNC. Using the example above, first noisy transition probability is 0.45, last valid transition probability is 0.49 and its corresponding frequency value in distribution is 1, than the first ratio value is $(0.49 / 1) * 1.5 = 0.74 > 0.45$. First ratio check is successful. Second ratio value states that the probability of the first noisy transition must be less than the probability of the last valid transition extended down by TNC: $0.49 / 1.5 = 0.33 < 0.45$. Second ratio check fails and this distribution is rejected. Trying the same to the 1:1, first and second ratio values will be 0.68 and 0.30 respectively. The probability of the first noisy transition for the distribution 1:1 is 0.03 which is less than both ratios, so the distribution 1:1 passes the separator control.

Next comes the pairs check. All transitions exceeding size of distribution are considered to be noise and neglected. Pairs are all the combinations of elements in given distribution. If the given distribution is 2:1 then only one pair (2-1) has to be checked, if it was 4:3:2:1 then 6 pairs {(4-3), (4-2), (4-1), (3-2), (3-1), (2-1)} had to be checked. The

main question here is: may transitions with probabilities of 0.49 and 0.45 be assigned frequency values of 2 and 1 respectively?

Table III.2. Frequency distributions of the first 6 classes.

1	2	3	4	5	6	...
1	2	3	4	5	6	...
	1:1	2:1	3:1	4:1	5:1	...
		1:1:1	2:2	3:2	4:2	...
			2:1:1	3:1:1	4:1:1	...
			1:1:1:1	2:2:1	3:3	...
				2:1:1:1	3:2:1	...
				1:1:1:1:1	3:1:1:1	…
					2:2:2	…
					2:2:1:1	…
					2:1:1:1:1	…
					1:1:1:1:1:1	…

Or more scientifically expressed question: is ratio 0.49 / 0.45 in the TNC neighborhood of the ratio 2 / 1. Ratio of probabilities equals to 1.09 and the TNC neighborhood of the ratio 2 / 1 is [1.33; 3], where TNC is assumed to be 1.5. It is clear that 1.09 is not in the mentioned neighborhood so this distribution that passed size and separator controls fails the pair ratio control. If all controls are applied to, for example, the distribution 1:1, then the result would be that this transition set fits to the given distribution. First two transitions receive frequencies of 1 and others are neglected as noise. Then function calculates *score* of the match. Calculation based on the determination of the *cos* value between two vectors in the *n*-dimensional space. Score value is calculated using the following formula:

$$score = \sqrt{\sum_{i=1}^{|D|}\left[\frac{\Pr(\delta_i) + D[i]}{\sqrt{\sum_{j=1}^{|D|}\Pr(\delta_j)^2} + \sqrt{\sum_{j=1}^{|D|}D[j]^2}}\right]^2} * \frac{1}{\sum_{k=1}^{|D|}D[k]}$$

Where δ_i denote the i^{th} transition of the sorted transition list, D is the distribution and $D[i]$ is the i^{th} element of the distribution. Score value of the first multiplier for the transitions given above and distribution 1:1 is 0.9998. Second multiplier of the score is used to reward distributions from smaller classes. After all calculations are over, given transitions may match to more than one distribution, so the matching process produces as a result many transition sets with their corresponding scores. Sometimes none of the distributions match. In this case algorithm in Fig. III.12 jumps from step 3b to step 3.

Step 3c is responsible for creating automata using the results of the transition matching: states and transition sets. The main goal of this step is to create all possible combinations of automata using each state and one of the matched transition sets associated to that state once for each automaton. For example: VSLA has 5 non-noisy states *a*, *b*, *c*, *d*, *e* and number of transition sets, matched to the distributions from the ranges of the relevant states, are 2, 3, 1, 5, 3 respectively. Numbers of transition sets for state, say *b*, mean that noisy transition set, containing transitions initiated from *b*, fit to 3 different frequency distributions. Hence, the subalgorithm involved in this step will use each transition set assigned to a state to create all possible combinations of automata. At the end, subalgorithm will produce 2 * 3 * 1 * 5 * 3 = 90 different automata for the example given above. Score of each automaton is the summation of scores of transition sets that automaton consists of.

Automata created in step 3c are filtered in step 3d. Filtering eliminates invalid automata. First control is a communication property. If an automaton contains a cycle then it must be possible to reach all states from any given state: $\Pr(\delta_{\varphi_i\varphi_j}^{(n)}) > 0 \ \forall i, j \in [1; S]$ and $n > 0$, where S is the number of states in an automaton and n is the path length. If there is a non-communicating pair of states this automaton cannot contain cycle and it is eliminated. Another property of an automaton containing a cycle is the frequency matching. The following property has to fit to each state:

$$\sum_{i=1}^{S} \nu\left(\delta_{ij}^{\tau_m}\right) = \sum_{k=1}^{S} \nu\left(\delta_{jk}^{\tau_n}\right)$$

Sum of the frequencies of transitions leading to a state must be equal to the sum of the frequencies of transitions initiating from that state. If not then this automaton has invalid structure and cannot contain a cycle.

Step 3e checks if any automaton left after the elimination step and if so, the automaton that owns the maximum score is the one this algorithm searches for.

If no automata found in step 3, then it means that current φ_l cannot be a real state as was assumed before. Step 5 removes φ_l with all its transitions and normalizes the VSLA.

Process continues until no states left in the VSLA (step 6). Pseudo code of the Noise Remover is given in Appendix C.

III.5. CYCLE DETECTOR

Cycle Detector is a simple module. It uses a second pass through a noisy sequence to detect cycle. Algorithm for Cycle Detector is shown in Fig. III.13. First of all it creates a buffer of data type "queue" with a size of a cycle that the FSLA may contain. Size of a cycle possibly stored in FSLA can be determined by summing up frequencies of all states or transitions. After that, Cycle Detector receives a token from an Input Channel, creates a copy of FSLA and adds that copy to the storage. Than, Cycle Detector presents received token to all copies of FSLA currently in storage and collects responses.

Algorithm

1. create buffer
2. receive current token τ_{curr}
3. create a copy of FSLA and add it to the storage area
4. present τ_{curr} to all FSLAs in storage
5. if one of FSLAs returned acceptance response
    a. pass buffer contents to Output Channel
    b. exit
6. remove FSLAs, that returned failure responses, from storage
7. insert τ_{curr} to the buffer
8. loop back to step 2 as long as there are tokens left in the sequence

Figure III.13. The algorithm for the detection of cycle.

Newly created FSLA marks a state that initiates transitions with current token as Currently Active State (CAS) and returns no response. This tells the Cycle Detector that FSLA is ready to receive another token. After receiving next token FSLA determines state that is associated with that token and marks it as the Next Active State (NAS). If a transition between CAS and NAS does not exists or its frequency value is 0 then FSLA returns failure response. If transition exists and its frequency is more than 0 then FSLA marks NAS as CAS and decreases the frequency of that transition. After that, FSLA checks frequencies of all transitions and if all are 0 except any transition that initiates from CAS with frequency of 1 then FSLA returns acceptance response and if not no response at all.

After receiving responses from all copies of FSLA, Cycle Detector searches for acceptance response and if it finds one it sends buffer contents to the Output Channel and exits because after acceptance response buffer will contain a short token sequence (cycle) that resulted in acceptance response by a copy of FSLA. If Cycle Detector finds no acceptance response it removes all FSLAs, which returned failure responses, from storage area and insert token to the buffer. This process continues until all tokens have been received.

PART IV

RESULTS

To assess the performance of VSLA in discovering the cycle of noisy periodic sequences, 100000 experiments were run with cycle lengths ranging within [5; 25] (with step 2), using sequences with noise percent ranging within [5; 60] (with step 5) and have observed the average recognition probability of VSLA.

The test results of the method are shown in Fig. IV.1. Each point in the mesh in Fig. IV.1 is an average of 1000 experiments. All experiments, were performed using a 10–element token alphabet, and a quite small learning parameter $\lambda = 0.01$ to approach to expected transition probabilities as close as possible. VSLAs constructed by the method display almost a full recognition capability (i.e., within [0.98; 1]) for all cycle lengths less than an alphabet length up to a level of 30% of noise. In general, VSLAs are able to discover cycles of sequences for all experimented cycle lengths and noise level less than 30% with a probability of not worse than 0.90. Increase in noise amount triggers a dramatic fall. The point on the graph in Fig. IV.1 with coordinates of $|C| = 25$ and noise 60% is 0. The reason for such a result is inability of Noisy Sequence Generator to generate a sequence with clean cycles. If VSLA works with smaller cycles ($|C| < 10$) it shows high recognition results even with 50% noise (the sequence where, in average, each second token is noisy). A final word on the size of token alphabet is that as token alphabets shrink, VSLA complexity grows.

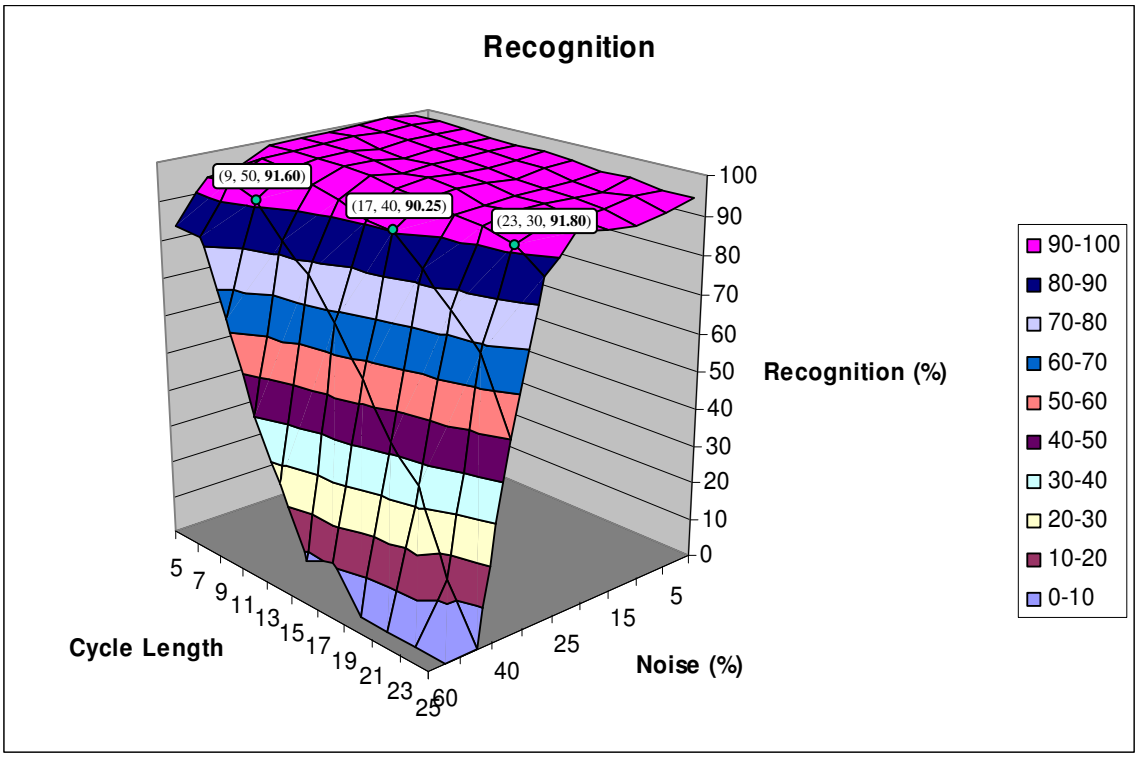

Figure IV.1. Illustration of results.

PART V

CONCLUSION

V.1. OVERVIEW

In this work, it is shown that, using the principles of reinforcement learning, VSLAs can be devised with a constructive approach to detect cycles or repetitive patterns in noisy periodic or partially periodic sequences.

V.2. FUTURE WORKS

However, some improvements can be made to the algorithm shown in this work. For example the values of the constants used in the Noise Remover directly affect the recognition probability, so their manual choice may not be the best; or this algorithm needs to be tested on the real world data; some special cases make it impossible for the Noise Remover to distinguish between two different cycles, so the algorithm of VSLA construction requires some changes; and the Cycle Detector's requirement for presence of clean cycles in the noisy sequence may not be necessary.

V.2.1 Determining Constants

Module that detects and eliminates noise (Noise Remover) is using some constant values to limit an area of search. Those constants (MCL, CNC, SNC and TNC) were determined manually and may not have the best possible values. Tuning the values of those constants may result in increase of the performance of the Noise Remover. One of the obvious methods is "brute force" – to try many possible values and chose the one that gives the best result.

V.2.2 Special Case

In certain cases, use of a smaller number of alphabet tokens enhances the repetition of tokens in the cycle. Sometimes it is possible that two sequences with different cycles may result in same VSLA. For example: consider sequences generated for cycles C_1 = (1 2 1 3 1 3 1 2) and C_2 = (1 2 1 3). Noise-free (for simplicity) VSLA that is a result of VSLA Constructor is shown in Fig. V.1. VSLA Construction process is based on 1st-degree Markovian process where each token depends only on the previous one.

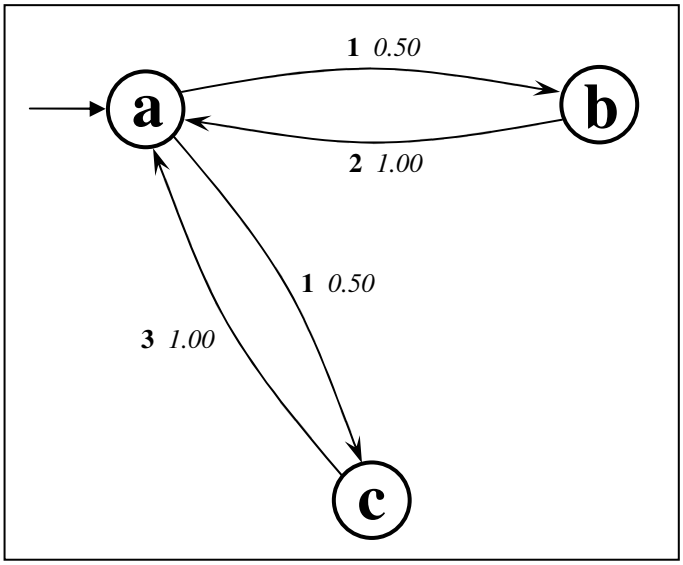

Figure V.1. Structure of noise-free VSLA for cycles C_1 = (1 2 1 3 1 3 1 2) and C_2 = (1 2 1 3).

Although 1st degree Markovian process is enough to recognize almost all cycles, in these cases a 2nd-degree Markovian process is required to distinguish between these two cycles, which is beyond the scope of this work. Increasing degree of Markovian process results in slight increase of recognition probabilities and exponential increase of runtime which is $O(|C|^n)$. VSLAs, constructed using 2nd degree, are shown in Fig. V.2 and Fig. V.3.

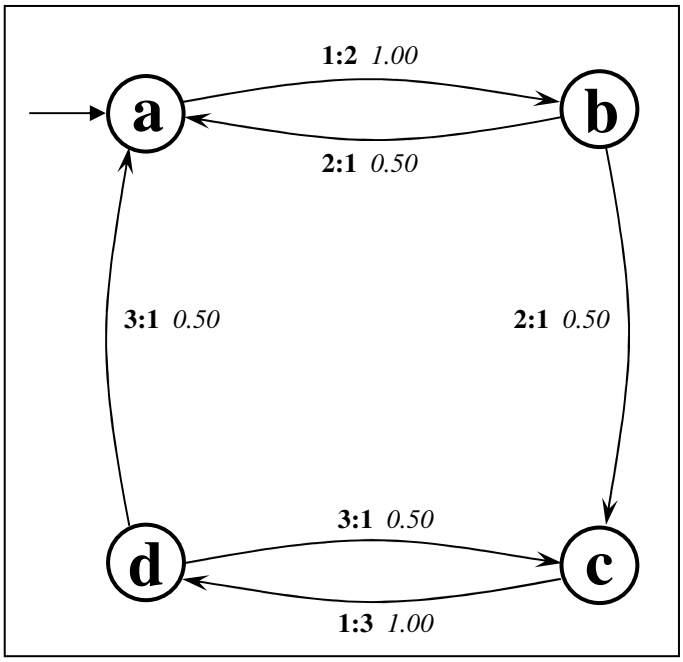

Figure V.2. VSLA, constructed using 2nd degree Markovian process, for C_1 = (1 2 1 3 1 3 1 2).

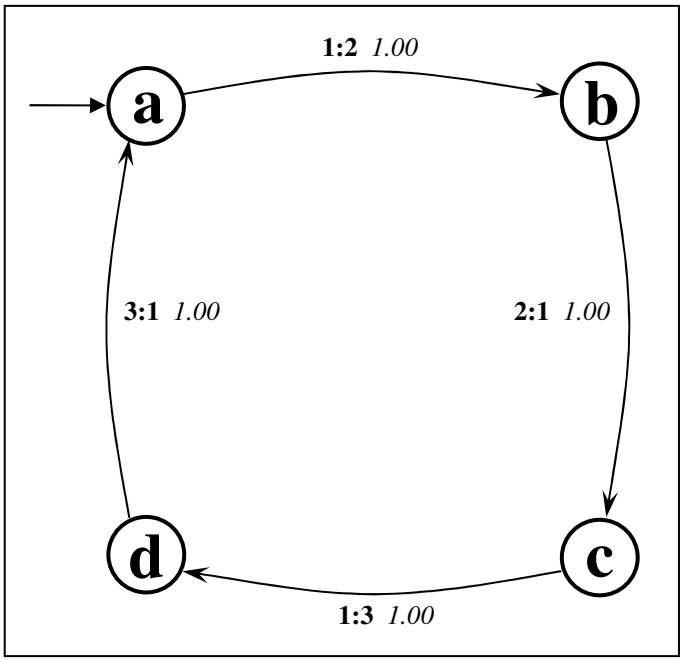

Figure V.3. VSLA, constructed using 2nd degree Markovian process, for C_2 = (1 2 1 3).

V.2.3 No Clean Cycles

Cycle Detector's algorithm requires presence of at least one clean cycle in noisy sequence to be able to detect it. This property is hardwired to the system and may not

easily be changed. If cycle detector was able to detect cycles in noisy input sequences using FSLAs without need for clean cycles it would be possible to use this approach to higher amounts of noise and to apply it to the systems that may not have a clean cycle at all like ECG, OCR or SR.

V.2.3 Real World Applications

Adaptive constructive approach to SPR may be useful in many real world applications like ECG (Electrocardiograms), OCR (Optical Character Recognition) and SR (Speech Recognition).

Graph of an electrocardiogram is a ready-to-recognize periodic sequence of heart beats (Fig. V.4). After quantization this graph may be converted into a sequence of tokens where each token is a distinct quantum.

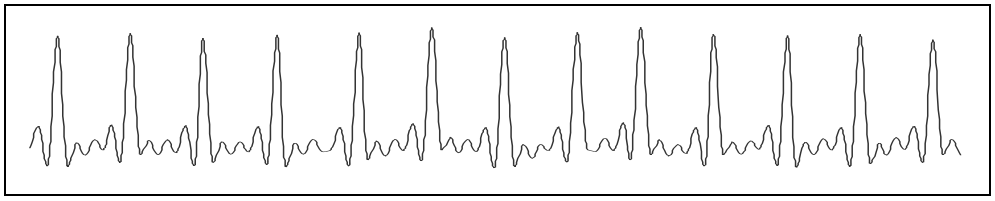

Figure V.4. Graph of the ECG signal.

OCR (Fig. V.5) is a recognition problem in 2–dimensional space which requires to be converted into 1–dimensional, like ECG. Here the possible algorithm might be to define and extract features of a letter image, and, then, sort them in an order as they appear in an image. Each feature is a single token. And a set of tokens/features extracted from an image is a noisy cycle. Performing same operation to large amount of letter images during training phase will result in a sequence for a specific letter and that sequence will be ready to recognize using constructive approach.

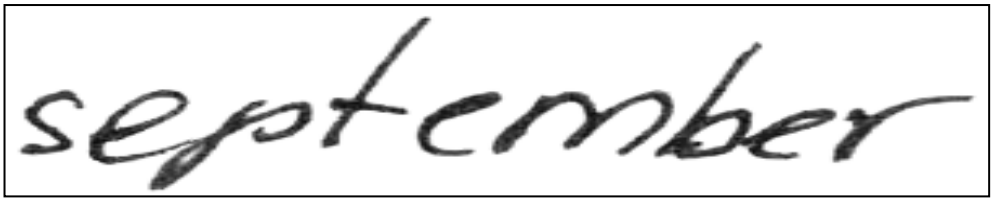

Figure V.5. Image of the word "september", used as an input to OCR.

SR is a combination of ECG and OCR (Fig. V.6). SR easily can be converted into a 1–dimensional graph, but to make it periodic it has to be decomposed into sounds, phonemes or syllables like in OCR where sequences are constructed for each letter.

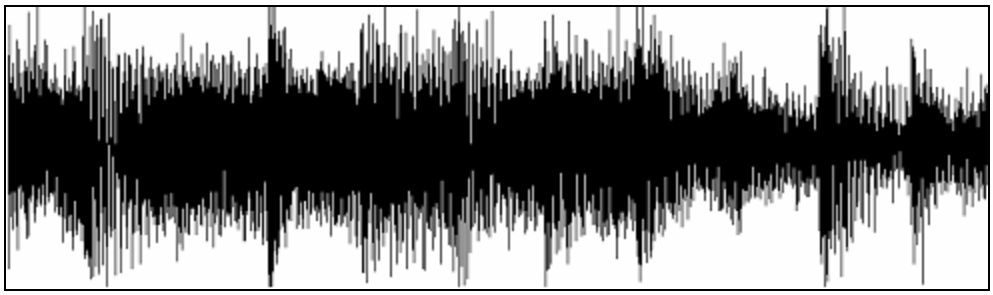

Figure V.6. Graph of the digitally recorded speech to use in SR.

REFERENCES

[1] Hopcroft, J. E.; , Motwani, R.; Ullman, J. D.: "Introduction To Automata Theory, Languages, And Computation", The Addison-Wesley press, (**2001**) 2-38.

[2] Alpaydin, E.: "Introduction to Machine Learning", The MIT Press, Cambridge, Massachusetts, London, England, (**2004**) 1-16.

[3] Baba, N.; Mogami, Y.: "A New Learning Algorithm for the Hierarchial Structure Learning Automata Operating in the Nonstationary S-Model Random Environment", *IEEE Transactions on Systems, Man and Cybernetics, Part:B*, 32 (6), (**2002**) 750–757.

[4] Narendra, K.; Thathachar, M. A. L.: "Learning Automata: An Introduction", Prentice Hall, Inc., (**1989**) 1-108.

[5] Sutton, R. S.; Barto, A. G.: "Reinforcement Learning: An Introduction", *A Bradford Book*, The MIT Press, Cambridge, Massachusetts, London, England, (**1998**).

[6] Huang, K.-Y.: "Syntactic Pattern Recognition For Seismic Oil Exploration", *Series in Machine Perception and Artificial Intelligence*, National Chiao Tung University, Taiwan, 46 (**2002**) 1-5.

[7] Fu, K. S.: "Syntactic Methods in Pattern Recognition", Springer-Verlag, (1974).

[8] Bishop, C. M.: "Neural Networks for Pattern Recognition", Oxford University Press, (**1995**) 1-7.

[9] Najim, K.; Poznyak A.: "Learning Automata: Theory and Applications", Elsevier Science Ltd., (**1994**).

[10] Najim, K.; Poznyak A.: "Learning Automata and Stochastic Optimization", Springer Verlag, (**1997**).

[11] Tumer M. B.; Belfore, L. A. II; Ropella, K. M.: "A Syntactic Methodology for Automatic Diagnosis by Analysis of Continuous Time Measurements using

Hierarchical Signal Representations", *IEEE Transactions on Systems, Man and Cybernetics, Part:B*, 33 (6), (**2003**) 951-965.

[12] Tumer M. B.; Belfore, L. A. I; Ropella, K. M.: "A Syntactic Methodology for Analysis of Continuous Time-Sampled Signals", *IEEE Transactions on Systems, Man and Cybernetics, Part:B*, 45, (**2003**) 951–965.

[13] Sörnmo, L.; Laguna, P.: "Bioelectrical Signal Processing In Cardiac And Neurological Applications", ACADEMIC PRESS, (**2005**) 1-22.

[14] Koski, A.; Juhola, M.; Meriste, M.: "Syntactic Recognition of ECG Signals by Attributed Finite Automata", *Pattern Recognition*, 28, (**1995**) 1927–1940.

[15] Trahanias, P.; Skordalakis, E.; Papakonstantinou G.: "A Syntactic Method For The Classification of the QRS Patterns", *Pattern Recognition Letters*, 9, (**1989**) 13–18.

[16] Trahanias, P.; Skordalakis, E.: "Syntactic pattern recognition of the ECG", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12, (**1990**) 648–657.

[17] G. Belforte, R. DeMori, and F. Ferraris, "A Contribution to the Automatic Processing of Electrocardiograms Using Syntactic Methods", *IEEE Transactions on Biomedical Engineering*, 26, (**1979**) 125–136.

[18] Vidal, E.; Castro, M. J.: "Classification of Banded Chromosomes using Error-Correcting Grammatical Inference (ECGI) and Multilayer Perceptron (MLP)", *VII National Symposium on Pattern Recognition and Image Analysis*, (**1997**) 31–36.

[19] Moayer, B.; Fu, K. S.: "A Syntactic Approach to Fingerprint Pattern Recognition", Pattern Recognition, 7, (**1975**) 1-23.

[20] Huckvale, M.: "A Syntactic Pattern Recognition Method For The Automatic Location of Potential Enhancement Regions In Running Speech", *Speech, Hearing and Language: Work in Progress, UCL*, 9 (1996) *in press*.

[21] Trahanias, P.; Skordalakis, E.: "Syntactic Pattern Recognition of the ECG", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12 (7), (**1990**), 648–657.

APPENDICES

APPENDIX A: The User Manual

An application that was written and used to test new constructive approach is attached to this work. Application was written in Java using v. 1.4.2 SDK (Standard Development Kit). The working principle depends on parameters that were designed to support as much distinct functionality as possible. Some of the parameters are optional, some mandatory.

There are 3 mandatory global parameters. First is "output_file" which specifies filename for storing the output. Second is "state_multiplier": the learning parameter $(1 - \lambda)$ used in L algorithm for rewarding/penalizing states. Last one is "transition_multiplier" that is $(1 - \lambda)$ for transition probability updates. The closer last two parameters to 1 the slower VSLA learns.

Application can take input in two different ways: from user defined file, or from Noisy Sequence Generator built into the system. First choice is rather simple one. To select it user must enable "input_file" parameter. Input file, whose filename is a value of an "input_file", has to be in the following format: data presented on each line is assumed to be a token so the number of lines an input file contains is the sequence length. Second choice is to use Noisy Sequence Generator that has some parameters of its own. First parameter "nig_series_number" specifies how many experiments the application has to run. This is an optional parameter the value of which (if not given explicitly) defaults to 1. Parameter "nig_period_size" defines the cycle length; "nig_period_number" – length of the sequence in terms of cycles; "nig_noise_percentage" specifies the ratio in percent of noisy tokens to all tokens in the sequence; "nig_noise_generation_strategy" takes 3 different values: "r", "e" and "u" which are *random*, *equal* and *user defined* respectively. Next three parameters ("nig_replacement_percentage", "nig_removal_percentage" and "nig_insertion_percentage") specify the distribution of each type of noise. So, if the

parameter “nig_noise_generation_strategy” has the value of “r” Noisy Sequence Generator will set the values of distribution parameters randomly; if the value is “e” the generator equally distributes all types of noise; and if the value is “u” then the generator uses the user defined values for three distribution parameters. Next parameter is “nig_alphabet”, which defines a token alphabet (Σ). Last parameter “nig_output_file” provides the name of the file that the generator will use to store the generated sequence(s).

When the application have been run for one sequence (using input file or single Noisy Sequence Generator sequence) output returned by the application contains VSLA and all the FSLAs, that were generated by the Noise Remover, with scores and detected cycles (from Cycle Detector). All kings of automata presented in output are in textural format consisting of list of states and transitions with probabilities and frequencies.

If the application has been run for more than one sequence, which is possible only using Noisy Sequence Generator, output file contains only input/output filenames and recognition summaries for each sequence.

APPENDIX B: The VSLA Structures

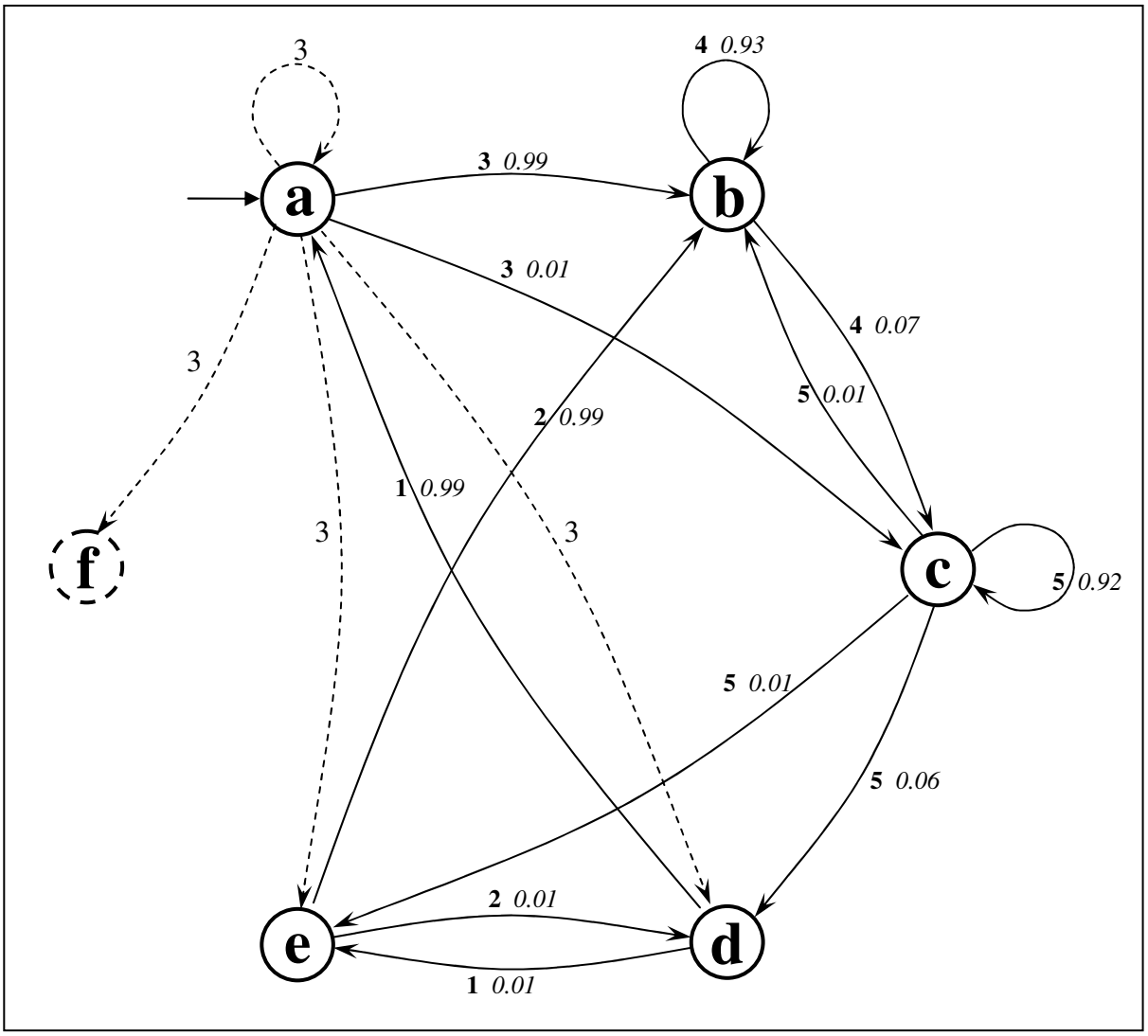

Figure B.1. Structure of the VSLA after 50 tokens has been processed.

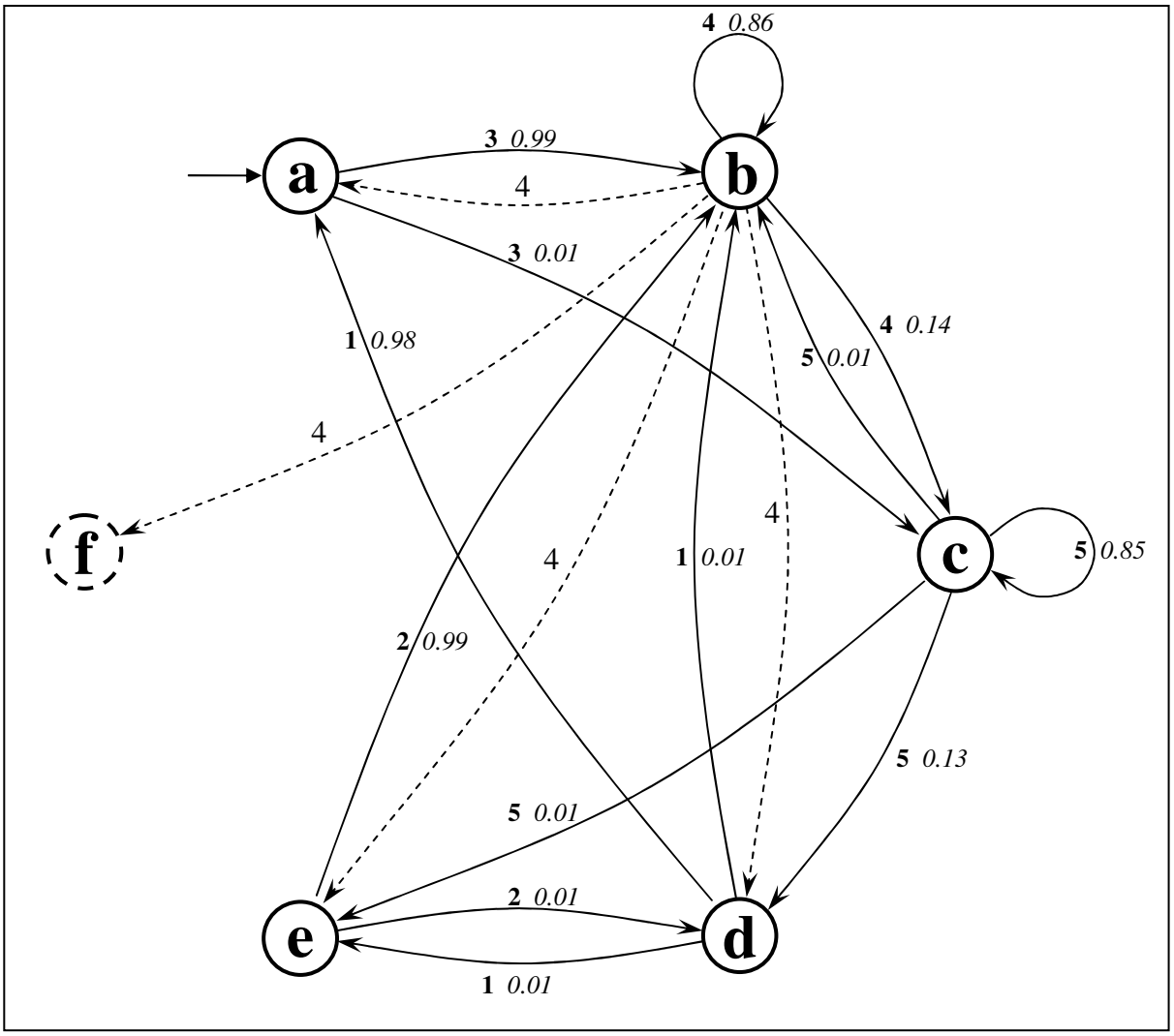

Figure B.2. The VSLA's structure, after first 100 tokens have been processed.

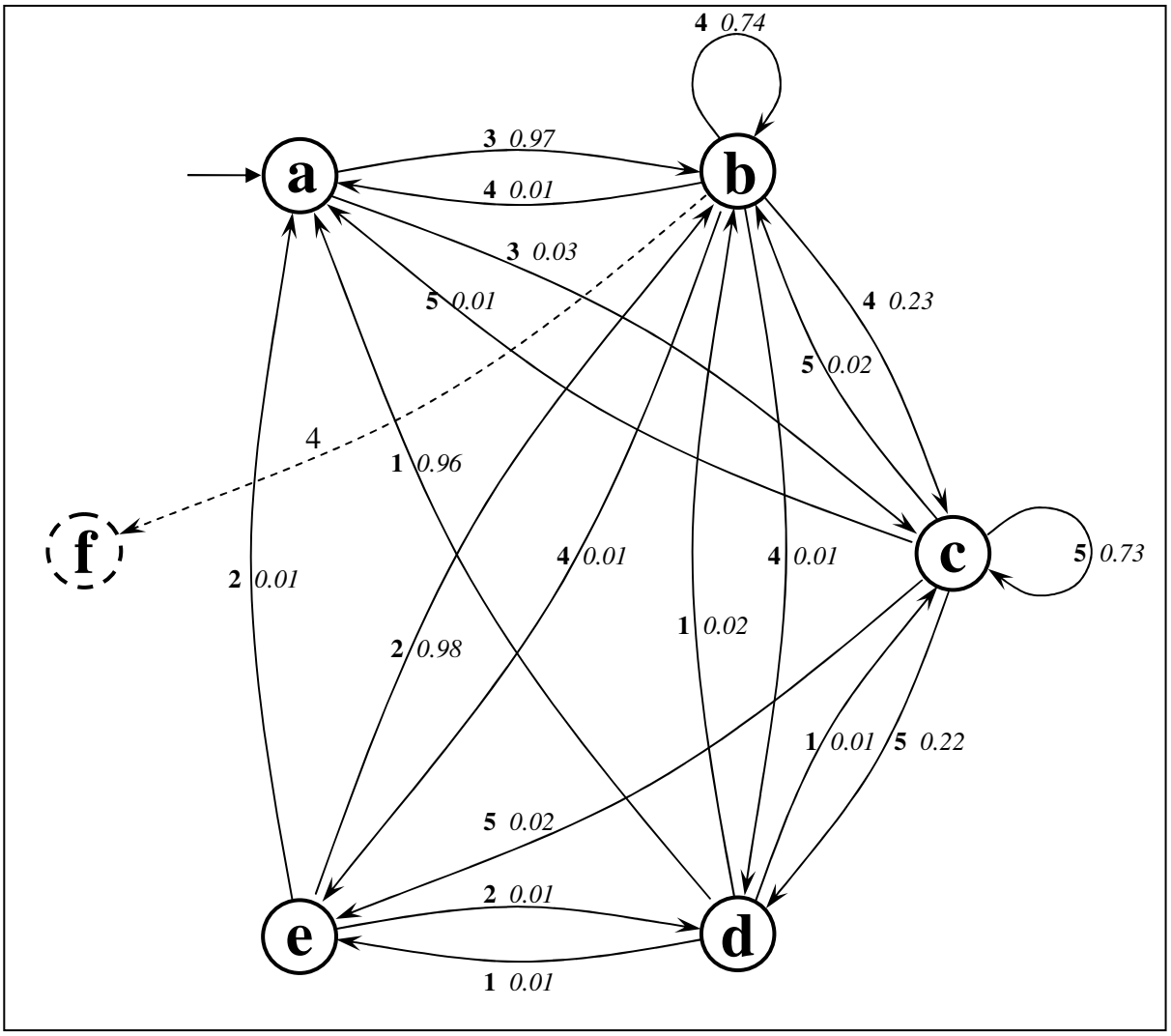

Figure B.3. The VSLA after 200th token (4) has been processed.

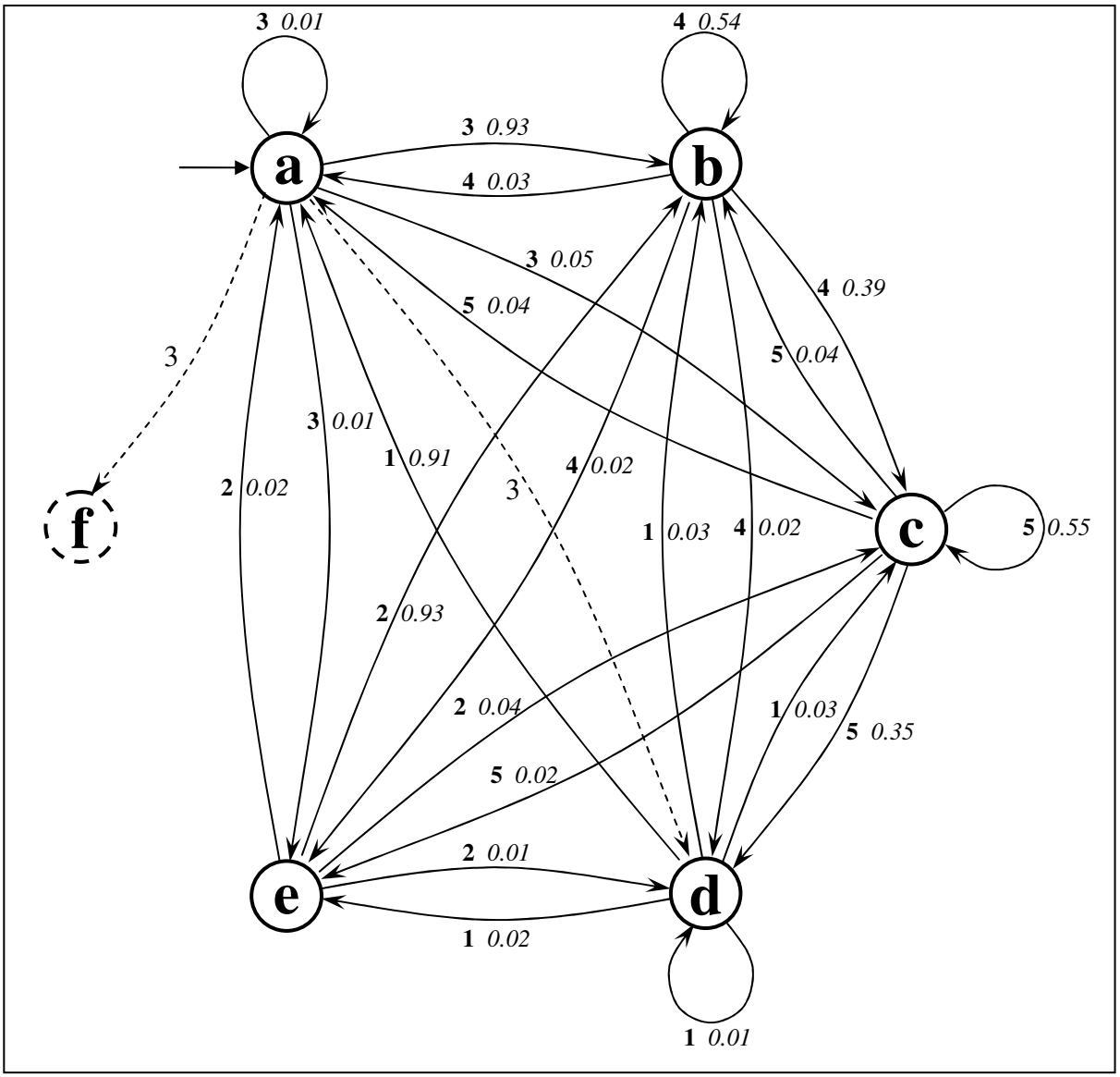

Figure B.4. Structure of the VSLA after 500 tokens.

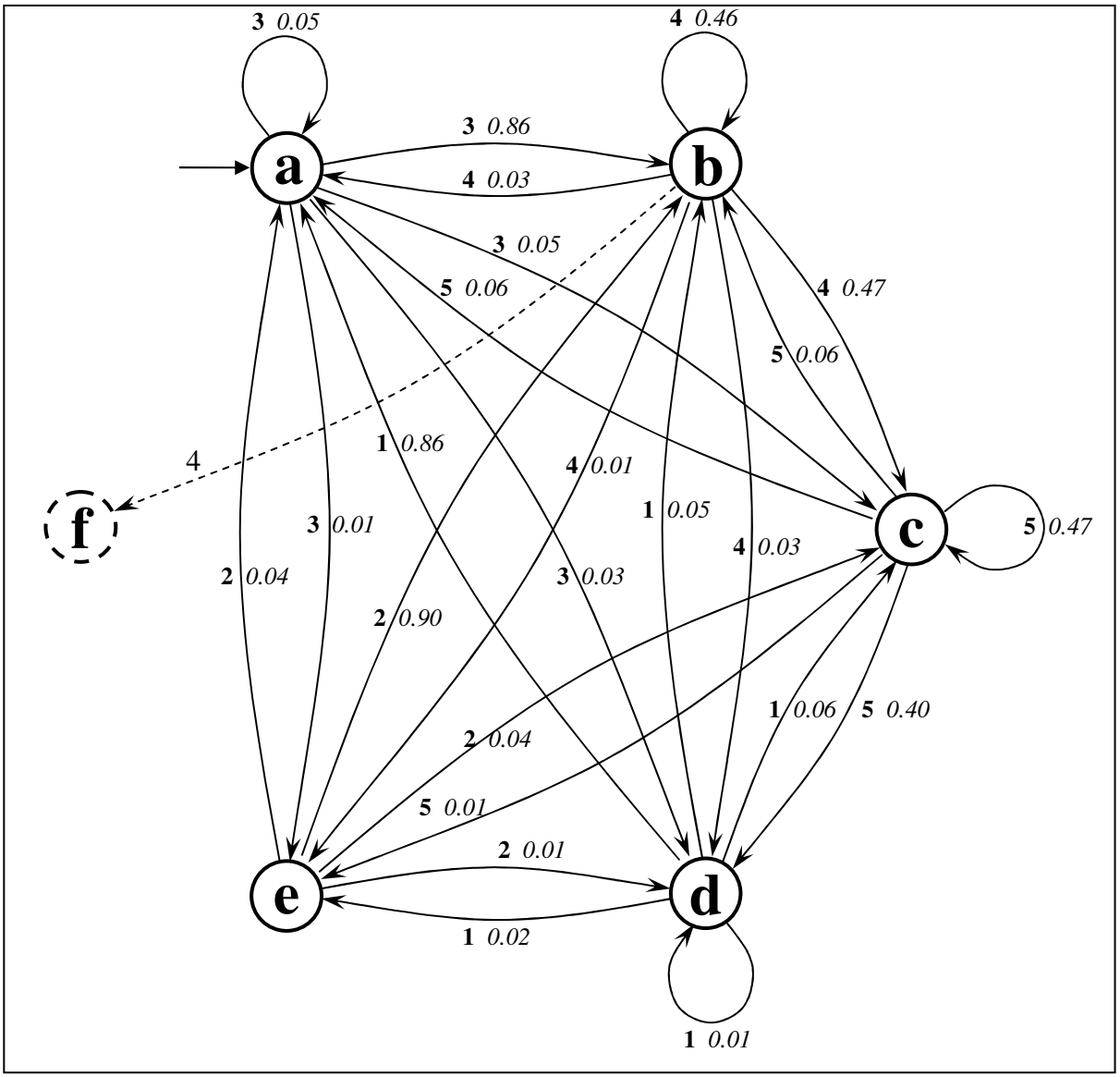

Figure B.5. The VSLA's structure after 1000 tokens have been processed.

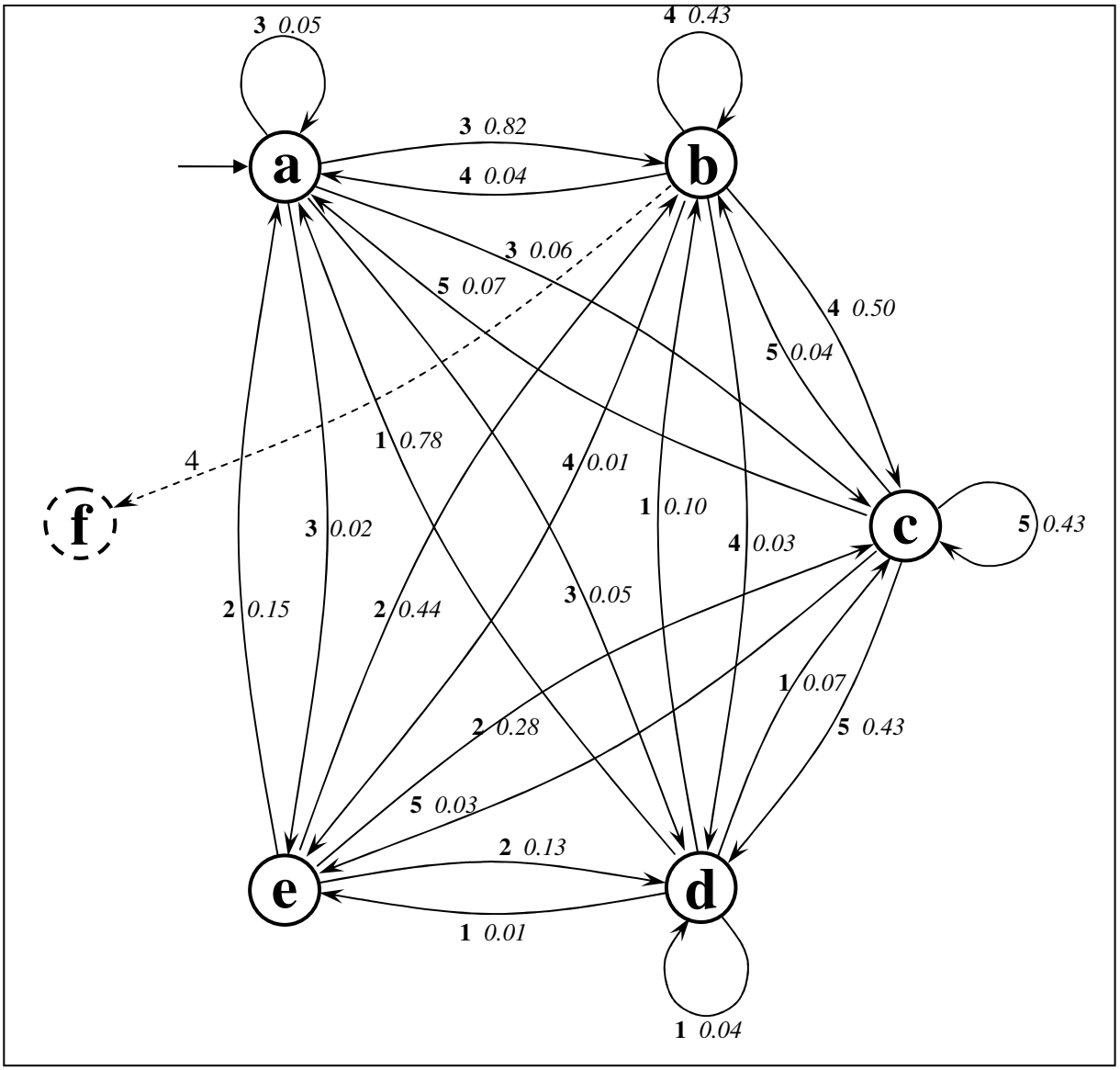

Figure B.6. The VSLA after process of the 10000th token (4).

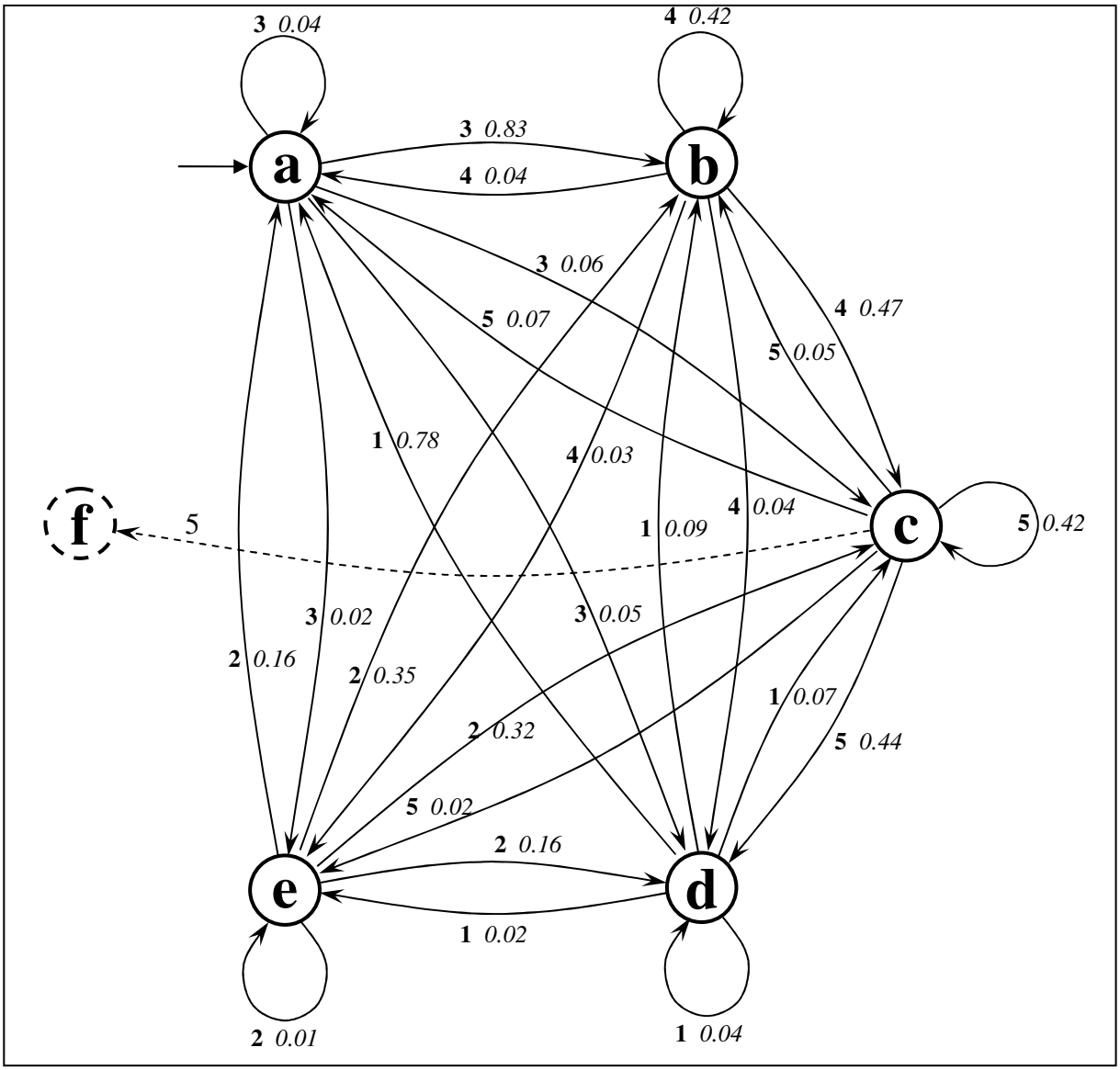

Figure B.7. Final VSLA structure after processing the entire sequence of 50000 tokens.

APPENDIX C: The Pseudo Code of the Noise Remover

This pseudo code is using the following notation. All variable names are written in *italics*. Keywords (loop, while, do, for, if etc.), data types (float, int, φ, FSLA, etc.) and system functions (add(), delete(), findDistributions() etc.) are written in **bold**. Introduction of basic type variables, like int, float or φ, is made by simply writing the variable name. Introduction of complex type variables is made by writing the variable name and its structure. Square brackets [] denote arrays or lists, curve brackets {} denote sets and straight brackets || denote number of elements in array, list or set. For example: *range*[**int**, **int**] means that the name of the introduced variable is "range" and [**int**, **int**] means that this variable is an array/list consisting of two integer values. More complex example is: *allMatchResults*{[**φ**, {[{**δ**}, **float**]}]}. Variable name is "allMatchResults", and it is a set of two-element arrays/lists. First element is a state, second is again a set of two-element arrays/lists which are a set of transitions and a floating point value. All arrays and lists are 1-based. Index value that may be seen after the set variable denotes the single element from that set.

MCL = 30

CNC = 1.3

SNC = 1.3

TNC = 1.5

φ_l

S

Figure C.1. Global variables. First 4 variables are constants which used to limit an area of search. Values of the last 2 are determined during the runtime.

```
removeNoise(VSLA)
1   minStateProbability = 1 / (MCL * CNC)
2   while Pr(φ_l) < minStateProbability do
3       VSLA = removeState(VSLA, φ_l)
4   loop
5   while S > 0 do
6       maxBase = floor(Pr(φ_l) * MCL * CNC)
7       for base = 1 to maxBase do
8           allMatchResults{[φ, {[{δ}, float]}]} = {}
9           for each φ_i in VSLA do
10              range[int, int] = calculateStateFrequencyRange(φ_i, base)
11              transitionsSet{[{δ}, float]} = matchTransitions({δ_ij^τk}_j=1^S, range)
12              allMatchResults = add(allMatchResults, {[φ_i, transitionsSet]})
13          loop
14          FSLAs{[FSLA, float]} = createAutomata(allMatchResults)
15          FSLAs{[FSLA, float]} = filterAutomata(FSLAs)
16          if |FSLAs| > 0 then
17              return FSLAs_i[1] where FSLAs_i[2] is maximum
18          end if
19      loop
20      VSLA = removeState(VSLA, φ_l)
21  loop
22  return nothing
```

Figure C.2. Main function of the Noise Remover. It takes VSLA as a parameter, removes noisy states and transitions, determines frequency for states and transitions of all resulting possible FSLAs and return one FSLA that gained the highest score.

removeState(*VSLA*, φ_i)

1 *VSLA* = **deleteTransitions**(*VSLA*, $\left\{\delta_{ij}^{\tau_k}\right\}_{j=1}^{S}$)

2 *VSLA* = **deleteTransitions**(*VSLA*, $\left\{\delta_{ji}^{\tau_m}\right\}_{j=1}^{S}$)

3 *VSLA* = **deleteStates**(*VSLA*, $\{\varphi_i\}$)

4 *VSLA* = normalize(*VSLA*)

5 **return** *VSLA*

Figure C.3. This function permanently removes state and all transitions related to that state. After removal process automaton is normalized.

normalize(*VSLA*)

1 $sumOfStateProbabilities = \sum_{i=1}^{S} \Pr(\varphi_i)$

2 **for each** φ_i **in** *VSLA* **do**

3 $\Pr(\varphi_i) = \Pr(\varphi_i)$ / *sumOfStateProbabilities*

4 $sumOfTransitionProbabilities = \sum_{j=1}^{S} \Pr\left(\delta_{ij}^{\tau_k}\right)$

5 **for each** $\delta_{ij}^{\tau_k}$ **in** $\left\{\delta_{ij}^{\tau_k}\right\}_{j=1}^{S}$ **do**

6 $\Pr\left(\delta_{ij}^{\tau_k}\right) = \Pr\left(\delta_{ij}^{\tau_k}\right)$ / *sumOfTransitionProbabilities*

7 **loop**

8 **loop**

9 **return** *VSLA*

Figure C.4. Normalization process ensures that sum of probabilities of all states and all transitions initiated from any state are 1.

```
calculateStateFrequencyRange(φ_i, base)
    if φ_i = φ_l then
        return [base, base]
    end if
    ratio = base * Pr(φ_i) / Pr(φ_l)
    lowest = floor(ratio / SNC)
    highest = ceil(ratio * SNC)
    return [lowest, highest]
```

Figure C.5. Function calculates a range of frequency distribution classes to apply to the transitions of the given state.

```
createAutomata(allMatchResults{[φ, {[{δ}, float]}]})
    FSLAs{[FSLA, float]} = {}
    matchResults[φ, {[{δ}, float]}] = allMatchResults_1
    allMatchResults = remove(allMatchResults, allMatchResults_1)
    for each matchResult[{δ}, float] in matchResults[2] do
        curFSLA[FSLA, float] = [
            createFSLA({matchResults[1]}, matchResult[1]),
            matchResult[2] ]
        subFSLAs{[FSLA, float]} = createAutomata(allMatchResults)
        newFSLAs{[FSLA, float]} = merge(curFSLA, subFSLAs)
        FSLAs = add(FSLAs, newFSLAs)
    loop
    if ROOT_OF_RECURSION then
        for each singleFSLA[FSLA, float] in FSLAs do
            singleFSLA[1] = normalize(singleFSLA[1])
        loop
    end if
    return FSLAs
```

Figure C.6. This function recursively combines and normalizes automata of FSLA types from states and transition sets.

```
merge(curFSLA[FSLA, float], subFSLAs{[FSLA, float]})
1    newFSLAs{[FSLA, float]} = {}
2    for each singleFSLA[FSLA, float] in subFSLAs[2] do
3        singleFSLA[1] = mergeAutomata(singleFSLA[1], currFSLA[1])
4        singleFSLA[2] = singleFSLA[2] + currFSLA[2]
5        newFSLAs = add(singleFSLA)
6    loop
7    return newFSLAs
```

Figure C.7. The goal of this function is to add an FSLA from the first parameter to each FSLA from the second parameter.

```
matchTransitions(transitions{δ}, range[int, int])
1    distributions{[int, …]} = findDistributions(range)
2    transitionsList[δ, …] = sortByProbability(transitions, DESCENDING)
3    transitionsSet{[{δ}, float]} = {}
4    for each distribution[int, …] in distributions do id = 1
5        sizeD = |distribution|
6        if sizeD > |transitionsList| then
7            next for 1
8        else if sizeD < |transitionsList| then
9            ratio1 = (Pr(transitionsList[sized]) / distribution[sizeD]) * TNC
10           ratio2 = Pr(transitionsList[sized]) / TNC
11           if Pr(transitionsList[sizeD + 1]) > min(ratio1, ratio2) then
12               next for 1
13           end if
14       end if
```

Figure C.8. This function tries to match given transition set to all frequency distributions in the given range of classes. If separator ratio and all pair ratios are suitable for the current frequency distribution matched set of transitions and match score is added to the result set.

```
15          for i = 1 to sizeD – 1 do
16              for j = i to sizeD do
17                  distributionRatio = distribution[i] / distribution[j]
18                  transitionRatio = Pr(transitionsList[i]) / Pr(transitionsList[j])
19                  lowest = transitionRatio / TNC
20                  highest = transitionRatio * TNC
21                  if distributionRatio ∉ [lowest; highest] then
22                      next for 1
23                  end if
24              loop
25          loop
26          score = sqrt( sum_{i=1}^{sizeD} [ (Pr(transitionsList[i]) + distribution[i]) / ( sqrt(sum_{j=1}^{sizeD} Pr(transitionsList[j])^2) + sqrt(sum_{j=1}^{sizeD} distribution[j]^2) ) ]^2 )
27          score = score * 1 / sum_{k=1}^{sizeD} distribution[k]
28          subSetOfTransitions{δ} = subListToSet(transitionsList, 1, sizeD)
29          transitionsSet = add(transitionsSet, [subSetOfTransitions, score])
30      loop
31      return transitionsSet
```

Figure C.8. This function tries to match given transition set to all frequency distributions in the given range of classes. If separator ratio and all pair ratios are suitable for the current frequency distribution matched set of transitions and match score is added to the result set. (continued)

```
filterAutomata(FSLAs{FSLA})
1     for each FSLA in FSLAs do id = 1
2           if not allStatesCommunicating(FSLA) then
3                 FSLAs = remove(FSLAs, FSLA)
4                 next for 1
5           end if
6           if not allFrequenciesMatch(FSLA) then
7                 FSLAs = remove(FSLAs, FSLA)
8                 next for 1
9           end if
10    loop
11    return FSLAs
```

Figure C.9. Function tries to find and eliminate (remove) all invalid automata in the given set.

CURRICULUM VITAE

He was born in Dushanbe (capital city of the Tajikistan) in 1980. He completed his primary and secondary educations in governmental school № 75 in Tursunzade. After secondary education, he was accepted to Tajik – Türk High School that was opened in Tajikistan by Hagi Kemal ERİMEZ, the Turkish investor. In 1998 he was admitted to Istanbul University, Faculty of Engineering, Department of Computer Engineering. He graduated in 2002 with a degree of BSc. A year later he started to pursue MSc degree in the same field in Marmara University.