UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE TECNOLOGIA E GEOCIÊNCIAS

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

**DANIEL KUDLOWIEZ FRANCH**

# DYNAMICAL SYSTEM MODELING WITH PROBABILISTIC FINITE STATE AUTOMATA

Recife
2017

**DANIEL KUDLOWIEZ FRANCH**

# DYNAMICAL SYSTEM MODELING WITH PROBABILISTIC FINITE STATE AUTOMATA

**Dissertação** submetida ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Pernambuco como parte dos requisitos para obtenção do grau de **Mestre em Engenharia Elétrica**.
Orientador: Prof. Cecilio José Lins Pimentel.
Coorientador: Prof. Daniel Pedro Bezerra Chaves
Área de Concentração: Comunicações

Recife
2017

**Universidade Federal de Pernambuco**

***Pós-Graduação em Engenharia Elétrica***

PARECER DA COMISSÃO EXAMINADORA DE DEFESA DE DISSERTAÇÃO DO MESTRADO ACADÊMICO DE

# DANIEL KUDLOWIEZ FRANCH

TÍTULO

**“DYNAMICAL SYSTEM MODELING WITH PROBABILISTIC FINITE STATE AUTOMATA”**

A comissão examinadora composta pelos professores: DANIEL PEDRO BEZERRA CHAVES, DES/UFPE; RICARDO MENEZES CAMPELLO DE SOUZA, DES/UFPE, e WILSON ROSA DE OLIVEIRA JUNIOR, DEI/UFRPE, sob a presidência do primeiro, consideram o candidato **DANIEL KUDLOWIEZ FRANCH** **APROVADO**.

Recife, 10 de março de 2017.

**MARCELO CABRAL CAVALCANTI**
Coordenador do PPGEE

**DANIEL PEDRO BEZERRA CHAVES**
Coorientador e Membro Titular Interno

**WILSON ROSA DE OLIVEIRA JUNIOR**
Membro Titular Externo

**RICARDO MENEZES CAMPELLO DE SOUZA**
Membro Titular Interno

Catalogação na fonte
Bibliotecária Valdicéa Alves, CRB-4 / 1260

F816d Franch. Daniel Kudlowiez.
Dynamical system modeling with probabilistic finite state automata / Daniel Kudlowiez Franch - 2017.
95folhas, Il.; Tabs; Abr.e Acr.; e Simbs.

Orientador: Prof. Dr. Cecilio José Lins Pimentel.
Coorientador: Prof. Dr. Daniel Pedro Bezerra Chaves.

Dissertação (Mestrado) – Universidade Federal de Pernambuco. CTG. Programa de Pós-Graduação em Engenharia Elétrica, 2017.
Include Reference e Appendix .

1. Engenharia Elétrica. 2. Clustering. 3. Dynamical systems. 4. Graph minimization. 5. Synchronization word. 6. Probabilistic finite state automata. I. Pimentel, Cecilio José Lins ( Orientador). II. Chaves, Daniel Pedro Bezerra. (Coorientador). III. Título.

UFPE

621.3 CDD (22. ed.) BCTG/2017 - 186

To my grandmother Zélia

To my parents and sister for all the support they gave me.

To my advisers for all the guidance provided.

# Abstract

Discrete dynamical systems are widely used in a variety of scientific and engineering applications, such as electrical circuits, machine learning, meteorology and neurobiology. Modeling these systems involves performing statistical analysis of the system output to estimate the parameters of a model so it can behave similarly to the original system. These models can be used for simulation, performance analysis or fault detection. The current work presents two new algorithms to model discrete dynamical systems from two categories (synchronizable and non-synchronizable) using Probabilistic Finite State Automata (PFSA) by analyzing sequences generated by the original system and applying statistical methods, machine learning algorithms and graph minimization techniques to obtain compact and efficient PFSA models. Its performance and time complexity are compared with other algorithms present in literature that aim to achieve the same goal by applying the algorithms to a series of examples.

**Keywords:** Clustering. Dynamical systems. Graph minimization. Synchronization word. Probabilistic finite state automata.

# RESUMO

Sistemas dinâmicos discretos são amplamente usados em uma variedade de aplicações cientifícas e de engenharia, por exemplo, circuitos elétricos, aprendizado de máquina, meteorologia e neurobiologia. O modelamento destes sistemas envolve realizar uma análise estatística de sequências de saída do sistema para estimar parâmetros de um modelo para que este se comporte de maneira similar ao sistema original. Esses modelos podem ser usados para simulação, referência ou detecção de falhas. Este trabalho apresenta dois novos algoritmos para modelar sistemas dinâmicos discretos de duas categorias (sincronizáveis e não-sincronizáveis) por meio de Autômatos Finitos Probabilísticos (PFSA, *Probabilistic Finite State Automata*) analisando sequências geradas pelo sistema original e aplicando métodos estatísticos, algoritmos de aprendizado de máquina e técnicas de minimização de grafos para obter modelos PFSA compactos e eficientes. Sua performance e complexidade temporal são comparadas com algoritmos presentes na literatura que buscam atingir o mesmo objetivo aplicando os algoritmos a uma série de exemplos.

**Palavras-chaves:** Agrupamento. Sistemas dinâmicos. Minimização de grafos. Palavra de sincronização. Autômatos probabilísticos de estados finitos.

# LIST OF FIGURES

# List of Tables

# List of Abbreviations and Acronyms

| **Acronym** | **Definition**. |
|---|---|
| PFSA | Probabilistic Finite State Automaton). |
| CSSR | Causal-State Splitting Reconstruction. |
| CRISSiS | Compression via Recursive Identification of Self-Similar Semantics. |
| DCGraM | D-Markov with Clustering and Graph Minimization |
| ALEPH | ALEPH algorithm. |
| RTP | Rooted Tree with Probabilities. |
| SVS | Shortest Valid Suffix. |
| SNR | Signal to Noise Ratio. |
| ACF | Autocorrelation Function. |
| DFC | Discrete Fading Channel. |
| OPTICS | Ordering Points to Identify the Clustering Structure. |
| DBSCAN | Density-based spatial clustering of applications with noise. |

# LIST OF SYMBOLS

| | |
|---|---|
| $L_{max}$ | Maximum depth of a probabilistic suffix tree. |
| $\chi^2$ | $\chi^2$ statistical test. |
| $\Sigma$ | An alphabet. Discrete set of symbols. |
| $\Sigma^n$ | Set of sequences of symbols from $\Sigma$ of length $n$. |
| $\Sigma^*$ | Set of all sequences of all length with symbols from $\Sigma$. |
| $\epsilon$ | Sequence with length 0. |
| $G$ | A labeled directed graph. |
| $Q$ | The set of states of a graph $G$. |
| $\delta$ | Graph $G$ transition function $\delta : Q \rightarrow Q$. |
| $\delta^*$ | Graph $G$ extended transition function $\delta^* : Q \times \Sigma^* \rightarrow Q$. |
| $F(q)$ | The right context of a state $q \in Q$ of graph $G$. |
| $\mathcal{L}$ | Language generated by a graph. |
| $\mathcal{P}$ | Partition of a set of states $Q$. |
| $L_q^{(h)}(G)$ | Set of sequences generated from state $q$ of graph $G$ with length at most $h$. |
| $\mathcal{M}_h$ | Partition of the states of a graph defined by the Moore equivalence of depth $h$. |
| $\pi$ | Probability of transition function of a PFSA, $\pi : Q \times \Sigma \rightarrow [0, 1]$. |
| $(G, \pi)$ | A PFSA with graph $G$ and probability of transition function $\pi$. |
| $\mathcal{V}(q)$ | Morph of a state $q$ of a PFSA $(G, \pi)$, $\{\pi(q, \sigma), \forall\sigma \in \Sigma\}$. |
| $\mathcal{V}(Q)$ | Set of the morphs of all states $q \in Q$ of a PFSA $(G, \pi)$. |
| $K$ | Number of clusters of the K-Means Algorithm. |
| $S$ | Sequence generated by a PFSA. |
| $N$ | Length of $S$. |
| $L_1$ | The maximum length of prefixes CRISSiS checks for a potential synchronization word. |
| $L_2$ | The maximum length of suffixes CRISSiS checks for a potential synchronization word. |
| $\omega_{syn}$ | Synchronization word of a PFSA. |
| $\alpha$ | Confidence level of a statistical test. |
| $\mathcal{T}$ | A rooted tree with probabilities. |
| $L$ | The depth of a rooted tree with probabilities $\mathcal{T}$. |

| | |
|---|---|
| $W$ | The maximum depth of a rooted tree with probabilities $\mathcal{T}$ that is checked by our synchronization word finding algorithm. |
| $\Gamma$ | List containing all of the valid synchronization word candidates. |
| $\Theta$ | List containing states that have already passed all tests for synchronization word validity. |
| $\Omega_{syn}$ | List that stores the found synchronization words. |
| $suffixes$ | Dictionary that maps a state as key to the set of words containing it as a suffix. |
| $V$ | Dictionary that contains states that need to be checked as their own SVS in the future as keys. |
| $\Psi$ | List of the descendants of a state that failed an statistic test. |
| $\Omega$ | Criterion to connect leaf states of a rooted tree with probabilities. |
| $h$ | Entropy rate of a dynamical system. |
| $h_\ell$ | $\ell$-order entropy rate of a dynamical system. |
| $D_\ell$ | $\ell$-order Kullback-Leibler divergence between sequences $S_1$ and $S_2$. |
| $D$ | Depth of a D-Markov machine. |
| $r$ | Logistic map parameter. |
| $X_k$ | Input of the binary fading channel at time $k$. |
| $Y_k$ | Output of the binary fading channel at time $k$. |
| $Z_k$ | Noise symbol of the binary fading channel at time $k$. |
| $R_k$ | Received symbol at time $k$. |
| $S_k$ | Transmitted symbol at time $k$. |
| $E_s$ | Energy of transmitted signal. |
| $N_k$ | Noise suffered by the transmitted symbol at time $k$. Independent and identically distributed zero-mean Gaussian random variable with variance $N_0/2$ |
| $E_s/N_0$ | Signal to noise ratio. |
| $\{A_k\}$ | Channel's fading process. |
| $J_0$ | Zero-order Bessel function. |
| $f_DT$ | Normalized maximum Doppler frequency. |
| $\sigma, a, b, r$ | Lorenz equations parameters. |

# TABLE OF CONTENTS

# Chapter 1

# Introduction

Dynamical systems are mathematical models describing how the state of a system evolves over time. It consists of two main components: a dynamic, which specifies the evolution of the system, and an initial condition from which the system starts [1]. They can be either continuous or discrete. Continuous dynamical systems have real numbers as both their inputs and outputs and they can be described by differential equations [2]. Discrete time dynamical systems generate symbols from a discrete set called alphabet at discrete time intervals. They can be described by difference equations or discrete state transform relations [3] [4]. The output of a continuous dynamical system can be sampled and quantized. By doing this, this new discrete time sequence describes a discrete dynamical system.

These systems provide a useful framework for analyzing phenomena in several fields of engineering and science such as electronic circuits [5], machine learning [6], meteorology [7], mechanics [8] and neurobiology [9]. Theys also might lead to chaotic behavior, which means that two inputs close to each other produce outputs that greatly diverge from each other, making them difficult to predict and seem almost random even though they are completely deterministic [2].

The aim of systems modeling is to obtain a simple analytic model that accurately reflects the statistical description of the system. This involves two main steps:

i. choosing a class of models capable of representing the system behavior;

ii. developing methods to parameterize the model using experimental sequences.

Therefore, we can use the statistics of the model to analyze the behavior of the system and apply it to system simulation[10], performance analysis[5] and fault detection [11].

In order to obtain models for dynamical systems there are methods and frameworks such as belief networks [12], probabilistic context free grammars [13] and hidden Markov models [14], but they tend do be complex and require large sampling times [15]. An alternative method, which is the one chosen in this work, is to use probabilistic finite state automata (PFSA), which can solve these issues and also produce good statistical models.

## 1.1 Probabilistic Finite State Automata

PFSA can be described as a finite labeled graph with probabilities associated to each edge. As in [15], we consider the PFSA framework for which symbol generation is probabilistic and the end state is unique, given an initial state and a certain sequence. This differs from the framework presented in [16] in which the symbol generation probabilities are not specified and there is a distribution over the possible final states. The advantages of using PFSA are that they are simple and the sample time required for learning them is easy to characterize [15] and they are also an efficient framework for learning the causal structure of observed dynamical behavior [17].

Some PFSA generate sequences with a synchronization word. The statistics of the symbols generated after a synchronization word do not depend on anything that came before it [15]. Thus the synchronization word is deemed to be a good starting point for analysis as anything coming before it can be considered a transient.

The algorithms that construct PFSA include D-Markov machines [18], which are Markov chains of a finite order $D$, meaning it uses the statistics of all subsequences of length $D$ to form its states; the Causal-State Splitting Reconstruction (CSSR) [19], which starts by assuming that the systems being analyzed outputs an independent, identically-distributed sequence with one causal state and splits it to a probabilistic suffix tree of depth $L_{max}$. Each node on the tree defines a state labeled with a suffix and any two nodes are merged if the hypothesis that their next-symbol generation probability is the same according to some statistical test (such as $\chi^2$ or Kolmogorov-Smirnov).

There is also the Compression via Recursive Identification of Self-Similar Semantics (CRISSiS) [15] which first finds a synchronization word in the sequence and uses it as a starting point to construct the PFSA. It tests its children (states that contain the synchronization word as prefix) using statistical tests, merging states if the test passes and creating new ones when it fails. This is done recursively until an irreducible PFSA is obtained. As it has been shown in [15] CRISSiS outperforms CSSR.

## 1.2 Objectives and Contributions

In the current work, we are interested in modeling discrete dynamical systems having only an observed discrete sequence. To achieve this goal, we developed two algorithms that analyze the statistics of these sequences and model their systems via PFSA. In order to obtain models that are less memory consuming, our algorithms apply techniques of graph minimization to obtain smaller PFSA. The first algorithm, ALEPH, is applied to sequences generated by synchronizable systems, i.e. systems that generate synchronization words. The modeling results are compared to other algorithms in the literature that seek similar goals.

As CRISSIS, ALEPH makes use of synchronization words. One contribution of this work is a novel method to find synchronization words which makes use of data structures in order to obtain performance gains over the brute force method used in CRISSIS.

The general structure of the ALEPH algorithm is composed of a few steps, when given an input sequence:

i. creating a tree structure with probabilities in which each state represents a subsequence;

ii. finding the synchronization words;

iii. group the states in equivalence classes using a statistical criterion;

iv. applying a graph minimization algorithm to obtain an irreducible PFSA.

The second algorithm is called D-Markov with Clustering and Graph Minimization (DCGraM) and it is applied to non-synchronizable systems and uses a clustering technique from machine learning, which is a variation of the K-Means algorithm. It works by following these steps:

i. construct a D-Markov model for a given integer $D$;

ii. cluster similar states together;

iii. apply a graph minimization algorithm to obtain an irreducible PFSA.

As non-synchronizable machines lack structures present in their synchronizable counterparts, the second method uses a refinement over the D-Markov model.

## 1.3 Dissertation Structure

This dissertation is organized in six chapters. Chapter 2 reviews the theoretical background discussing discrete sequences, PFSA, clustering and graph minimization algorithms while also showing the CRISSiS and D-Markov Machine algorithms used in the literature. Chapter 3 presents the ALEPH algorithm and then analyzes its time complexity. Chapter 4 presents some synchronizable dynamical systems and shows the comparative results of ALEPH, CRISSiS and D-Markov when recovering the original PFSA. Chapter 5 shows applications modeled as non-synchronizable dynamical systems and an alternative algorithm to be applied in such situation and how this algorithm performs compared to the ones present in the literature. Finally, in Chapter 6, a conclusion is discussed and plans for future works to improve the algorithms are presented.

# Chapter 2

# Preliminaries on Graphs, Probabilistic Finite State Automata and Machine Learning

In this chapter we revise concepts from graphs and PFSA [3][16] that will be required in the subsequent chapters. The concept of graph minimization is presented and a mainstream algorithm to achieve this, Moore, is described (another algorithm, Hopcroft, is shown in Appendix A). The problem of clustering and a basic algorithm that implements it is show in Section 2.5, which is important because of a variation of it is used in Chapter 5. Finally, two well known algorithms to model dynamic systems with PFSA are presented, namely, D-Markov and CRISSiS.

## 2.1 Sequences of Discrete Symbols

This section provides tools to describe sequences of discrete symbols. A finite sequence $u$ of symbols from an alphabet $\Sigma$ is called a word and its length is denoted by $|u|$. The empty word $\varepsilon$ is defined as the sequence with length 0. The set of all possible words of length $n$ symbols from $\Sigma$ is $\Sigma^n$ and the set of all sequences of symbols from $\Sigma$ with all possible lengths, including the empty sequence $\varepsilon$, is $\Sigma^*$.

Two words $u$ and $v \in \Sigma^*$ can be concatenated to form a sequence $uv$. For example, using a binary alphabet, $\Sigma = \{0, 1\}$, the concatenation of $u$ = 1010 and $v$ = 111 is $uv$ = 1010111. Note that $|uv| = |u| + |v|$. In $|\Sigma|^*$, concatenation satisfies closure and is associative, which means $u(vw)$ = $(uv)w$ = $uvw$, but it is not commutative, as $uv$ is not necessarily equal to $vu$. The empty word $\varepsilon$ is a neutral element for concatenation, that is, $\varepsilon u = u\varepsilon = u$. This means that $\Sigma^*$ with the operation of concatenation is a Monoid, as it is a set with an associative operation with an identity element [20].

A word $v \in \Sigma^*$ is called a suffix of a word $w \in \Sigma^*$ ( $|w| > |v|$) if $w$ can be written as a concatenation $uv$, where $u \in \Sigma^*$. In this same sense, the sequence $u$ is called a prefix of $w$.

## 2.2 Graphs

Throughout this dissertation, graph is used to denote a labeled directed graph as defined in Definition 2.1.

**Definition 2.1 – Graph**

*A graph G over the alphabet $\Sigma$ consists of a triple $(Q, \Sigma, \delta)$:*

▷ *$Q$ is a finite set of states with cardinality $|Q|$;*

▷ *$\Sigma$ is a finite alphabet with cardinality $|\Sigma|$;*

▷ *$\delta$ is the state transition function $Q \times \Sigma \rightarrow Q$;* □

Each state $q \in Q$ can be represented as a dot or circle and if $\exists\, \delta(q, \sigma) = q'$, for $q, q' \in Q$ and $\sigma \in \Sigma$, this transition can be represented with a directed arrow from state $q$ to state $q'$ labeled with the symbol $\sigma$. This realization of the transition function is called the outgoing edge from $q$ to $q'$ with symbol $\sigma$. Figure 2.1 shows an example of a three-state graph over a binary alphabet from where it is possible to see there is an outgoing edge from state $A$ to state $B$ with the symbol 1, thus $\delta(A, 1) = B$.

It is possible to extend the transition function so it accepts words and not just symbols. Given $\omega \in \Sigma^n$, where $\omega = \sigma_1\sigma_2 \ldots \sigma_n$ with $\sigma_m \in \Sigma$, for $m = 1 \ldots n$, and given states $q_0, q_1, \ldots, q_n \in Q$, we define the function $\delta^*(q_0, \omega) = q_n$ if $\delta(q_0, \sigma_1) = q_1, \delta(q_1, \sigma_2) = q_2, \ldots, \delta(q_{n-1}, \sigma_n) = q_n$. If $\exists\, \omega \in \Sigma^*$ such that for two states $q_1, q_2 \in Q$, $\delta^*(q_1, \omega) = q_2$, it is said there is a path between $q_1$ and $q_2$ and that $\omega$ is generated by $G$. In Figure 2.1 the path starting at state $A$ and going through $A$, $A$, $B$, $C$, $B$, $A$ generates the word $\omega$ = 001110, that is $\delta^*(A, 001110) = A$.

**Definition 2.2 – Right Context**

*The right context of a state $q \in Q$ is defined as the set of all possible words generated by paths that start at q and end in a state of Q:* □

$$F(q) = \{\omega \in \Sigma^* | \delta^*(q, \omega) \in Q\}.$$

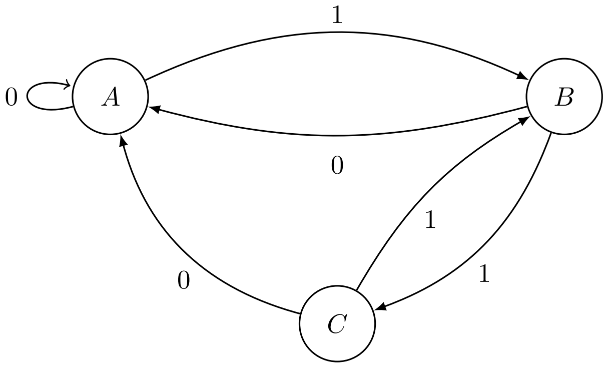

**Figure 2.1:** *A three-state graph with Q = {A, B, C} and* $\Sigma = \{0, 1\}$.

**Definition 2.3 – Language of a Graph**

*The language* $\mathcal{L} \subset \Sigma^*$ *of a graph G is the set of the union of right contexts of each state* $q \in Q$*:*□

$$\mathcal{L} = \bigcup_{q \in Q} F(q).$$

A word $\omega \in \Sigma^*$ is called a synchronization word of $G$ if starting from any state $q \in Q$ that generates $\omega$ the same state $q_{syn} \in Q$ is reached. That is, if $\omega$ is a synchronization word, $\delta^*(q, \omega) = q_{syn}$, for any $q \in Q$ that generates $\omega$. We say that $\omega$ synchronizes to $q_{syn}$ and $q_{syn}$ is called a synchronization state. In the graph of Figure 2.1, 0 is a synchronization word that synchronizes to state $A$.

## 2.3 Graph Minimization

In this section the topic of graph minimization is discussed and the Moore algorithm is shown as an example of a graph minimization algorithm. An alternative algorithm for graph minimization, the Hopcroft algorithm, is shown in Appendix A. The aim of graph minimization is to obtain the graph with the minimum number of states that generates a certain language $\mathcal{L}$.

### 2.3.1 Preliminary Notions

Suppose that two graphs $G_1 = \{Q_1, \Sigma, \delta_1\}$ and $G_2 = \{Q_2, \Sigma, \delta_2\}$ with $|Q_1| \neq |Q_2|$ and are capable of generating the same language $\mathcal{L}$. This is an indication that for a given language $\mathcal{L}$ there might be several graphs that generate it. Among them, it is desirable to use the one with the smallest amount of states, as this lowers the memory requirement.

**Definition 2.4 – Minimal Graph**

*For a given language $\mathcal{L}$ there is a minimal graph $G_{min} = \{Q, \Sigma, \delta\}$ capable of generating it. The minimal graph is the one for which each state $q \in Q$ has a unique right context.* □

If a graph $G_1 = \{Q_1, \Sigma, \delta_1\}$ has two distinct states $q_1$ and $q_2$ with the same right context, a new graph $G_2\{Q_2, \Sigma, \delta_2\}$ that generates the same language can be obtained by merging these states, i.e. $G_2$ has the same states as $G_1$ with the exception of $q_1$ and $q_2$. It has, instead a state $q'$ such that if any state $s$ of $G_1$ has a $\sigma$-transition (for any $\sigma \in \Sigma$) to either $q_1$ or $q_2$, the state $s$ of $G_2$ has $\sigma$-transition to $q'$. This means that for any $s \in Q_1$ that has $\delta_1(s, \sigma) = q_1$ or $q_2$ for any $\sigma \in \Sigma$, $\delta_2(s, \sigma) = q'$. The transitions from $q'$ are $\delta_2(q', \sigma) = \delta_1(q_1, \sigma) = \delta_1(q_2, \sigma)$, $\forall \sigma \in \Sigma$. The languages generated by $G_1$ and $G_2$ are the same.

From Definition 2.4 it is possible to define an equivalence relation called the Nerode equivalence [21]:

$$p, q \in Q, p \equiv q \Leftrightarrow F(p) = F(q).$$

A graph is considered minimal if and only if its Nerode equivalence is the identity. The problem of minimizing a graph is that of computing the Nerode equivalence. The minimal graph accepts the same language as the original graph.

Given a graph $G$ there are two main algorithms used to obtain a minimal graph from it: Moore and Hopcroft [21]. Both will be described in this section, but some definitions are due before getting into the algorithms.

**Definition 2.5 – Partitions and Equivalence Relations**

*Given a set $E$, a partition of $E$ is a family $\mathcal{P}$ of nonempty, pairwise disjoint subsets $P$ of $E$ such that $\bigcup_{P \in \mathcal{P}} P = E$. The index of the partition is its number of elements. The partition $\mathcal{P}$ defines an equivalence relation on $E$ and the set of all equivalence classes of an equivalence relation in $E$ defines a partition of the set.* □

When a subset $F$ of $E$ is the union of classes of $\mathcal{P}$ it is said that $F$ is saturated by $\mathcal{P}$. Given $\mathcal{Q}$, another partition of $E$, it is said to be a *refinement* of $\mathcal{P}$ (or that $\mathcal{P}$ is coarser than $\mathcal{Q}$) if every class of $\mathcal{Q}$ is contained in some class of $\mathcal{P}$ and it is written as $\mathcal{Q} \leq \mathcal{P}$. The index of $\mathcal{Q}$ is greater than the index of $\mathcal{P}$.

Given partitions $\mathcal{P}$ and $\mathcal{Q}$ of $E$, $\mathcal{U} = \mathcal{P} \wedge \mathcal{Q}$ denotes the coarsest partition which refines $\mathcal{P}$ and $\mathcal{Q}$. The elements of $\mathcal{U}$ are non-empty sets $P \cap Q$, such that $P \in \mathcal{P}$ and $Q \in \mathcal{Q}$. The notation is extended for multiple sets as $\mathcal{U} = \mathcal{P}_1 \wedge \mathcal{P}_2 \wedge \ldots \wedge \mathcal{P}_n = \bigwedge_{j=1}^{n} \mathcal{P}_j$.

Given $F \subseteq E$, a partition $\mathcal{P}$ of *E* induces a partition $\mathcal{P}'$ of *F* by intersection. $\mathcal{P}'$ is composed by the sets $P \cap F$ with $P \subseteq \mathcal{P}$. If $\mathcal{P}$ and $\mathcal{Q}$ are partitions of *E* and $\mathcal{Q} \leq \mathcal{P}$, the restrictions $\mathcal{P}'$ and $\mathcal{Q}'$ to *F* maintain $\mathcal{Q}' \leq \mathcal{P}'$.

Given a set of states $P \subset Q$ and a symbol $\sigma \in \Sigma$, let $\sigma^{-1}P$ denote the set of states $q \in Q$ such that $\delta(q, \sigma) \in P$. Consider $P, R \subset Q$ and $\sigma \in \Sigma$, the partition of $R$

$$(P, \sigma)|R$$

is the partition composed of possibly two non-empty subsets:

$$R \cap \sigma^{-1}P = \{r \in R|\delta(r, \sigma) \in P\} \tag{2.1}$$

and

$$R \backslash \sigma^{-1}P = \{r \in R|\delta(r, \sigma) \notin P\}. \tag{2.2}$$

The pair $(P, \sigma)$ is called a splitter. Observe that $(P, \sigma)|R = R$ if either $\delta(r, \sigma) \subset P$ or $\delta(r, \sigma) \cap P = \emptyset, \forall r \in R$ and $(P, \sigma)|R$ is composed of two classes if both $\delta(r, \sigma) \cap P \neq \emptyset$ and $\delta(r, \sigma) \cap P^c \neq \emptyset, \forall r \in R$ or equivalently if $\delta(r, \sigma) \not\subset P$ and $\delta(r, \sigma) \not\subset P^c, \forall r \in R$ . If $(P, \sigma)|R$ contains two classes, then we say that $(P, \sigma)$ splits $R$. This notation can also be extended to sequences, using a sequence $\omega \in \Sigma^*$ instead of the symbol $\sigma \in \Sigma$.

**Proposition 2.1**

*The partition corresponding to the Nerode equivalence is the coarsest partition $\mathcal{P}$ such that no splitter $(P, \sigma)$, with $P \in \mathcal{P}$ and $\sigma \in \Sigma$, splits a class in $\mathcal{P}$, such that $(P, \sigma)|R = R$ for all $P, R \in \mathcal{P}$ and $\sigma \in \Sigma$.* □

### 2.3.2 Moore Algorithm

An important minimization algorithm is the Moore algorithm [22]. It is based on the idea of taking an initial partition with a very wide criteria and then refining it until the Nerode equivalence classes are obtained. The outline of the algorithm is shown in Algorithm 1, which finds the minimal graph that generates the language generated by the graph $G$ given the initial partition $\mathcal{P}$. The usual initial partition for Moore is to group states that have outgoing edges with the same labels together.

Given a graph *G = (Q,* $\Sigma, \delta$*)* and the initial partition $\mathcal{P} = \{P_1, \ldots, P_n\}$, the set $L_q^{(h)}$ is defined as:

$$L_q^{(h)}(G) = \{w \in \Sigma^* || w| \leq h, \delta^*(q,w) \in P_j\},$$

where $\mathcal{P}_j \in \mathcal{P}$ is comprised of all words up to length $h$ that can be generated starting from a certain $q \in Q$ and reaching a state in the equivalence class $\mathcal{P}_j$. The Moore equivalence of order $h$ (denoted by $\equiv_h$) is defined by:

$$p \equiv_h q \Leftrightarrow L_p^{(h)}(G) = L_q^{(h)}(G).$$

This equivalence relation states that two states are equivalent if they generate the same words of length up to $h$ that reach a state in $\mathcal{P}_j$. The depth of the Moore algorithm on a graph $G$ is the integer $h$ such that the Moore equivalence $\equiv_h$ becomes equal to the Nerode equivalence $\equiv$ and it is dependent only on the language of the graph. The depth is the smallest $h$ such that $\equiv_h$ equals $\equiv_{h+1}$, which leads to an algorithm that computes successive Moore equivalences until it finds two consecutive equivalences that are equal, making it halt.

**Proposition 2.2**

*For two states $p, q \in Q$ and $h \geq 0$, one has*

$$p \equiv_{h+1} q \Longleftrightarrow p \equiv_h q \text{ and } \delta(p,\sigma) \equiv_h \delta(q,\sigma),\ \forall \sigma \in \Sigma. \tag{2.3}$$

Using this formulation and defining $\mathcal{M}_h$ as the partition defined by the Moore equivalence of depth $h$, the following proposition holds:

**Proposition 2.3**

*For h $\geq$ 0, one has*

$$\mathcal{M}_{h+1} = \mathcal{M}_h \wedge \bigwedge_{\sigma \in \Sigma} \bigwedge_{P \in \mathcal{M}_h} (P,\sigma)|Q. \tag{2.4}$$

**Algorithm 1** Moore($G, \mathcal{P}$)

```
repeat
    $\mathcal{P}' \leftarrow \mathcal{P}$
    for all $\sigma \in \Sigma$ do
        $\mathcal{P}_\sigma \leftarrow \bigwedge_{P \in \mathcal{P}} (P,\sigma)|Q$
    $\mathcal{P} \leftarrow \mathcal{P} \wedge \bigwedge_{\sigma \in \Sigma} \mathcal{P}_\sigma$
until $\mathcal{P} = \mathcal{P}'$
```

This computation means that for each symbol $\sigma \in \Sigma$ and for each equivalence class $P \in \mathcal{M}_h$ (where $\mathcal{M}_h$ is the partition of the previous iteration) a splitter $(P, \sigma)$ is created and applied to the original set of states $Q$. This will create partitions that show which states of $Q$ reach states in $P$ with symbol $\sigma$ and which do not. Then the coarsest partition $\bigwedge_{P \in \mathcal{M}_h}(P, \sigma)|Q$ is taken, which will separate the equivalence classes of states of $Q$ that reach different equivalence classes of $\mathcal{M}_h$ with a given symbol $\sigma$. After this, the coarsest partition $\bigwedge_{\sigma \in \Sigma} \bigwedge_{P \in \mathcal{M}_h}(P, \sigma)|Q$ between these equivalence classes are taken, separating $Q$ in classes that reach classes in $\mathcal{M}_h$ with each different symbol of $\Sigma$. This effectively computes the $\delta(p, \sigma) \equiv_h \delta(q, \sigma),\ \forall \sigma \in \Sigma$ part of (2.4). To finish (2.4) the coarsest partition of this last step and of $\mathcal{M}_h$ is taken, resulting in the effective computation of an increment in the Moore equivalence classes.

This previous computation is performed in Algorithm 1 in which the loop refines the current partition until no change occurs between $\mathcal{M}_h$ and $\mathcal{M}_{h+1}$, which means the Nerode equivalence is reached. As it will be explored in this work, the initial partition can be created with different criteria. For a graph, it is done by grouping together states in $Q$ which have outgoing edges with the same labels, but another criterion is used in the probabilistic case (Section 2.4.1).

Moore algorithm of the refinement of $k$ partition of a set with $n$ elements can be done in time $O(kn^2)$. Each loop is processed in time $O(kn)$, so the total time is $O(mkn)$, where $m$ is the total number of refinement steps needed to compute the Nerode equivalence.

#### An Example

To illustrate how the Moore algorithm works, this example will apply it to the graph of Figure 2.2, which has $Q = \{A, B, C, D, E\}$, $\Sigma = \{0, 1\}$ and it is not minimal. The first step is to create the initial partition $\mathcal{P}$ based on the criterion of grouping states together in equivalence classes if they have the same outgoing edges with the same labels. In Figure 2.2, states $A$ and $D$ have outgoing edges labeled with 0 and 1 while $B, C$ and $E$ have only an outgoing edge labeled with 0. Thus, the initial partition has two equivalence classes, $\mathcal{P} = \{\{A, D\}, \{B, C, E\}\}$.

Applying the Moore algorithm, $\mathcal{P}'$ will store the current state of $\mathcal{P}$. First, consider $\sigma = 0$. To create the equivalence class $\mathcal{P}_0$, we consider the splitters $(\{A, D\}, 0)$ and $(\{B, C, E\}, 0)$ applied to $Q$. First, take $(\{A, D\}, 0)|Q$. From (2.1) we have

$$Q \cap 0^{-1}\{A, D\} = \{B, E\}$$

and, from (2.2),

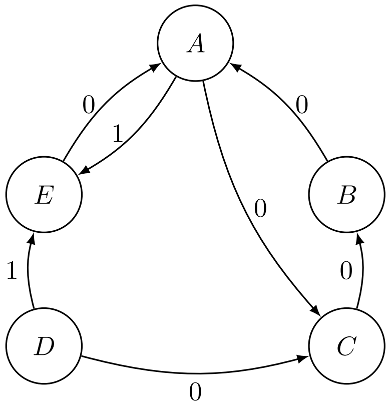

**Figure 2.2:** *An example of a graph that is not minimal.*

$$Q\backslash 0^{-1}\{A,D\} = \{A,C,D\}.$$

The same process is repeated for the splitter $(\{B,C,E\},0)$. From (2.1):

$$Q \cap 0^{-1}\{B,C,E\} = \{A,C,D\}$$

and, from (2.2),

$$Q\backslash 0^{-1}\{B,C,E\} = \{B,E\}.$$

$\mathcal{P}_0$ is then the coarsest partition between $(\{A,D\},0)|Q = \{\{B,E\},\{A,C,D\}\}$ and $(\{B,C,E\},0)|Q = \{\{A,C,D\},\{B,E\}\}$. Thus $\mathcal{P}_0 = \{\{A,C,D\},\{B,E\}\}$.

This process is repeated to obtain $\mathcal{P}_1$. $(\{A,D\},1)|Q$. From (2.1),

$$Q \cap 1^{-1}\{A,D\} = \emptyset$$

and, from (2.2),

$$Q\backslash 1^{-1}\{A,D\} = \{A,B,C,D,E\}$$

and, for $(\{B,C,E\},1)|Q$,

$$Q \cap 1^{-1}\{B,C,E\} = \{A,D\}$$

and

$$Q\backslash 0^{-1}\{A,D\} = \{B,C,E\}.$$

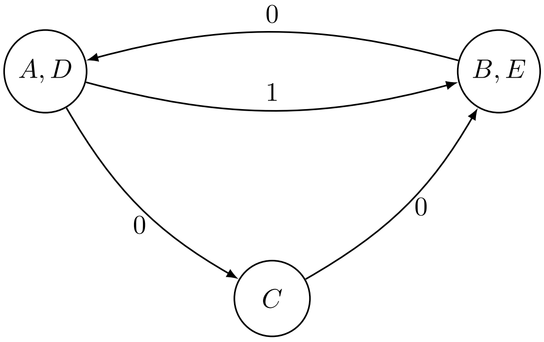

**Figure 2.3:** *Application of Moore algorithm to the initial partition.*

The coarsest partition between $\{\{A, B, C, D, E\}\}$ and $\{\{A, D\}, \{B, C, E\}\}$ is $\{\{A, D\}, \{B, C, E\}\} = \mathcal{P}_1$.

The next step is to take the coarsest partition between $\mathcal{P}_0$ and $\mathcal{P}_1$, which is $\{\{A, D\}, \{C\}, \{B, E\}\}$. Then the coarsest partition between this result and the current $\mathcal{P}$ is taken, which leaves it unchanged. This result is then stored as the new partition $\mathcal{P}$ and it is shown in Figure 2.3.

As the current $\mathcal{P}$ is different from the one stored in $\mathcal{P}'$, a new iteration has to be performed. Now, $\mathcal{P}$ overwrites the old $\mathcal{P}'$ and we have to compute $\mathcal{P}_0$ and $\mathcal{P}_1$.

First, the result of the splitters for 0 are $(\{A, D\}, 0)|Q = \{\{B, E\}, \{A, C, D\}\}$, $(\{C\}, 0)|Q = \{\{A, D\}, \{B, C, E\}\}$ and $(\{B, E\}, 0)|Q = \{\{C\}, \{A, B, D, E\}\}$ which results in $\mathcal{P}_0 = \{\{B, E\}, \{A, D\}, \{C\}\}$. Similarly, $(\{A, D\}, 1)|Q = \{\{A, B, C, D, E\}\}$, $(\{C\}, 1)|Q = \{\{A, B, C, D, E\}\}$ and $(\{B, E\}, 1)|Q = \{\{A, D\}, \{B, C, E\}\}$ and results in $\mathcal{P}_1 = \{\{A, D\}, \{B, C, E\}\}$. The coarsest partition between $\mathcal{P}_0$ and $\mathcal{P}_1$ is $\{\{A, D\}, \{C\}, \{B, E\}\}$ which remains unchanged when its coarsest partition is taken with $\mathcal{P}$. This result is then stored in $\mathcal{P}$ and it is equal to $\mathcal{P}'$, which means the algorithm has converged and the minimal graph is shown in Figure 2.3.

The Hopcroft Algorithm, which achieves a lower complexity of $O(n \log n)$ is presented in Appendix A and it is usually the algorithm applied when graph minimization is needed.

## 2.4 Probabilistic Finite State Automata

**Definition 2.6 – Probabilistic Finite State Automata**

*A PFSA is defined as a graph G and a probability function $\pi$ associated to each of its outgoing edges, i.e. $(G, \pi)$. The function $\pi : Q \times \Sigma \to [0, 1]$ such that for a state $q \in Q$, $\sum_{\sigma \in \Sigma} \pi(q, \sigma) = 1$, defines a probability distribution associated with each state of G.* □

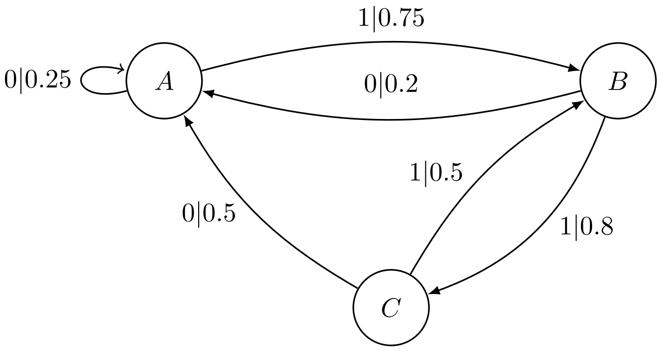

**Figure 2.4:** *A PFSA with the same graph of Figure 2.1.*

**Definition 2.7 – Morph**

*Given a state $q \in Q$, the probability distribution $\mathcal{V}(q) = \{\pi(q, \sigma); \forall\sigma \in \Sigma\}$ associated with $q$ is called its morph.* □

A PFSA is drawn with its graph with each outgoing edge labeled with a symbol and the probability $\pi(q, \sigma)$ associated with that transition. An example of a PFSA is shown in Figure 2.4. for which $Q$ = $\{A, B, C\}$, $\Sigma = \{0, 1\}$. It is the same graph from Figure 2.1 with probabilities associated to its edges to create a PFSA.

Given a PFSA $\{G, \pi\}$, there is a probability associated with each word $\omega \in \Sigma^*$ that can be generated from each state of $G$. From Figure 2.4, starting at the state $A$, it is possible to generate the word $\omega = 1011001$ (as $\delta^*(A, \omega) = B$) by taking a path going to states $B, A, B, C, A, A$ and $B$ and concatenating the labels of the path from each of these transitions. By multiplying the probabilities of these edges, it is seen that = $\Pr(\omega|A) = 0.75 \times 0.2 \times 0.75 \times 0.8 \times 0.5 \times 0.25 \times 0.75 = 0.0084375$.

It is useful to adapt the concept of synchronization word to the context of PFSA as defined in [15].

**Definition 2.8 – PFSA Synchronization Word**

*The word $w$ is a synchronization word if, $\forall u, v \in \Sigma^*$,*

$$\Pr(u|w) = \Pr(u|vw). \tag{2.5}$$

□

Definition 2.8 means that the probability of obtaining any sequence after the synchronization word $w$ does not depend on whatever came before $w$. The main problem with this definition is the fact that is not possible to check (2.5) for all $u \in \Sigma^*$ and for all $v \in \Sigma^*$ as there are an infinite number of sequences.The solution is to use (2.6),

$$\Pr(u|w) = \Pr(u|vw), \forall u \in \cup_{i=1}^{L_1} \Sigma^i, \forall v \in \cup_{j=1}^{L_2} \Sigma^j, \tag{2.6}$$

where $L_1$ and $L_2$ are called precision parameters. This means that all words $u$ of length up to $L_1$ are checked as past previous to $w$ and all words $v$ up to length $L_2$ are checked as continuations. This limits the number of tests to be performed, as the tests have to check $|\Sigma|^{L_1+1} - 1$ previous words and $|\Sigma|^{L_2+1} - 1$ continuation words.

A synchronization word is a good starting point to model a system from its output sequence because the probability of its occurrence does not depend on what comes before it. Therefore, its prefix can be regarded as a transient.

### 2.4.1 Initial Partition for PFSA

In the current work, when applying a graph minimization algorithm (such as Moore or Hopcroft) on a graph $G$ of a PFSA, the following criterion is used to create the initial partition $\mathcal{P}$:

▷ Two states $p$, $q \in Q$ are grouped together in an equivalence class if their morphs are equivalent via a statistical test, i.e., the null hypothesis $\mathcal{V}(p) = \mathcal{V}(q)$ is true for a confidence level $\alpha$;

▷ Use clustering techniques such as the one described in the next section to group states with similar morphs together.

## 2.5 Clustering

The algorithm presented in Chapter 5, a technique from machine learning called clustering, is used to reduce the size of a PFSA.

Clustering refers to a series of techniques used take a set of $M$ objects and gather them into $K$ groups, called clusters, that share some sort of similarity [23]. There are several motivations and applications for clustering, but in this current work we are interested in applying it to cluster states that share similar morphs.

### 2.5.1 K-Means

The K-Means algorithm, also called the Lloyd's Algorithm [24], aims to put $M$ data points in an $I$-dimensional space into $K$ clusters. Each cluster $k$ is represented by a vector $\mathbf{m}^{(k)}$ called its mean or its centroid. The data points are denoted by $\{\mathbf{x}^{(n)}\}$ with $n = 1, \ldots, M$ and each $\mathbf{x}^{(n)}$ has components $x_i$ with $i = 1, \ldots, I$. The centers of the clusters (the means $\mathbf{m}^{(k)}$) are initialized to points in $\mathbb{R}^I$, the data points closest to each mean are clustered together and the value of these means are updated to the mean value of the recently-clustered data points. The process is repeated until the means do not move after a new iteration, which means that they are at the centroid of the final set of clusters.

It is necessary to define a metric to measure the distance between data point vectors. For simplicity, it is assumed that $\mathbf{x}^{(n)} \in \mathbb{R}^I$, for $n = 1, \ldots, M$ and the following equation is used as the distance between vectors **x** and **y**:

$$d(\mathbf{x}, \mathbf{y}) = \frac{1}{2} \sum_i (x_i - y_i)^2. \tag{2.7}$$

The algorithm works by first assigning initial values to $\mathbf{m}^{(k)}$. It then assigns each data point **x** of $\mathbf{x}^{(n)}$ to a cluster $j$ for which the distance $d(\mathbf{x}, \mathbf{m}^{(j)})$ given by (2.7) is minimum. The values of $\{\mathbf{m}^{(k)}\}$ are then updated to be the mean value of the data points assigned to each cluster. These assignment and update steps are repeated until no assignment is changed after an iteration.

The full algorithm is shown in Algorithm 2. Its steps are discussed in more detail in the following sections.

**Algorithm 2** K-Means

```
Inputs: $\{\mathbf{x}^{(n)}\}, K$
Outputs: $K$ clusters containing all data points $\{\mathbf{x}^{(n)}\}$
## Step 1: Initialization:
for $k \in 1, \ldots, K$ do
    $\mathbf{m}^{(k)} \leftarrow \textit{random}()$
## Step 2: Assignment
for $i = 1, \ldots, M$ do
    $k^{(i)} \leftarrow \arg\min_j \{d(\mathbf{m}^{(j)}, \mathbf{x}^{(i)})\}$
    $r_k^{(i)} \leftarrow \textit{assign}(k^{(i)})$
## Step 3: Update
for $j = 1, \ldots, K$ do
    $R^{(j)} \leftarrow \sum_n r_j^{(n)}$
    $\mathbf{m}^{(j)} \leftarrow \frac{\sum_n r_j^{(n)} \mathbf{x}^{(n)}}{R^{(j)}}$
## Step 4: Repeat
if Assignments change then
    Repeat steps 2 and 3
else
    return $\{r_k^{(n)}, \forall n \in 1, \ldots, M\}$
```

#### Step 1: Initialization

The most common initialization method for the means $\mathbf{m}^{(k)}$ is to assign random values of $\mathbb{R}^I$ to each mean. There are other heuristics that can be used [25], but the random initialization is usually enough for most applications.

#### Step 2: Assignment

For each $\mathbf{x}^{(i)}$ with $i = 1, \ldots, M$, the distance $d(\mathbf{x}^{(i)}, \mathbf{m}^{(j)})$ is calculated for every $\mathbf{m}^{(j)}$ with $j = 1, \ldots, K$. The $j$ for which the distance is minimum is stored in $k^{(j)}$.

The function $r_k^{(n)}$ is defined as

$$r_k^{(n)} = \begin{cases} 1, & \text{if } k^{(n)} = k \\ 0, & \text{otherwise} \end{cases} \tag{2.8}$$

it is used to label each data point, as it is 1 for $k = k^{(n)}$ and this value of $k^{(n)}$ is found in the previous operation and (2.8) is updated for the values found.

#### Step 3: Update

This step updates the values of the means by taking the mean position of the data points contained in each cluster. For a given cluster $j$, $\mathbf{m}^{(j)}$ is updated with $\sum_n r_j^{(n)} \mathbf{x}^{(n)} / R^{(j)}$, in which $\sum_n r_k^{(n)} \mathbf{x}^{(n)}$ gives only the data points in cluster $j$ as $r_j^{(n)}$ is 0 for any data point outside of the cluster and $R^{(j)} \leftarrow \sum_n r_j^{(n)}$ is the total amount of data points in cluster $j$.

#### Step 4: Repeat

Steps 2 and 3 are then repeated until the clustering before and after the iteration are the same, which means that the means reached the centroid of each cluster and that the task is finished.

#### Time Complexity

There are many methods to obtain the K-Means algorithm complexity but the average scenario is given by $O(MKIj)$ [26], where $M$ is the number of data points, $K$ is the number of clusters, $I$ is the dimension of the data point set and $j$ is the number of iterations until convergence, which is usually small.

The next section describes two PFSA construction algorithms present in the literature.

**Table 2.1:** *Probabilities of words of length up to 3 obtained from a binary sequence S.*

| $\ell = 1$ | Prob. | $\ell = 2$ | Prob. | $\ell = 3$ | Prob. |
|---|---|---|---|---|---|
| 0 | 0.51 | 00 | 0.27 | 000 | 0.15 |
| 1 | 0.49 | 01 | 0.23 | 001 | 0.12 |
| | | 10 | 0.24 | 010 | 0.12 |
| | | 11 | 0.25 | 011 | 0.11 |
| | | | | 100 | 0.12 |
| | | | | 101 | 0.12 |
| | | | | 110 | 0.11 |
| | | | | 111 | 0.14 |

## 2.6 PFSA Modeling Algorithms

In this section, two algorithms that construct a PFSA from a sequence $S$ of length $N$ over an alphabet $\Sigma$ are presented: D-Markov Machines[27] and CRISSiS[15].

### 2.6.1 D-Markov Machines

A D-Markov machine is a PFSA that generates symbols that depend only on the history of at most $D$ previous symbols, in which $D$ is the machine's depth. It generates a Markov process $\{s_n\}$ of order $D$,

$$\Pr(s_n|\dots s_{n-D}\dots s_{n-1}) = \Pr(s_n|s_{n-D}\dots s_{n-1}).$$

To construct a D-Markov Machine, first $|\Sigma|^D$ states are created in the set $Q$ labeled with each of the $D$-length subsequences of alphabet $\Sigma$. For each $\tau \in \Sigma$, the $\tau$-labeled transition from state $q$ is determined as follows. Consider that state $q$ labeled as $q = \sigma_1\sigma_2\dots\sigma_D$ with $\sigma_n \in \Sigma$, for $n = 1, 2, \dots, D$. There is a transition from $q$ to $q' = \sigma_2\dots\sigma_D\tau$ for $\tau \in \Sigma$, that is $\delta(q,\tau) = q'$, with probability:

$$\Pr(\tau|q) = \frac{\Pr(q\tau)}{\Pr(q)}, \tag{2.9}$$

where $\Pr(q)$ and $\Pr(q\tau)$ are the probabilities of $q$ and $q'$ occurring in the original sequence, respectively.

For example, consider a binary sequence S with the probabilities of words of length $\ell \leq 3$ shown in Table 2.1. To build a 2-Markov Machine, the states are 00, 01, 10 and 11. Using Equation (2.9), the D-Markov machine shown in Figure 2.5 is built.

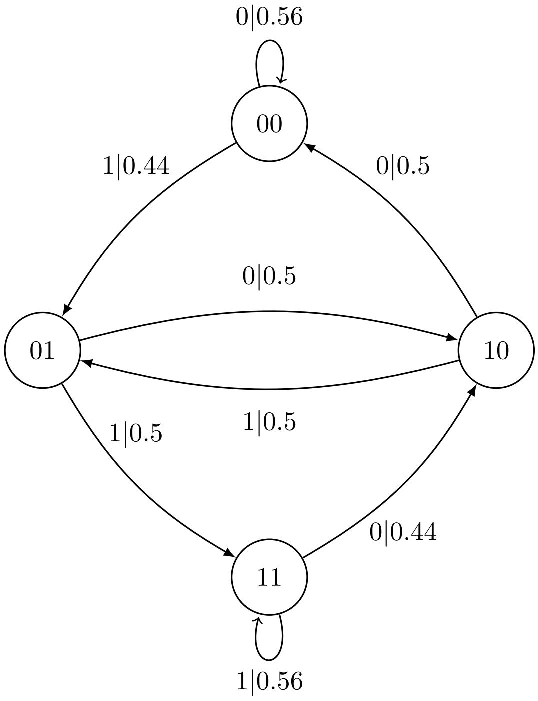

**Figure 2.5:** *A D-Markov machine with sequence S and D = 2.*

### 2.6.2 CRISSiS

The Compression via Recursive Identification of Self-Similar Semantics (CRISSiS) algorithm is presented in [15]. It assumes that a sequence *S* over an alphabet $\Sigma$ of length *N* is generated by a synchronizable and irreducible PFSA. CRISSiS is shown in Algorithm 3 and it consists of three steps:

#### Identification of Shortest Synchronization Word

Using the definition of a synchronization word given in (2.6), CRISSiS uses brute force to find the shortest synchronization word with fixed parameters $L_1$ and $L_2$. This is shown in Algorithm 4 where each state morph is checked with the morph of its extensions up to a length $L_2$. If all statistical tests are positive for a given word $\omega$, it is returned as the synchronization word $\omega_{syn}$.

The auxiliar function *hypothesisTest* is used to check (2.6) for a given value of $\alpha$. It can be implemented either as the $\chi^2$ test or the Kolmogorov-Smirnov test. It returns True when the test states that the probabilities are statistically the same or False when they are not.

**Algorithm 3** CRISSiS

```
**Inputs:** Symbolic string $S, \Sigma, L_1, L_2$, significance level $\alpha$
**Outputs:** PFSA P̂ = $\{G, \pi\}$
**## Identification of Shortest Synchronization Word:**
$\omega_{syn} \leftarrow$ null
$d \leftarrow 0$
**while** $\omega_{syn}$ is null **do**
    $\Omega \leftarrow \Sigma^d$
    **for all** $\omega \in \Omega$ **do**
        **if** (isSynString($\omega, L_1, L_2$)) **then**
            $\omega_{syn} \leftarrow \omega$
            **break**
    $d \leftarrow d + 1$
**## Recursive Identification of States:**
$Q \leftarrow \{\omega_{syn}\}$
Q̃ $\leftarrow \{\omega_{syn}\sigma, \forall\sigma \in \Sigma\}$
$\delta(\omega_{syn}, \sigma) = \omega_{syn}\sigma\forall\sigma \in \Sigma$
**for all** $\omega \in$ Q̃ **do**
    **if** $\omega$ occurs in *S* **then**
        $\omega^* \leftarrow$ matchStates($\omega, Q, L_2$)
        **if** $\omega^*$ is null **then**
            Add $\omega$ to $Q$
            Add $\omega\sigma$ to Q̃ and $\delta(\omega, \sigma) = \omega_{syn}\sigma$, $\forall\sigma \in \Sigma$
        **else**
            Replace all $\omega$ by $\omega^*$ in $\delta$
**## Estimation of Morph Probabilities:**
Find $k$ such that $S[k]$ is the symbol after the first occurrence of $\omega_{syn}$ in *S*
Initialize $\pi$ to zero
*state* $\leftarrow \omega_{syn}$
**for all** i $\geq$ *k* in *S* **do**
    $\pi($*state*$, S[i]) \leftarrow \pi($*state*$,S[i]) + 1$
    *state* $\leftarrow \delta($*state*$,S[i])$
Normalize $\pi$ for each state
```

**Recursive Identification of States**

States are equivalence class of strings under Nerode equivalence class. To check if two states $q_1$ and $q_2$ are equivalent, it would be necessary to test that

$$\Pr(v|q_1) = \Pr(v|q_2), \forall v \in \Sigma^*, \tag{2.10}$$

but as it is not feasible to test for strings up to infinite length, a simplified version checks for string up to length $L_2$,

$$\Pr(v|q_1) = \Pr(v|q_2), \forall v \in \Sigma^d, d = 1, \ldots, L_2. \tag{2.11}$$

If two states pass the statistical test using (2.11), they are considered to be statistically the same. Strings $q_1$ and $q_2$ need to be synchronization words in order to use (2.11). If $\omega$ is a synchronization word for some $q_i \in Q$, then $\omega\tau$ is also a synchronization word for $q_j = \delta(q_i, \tau)$.

The next procedure starts by letting $Q$ be the set of states that will receive the states for the final machine found by the algorithm and $\tilde{Q}$ is the set of states to be checked if they are equivalent to some state in $Q$ or if they are a state on their own. It is initialized with the descendants of $\omega_{syn}$. The function $\delta$ for the machine is initialized with $\delta(\omega_{syn}, \sigma)$ equal to $\omega_{syn}\sigma$ for all $\sigma \in \Sigma$. This is represented by a tree using $\omega_{syn}$ as the root node to $|\Sigma|$ children, which are the states in $\tilde{Q}$. Each one of the children nodes is regarded as a candidate state. Each one of them is tested using a statistical test with confidence level $\alpha$ with each of the states in $Q$. If a match between some $\omega \in \tilde{Q}$ and some $\omega^* \in Q$ is found, the child state is removed and all the transitions to $\omega$ are redirected to $\omega^*$ (i.e. every $\delta(q, \sigma) = \omega$ now becomes $\delta(q, \sigma) = \omega^*$ for any $q \in Q$ and any $\sigma \in \Sigma$). If it does not match any state in $Q$, it is considered a new state and it is then added to $Q$ and it should also be split in $|\Sigma|$ new candidate states which are added to $\tilde{Q}$. This procedure is repeated until no new candidate states have

**Algorithm 4** isSynString($\omega, L_1, L_2$)

```
Outputs: true or false
for $D$ = 0 to $L_1$ do
    for all $u \in \Sigma^D$ do
        for $d$ = 0 to $L_2$ do
            for all $v \in \Sigma^d$ do
                if hypothesisTest($\Pr(u|\omega v), \Pr(u|\omega v), \alpha) =$ False then
                    return False
return true
```

**Algorithm 5** matchStates($\omega, Q, L_2$)

```
for all $q \in Q$ do
    for $d = 0$ to $L_2$ do
        for all $v \in \Sigma^d$ do
            if hypothesisTest($\Pr(\omega v|q), \Pr(v|q), \alpha$) = False then
                return $q$
return null
```

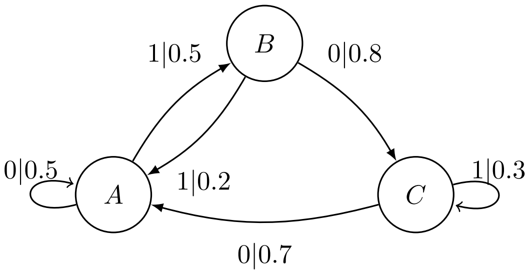

**Figure 2.6:** *The Tri-Shift PFSA.*

to be visited. As CRISSiS should be applied to estimate a finite PFSA, this procedure is guaranteed to terminate.

### Estimation of Morph Probabilities

To recover the morphs of each state in $Q$ found in the last step, the sequence *S* (starting from the first occurrence of $\omega_{syn}$) is fed to the PFSA starting at state $\omega_{syn}$ and transition following the symbols of the original sequence. Each transition is counted and then normalized in order to recover an estimation of each state morph.

### Example

The PFSA in Figure 2.6, which is called Tri-Shift in this work, is presented in [15]. It is synchronyzable and works over a binary alphabet. It is used in this example to generate a string *S* of length 10000. Table 2.2 gives the estimated probabilities of subsequences occurring in *S*. In this example, $L_1 = L_2 = 1$.

First, the synchronization word needs to be found. States 0, 1 and so on are checked with (2.6). Starting by 0, $\Pr(0|0) = 0.5607$ is not equal to $\Pr(1|0) = \Pr(01) = 0.4393$ which means they do not pass the $\chi^2$ test. Then, the state 1 is tested, which also fails ($\Pr(0|1) = 0.7387 \neq 0.2613 = \Pr(1|1)$. For state 00, the probabilities are relatively close ($\Pr(0|00) = 0.5 = \Pr(1|00)$) and it passes the test, giving 00 the status of synchronization word.

**Table 2.2:** *Probabilities of words generated by the Tri-Shift.*

| $\ell = 1$ | Prob. | $\ell = 2$ | Prob. | $\ell = 3$ | Prob. | $\ell = 4$ | Prob. | $\ell \geq 5$ | Prob. |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.62711 | 00 | 0.35164 | 000 | 0.17565 | 0000 | 0.08673 | 00100 | 0.09881 |
| 1 | 0.37291 | 01 | 0.27546 | 001 | 0.17599 | 0001 | 0.08892 | 00101 | 0.04181 |
| | | 10 | 0.27546 | 010 | 0.21451 | 0010 | 0.14062 | 001000 | 0.0499 |
| | | 11 | 0.09745 | 011 | 0.06094 | 0011 | 0.03536 | 001001 | 0.04891 |
| | | | | 100 | 0.17599 | 0100 | 0.14206 | 001010 | 0.02926 |
| | | | | 101 | 0.09946 | 0101 | 0.07245 | 001011 | 0.01255 |
| | | | | 110 | 0.06094 | 1000 | 0.08892 | | |
| | | | | 111 | 0.03651 | 1001 | 0.08707 | | |
| | | | | | | 1100 | 0.03393 | | |
| | | | | | | 1101 | 0.02701 | | |

The second step starts by defining the synchronization word state 00 adding it to $Q$ and splitting it into two candidates states, 000 and 001 (Figure 2.7). Each candidate has its morphs compared to that of 00, which is the only state in $Q$, via (2.11). $\mathcal{V}(000) = [0.494, 0.506]$ is considerably close to $\mathcal{V}(00) = [0.500, 0.500]$, so they pass the statistical test and 00 and 000 are considered to be an equivalent state. 000 is removed and the edge going from 00 to 000 becomes a self-loop from 00 to itself. On the other hand, $\mathcal{V}(001) = [0.800, 0.200]$ is considerably different from $\mathcal{V}(00)$, therefore it is considered a state and added to $Q$ (which now becomes {00, 001}) and then it is split into two new candidates (Figure 2.8).

The same procedure is then repeated for the candidates 0010 and 0011. $\mathcal{V}(0010) = [0.703, 0.297]$ is different from both 00 and 001, therefore it is a new state, it is added to $Q$ and split into the new candidates 00100 and 00101. $\mathcal{V}(0011) = [0.500, 0.500]$ passes the test with $\mathcal{V}(00)$, which means that 0011 is removed and the edge from 001 to 0011 goes back to 00. This leads to the configuration in Figure 2.9, with $Q$ = {00, 001, 0010}.

The next candidates are similar to two states in $Q$ ($\mathcal{V}(00100) = [0.505, 0.495]$ passes with 00 and $\mathcal{V}(00101) = [0.700, 0.300]$ passes with $\mathcal{V}(0010)$), so both are removed and its edges rearranged to the configuration in Figure 2.10, which is the same graph as the original Tri-Shift, showing that CRISSiS recovered the PFSA graph. All that is left is to feed the input sequence to the graph and computing the morph probabilities, which recovers an accurate Tri-Shift PFSA.

### Time Complexity

As shown in [15], CRISSiS operates with a time complexity of $O(N) \cdot (|\Sigma|^{O(|Q|^3)+L_1+L_2} + |Q||\Sigma|^{L_2})$, where $N$ is the length of the input sequence, $|\Sigma|$ is the alphabet size, $|Q|$ is the number

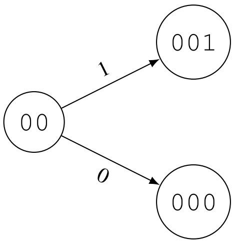

**Figure 2.7:** *Tree with 00 at its root, Q = {00}.*

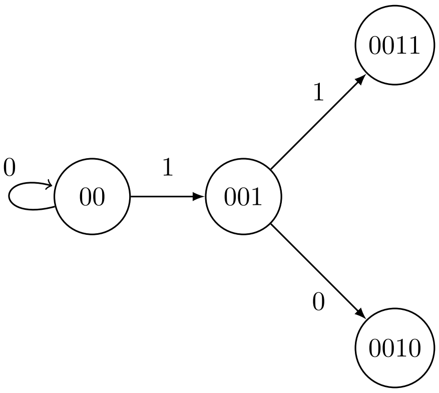

**Figure 2.8:** *Second iteration of three, Q = {00, 001}.*

of states in the original PFSA and $L_1$ and $L_2$ are parameters determining how much of the past and future of a state is needed to determine it. It is stated that as $L_1$ and $L_2$ are both usually small, it does not affect the performance greatly, even though the algorithm is exponential in these parameters.

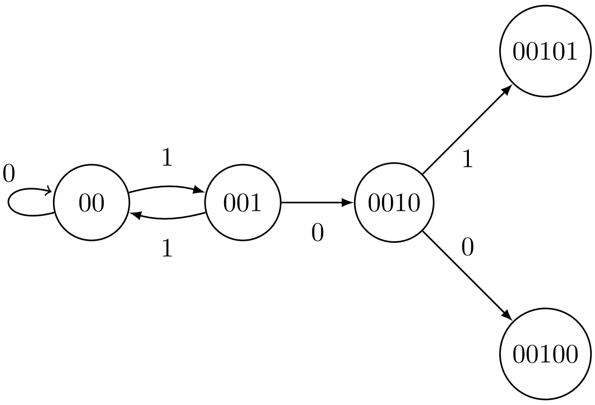

**Figure 2.9:** *Third iteration of the three, Q = {00, 001, 0010}.*

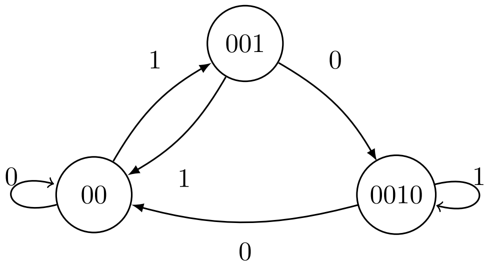

**Figure 2.10:** *Recovered Tri-Shift topology.*

# Chapter 3

# An Algorithm for Synchronizable Dynamical Systems

In this chapter, the proposed PFSA construction algorithm to model a dynamical system from its output sequence $S$ is presented. The first section discusses an algorithm to find synchronization words which has lower complexity than the brute force method used by CRISSiS. Later, the PFSA construction algorithm is shown. It is further divided in two parts: the leaf connection, which makes sure that all states have outgoing edges and the full ALEPH algorithm which creates the final PFSA for the provided parameters.

Thus, the proposed algorithm consists of the following three steps:

1 Find synchronization words from sequence $S$;

2 Apply a leaf connection criterion for the rooted tree with probabilities $\mathcal{T}$ based on $S$;

3 Apply the ALEPH algorithm for PFSA construction.

## 3.1 A New Algorithm for Finding Synchronization Words

Given a sequence $S$ of length $N$ over an alphabet $\Sigma$ generated by a dynamical system, we introduce in this section an algorithm to find possible synchronization words in $S$. The CRISSiS method uses (2.6) for an extensive, brute force search. The proposed algorithm uses data structures in order to speed up the process. This implies using a structured search to realize less statistical tests, which reduces the time complexity of the algorithm, while also finding not only just one synchronization word, but all of them up to a given length $W$.

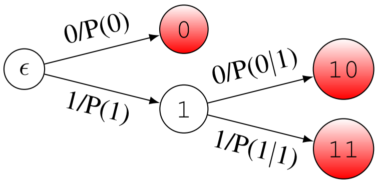

**Figure 3.1:** *Example of a rooted tree with probabilities.*

The proposed algorithm uses a rooted tree with probabilities $\mathcal{T}$ over an alphabet $\Sigma$ to search for synchronization words. At the beginning of the algorithm, all states of $\mathcal{T}$ are considered valid candidates to be synchronization words. A search is performed in $\mathcal{T}$ starting by its root using a statistical test (which compares two state morphs via a test such as $\chi^2$ or Kolmogorov-Smirnov for a given confidence level $\alpha$) to determine whether a state should be expanded. The way the tree is explored guarantees that a state is only tested against other states that have it as a suffix. When a test fails, an expansion algorithm is used to determine how the next states are to be tested. On the other hand, when the test is successful, the state keeps its status as a valid candidate.

A rooted tree with probabilities (RTP) $\mathcal{T}$ over $\Sigma = \{0, 1\}$ is presented via an example in Figure 3.1. It consists of a set of states connected by edges. All states have exactly one predecessor (with the exception of the root state, labeled with the empty string $\epsilon$, which has no predecessors). Leaf states ($0, 10$ and $11$ in the example) have no successors, while the other states have $|\Sigma|$ successors as each element of $\Sigma$ labels its outgoing edges. Those edges are also labeled with the probability of leaving the state with that symbol. Each state is labeled with the string formed from concatenating the symbols in the branches in the path from the root to the current state. The probability of reaching a state is given by multiplying the probabilities labeling the branches in the path from the root state to the current state. For example, consider the leaf state *10*. The path taken from the root state $\epsilon$ is first *1* and then *0*. The probability of reaching this state is $P(1) \times P(0|1)$, that is the probability of leaving the root state with 1 (which is $P(1)$) multiplied by the probability of leaving the state 1 with 0 (that is, $P(0|1)$). The edge probabilities of $\mathcal{T}$ are taken from the conditional probabilities of sub-sequences of *S*.

An RTP has its maximum depth *L* ultimately constrained by the length *N* of *S*. It is good to remind that as the chance of sub-sequences occurring gets smaller as their length increases, the statistics of large sub-sequences might be really poor for a given *N*. This means that using a very large *L* implies that the probabilities of states closer to the leaves tend to be unreliable.

Another data structure used in the algorithm is a dictionary (also called a hash table) [28]. A dictionary $d$ is a mapping between two sets $d : X \to Y$. The elements from $X$ are called the dictionary keys. An entry in the dictionary is the element $y \in Y$ associated to the key $x \in X$ and is denoted by $d[x]$, which is also called the *value* of $x$ in $d$. An entry $d[x]$ might be updated and even deleted from $d$.

As mentioned before, all states of $\mathcal{T}$ start as possible candidates for synchronization words. This is represented by a dictionary called $candidacy$ which takes states as keys. For a given key $k$, $candidacy[k]$ is a boolean value: it is True when $k$ is still a possible synchronization word and False when it is not (i.e. when it failed a statistic test). Thus, $candidacy$ is initialized with a True value for all of its keys. When $candidacy[k] =$ True, $k$ is called a valid state.

The concept of a shortest valid suffix (SVS) also needs to be explained as it is important in one of the algorithm steps via an auxiliary function called *shortestValidSuffix*. For a given word $\omega \in \Sigma^*$, its SVS is the word $\zeta \in \Sigma^*$, $|\zeta| \leq |\omega|$, that is a shortest suffix of $\omega$ for which the state labeled with $\zeta$ in $\mathcal{T}$ is still a valid candidate for synchronization word. The function *shortestValidSuffix* receives the word $\omega = \sigma_1\sigma_2\ldots\sigma_n \in \Sigma^*$, the tree $\mathcal{T}$ and the dictionary *candidacy* as inputs. First $\omega = \sigma_1\sigma_2\ldots\sigma_n$ is reversed, $\omega_{rev} = \sigma_n\sigma_{n-1}\ldots\sigma_1$. Then, the tree $\mathcal{T}$ is traversed according to $\omega_{rev}$, starting at the root $\epsilon$. *candidacy[$\epsilon$]* is checked and if it is true, $\epsilon$ is returned. If not, the state $\delta(\sigma_n, \epsilon)$ is evaluated. At each level $k$ of $\mathcal{T}$, the current state is $c = \delta^*(\sigma_n \ldots \sigma_{n-k}, \epsilon)$. Let $c_{rev}$ be the reversed label of the current candidate (i.e. if $c = \tau_1\tau_2\ldots\tau_m$, $c_{rev} = \tau_m\tau_{m-1}\ldots\tau_1$). If *candidacy*[$c_{rev}$] is True, $c_{rev}$ is returned. If not the next iteration is processed until $\omega_{rev}$ is reached, which means that $\omega$ is its own shortest valid suffix if its candidacy is True or that it has no valid suffix if its candidacy is False.

As an example, take the tree $\mathcal{T}$ represented in Figure 3.2, where the filled states indicate that their candidacy status is True while the white states have them as False. If we wish to check which state is the shortest valid suffix for $\omega = 110$ we first take $\omega_{rev} = 011$ and go to the root. As *candidacy*[$\epsilon$] is False, we go to the next iteration, taking $c = \delta(0, \epsilon) = 0$. The candidacy of $c_{rev} = 0$ is checked, which once again is false and takes us to the next iteration. Now $c = \delta^*(01, \epsilon) = 01$, $c_{rev} = 10$ and *candidacy*[$c_{rev}$] = *candidacy*[10] = True and the function returns $c_{rev} = 10$, i.e. 10 is the shortest valid suffix of 110.

Along with the *candidacy* dictionary, a second dictionary called *suffixes* is created. It also has the states from $\mathcal{T}$ as keys. The associated value to each key is a list of states for which the key is the shortest valid suffix, i.e. the key state is the shortest state to have a *True* value for its candidacy and also is a suffix for all the word in the associated list. Another dictionary, $V$ is created to be used in the expansion algorithm and it is explained later in the context of Algorithm 7.

To find the synchronization words, Algorithm 6 is used. Its inputs are the rooted tree with prob-

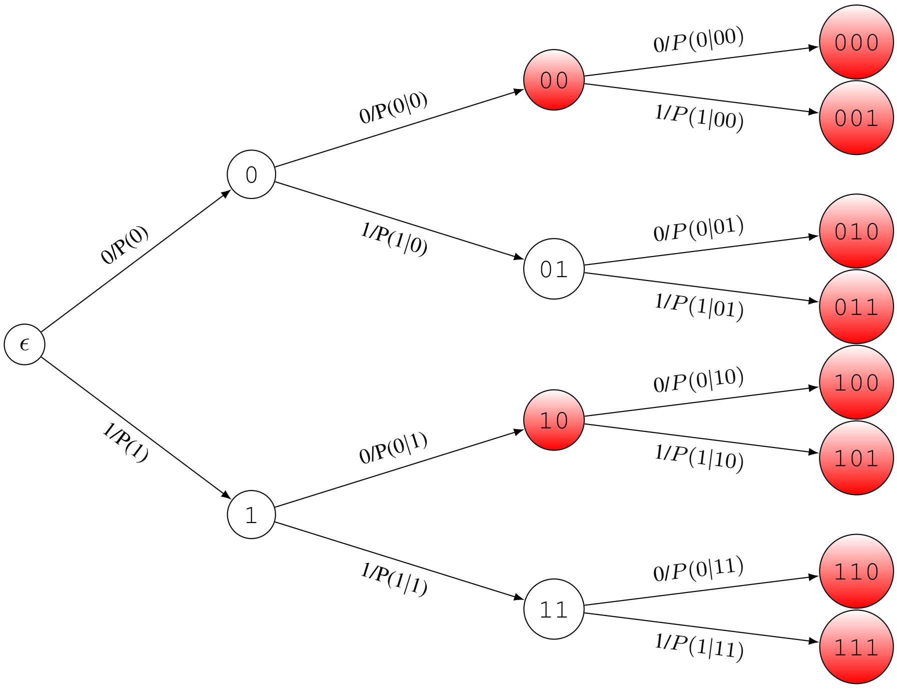

**Figure 3.2:** *Example of binary RTP with $L = 3$.*

abilities $\mathcal{T}$ with maximum depth $L$, the maximum window size $W$, which is a parameter that determines how deep in the tree the algorithm searches. The algorithm starts by creating the queue $\Gamma$ which contains states from $\mathcal{T}$ that are not fully tested for the synchronization word hypothesis during the current iteration. As $\epsilon$ is the only value to be tested in the beginning of the algorithm, only *suffixes*$[\epsilon]$ is initialized with a list of the states $\sigma \in \Sigma$ as they all have $\epsilon$ as their shortest valid suffix. A list $\Theta$ is created and initialized empty. It receives the states from $\mathcal{T}$ which currently have passed the statistical tests. The statistical tests are implemented as either $\chi^2$ or Kolmogorov-Smirnov tests for a given confidence level $\alpha$. This is implemented as the auxiliary function *statisticalTest* which takes as inputs two states and a significance level $\alpha$ and compares the state morphs with a statistical test with the given confidence level and returns True if the test passes and False otherwise.

The main loop then begins. At the start of each iteration, the variable $c$ receives the first element of $\Gamma$ via dequeueing (as $\Gamma$ is a queue, the first element to be inserted into it is the first to be removed) which is represented by the dequeue auxiliary function in line 10 of Algorithm 6. It takes the queue $\Gamma$ as input and returns the elements in its first position. If the label of $c$ is not larger than $W$, a flag $p$ is set to True. If *suffixes*$[c]$ is empty, $p$ stays True. Otherwise, each element $\lambda$ of *suffixes*$[c]$ goes through the statistical test with $c$ in order to check (2.6) and $p$ receives the result of *statisticalTest*$(c, \lambda, \alpha)$. If after all the tests are performed, $p$ retains its True value, $c$ keeps its status as a valid candidate for synchronization word and it is appended at $\Theta$ . If one of the tests fails, $p$ is set to False and no more tests need to be done for $c$. The *candidacy*($c$) is set to False, the list $\Gamma$ and the dictionaries will be expanded according to Algorithm 7 (which will be explained later) and as each element $\theta \in \Theta$ needs to be tested again for the new elements appended to *suffixes*$[\theta]$ after the expansion, all elements of $\Theta$ are queued at the end of $\Gamma$ (using the auxiliary function queue) and then $\Theta$ is set to the empty set. This procedure is repeated until either the queue $\Gamma$ is empty or if all the elements in $\Gamma$ have labels longer than $W$. After one of these conditions is met, it stores $\Theta$ in the list $\Omega_{syn}$, which contains all the elements that passed in all their statistical tests, meaning that they are synchronization words according to (2.6). $\Omega_{syn}$ is then returned as the final value.

**Algorithm 6** findSynchWords($W, \mathcal{T}$)

```
procedure INITIALIZATION
    $\Gamma \leftarrow \{\epsilon \in \mathcal{T}\}$
    $suffixes[\epsilon] \leftarrow \{\delta(\sigma, \epsilon) \forall \sigma \in \Sigma\}$
    $V \leftarrow$ empty dictionary
    for $s \in \mathcal{T}$ do
        $candidacy[s] = $ True
    $\Theta \leftarrow \emptyset$
procedure MAINLOOP
    while $\Gamma \neq \emptyset$ do
        $c \leftarrow$ dequeue($\Gamma$)
        if length($c$) $< W$ then
            $p \leftarrow$ True
            if $suffixes[c] \neq \emptyset$ then
                for every $\lambda \in suffixes[c]$ do
                    $p \leftarrow statisticalTest(c, \lambda, \alpha)$
                    if $p = $ False then
                        candidacy[$c$] $\leftarrow$ False
                        expand($c, V, \mathcal{T}, \Gamma$, candidacy, suffixes)
                        for every $\theta \in \Theta$ do
                            queue($\Gamma, \theta$)
                        $\Theta \leftarrow \emptyset$
                        break
            if $p = $ True then
                $\Theta \leftarrow \Theta \cup \{c\}$
    $\Omega_{syn} \leftarrow \Theta$
    return $\Omega_{syn}$
```

Algorithm 7 updates $\Gamma$ and the dictionaries *suffixes* and $V$ after a statistical test fails. Its goal is to take the descendants of the element $c$ that failed the test and queue them into the end of $\Gamma$ and in turn take their descendants, find their SVS and append them to their SVS *suffixes* dictionary entry. There are some caveats: for an element to be queued into $\Gamma$, it needs to be its own SVS, otherwise it means that there are shorter states that need to be checked first. For every element $d$ that is a descendant of $c$, its SVS $\zeta$ is found. If $d$ is not its own SVS (i.e. $d \neq \zeta$), $d$ is appended to the list $V[\zeta]$. This is done because if in a later iteration $\zeta$ (which should be in $\Gamma$) fails its test, $d$ has the opportunity to check again if it became its own SVS so it might be queued into $\Gamma$.

First, a list $\Psi$ with all the descendants of the state $c$ is created. This list holds the elements that need to be checked if they can be queued into $\Gamma$. They will be queued if they are their own SVS. The dictionary $V$ uses states of $\mathcal{T}$ as keys (for a given key $k$, $V[k]$ is a list of states that have $k$ as SVS). When an element is not its own SVS it cannot be added to $\Gamma$ and so it is added to a list in $V$. In a later call to Algorithm 7 it might have become its own SVS and so it has to be checked again. If there is a list associated to $c$ in $V$, all its elements are appended to $\Psi$. The entry $V[c]$ is then deleted as it no longer has a use.

The next step is to check if each element $d$ in $\Psi$ are their own SVS using the *shortestValidSuffix* function. This function will return a state $\zeta$ which is the SVS of $d$. If $d = \zeta$, $d$ is queued at the end of $\Gamma$. After this, for each descendant $t$ of $d$, $t$ has its SVS $\tau$ found and *suffixes*[$\tau$] has $t$ appended to it, as now $t$ has to be checked against $\tau$.

When $\zeta \neq d$, $V[\zeta]$ has $d$ appended to it. Later on, if $\zeta$ fails one of its tests, $d$ has to be checked again to see if it is now its own SVS.

### 3.1.1 An Example

To illustrate how the algorithm works, the synchronization word finding algorithm is applied to the Tri-Shift (from Section 2.6.2), as the results can be compared to CRISSiS. All statistical tests in this section use the $\chi^2$ test with $\alpha = 0.95$. The initial RTP with $L = 4$ is shown in Figure 3.3. We consider $W = 3$. The queue $\Gamma$ is initialized with the root of $\mathcal{T}$. The dictionary *suffixes* is initialized with *suffixes*[$\epsilon$] = $\{0, 1\}$. $V$ is initialized as an empty dictionary, $\Theta$ is initialized as an empty list and all the states start with their candidacy set to True.

As $\Gamma$ is not empty, it is dequeued and $c = \epsilon$, which has a label length of zero and is shorter than $W = 3$. It then proceeds to iterate through *suffixes*[$c$] = *suffixes*[$\epsilon$] = $\{0, 1\}$ and $p$ is set to true. It first compares *statisticalTest*$(\epsilon, 0, \alpha)$. As the morphs are [0.6276, 0.3724] and [0.5615, 0.4385], the test fails, which means $\epsilon$ candidacy is set to False and the expansion algorithm is called.

**Algorithm 7** expand($c, V, \mathcal{T}, \Gamma$, candidacy, suffixes)

```
procedure EXPAND Γ
    $\Psi \leftarrow \{\delta(\sigma, c), \forall\sigma \in \Sigma\}$
    if $c$ is a key of $V$ then
        $\Psi \leftarrow \Psi \cup V[c]$
        delete $V[c]$
    for every $d \in \Psi$ do
        $\zeta \leftarrow$ shortestValidSuffix($\mathcal{T}, d,$ candidacy)
        if $\zeta = d$ then
            queue(Γ, ζ)
            for $t \in \{\delta(\sigma, \zeta)\forall\sigma \in \Sigma\}$ do
                $\tau \leftarrow$ shortestValidSuffix($\mathcal{T}, t,$ candidacy)
                suffixes$[\tau] \leftarrow$ suffixes$[\tau] \cup t$
        else
            $V[\zeta] \leftarrow V[\zeta] \cup \{d\}$
```

The list $\Psi$ is initialized with the direct descendants of $c = \epsilon$, that is $\Psi = \{0, 1\}$. $V[\epsilon]$ is empty and can be disregarded. It is easy to check that all elements in $\Psi$ are their own shortest valid suffixes after $\epsilon$ candidacy becomes false (as seen in Figure 3.4, since the state is not filled). This means both of them are queued into Γ, so that $\Gamma = \{0, 1\}$. For both 0 and 1, they are their direct descendants shortest valid suffixes, which means that *suffixes*[0] = {00, 10} and *suffixes*[1] = {01, 11}. The expansion algorithm returns to the synchronization algorithm. The list Θ is appended to the end of Γ, but as it is currently empty it does not change Γ. This ends the first iteration.

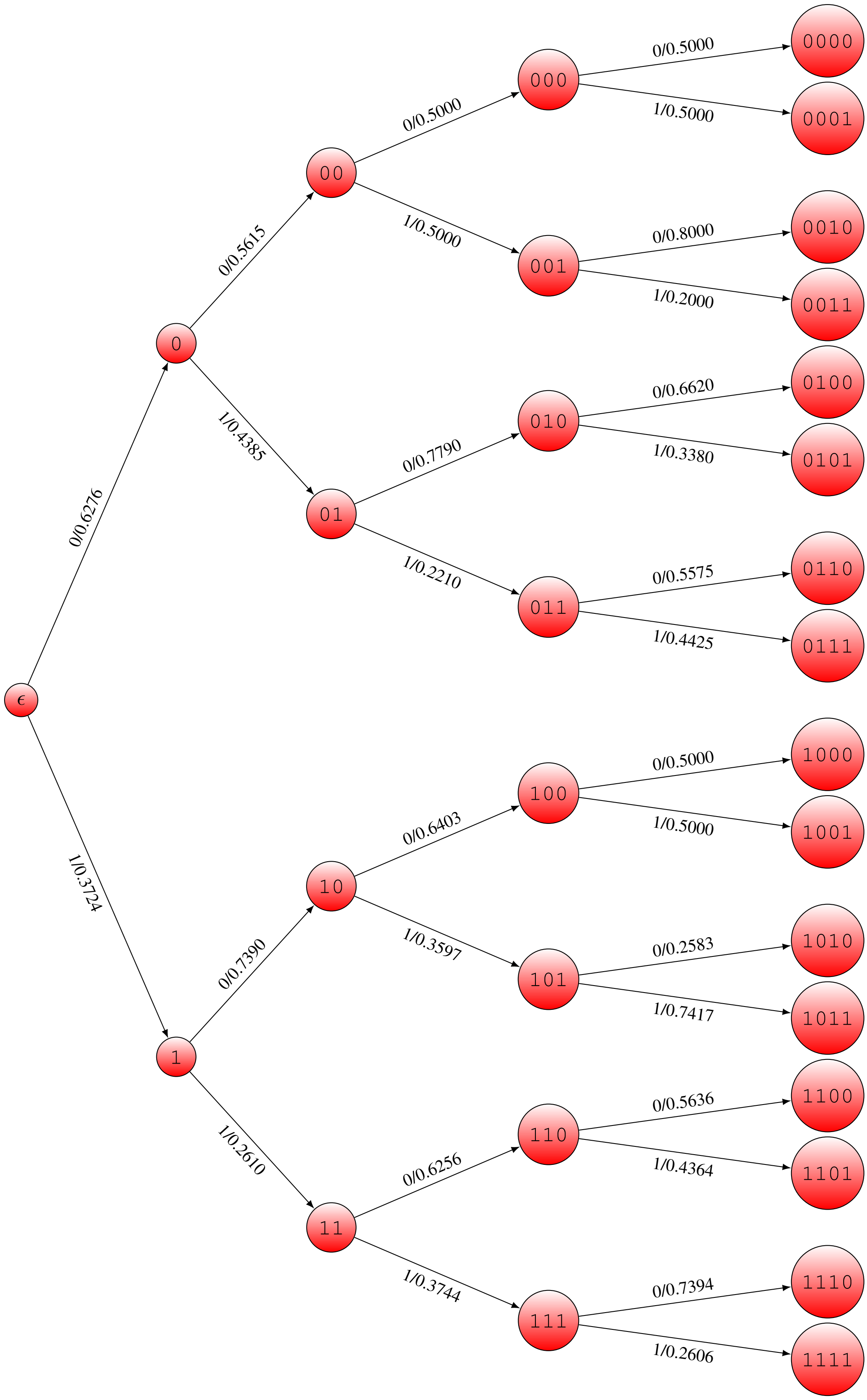

**Figure 3.3:** *Input Rooted Tree with Probabilities $\mathcal{T}$ for the Tri-Shift Example.*

At the beginning of the next iteration $\mathcal{T}$ is shown in Figure 3.4, $\Gamma = \{0, 1\}$ and when it is dequeued, $c = 0$, whose label is still shorter than $W$. The list *suffixes*[0] $= \{00, 10\}$ has each of its elements tested. First to be tested is 00 and *statisticalTest*$(0, 00, \alpha)$ returns False as $\mathcal{V}(0) = [0.5615, 0.4385]$ diverges significantly from $\mathcal{V}(00) = [0.5, 0.5]$. This means that *candidacy*[0] is set to False and the expansion algorithm is called.

For $c = 0$, the expansion algorithm has $\Psi = \{00, 01\}$ and 0 is not among the keys of $V$, so no other elements are appended to $\Psi$. First, the SVS is checked for 00 and by examining the tree, it is observed that it is its own shortest valid suffix. This means that 00 is queued into $\Gamma$. Its children, 000 and 001 have 00 and 1 as shortest valid suffixes, so the *suffixes* dictionary is updated to *suffixes*[000] = $\{000\}$ and *suffixes*[1] = $\{01, 11, 001\}$. Next, the shortest valid suffix of 01 is shown to be 1, which means it is not its own shortest valid suffix. This means it has to be appended to $V[1]$, which makes it $V[1] = \{01\}$. The empty list $\Theta$ is once again appended to $\Gamma$ and then emptied.

In the beginning of the next iteration, we have $\Gamma = \{1, 000\}$, $V[1] = \{01\}$, $\Theta = \emptyset$, suffixes[1] = $\{01, 11, 001\}$, suffixes[00] = $\{000\}$ and $\mathcal{T}$ is shown in Figure 3.5. $\Gamma$ is dequeued and $c = 1$, *suffixes*[1] = $\{01, 11, 001\}$ is iterated through. First, *statisticalTest*$(1, 01, \alpha)$ is checked to be false ($[0.779, 0.221]$ against $[0.739, 0.261]$) making *candidacy*[1] = False and the call to the expansion algorithm.

In the expansion algorithm, $\Psi = \{10, 11\}$ and it is appended of 01 because $V[1] = \{01\}$, making $\Psi = \{10, 11, 01\}$. Now that both *candidacy*[0] = *candidacy*[1] = False, all of them are their own shortest valid suffixes and they are their children nodes' shortest valid suffixes. Thus, $\Gamma = \{00, 01, 10, 11\}$ and *suffixes*[00] = $\{000, 100\}$, *suffixes*[01] = $\{001, 101\}$, *suffixes*[10] = $\{010, 110\}$ and *suffixes*[11] = $\{011, 111\}$. Once again $\Theta$ is appended in $\Gamma$ and emptied.

The fourth iteration has $c = 00$, *suffixes*$[c]$= $\{000, 100\}$ and $\mathcal{T}$ as in Figure 3.6. All the states in *suffixes*$[c]$ have the same morph as $c$, so it passes all its tests, keeps its candidacy as True and it is added to $\Theta$.

At the beginning of the next iteration, $\Gamma = \{01, 10, 11\}$, $\Theta = \{00\}$, *suffixes*[01] = $\{001, 101\}$, *suffixes*[10] = $\{010, 110\}$ and *suffixes*[11] = $\{011, 111\}$ and $\mathcal{T}$ is still as in Figure 3.6.After dequeueing, $c = 01$ and *suffixes*$[c]$= $\{001, 101\}$. The test *statisticalTest*$(01, 001)$ fails ([0.779, 0.221] against [0.8, 0.2]). During the expansion, $\Psi = \{010, 011\}$ and $V[01] = \emptyset$. 010 is its own shortest valid suffix, but 011 is not (its shortest valid suffix is 11). This means $V[11] = \{011\}$ and $\Gamma$ appends 010. The children of 010 are 0100 and 0101 and will be added to *suffixes*[00] and *suffixes*[101]. After the expansion, $\Theta = \{00\}$ is appended to $\Gamma$.

In the sixth iteration, $\Gamma = \{10, 11, 010, 00\}$, $V[11] = \{011\}$, *suffixes*[10] = $\{010, 110\}$, *suffixes*[11] = $\{011, 111\}$, *suffixes*[010] = $\emptyset$ and *suffixes*[00] = $\{000, 100, 0100\}$ and $\mathcal{T}$ as in Figure 3.7. $c = 10$, *suffixes*[$c$]= $\{010, 110\}$ and *statisticalTest*$(10, 010)$ fails ([0.6403, 0.3597] against [0.662, 0.338]). The expansion has $\Psi = \{100, 101\}$. 100 has 00 as shortest valid suffix, therefore it is not appended to $\Gamma$ and $V[00] = \{100\}$. 101 is its own shortest valid suffix so it is queued into $\Gamma$ and its children are 1010 and 1011 which are added to *suffixes*[010] and *suffixes*[11].

The following iteration has $\Gamma = \{11, 010, 00, 101\}$, $V[11] = \{011\}$, $V[00] = \{100\}$, *suffixes*[11] = $\{011, 111, 1011\}$, *suffixes*[010] = $\{1010\}$, *suffixes*[00] = $\{000, 100, 0100\}$ and *suffixes*[101] = $\{0101\}$ and $\mathcal{T}$ as in Figure 3.8. $c = 11$ and *suffixes*[$c$]= $\{011, 111, 1011\}$. The test *statisticalTest*$(11, 011)$ fails ([0.6256, 0.3744] against [0.5575, 0.4425]). In the expansion for $c = 11$, $\Psi = \{110, 111, 011\}$ (because $V[11] = \{011\}$). All of them are their own shortest valid suffixes, so they are appended to $\Gamma$ and suffixes is updated with *suffixes*[00] receiving 1100; *suffixes*[101] receives 1101; *suffixes*[110], 1110 and 0110; *suffixes*[111], 1111 and 0111.

In the eighth iteration, $\Gamma = \{010, 00, 101, 110, 111, 011\}$, $V[00] = \{100\}$, *suffixes*[010] = $\{1010\}$, *suffixes*[00] = $\{000, 100, 0100, 1100\}$, *suffixes*[101] = $\{0101, 1101\}$, *suffixes*[110] = $\{1110, 0110\}$, *suffixes*[111] = $\{1111, 0111\}$ and *suffixes*[011] = $\{1011\}$ and $\mathcal{T}$ as in Figure 3.9. $c = 010$ which is now equal in length to $W = 3$, which means it is no longer tested.

In the ninth iteration, $c = 00$ and *suffixes*$[c] = \{000, 100, 0100, 1100\}$. All of these states have morphs close to [0.5, 0.5] and they pass in all statistical tests. This keeps 00 candidacy as True and it is once again added to $\Theta$. The rest of the elements in $\Gamma = \{101, 110, 111, 011\}$ have labels with length grater than or equal to $W$ so they are all skipped and the algorithm returns $\Theta = \{00\}$. This result is the same as the one found by CRISSiS.

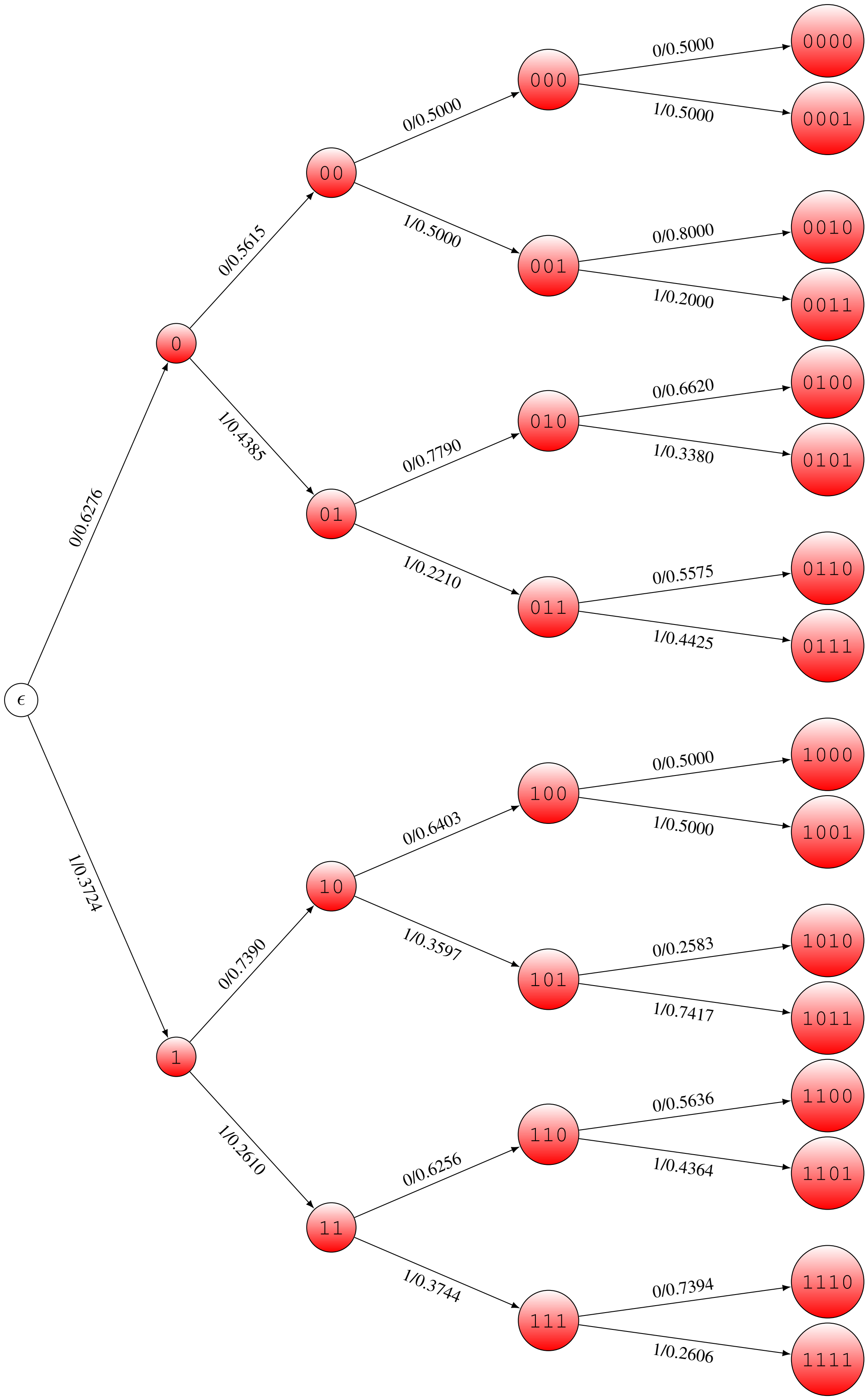

**Figure 3.4:** *Rooted Tree with Probabilities $\mathcal{T}$ for the Tri-Shift Example after the first iteration.*

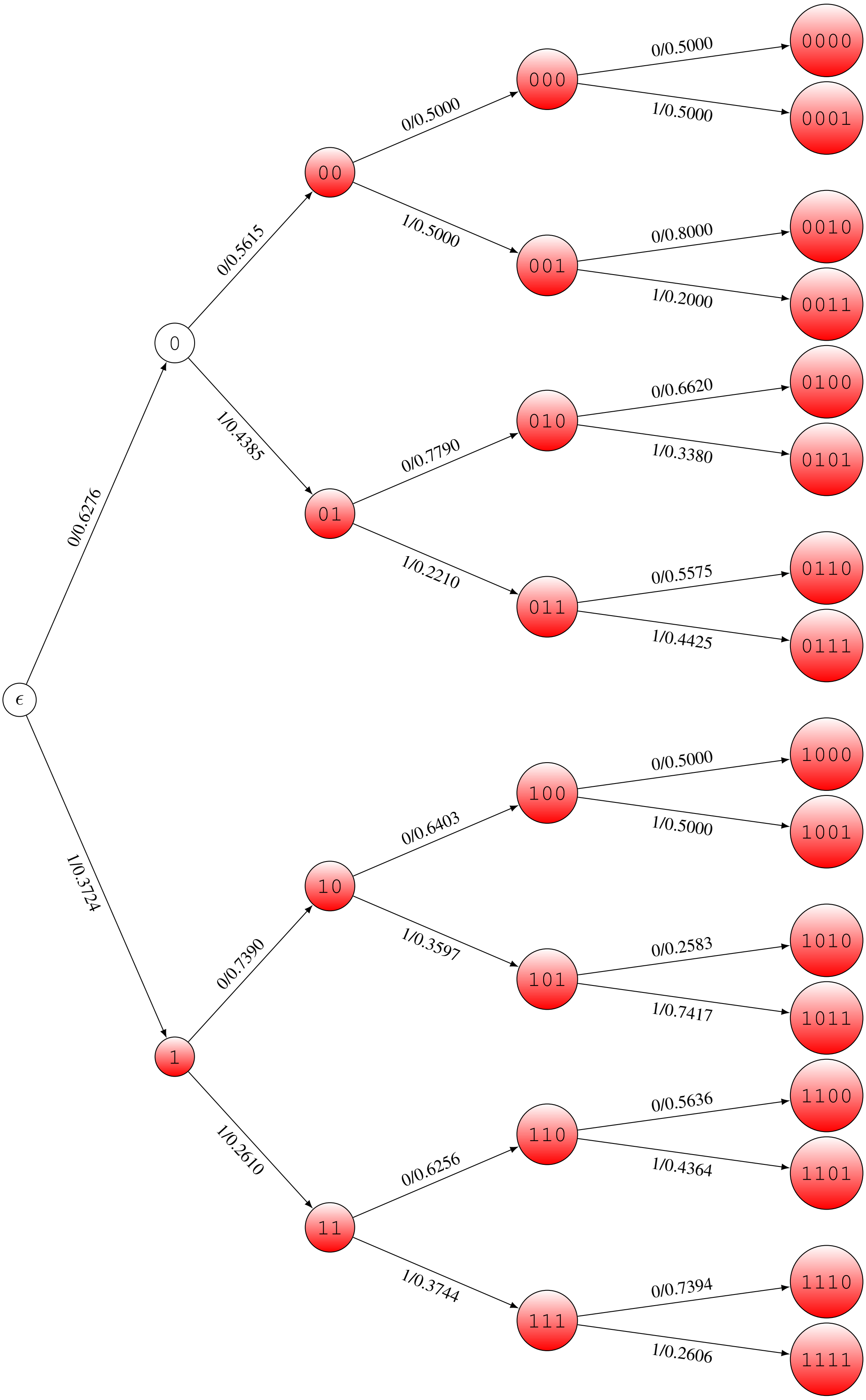

**Figure 3.5:** *Rooted Tree with Probabilities $\mathcal{T}$ for the Tri-Shift Example after the second iteration.*

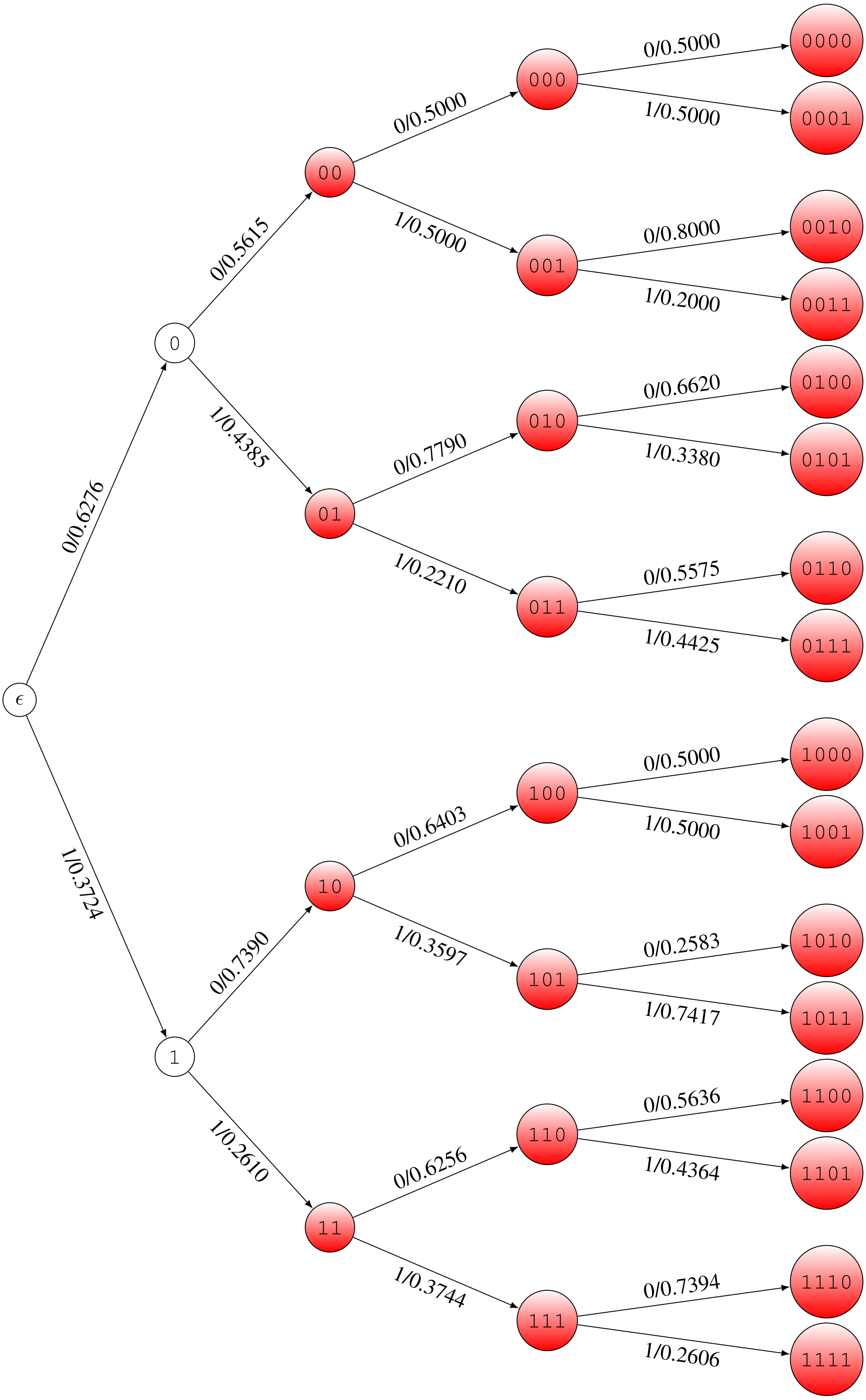

**Figure 3.6:** *Rooted Tree with Probabilities $\mathcal{T}$ for the Tri-Shift Example after the third and fourth iterations.*

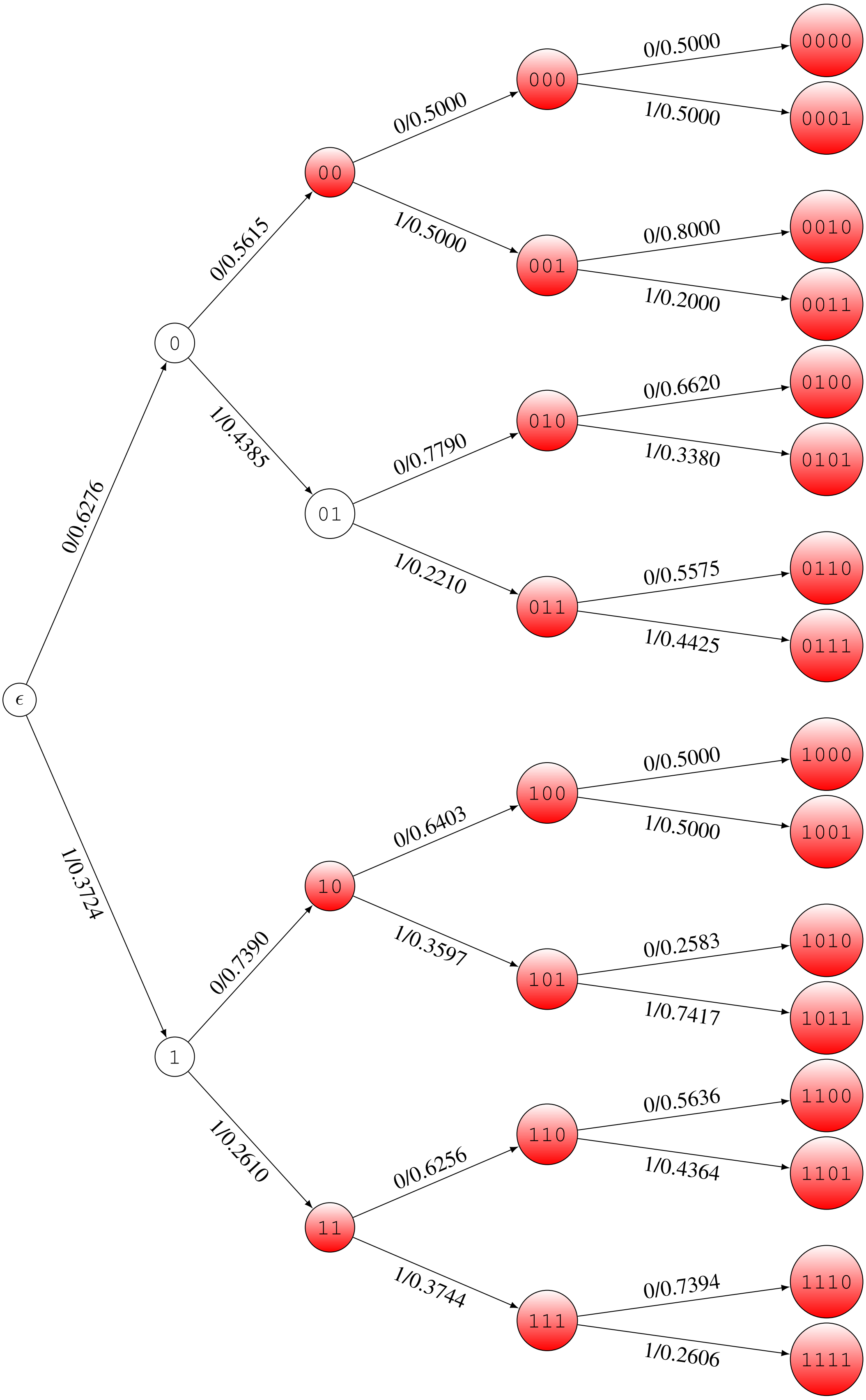

**Figure 3.7:** *Rooted Tree with Probabilities $\mathcal{T}$ for the Tri-Shift Example after the fifth iteration.*

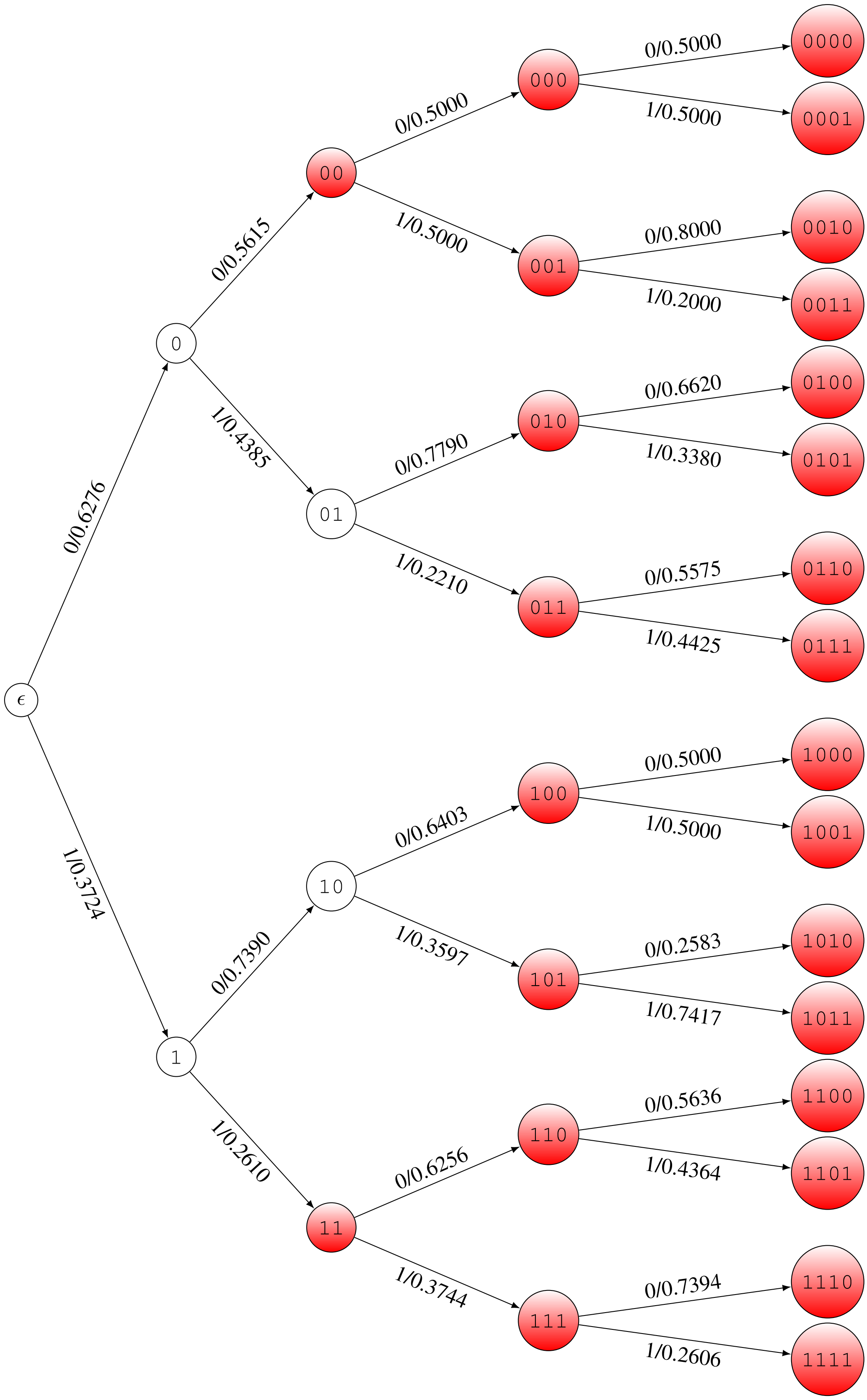

**Figure 3.8:** *Rooted Tree with Probabilities $\mathcal{T}$ for the Tri-Shift Example after the sixth iteration.*

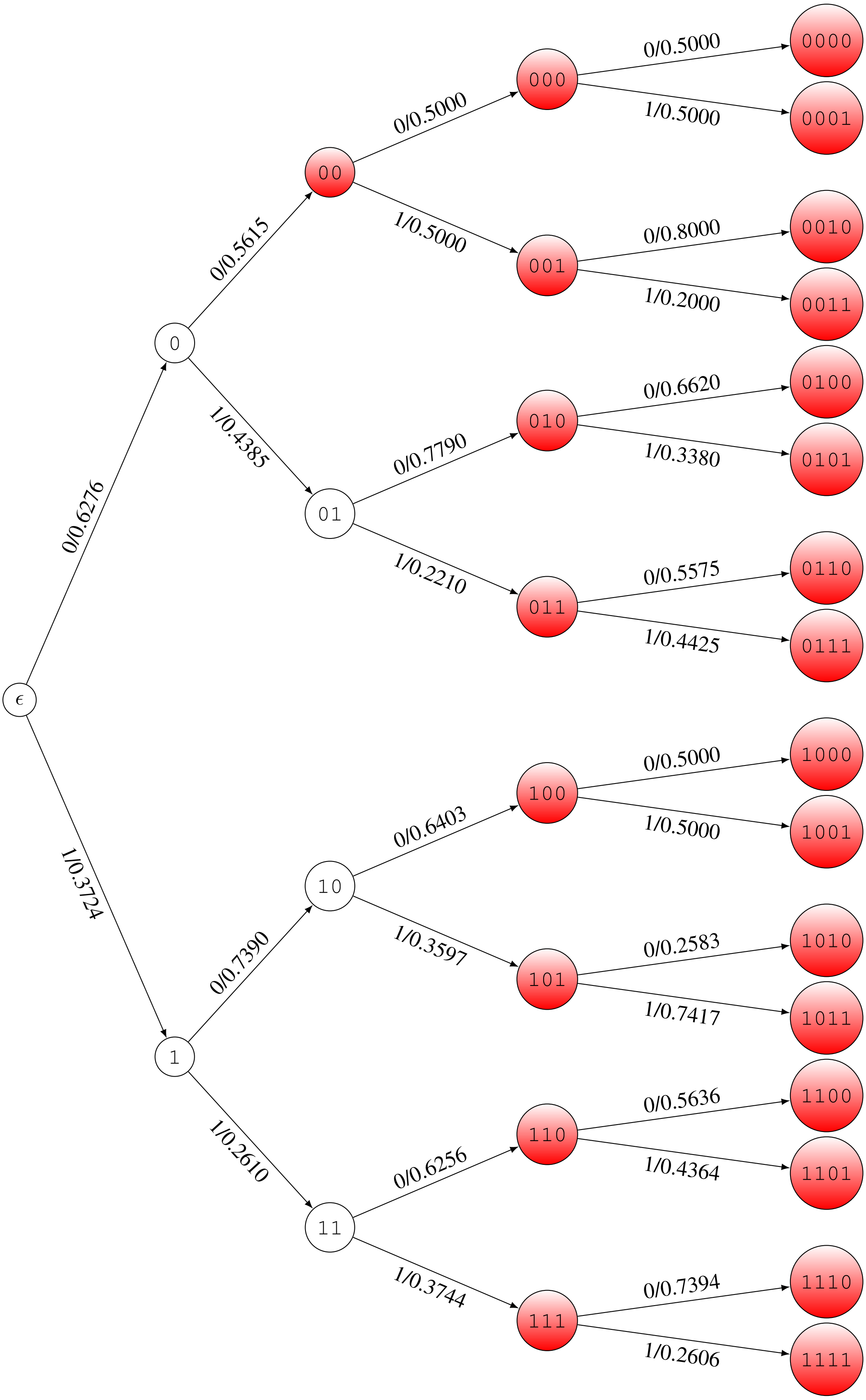

**Figure 3.9:** *Rooted Tree with Probabilities $\mathcal{T}$ for the Tri-Shift Example after the seventh iteration.*

## 3.2 PFSA Construction

In this section, we discuss the ALEPH algorithm that constructs a PFSA from the RTP $\mathcal{T}$. The first step is to transform $\mathcal{T}$ into a graph as no leaf states (i.e. states with no outgoing edges) can exist during the PFSA construction. This transformation is done via the criterion described in Section 3.2.1.

The final procedure groups states in equivalence classes of states that have statistically similar morphs (checked by $\chi^2$ or Kolmogorov-Smirnov for a given confidence level $\alpha$ using the same *statisticalTest* auxiliary function described in Section 3.1) and the partition given by these equivalence classes is used as an initial partition for a graph minimization algorithm (such as Moore or Hopcroft) to obtain the final reduced PFSA. This is the main contribution of the ALEPH algorithm. It manages to first get a reduced set of equivalence classes, with states in each class sharing statistically similar morphs, and then breaks them apart to obtain a final set of states that generate sequences similar to the original while having as little redundant states as possible for the given input parameters.

### 3.2.1 RTP Leaf Connection Criteria

The ALEPH algorithm creates equivalence classes for states with statistically similar morphs. In order to have every state in an equivalence class, all of them need to have a morph, which is not the case for leaf states. The $\Omega$ criterion is used to connect the leaf states of the RTP $\mathcal{T}$ and turn it into a PFSA. It depends on the system having synchronization words and is able to create connections that better represent the original system.

**$\Omega$ Connection**

For each state $p$ in level $L+1$ of $\mathcal{T}$, this criterion checks via statistical test if $p$ has similar morph to any of the synchronization words states. If it has not, it subsequently tests with the morphs of each extension of synchronization words up to length $L$. If any of these tests succeeds, the state $q$ in level $L$ that has $\delta(\tau, q) = p$ for $\tau \in \Sigma$ has this edge reassigned for the state with which the test was successful. In case no test passes, the D-Markov criteria is used for $q$. This is shown in Algorithm 8 whose inputs are the RTP $\mathcal{T}$, the desired last level $L$ and a list of synchronization words $\Omega_{syn}$.

### 3.2.2 ALEPH Algorithm

The full ALEPH algorithm is shown in Algorithm 9. It takes an RTP $\mathcal{T}$, the maximum considered depth $L$ and a list of synchronization words $\Omega_{syn}$ as inputs. It is further broken in 4 steps:

**Algorithm 8** omegaConnection($\mathcal{T}, L, \Omega_{syn}$)

```
procedure CONNECT
    $\Psi \leftarrow \{p \in \mathcal{T}$ if $p$ in level $L\}$
    for $q \in \Psi$ do
        next = NULL
        for $\tau \in \Sigma$ do
            $q' = \delta(\tau, q)$
            for $\omega \in \Omega_{syn}$ do
                $r \leftarrow$ statisticalTest($q', \omega, \alpha$)
                if $r = True$ then
                    next $\leftarrow \omega$
                    break
            if next = NULL then
                $\eta \leftarrow \{$All extensions of $\omega$ up to length $L, \forall\omega \in \Omega_{syn}\}$
                for $e \in \eta$ do
                    $r \leftarrow$ statisticalTest($q', e, \alpha$)
                    if $r = True$ then
                        next $\leftarrow e$
                        break
            if next = NULL then
                Given that $q = \sigma_0 \ldots \sigma_L$
            $\delta(\tau, m) \leftarrow \sigma_1 \ldots \sigma_L\tau$
```

i the RTP leaf connection;

ii a state reduction;

iii grouping the states in an initial partition of equivalence classes and

iv applying a graph minimization algorithm.

**Step 1: RTP Leaf Connection**

The $\Omega$ connection which is previously described is used in $\mathcal{T}$. The output of this step is a complete probabilistic graph instead of a tree with leaf nodes.

**Step 2: State Reduction**

After the RTP connection process is finished a technique is used to discard some branches of $\mathcal{T}$ in order to obtain fewer starting states in the next two steps, which implies in a smaller complexity. Each state $q' \in \mathcal{T}$ is visited, starting by the synchronization word states and it is checked if any of its descendents (i.e. $\delta(q', \sigma)$ for some $\sigma \in \Sigma$) has a synchronization word $\omega$ as suffix. In the affirmative case, the outgoing edge of $q'$ is reassigned to $\omega$. This is done because a state that has a synchronization word as suffix has a morph similar to that of the synchronization word and generates the same sequences with the same probabilities, which means these states that have synchronization words as suffixes can be discarded. By discarding them, the rest of its branch is also discarded, reducing the number of states even further.

This is done by creating a copy of $\Omega_{syn}$ called $Q_0$ and an empty list called $P_0$. The first element of $Q_0$ is taken by a dequeue and stored in $q_0$, which means starting to visit the state of $\mathcal{T}$ by the synchronization words. All $q_0$ descendants are checked to see if they have one of the synchronization words as suffix, and in the affirmative case, $\delta(q_0, \sigma)$ is reassigned to that synchronization word (lines 9 to 12 of Algorithm 9). If a descendant of $q_0$ does not end in a synchronization word, the descendant is added to the list $Q_0$ if it is not already in it and neither in $P_0$ (lines 13 to 15 of Algorithm 9). This is done to make sure that states are not visited twice. After all descendants of $q_0$ are checked, $q_0$ is appended to $P_0$.

Finally, the initial equivalence classes are created: one for each synchronization word (line 17 of Algorithm 9).

### Step 3: Grouping the States in Equivalence Classes

The initial equivalence classes created in the previous step are stored in the list $\mathcal{P}$. Given an equivalence class $C$, its head state is the first state that is added to $C$ and it is denoted by $C[0]$. A list $\tilde{\text{Q}}$ is created containing the descendants of the head states in each of the initial equivalence classes.

For each element $q$ of $\tilde{\text{Q}}$, statistical tests are performed with the head state of each equivalence class $C$ already present in $\mathcal{P}$. If a test result is positive, the state $q$ is added to the partition $C$. If no test is successful, a new equivalence class is created for $q$ and this class is subsequently added to $\mathcal{P}$. This process is repeated until every state of $\mathcal{T}$ is present in one equivalence class of the partition $\mathcal{P}$.

### Step 4: Graph Minimization

Once every state of $\mathcal{T}$ is in one class of the initial partition $\mathcal{P}$, a graph minimization algorithm (either Moore or Hopcroft) is applied using $\mathcal{P}$ as the initial partition. This initial partition guarantees that the states in the same equivalence class have the same morph and the reduction algorithm breaks this class if paths starting at these states eventually reach states with different morphs. The *graph-Minimization* function then proceeds to refine the initial partition until the partition $G_o$ is obtained, in which each equivalence only contains states that generate the sequences in their right context with similar probabilities.

To obtain a PFSA, the transition probability function $\pi_o$ must also be obtained, which is done by the averageMorphs function. This function computes the average morph of each equivalence class of $G_o$. The equivalence classes of $G_o$ are then turned into states with the morph computed by *averageMorphs*. The final result is the PFSA $(G_o, \pi_o)$ that represents the original system for the given parameters. The accuracy and the number of states depend on the parameters $L$ and the confidence level $\alpha$.

## 3.3 Time Complexity

The main improvements of the ALEPH algorithm are that, unlike CRISSiS, it does not depend on the original system being synchronizable (the original system does not even need to be represented by a PFSA and ALEPH will generate a PFSA that approximates it) . As seen in [15], CRISSiS complexity depends on the number of states of the original system which (seen in Section 2.6.2), in practical applications, remains unknown until the end of the algorithm. As it is discussed in this section, the complexity of ALEPH depends only on parameters known prior to the algorithm execution.

**Algorithm 9** ALEPH($\mathcal{T}, \Omega_{syn}, L$)

```
procedure
    ## Step 1: RTP Leaf State Connection
    $\mathcal{T} \leftarrow$ omegaConnection($\mathcal{T}, L$)
    ## Step 2: State Reduction
    $Q_0 \leftarrow \Omega_{syn}$
    $P_0 \leftarrow \emptyset$
    while $Q_0 \neq \emptyset$ do
        $q_0 \leftarrow$ dequeue($Q_0$)
        for $\sigma \in \Sigma$ do
            $q_0' \leftarrow \delta(q_0, \sigma)$
            if for some $\omega \in \Omega_{syn}, \omega$ is a suffix of $q_0'$ then
                $\delta(\sigma, q_0) \leftarrow \omega$
            else
                if $q_0' \notin Q_0$ and $q_0' \notin P_0$ then
                    $Q_0 \leftarrow Q_0 \cup \{q_0'\}$
            $P_0 \leftarrow P_0 \cup \{q_0\}$
    $\mathcal{P} \leftarrow \{\{\omega\}, \forall \omega \in \Omega_{syn}\}$
    ## Step 3: Grouping in Equivalence Classes
    $Q \leftarrow \{\delta(\sigma, C[0]), \forall \sigma \in \Sigma$ and $\forall C \in \mathcal{P}\}$
    for $q \in Q$ do
        $r \leftarrow$ False
        for $C \in \mathcal{P}$ do
            $r \leftarrow$ statisticalTest($q, C[0], \alpha$)
            if $r =$ True then
                $C \leftarrow C \cup \{q\}$
                break
        if $r =$ False then
            $R \leftarrow \{q\}$
            $\mathcal{P} \leftarrow \mathcal{P} \cup \{R\}$
        $Q \leftarrow Q \cup \{\delta(\sigma, q), \forall \sigma \in \Sigma | \delta(\sigma, q)$ not in any $p \in \mathcal{P}\}$
    ## Step 4: Graph Minimization (either Moore or Hopcroft)
    $G_o \leftarrow$ graphMinimization($\mathcal{T}, \mathcal{P}$)
    $\pi_o \leftarrow$ averageMorphs($G_o$)
    return $(G_o, \pi_o)$
```

The complexity of each part of the algorithm is discussed individually and a final complexity is given in the end.

### 3.3.1 RTP Construction

To construct the RTP, the original sequence $S$ of length $N$ has to be parsed $L$ times, which depends mainly on the sequence length, giving a final complexity $O(N)$.

### 3.3.2 Synchronization Word Search

For a given state with length $n < W$, the maximum amount of statistical tests it goes through is $n$ for each of its suffixes, starting by $\epsilon$. For each of these tests, one search for SVS is performed. This search for SVS has complexity $O(m)$ for a SVS of length $m$. Thus, for the given state of length $n$, searches for SVS of length 0 to $n-1$ are performed, resulting in a complexity of $O(n^2)$ for the searches. As in CRISSiS, a complexity of $O(1)$ is used for the statistical test. This implies that for a given state of length $n$, $O(n^3)$ operations are performed. This can be simplified if after a search for SVS the result is stored. When a new search for SVS has to be performed, it can start from where the last one finished. This means that for a state of length $n$, the searches have complexity $O(n)$ and the final complexity is $O(n^2)$.

Looking at $\mathcal{T}$, for a given level $d$, $|\Sigma|^d$ tests and searches for SVS of length $d$ are performed, giving a complexity of $O(|\Sigma|^d d^2)$ per level. The total complexity is the sum of all levels from 1 to $W$. Therefore, it is $O(\sum\limits_{d=1}^{W} |\Sigma|^d d^2)$ and as usually $W > |\Sigma|$, the final complexity for the synchronization word search is $O(\frac{|\Sigma|^{W+1} W^2}{|\Sigma|-1})$.

The complexity for the same operation in CRISSiS is $|\Sigma|^{(|Q|^3+L_1+L_2)}$. As it is exponential on the cube of the number of states of the original machine, it grows much faster than our solution.

### 3.3.3 RTP Leaf Connection

In the D-Markov connection, each of the $|\Sigma|^L$ elements in the last level has their $|\Sigma|$ edges reassigned, giving a complexity of $O(|\Sigma|^{L+1})$.

The $\Omega$ connection is a little more complex to analyze. It also needs to perform operations for each of the $|\Sigma|$ outgoing edges of each of the $|\Sigma|^L$ states in level $L$, but those operations are not simply reconnections of complexity $O(1)$. It performs tests with all states in $\Omega_{syn}$ and its descendants up to length $L$, which in a worst case scenario means testing against states from level 0 to $L$ in a total of $O(|\Sigma||\Omega_{syn}|L^2)$ tests per state in the last level and a final complexity of $O(|\Sigma|^{L+1}|\Omega_{syn}|L^2)$.

### 3.3.4 PFSA Construction

When there are synchronization words, the algorithm starts by checking if all the $|\Sigma|^L$ states have an outgoing edge that could be substituted by a synchronization word, giving a final complexity of $O(|\Sigma|^{L+1})$ for this additional step.

The worst case scenario occurs when all the $|\Sigma|^{L+1}$ states of $\mathcal{T}$ have their own equivalence classes. In this case, the $d^{th}$ has to be tested against the $d-1$ previous equivalence classes, giving a complexity of $O(d)$. The complexity for all states is $O(1+2+\ldots+|\Sigma|^{L+1}) = O(|\Sigma|^{2L+2})$. As seen in [21], this is the same procedure that dominates the complexity of graph reduction algorithms, therefore their complexity by the end of the ALEPH algorithm does not need to be considered.

When there are synchronization words and the first step of complexity $O(|\Sigma|^{L+1})$ is applied, the number of states that will be organized in equivalence classes might be dramatically reduced, which also reduces the complexity of that step. But in the worst case scenario, the $O(|\Sigma|^{2L+2})$ factor dominates the PFSA construction step and is its final complexity.

# CHAPTER 4

# RESULTS

In this chapter, the efficiency of the algorithms proposed in Chapter 3 to construct a PFSA is verified for some examples of dynamic systems that can be represented by a PFSA. Results for more practical and complex systems are discussed in the next chapter. First, from the original system, a discrete sequence *S* over the alphabet $\Sigma$ of length $N = 10^7$ is generated. Then, we calculate the probabilities of subsequences occurring in *S* up to a length $L_{max}$ and construct an RTP from these probabilities. After this, a series of PFSA are created using the D-Markov Machine, CRISSiS and the ALEPH algorihm with different values of their parameters, that is, $D$ (for D-Markov Machines), $L$ (for the ALEPH algorithms) and $L_2$ (for CRISSiS). Finally, the accuracy of each of those PFSA are compared using the quantifiers explained in Section 4.1 and the comparison results are explained in Section 4.2.

## 4.1 Performance Quantifiers

This section presents the two quantifiers that are used to compare the performance of the PFSA generated by the algorithms. The first one is the conditional entropy that is used to approximate the entropy rate, which gives a sense of the memory of the system. The other one, the Kullback-Leibler Divergence compares sequences generated by the models with the original one and estimate how similar they are. The smaller the divergence, the more similar to the original system are the models.

### 4.1.1 Entropy Rate

Let $\{X_k\}_{k=1}^{\infty}$ be a discrete random process over $\Sigma$. Its entropy rate is defined as [29]:

$$h \triangleq \lim_{k\to\infty} H(X_k|X_1X_2\dots X_{k-1}) = -\lim_{k\to\infty} \sum_{x\in\Sigma^k} \Pr(x)\log\Pr(x_k|x_1x_2\dots x_{k-1}). \tag{4.1}$$

For a stationary process, the conditional entropy $H(X_k|X_1\dots X_{k-1})$ is non-increasing in $k$ and converges to $h$ as $k$ approaches infinity [29]. As it is not feasible to compute (4.1) the $\ell$-order conditional entropy is used, defined as:

$$h_\ell \triangleq H(X_\ell|X_1X_2\dots X_{\ell-1}), \tag{4.2}$$

which measures the uncertainty of a random variable $X_\ell$ given the previous $\ell - 1$ samples. The comparison of $h_\ell$ of the generated PFSA with the one from the original system is useful to test if the generated one correctly captures the system memory.

### 4.1.2 Kullback-Leibler Divergence

For the purpose of comparing the algorithms, consider two sequences $S_1$ and $S_2$ over a common alphabet $\Sigma$. They can be either the original sequence $S$ or a sequence generated by a PFSA. Let $\omega \in \Sigma^\ell$ be a word of length $\ell$ and $P_1(\omega)$ and $P_2(\omega)$ be the probabilities of occurrence of $\omega$ in $S_1$ and $S_2$ respectively. For a given $\ell$ we take the $\ell$-order Kullback-Leibler Divergence as:

$$D_\ell(S_1||S_2) = \sum_{\omega\in\Sigma^\ell} P_1(\omega)\log\Big(\frac{P_1(\omega)}{P_2(\omega)}\Big). \tag{4.3}$$

Although it is technically not a distance, as it does not obey the triangle inequality nor is necessarily commutative, the Kullback-Leibler Divergence is useful to give an idea of how similar two distributions are. A small divergence indicates that the sequence generated by a PFSA is statistically close to the original sequence, which shows that the PFSA is a good estimate for the original system.

## 4.2 Construction of PFSA for Dynamic Systems

We consider the following examples of dynamic systems with known representations as PFSA. The goal is to apply the D-Markov Machine, CRISSiS and ALEPH algorithm to recover a good PFSA and compare their number of states.

In all examples, the ALEPH algorithm is able to recover the original PFSA for some value of $L$ as well as CRISSiS for some value of $L_2$. Usually, D-Markov Machines are not capable of retrieving the original PFSA, but by increasing $D$, better machines are obtained in expense of an exponential

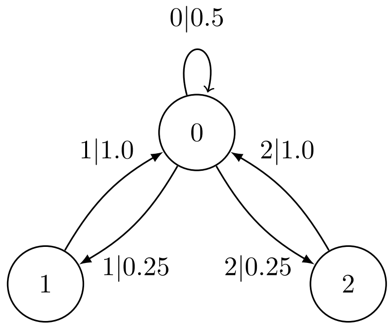

**Figure 4.1:** *A PFSA of a Ternary Even-Shift.*

**Table 4.1:** *Synchronization Words for Ternary Even Shift.*

| | $\alpha$ | |
|---|---|---|
| $W$ | 0.95 | 0.99 |
| 2 | 0 | 0 |
| 3 | 0, 12, 21 | 0, 12, 21 |
| 4 | 0, 12, 21 | 0, 12, 21 |
| 5 | 0, 12, 21 | 0, 12, 21 |
| 6 | 0, 12, 21 | 0, 12, 21 |

growth in the number of states. The results for two D-Markov Machines are shown for each example: one for the $D$ that achieves a performance similar to the original PFSA and another for $D-1$ to show a more compact with lower performance.

In the following cases we consider $\ell = 10$. All the PFSA were constructed using the $\chi^2$ test with $\alpha = 0.95$ and for the synchronization words, two values of $\alpha$ (0.95 and 0.99) were used.

### 4.2.1 Ternary Even Shift

The ternary even shift is a symbolic dynamic system with a ternary alphabet $\Sigma = \{0, 1, 2\}$ where there must be an even number of consecutive non-zero symbols between zeroes. A PFSA that satisfies this restriction is shown in Figure 4.1.

The synchronization words found by our algorithm with $2 \leq W \leq 6$ are shown in Table 4.1 for two values of $\alpha$ (0.95 and 0.99) and the same words ($\Omega_{syn} = \{0, 12, 21\}$) are found for any $W \geq 3$. It is possible to check in the graph of Figure 4.1 that all found synchronization words are indeed valid and each one synchronizes to one state of the graph. They can all be used as starting points for the ALEPH algorithm.

The results of the ALEPH algorithm are compared to D-Markov and CRISSiS in Table 4.2. The ALEPH algorithm obtained the same results for any L greater than 2. D-Markov machines of

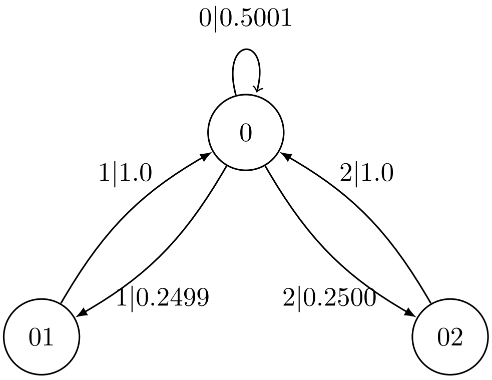

**Figure 4.2:** *PFSA of a Ternary Even-Shift generated by the ALEPH algorithm and by CRISSiS.*

**Table 4.2:** *Results for Ternary Even Shift.*

| | D-Markov | | ALEPH/CRISSiS |
|---|---|---|---|
| | $D = 8$ | $D = 9$ | $L = 2/L_2 = 1$ |
| # of States | 169 | 339 | 3 |
| $h_{10}$ | 1.0084 | 1.0058 | 1.0003 |
| $D_{10}$ | $2.7 \cdot 10^{-3}$ | $4.16 \cdot 10^{-5}$ | $9.55 \cdot 10^{-5}$ |

$D = 8$ and $D = 9$ are considered. CRISSiS was tested using $L_2 = 1$. Both CRISSiS and ALEPH reconstruct the same PFSA (shown in Figure 4.2), therefore their results are statistically the same and they are a good estimate to the original 3-state PFSA while a large D-Markov machine with $D = 9$ with 339 states is needed to obtain approximately the same performance. These D-Markov machines with $D = 8$ and 9 do not have $3^8$ and $3^9$ states respectively because there are forbidden words in the original system, which results in some states being non-existent in the RTP. The original system had $h_{10} = 1.0003$, which is close to the value found by all the algorithms.

### 4.2.2 Tri-Shift

The Tri-Shift was previously discussed in Section 2.6.2 and a PFSA that represents it is repeated here in Figure 4.3. The synchronization words found by the algorithm are shown in Table 4.3 and 00 appeared, as expected (see Section 2.6.2), and 0110 synchronizes to the same state as 00, thus $\Omega_{syn} = \{00, 0110\}$. The comparative results are shown in Table 4.4. Once again this is an example where our algorithm and CRISSiS are able to recover the three states from the original PFSA with a good estimate for the morphs as seen in Figure 4.5, which means their results are again statistically the same. To obtain a similar performance with a D-Markov machine, 256 states might be needed. The original system presented has $h_{10} = 0.4873$, showing that our algorithm, CRISSiS and the

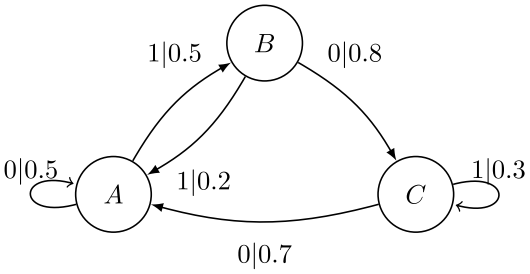

**Figure 4.3:** *The Tri-Shift PFSA.*

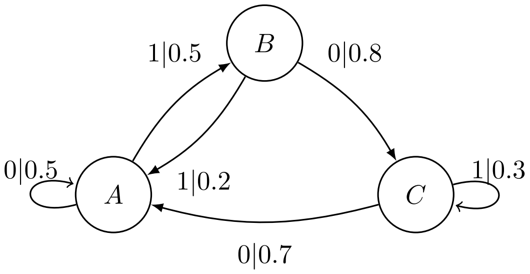

**Figure 4.4:** *The Tri-Shift PFSA.*

8-Markov Machine are able to capture the system memory.

### 4.2.3 A Six-State PFSA

Figure 4.6 shows a PFSA with six states that shows how CRISSiS might need $L_2 \geq 1$ to retrieve the original machine. This system has 4 synchronization words: 00, 01, 10 and 1111, as shown in Table 4.5. The comparative results between the algorithms is shown in Table 4.6.

Using CRISSiS with $L_2$ larger than 3 and ALEPH with $L$ larger than 4, it is possible to reconstruct a good estimate to the original system, shown in Figure 4.7, providing statistically similar results. For a D-Markov Machine to perform similarly, it is necessary to use $D = 4$, obtaining a PFSA with 11 states. Once again, some sequences do not occur, therefore the D-Markov Machine in those cases

**Table 4.3:** *Synchronization Words for Tri-Shift.*

| | $\alpha$ | |
|---|---|---|
| $W$ | 0.95 | 0.99 |
| 2 | None | None |
| 3 | 00 | 00 |
| 4 | 00 | 00 |
| 5 | 00, 0110 | 00, 0110 |
| 6 | 00, 0110 | 00, 0110 |

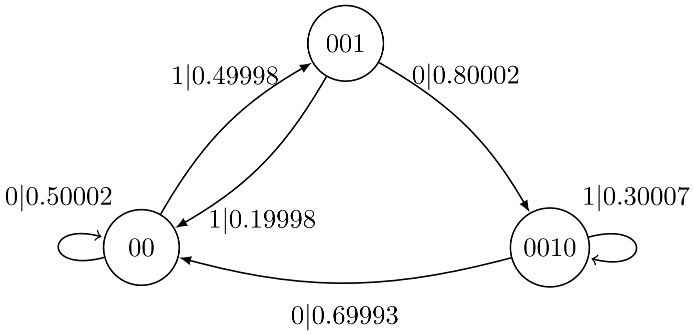

**Figure 4.5:** *The Tri-Shift PFSA generated by our algorithm and by CRISSiS.*

**Table 4.4:** *Results for the Tri-Shift.*

| | D-Markov | | ALEPH/CRISSiS |
|---|---|---|---|
| | $D = 7$ | $D = 8$ | $L = 4/L_2 = 1$ |
| # of States | 128 | 256 | 3 |
| $h_{10}$ | 0.4870 | 0.4867 | 0.4872 |
| $D_{10}$ | $4.1 \cdot 10^{-3}$ | $1.65 \cdot 10^{-3}$ | $1.16 \cdot 10^{-3}$ |

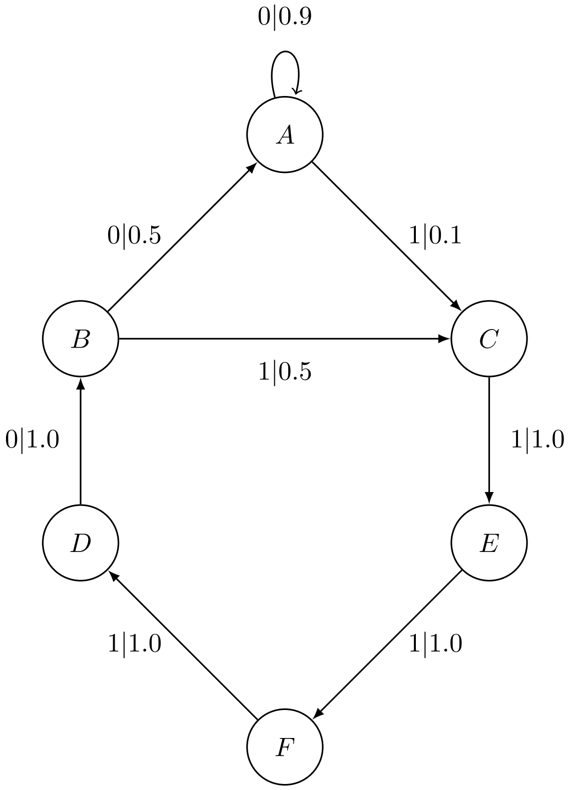

**Figure 4.6:** *A Six-State PFSA.*

**Table 4.5:** *Synchronization Words for the Six-State PFSA.*

| | $\alpha$ | |
|---|---|---|
| $W$ | 0.95 | 0.99 |
| 2 | None | None |
| 3 | 00, 01, 10 | 00, 01, 10 |
| 4 | 00, 01, 10 | 00, 01, 10 |
| 5 | 00, 01, 10, 1111 | 00, 01, 10, 1111 |
| 6 | 00, 01, 10, 1111 | 00, 01, 10, 1111 |

**Table 4.6:** *Results for the Six-State PFSA.*

| | D-Markov | | ALEPH/CRISSiS |
|---|---|---|---|
| | $D = 3$ | $D = 4$ | $L = 4/L_2 = 3$ |
| # of States | 7 | 11 | 6 |
| $h_{10}$ | 0.5341 | 0.3344 | 0.3344 |
| $D_{10}$ | 1.1980 | $4.0499 \cdot 10^{-6}$ | $5.6969 \cdot 10^{-5}$ |

will not have $2^D$ states.

As ALEPH uses all synchronization words, there are multiple starting points and the graph minimization algorithm step by the end is useful to differentiate states that will have different follower sets. The original system has a $h_{10} = 0.3344$, showing that both the 4-Markov Machine, the ALEPH algorithm and CRISSiS are able to estimate the PFSA with good precision.

### 4.2.4 Maximum Entropy $(d, k)$-Constrained Code

As seen in [30], a $(d, k)$-constrained code is a code used in digital recording devices and other systems in which a long sequences of 1's might cause desynchronization issues. This code guarantees that at most $k$ and at least $d$ 1's are generated between occurrences of 0's. A Maximum Entropy $(d, k)$-Constrained Code is a PFSA that generates sequences with those restrictions and that also have maximum entropy rate. The algorithms are tested to recover a Maximum Entropy (3,5)-Constrained Code PFSA shown in Figure 4.8. The synchronization words for this system are 0 and 11111, as shown in Table 4.7.

The results for this system are shown in Table 4.8. This is a practical case where CRISSiS needs $L_2 \geq 3$ to obtain a correct estimate. When $L_2$ is 3, CRISSiS recovers the same PFSA as the ALEPH algorithm with $L = 6$ (shown in Figure 4.9), presenting statistically similar results. The original $h_{10}$ is 0.3218. For a D-Markov Machine to have a similarly good performance, a $D$ of 5 is needed, generating machines with 7 states, which is larger than the original PFSA.

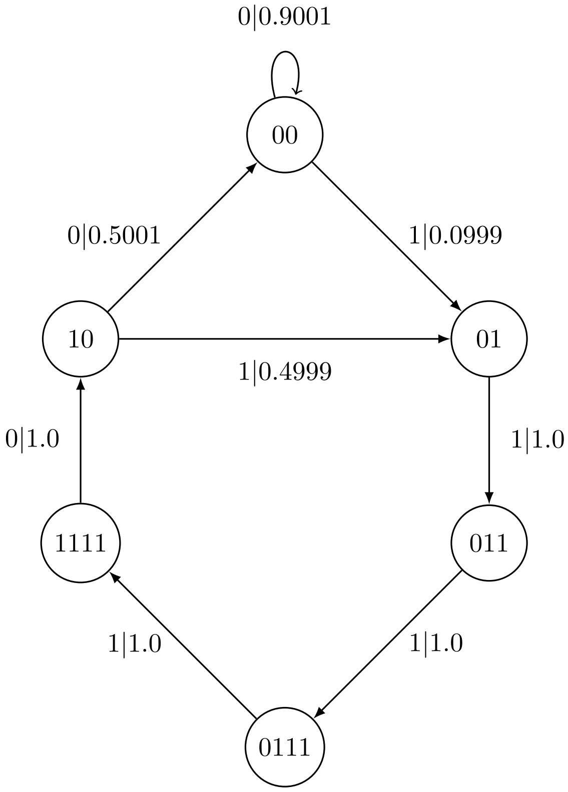

**Figure 4.7:** *The Recovered Six-State PFSA by our algorithm.*

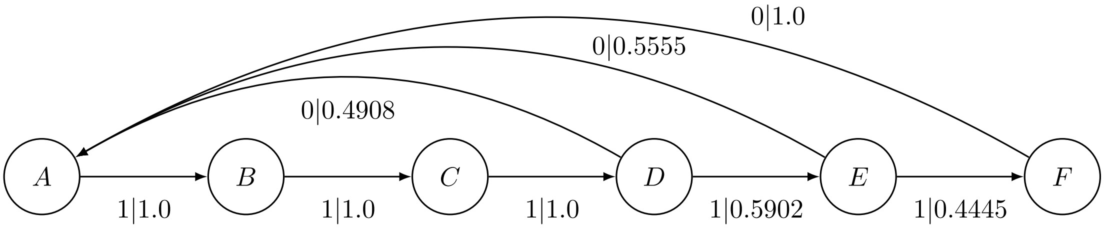

**Figure 4.8:** *The Maximum Entropy (3,5)-Constrained Code PFSA.*

**Table 4.7:** *Synchronization Words for the Maximum Entropy (3,5)-Constrained Code.*

| | $\alpha$ | |
|---|---|---|
| $W$ | 0.95 | 0.99 |
| 2 | 0 | 0 |
| 3 | 0 | 0 |
| 4 | 0 | 0 |
| 5 | 0 | 0 |
| 6 | 00, 11111 | 00, 11111 |
| 7 | 00, 11111 | 00, 11111 |

**Table 4.8:** *Results for the Maximum Entropy (3,5)-Constrained Code PFSA.*

| | D-Markov | | ALEPH/CRISSiS |
|---|---|---|---|
| | $D = 4$ | $D = 5$ | $L = 6/L_2 = 3$ |
| # of States | 5 | 7 | 6 |
| $h_{10}$ | 0.3575 | 0.3218 | 0.3218 |
| $D_{10}$ | 0.1793 | $7.0139 \cdot 10^{-7}$ | $5.9715 \cdot 10^{-7}$ |

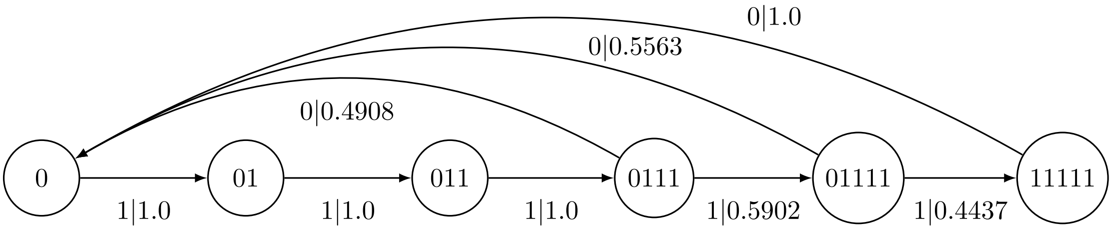

**Figure 4.9:** *The Maximum Entropy (3,5)-Constrained Code PFSA recovered by ALEPH algorithm and by CRISSiS.*

# Chapter 5

# An Algorithm for Non-Synchronizable Dynamical Systems

In this chapter we approach discrete dynamical systems that are not synchronizable. As there are no synchronization words, the methods using the ALEPH algorithm are not applicable for this category of systems. In order to model them we present another algorithm, the D-Markov with Clustering and Graph Minimization (DCGraM) algorithm. In Section 5.1 we describe the DCGraM algorithm step by step. In Section 5.2, four examples of applications are shown in order to compare the results obtained by DCGraM against those from a D-Markov model. The first example is of a synchronizable machine, the ternary even shift presented in Section 4.2.1, so it is possible to see that although DCGraM is not intended for this kind of application it still can produce considerably good results, although worse than those from ALEPH. We also consider two examples of non-synchronizable dynamical systems: the logistic map, which is a discrete mapping that can present chaotic behavior, and the binary communication channel with fading, that is useful for modeling wireless communications and the Lorenz attractor, a chaotic attractor obtained from the Lorenz equations, a system of non-linear differential equations.

## 5.1 DCGraM Algorithm

D-Markov models usually achieve good performance modeling non-synchronizable dynamical systems as $D$ grows, which means that the cost for this improved performance is an exponential growth in the number of states. The DCGraM algorithm comes as an improvement of D-Markov models using clustering and graph minimization techniques presented before to reduce the size of the final machine.

In order to use the clustering algorithm morphs must be represented as points in a $|\Sigma|$-dimensional space. For a given state $q$, its morph is represented as a point $(\Pr(\sigma_1|q), \Pr(\sigma_2|q), \ldots, \Pr(\sigma_{|\Sigma|}|q)$ for $\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|} \in \Sigma$ in the $|\Sigma|$-dimensional space. As the morphs are probability distributions, $\sum\limits_{\sigma \in \Sigma} \Pr(\sigma|q) = 1, \forall q \in Q$, the morph-points are restrained to a $(|\Sigma| - 1)$-dimensional subspace. Figure 5.1 shows an example of the morphs of a D-Markov machine with $D = 4$ represented in a two dimensional space.

In order to efficiently create an initial partition $\mathcal{P}$ of the states of a D-Markov machine $(G_D, \pi_D)$, a clustering algorithm is used to group states with similar morphs together. For example, Figure 5.2 shows how the morphs from Figure 5.1 might be clustered together in four groups by their proximity. In order to perform this clustering into $K$ groups, a variation of the K-Means algorithm is used with the parameter $K$, which is determined empirically, and a different method to determine the proximity of the state morphs.

The central idea of DCGraM is to use a graph minimization technique with the initial partition $\mathcal{P}$ determined by the clustering algorithm, producing a reduced graph with similar statistical behavior to the D-Markov model of the original system. As the states are clustered according only to their morphs, they have similar probabilities of generating a symbol, but the probability of generating longer sequences might differ. This is the criterion used to split these clusters with the graph minimization algorithm. After the graph minimization algorithm finishes refining the initial partition, the states in the same cluster generate the sequences in their right context with similar probabilities.

For example, consider that two states $A$ and $B$ from a binary graph $G$ are clustered together because they have equal morphs $\{\pi(0, q) = 0.5, \pi(1, q) = 0.5\}$ for $q = A, B$. But $A$ has a 0-transition $\delta(0, A) = C$ and the $C$ is in a cluster for states with morph $\{\pi(0, C) = 0.75, \pi(1, C) = 0.25\}$, while $B$ has a 0-transition $\delta(0, B) = D$ and the $D$ is in a cluster for states with morph $\{\pi(0, D) = 0.2, \pi(1, D) = 0.8\}$. This means that the probability of generating the sequence $00$ from $A$ is $0.375$, while the probability of generating $00$ from $B$ is $0.1$, which, in turn, means that $A$ and $B$ should not be in the same cluster and their cluster should be split in the next iteration of the graph minimization algorithm.

The full algorithm is shown in Algorithm 10 and each of its steps is discussed in the following.

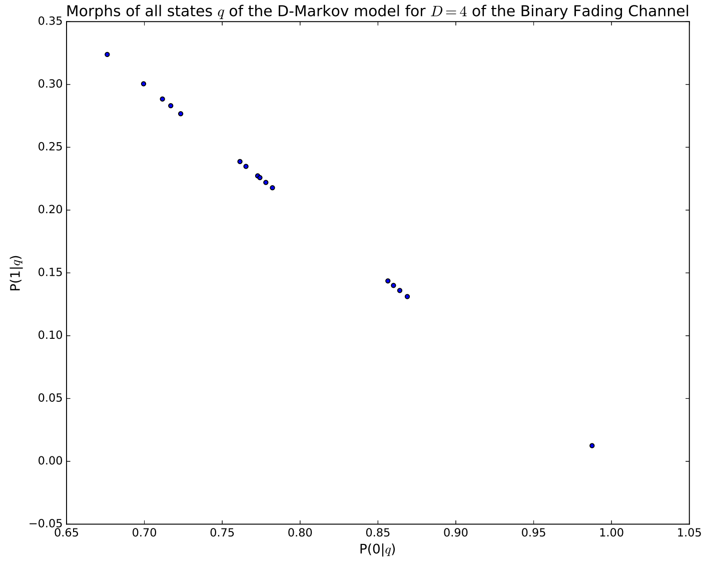

**Figure 5.1:** *Each point represents the morph of a state q from the D-Markov machine with $D = 4$ of the binary fading channel presented in Section 5.2.3. The x-axis shows the probability of q transitioning with 0, while the y-axis represents its probability of transitioning with 1.*

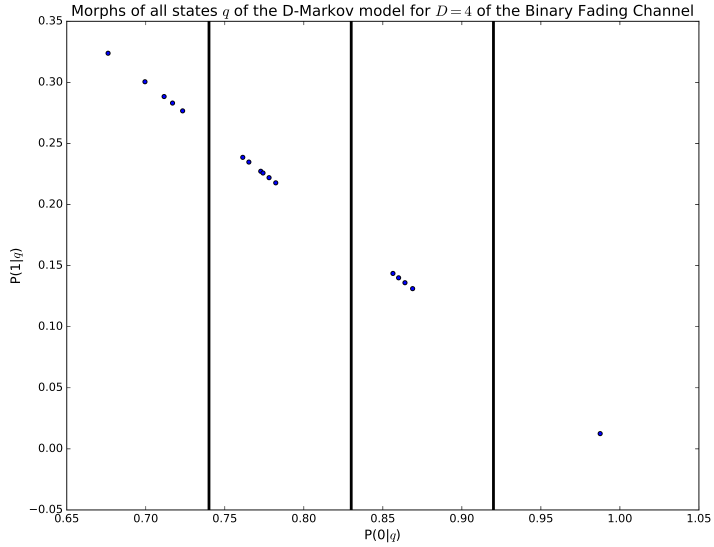

**Figure 5.2:** *The morphs of Figure 5.1 clustered into four groups.*

#### Step 1: Create D-Markov Machine

DCGraM starts by creating a D-Markov model $(G_D, \pi_D)$ for an input $D$ and the original sequence $S$ value which should have its number of states reduced in the following steps.

#### Step 2: Cluster states

In this step, a variation of the K-Means algorithm presented in Section 2.5.1 is applied to $\mathcal{V}(Q)$ in order to generate an initial partition to be used by the graph minimization algorithm. We use the notation $\mathcal{V}(q)$ to represent the morph of a state $q \in Q$ where $Q$ is the set of states of a PFSA. Now, the notation $\mathcal{V}(Q)$ is used to represent the set of the morphs of all states in $Q$.

The main difference between the regular K-Means algorithm presented in Section 2.5.1 and the one used in DCGraM is that instead of using the Euclidean metric (2.7) to measure the distance between data points, the Kullback-Leibler divergence is used to measure the information divergence between the morphs. Given morphs $\mathcal{V}(q)$ and $\mathcal{V}(q')$, the Kullback-Leibler divergence $D(\mathcal{V}(q) \| \mathcal{V}(q'))$ is given by:

$$D(\mathcal{V}(q) \| \mathcal{V}(q')) = \sum_{\sigma \in \Sigma} \Pr(\sigma | q) \log \frac{\Pr(\sigma | q)}{\Pr(\sigma | q')}. \tag{5.1}$$

Although the Kullback-Leibler divergence is not technically a metric because $D(\mathcal{V}(q) \| \mathcal{V}(q'))$ is not necessarily equal to $D(\mathcal{V}(q') \| \mathcal{V}(q))$ [28], keeping $q'$ the same as each cluster centroid whenever the divergence is computed is a good method to measure how the states diverge from the clusters.

The states of $G_D$ are grouped together using the auxiliary function *kmeans*$(K, Q, \mathcal{V}(Q))$. This applies the modified K-Means algorithm on $\mathcal{V}(Q)$ to create $K$ clusters and then partition the set of states $Q$ from $G_D$ following the labels of these $K$ clusters, i.e. states that receive the same label by K-Means are grouped together in the same equivalence class. This results in the initial partition $\mathcal{P}$ of the set of states of $G_D$ and it is used as the input to a graph minimization algorithm.

#### Step 3: Apply graph minimization algorithm

The final step consists of applying a graph minimization algorithm (either Moore or Hopcroft) to the equivalence classes in partition $\mathcal{P}$ in order to generate the final refined partition $G_o$ in which each the states in each equivalence class have the same right context and generate the sequences in the right context with similar probabilities.

Once the $G_o$ is obtained, $\pi_o$ is computed with the *averageMorph* function, as described in Section 3.2.2, and the equivalence classes of $G_o$ are turned into states with morphs that are the average morph of the states in that equivalence class. Finally, the reduced PFSA $(G_o, \pi_o)$ is returned as the output of DCGraM.

**Algorithm 10** DCGraM($S, D, K$)

```
procedure DCGRAM
    ## Step 1: Create D-Markov Machine
    $(G_D, \pi_D) \leftarrow dmarkov(S, D)$
    ## Step 2: Cluster states
    $\mathcal{P} \leftarrow kmeans(K, Q, \mathcal{V}(Q))$
    ## Step 3: Apply graph minimization algorithm
    $G_o \leftarrow graphMinimization(G_D, \mathcal{P})$
    $\pi_o \leftarrow averageMorphs(G_o)$
    return $(G_o, \pi_o)$
```

### 5.1.1 Time Complexity

Generating the D-Markov machine from the original sequence $S$ of length $N$ has a complexity of $O(N)$. The main component of the complexity of DCGraM is the partitioning into equivalence classes. As in ALEPH, this step requires visiting each state and comparing it to the current equivalence classes, which for a D-Markov machine with $\Sigma^D$ states and over an alphabet $\Sigma$ takes $O(|\Sigma|^D)$.

The K-Means algorithm complexity is given by $O(MKIi)$, for which $M$ is the number of entities to be clustered, $I$ is the number of dimensions, $K$ is the number of clusters to be used and $i$ is the number of iterations needed until convergence, which is usually a small number when compared to $M$ and results only in a slight improvement after the first few iterations. For the DCGraM algorithm, this becomes $M = |\Sigma|^D$, $I = |\Sigma| - 1$, $K$ is given as an input, giving a final complexity of $O(|\Sigma|^D(|\Sigma| - 1)Ki) = O(|\Sigma|^{D+1}Ki)$. The value of $i$ is usually small compared to the other parameters of this component of the complexity.

Finally, the graph minimization algorithm can be implemented with either Moore, with a complexity of $O(M^2)$, or Hopcroft, with a complexity of $O(|\Sigma|M \log M)$. Considering that Hopcroft is used and that $M = |\Sigma|^D$ is the number of states, the complexity of this step is $O(D|\Sigma|^{D+1} \log |\Sigma|)$.

Adding up the complexities of all steps leaves the DCGraM complexity as $O(N + K|\Sigma|^{D+1} + |\Sigma|^D \log |\Sigma|) = O(N + K|\Sigma|^{D+1}i + D|\Sigma|^{D+1} \log |\Sigma|) = O(N + |\Sigma|^{D+1}(Ki + D \log |\Sigma|))$. The main component depends on whether the original sequence length or the size of the generated D-Markov is larger.

Although DCGraM is necessarily more complex than D-Markov (as one of its steps includes generating a D-Markov machine), it is shown in the next section that its final results are considerably more compact than those from D-Markov machines, with better or similar performance. Generating the PFSA is a one-time endeavor, while using it in its applications is probably done more frequently, which means it compensates to have a smaller PFSA even if it takes longer to obtain it.

## 5.2 Applications

In this section, four examples of applications of the DCGraM algorithm are shown. First, DCGraM is applied to the ternary even shift, that was previously covered in Section 4.2.1 and is a synchronizable system. This is done in order to verify how well it behaves even when applied to a category of systems for which it is not designed. The two other examples are non-synchronizable systems: the logistic map, which can show chaotic behavior, and the binary fading channel, which is an important system to simulate wireless communication systems. The values of $K$ used in each case were obtained empirically to obtain the best results.

### 5.2.1 Ternary Even Shift

As seen in Section 4.2.1, the ternary even shift is a synchronizable dynamical system with three states over a ternary alphabet. When the ALEPH algorithm is applied to a sequence $S$ of length $10^7$ generated by the ternary even shift, it is capable of recovering the original PFSA.

Figures 5.3 and 5.4, respectively show the results for conditional entropy and the Kullback-Leibler divergence for $\ell = 10$ when DCGraM is applied with $K = 5$ for $D$ from 4 to 9. Each mark in the curves of Figures 5.3 and 5.4 indicates one machine, starting with $D = 4$ in the upper left. In Figure 5.3 (and subsequent conditional entropy graphs), the original sequence baseline indicates the value of $h_{10}$ of the original sequence (in this case, the value of 1.003 previously shown in Section 4.2.1). Although DCGraM is not capable of retrieving the original machine as ALEPH, for $D = 8$ it is possible to see that it creates a machine that performs similarly to the original PFSA with 17 states while the correspondent D-Markov machine has 681. The DCGraM is not significantly larger than the original three state machine and still has a smaller size compared to the D-Markov machine.

In fact, it is observed that for any $D$, the machine obtained by DCGraM is much more compact than the correspondent D-Markov machine. For $D = 4$ the difference is of $78.0\%$ while for $D = 8$ it reaches $97.5\%$. This shows that although DCGraM is not optimal for synchronizable systems, its results are still considerably good in these cases. When it is uncertain if the system to be modeled is synchronizable or not it might be a good idea to apply DCGraM as it probably can achieve good results.

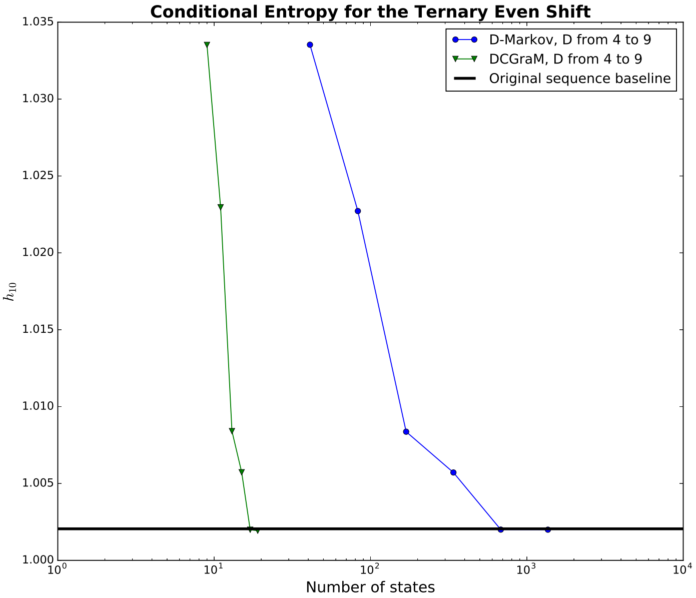

**Figure 5.3:** *Conditional entropy $h_{10}$ versus the number of states of sequences generated by D-Markov machines and DCGraM for $K = 5$ for the ternary even shift with $D$ ranging from 4 to 9.*

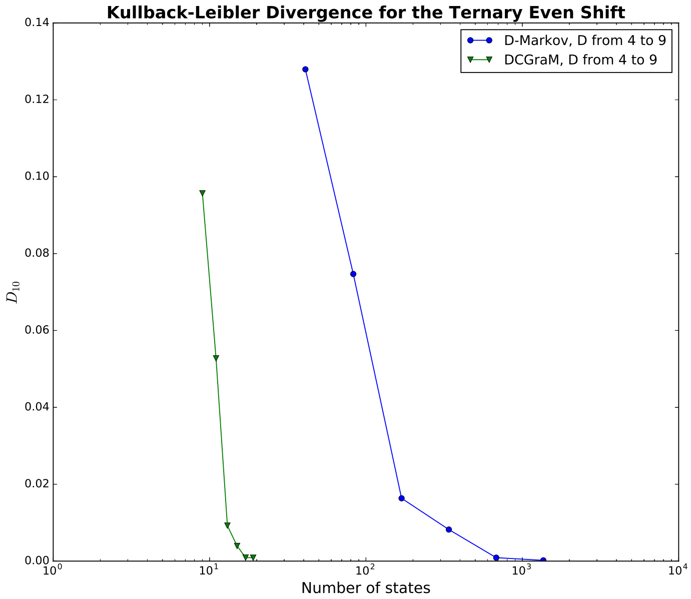

**Figure 5.4:** *Kullback-Leibler divergence $D_{10}$ versus the number of states of sequences generated by D-Markov machines and DCGraM for $K = 5$, compared to the original sequence $S$ of the ternary even shift with $D$ ranging from 4 to 9.*

### 5.2.2 Logistic Map

A non-synchronizable dynamical system is the Logistic Map, a symbolic dynamic system whose outputs are given by the difference equation [2]:

$$x_{k+1} \triangleq r x_k (1 - x_k), \text{ for } k = 0, 1, 2, \dots \tag{5.2}$$

with $x_k, r \in \mathbb{R}$. This system shows chaotic behavior for several values of $r$. As in [27], $x_0$ is set to $0.5$ and $r = 3.75$. A sequence of length $10^7$ is generated from this equation and then it is quantized with a ternary alphabet: values $x_k \leq 0.67$ are mapped to $0$; when $0.67 < x_k \leq 0.79$, it is mapped to $1$ and when $x_k > 0.79$ it is mapped to $2$. A part of that sequence and the specified threshold are shown in Figure 5.5.

D-Markov machines are generated from the ternary sequence $S$ for $D$ from 4 to 8 and they are used in DCGraM machines for the same range. The comparative results for conditional entropy for $\ell = 10$ versus the number of states are shown in Figure 5.6 and the results for Kullback-Leibler divergence for $\ell = 10$ versus number of states are shown in Figure 5.7. For these results the parameter for DCGraM is $K = 5$.

As the scale of the x-axis is logarithmic it is possible to see that the states of both D-Markov machines and DCGraM PFSA grow exponentially with $D$, although the number of states of DCGraM grows slower than the D-Markov machines. For a given $D$ it is possible to see that the quality of the D-Markov machine and of the DCGraM PFSA are similar, indicating that the DCGraM machine is a more suitable candidate to model the original system as it is more compact while performing similarly to the D-Markov machines. This is observed by noting that for $D = 4$, DCGraM is capable of obtaining a machine with $52.4\%$ less states than the D-Markov for the same $D$, while for $D = 8$, DCGraM is $71.2\%$ more compact than the respective D-Markov machine.

In Figure 5.6, it is shown that for $D = 8$ DCGraM is achieves similar system memory (as verifies by its conditional entropy) with 30 states while D-Markov achieves this for $D =$ with 104 states. Similarly, in Figure 5.7, the divergence of machine with $D = 8$ in relation to the original becomes sufficiently close to zero, which means for $D \geq 8$ DCGraM is able to create PFSAs that generate sequences similar to the original system. Using $D < 8$ might be useful to obtain more compact models, in expense of them being less precise.

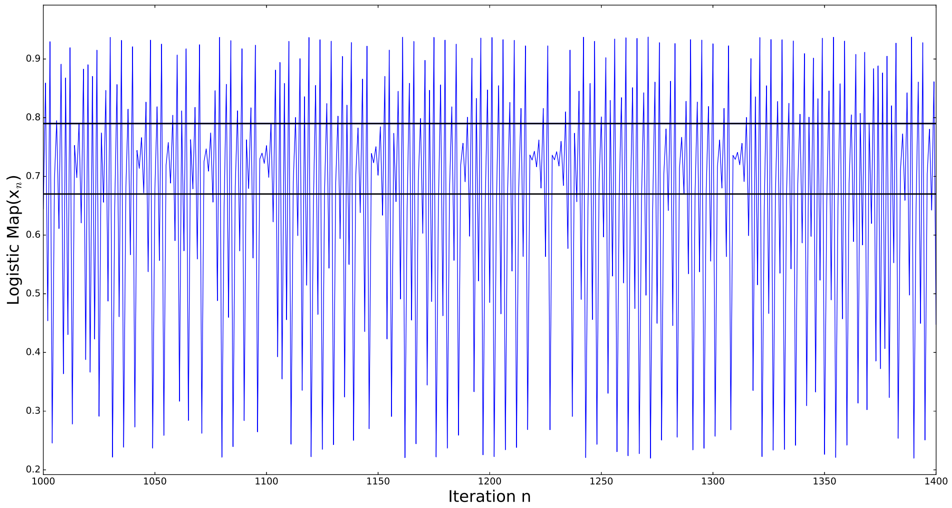

**Figure 5.5:** *Samples of the Logistic Map sequence generated by (5.2) with $x_0$ = 0.5 and $r$ = 3.75.*

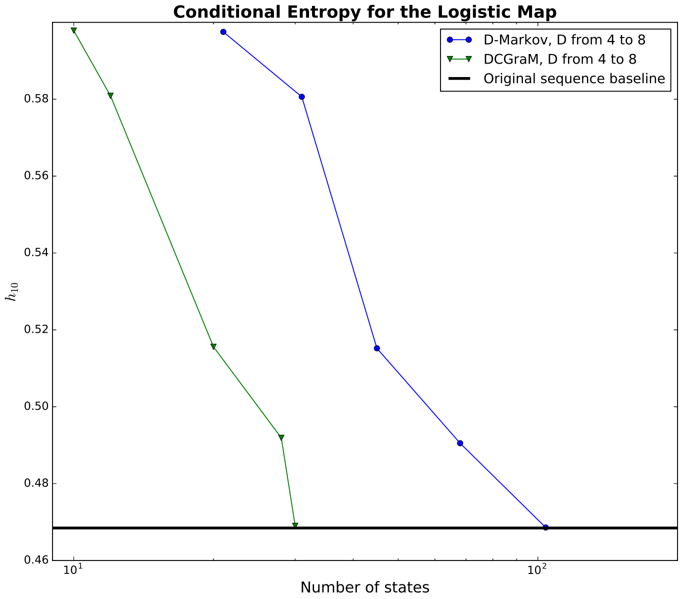

**Figure 5.6:** *Conditional entropy $h_{10}$ versus the number of states of sequences generated by D-Markov machines and DCGraM for $K = 5$ for the logistic map with $D$ ranging from 4 to 8.*

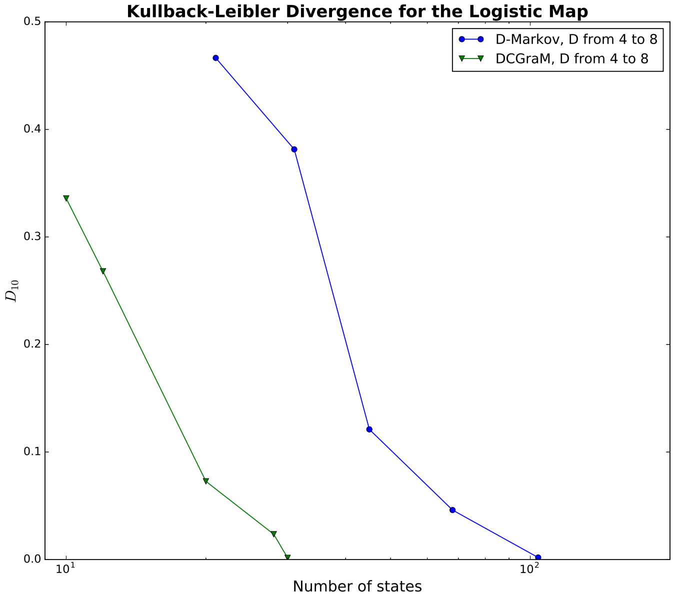

**Figure 5.7:** *Kullback-Leibler divergence $D_{10}$ versus the number of states of sequences generated by D-Markov machines and DCGraM for $K = 5$ compared to the original sequence $S$ of the logistic map for $D$ ranging from 4 to 8.*

### 5.2.3 Binary Fading Channel

Consider a binary-input binary-output discrete fading channel (DFC) with a binary input process $\{X_k\}_{k=1}^{\infty}$, $X_k \in \{0,1\}$, and a binary output process $\{Y_k\}_{k=1}^{\infty}$, $Y_k \in \{0,1\}$.

The DFC is composed of a BPSK modulator, a time-correlated flat Rayleigh fading channel with AWGN, and a hard-quantized coherent demodulator. The received symbol at the $k$th signaling interval is written as

$$R_k = \sqrt{E_s} A_k S_k + N_k, \qquad k = 1, 2, \cdots$$

where $\{S_k\} = \{(2X_k - 1)\}$, $E_s$ is the energy of the transmitted signal and $\{N_k\}$ is a sequence of independent and identically distributed zero-mean Gaussian random variables with variance $N_0/2$. The signal to noise ratio (SNR) is defined as $E_s/N_0$. Furthermore, $\{A_k\}$ is the channel's fading process with $A_k = |G_k|$, where $\{G_k\}$ is a complex Gaussian process with zero-mean, unit variance and Clarke's ACF [31] $R[k] = J_0(2\pi f_D T|k|)$, where $J_0(x)$ is the zero-order Bessel function of the first kind and $f_D T$ is the normalized maximum Doppler frequency. The random variable $A_k$ has the Rayleigh probability density function with unit second moment, $p_A(a) = 2ae^{-a^2}$, for $a > 0$. The output symbol is $Y_k = 0$ if $R_k \leq 0$, or $Y_k = 1$ if $R_k > 0$.

The input and output symbols explicitly are expressed in terms of a noise symbol $Z_k \in \{0,1\}$ as $Y_k = X_k \oplus Z_k$, where $\oplus$ denotes addition modulo 2 and the input and noise symbols are statistically independent. For a given DFC specified by its parameters $f_D T$ and SNR, a binary noise sequence $\{Z_k\}$ of the DFC of length $N = 3 \times 10^7$ is generated by computer simulation and the parameters of the DCGraM are estimated using the algorithm proposed in the previous section.

The following results are for a binary fading channel with $f_D T = 0.005$ and $SNR = 10$ dB modeled with DCGraM with $D$ varying from 4 to 8 and $K = 10$ and for D-Markov with $D$ ranging from 4 to 9. Figure 5.8 shows the result of conditional entropy versus the number of states for $\ell = 10$ and Figure 5.9 shows the result for Kullback-Leibler divergence versus the number of states for $\ell = 10$.

The results for conditional entropy for $D < 8$ are not too good as they are either almost equivalent to the original D-Markov or, in the case of $D = 7$ slightly worse, although being more compact. But for $D = 8$, the DCGraM machine $h_{10}$ is similar to the original system entropy and performs similarly to the D-Markov machine with $D = 9$.

On the other hand, the results for Kullback-Leibler divergence are better. The DCGraM machine for $D = 8$ with 199 states shows similar performance to the D-Markov machine with $D = 9$ with 512 states, which represents a reduction of $61.1\%$ in the number of states. This indicates that for $D = 8$

the DCGraM machines are capable of generating sequences similar to the ones from the original system and also retrieve its system memory with similar quality of a larger D-Markov machine.

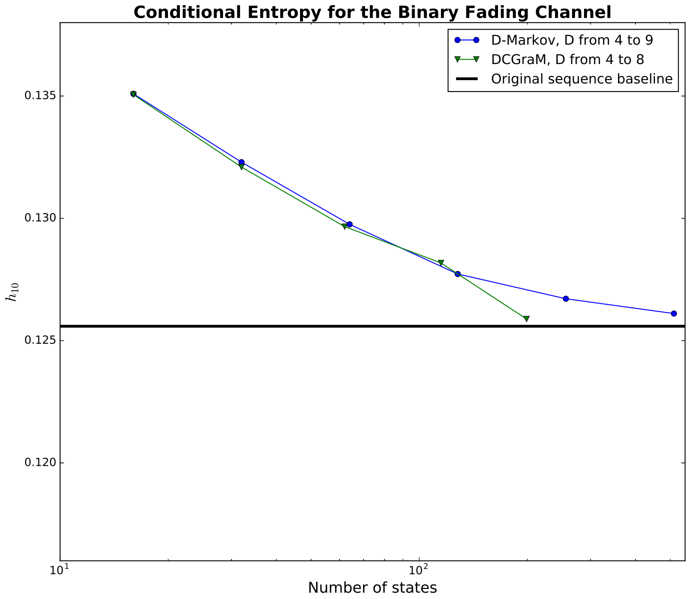

**Figure 5.8:** *Conditional entropy $h_{10}$ versus the number of states of sequences generated by D-Markov machines ($D$ ranging from 4 to 9) and DCGraM with $K = 10$ ($D$ ranging from 4 to 8) for the binary fading channel with $f_D T = 0.005$ and $SNR = 10$ dB.*

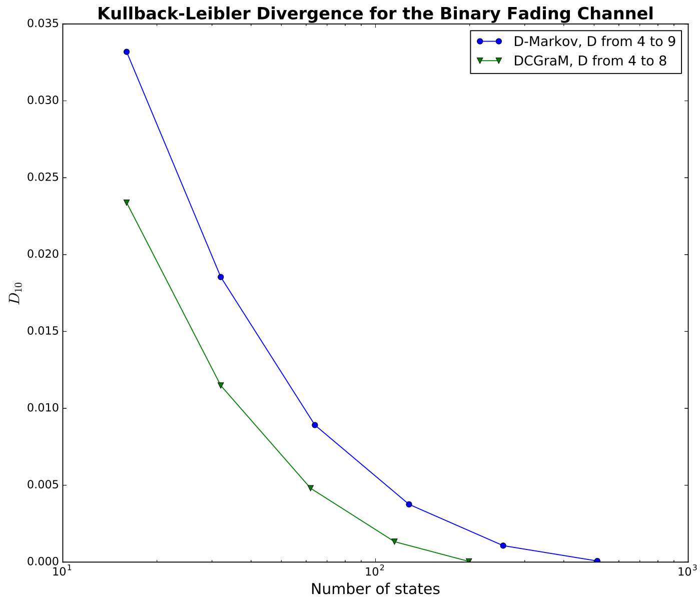

**Figure 5.9:** *Kullback-Leibler divergence $D_{10}$ versus the number of states of sequences generated by D-Markov machines (D ranging from 4 to 9) and DCGraM with $K = 10$ (D ranging from 4 to 8) compared to the original sequence $S$ for the binary fading channel $f_D T = 0.005$ and $SNR = 10$ dB.*

### 5.2.4 Lorenz Attractor

The Lorenz attractor is an attractor (i.e. an indecomposable set of points towards which a dynamical system tends to evolve) with associated dynamics that have sensitive dependence on initial conditions. It is determined by the Lorenz equations, given by the following system of differential equations [2]:

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y-x), \\ \frac{dy}{dt} &= rx - y - xz, \\ \frac{dz}{dt} &= xy - bz, \end{aligned} \tag{5.3}$$

where $x, y, z, \sigma, r, b \in \mathbb{R}$ and $r, b$ and $\sigma$ are parameters. Using $\sigma = 10, b = \frac{8}{3}$ and $r = 28$ and the initial condition $x(0) = 0, y(0) = 1$ and $z(0) = 1$, and plotting $x$ by $z$, the evolution of the system is shown in Figure 5.10, which is a projection of the three-dimensional Lorenz attractor on the $xz$-plane.

To generate a discrete sequence of binary symbols from the Lorenz attractor, we take a point in the section of the attractor shown by the red and green lines in Figure 5.10. Starting by a point in the green line means that the next sample is located on the same side of the graph in relation to the $z$-axis. On the other hand, if the point is on the red line, the next sample is on the other side in relation to the $z$-axis. A sequence of such points is shown in Figure 5.11. The points above the upper red line in Figure 5.11 and below the lower red line are mapped as 1, while the others are mapped as 0 and a binary sequence $S$ of length $10^6$ is then created and used as input for the DCGraM algorithm.

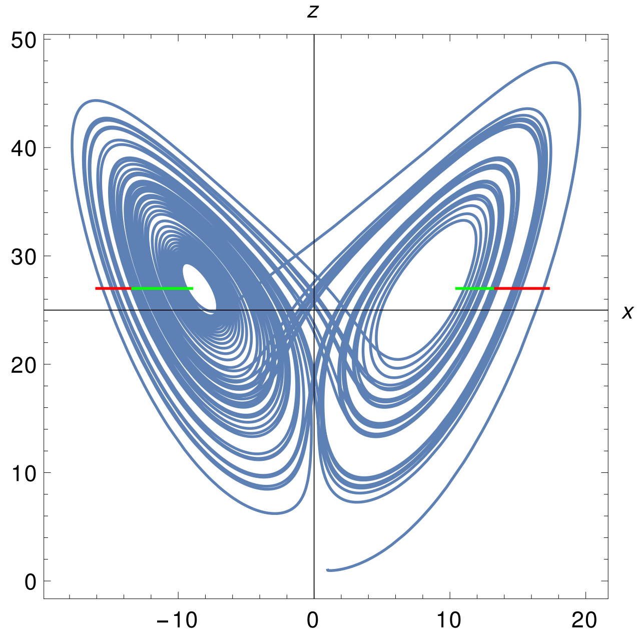

**Figure 5.10:** *The $xz$-projection of the Lorenz attractor for $\sigma = 10, b = \frac{8}{3}$ and $r = 28$ and the initial condition $x(0) = 0, y(0) = 1$ and $z(0) = 1$. Following the dynamic of the system on the section of the attractor given by the green and red lines, points on the green lines end in points of the same side in relation to the $z$-axis. Points on the red lines end in the opposite side in relation to the $z$-axis.*

The results of the conditional entropy $h_{10}$ and Kullback-Leibler divergence $D_{10}$ for D-Markov and DCGraM applied to the Lorenz attractor with $K = 5$ and $D$ ranging from 4 to 9 are shown in Figures 5.12 and 5.13, respectively. Although the results oscillate for $D = 5$ for both $h_{10}$ and $D_{10}$ and for $D = 7$ for $D_{10}$, it is possible to see that performance similar to the original system is obtained for $D = 9$. While the D-Markov machine achieves this results with 68 states, the DCGraM PFSA presents similar performance with 18 states, which represents a reduction of $70.1\%$ in the number of states.

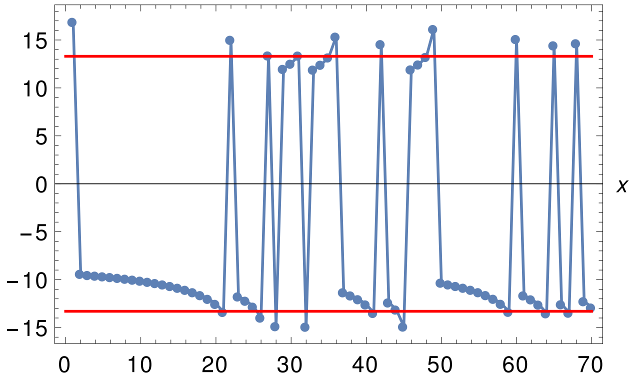

**Figure 5.11:** *The sequence generated by the section of the attractor shown in Figure 5.10. Samples above the upper red line or below the lower red line are mapped as 0, while the rest of the samples are mapped as 1.*

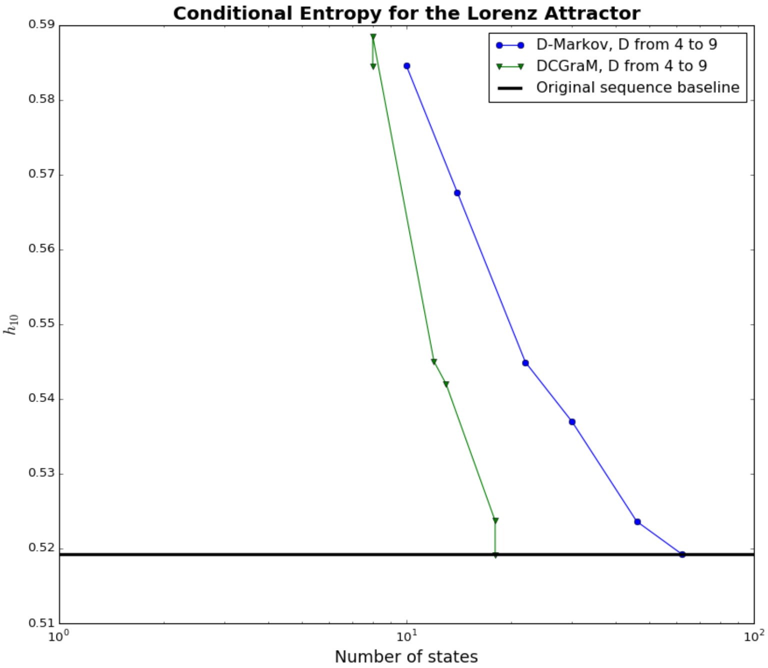

**Figure 5.12:** *Conditional entropy $h_{10}$ versus the number of states of sequences generated by D-Markov machines and DCGraM for $K = 5$ for the Lorenz attractor with $D$ ranging from 4 to 9.*

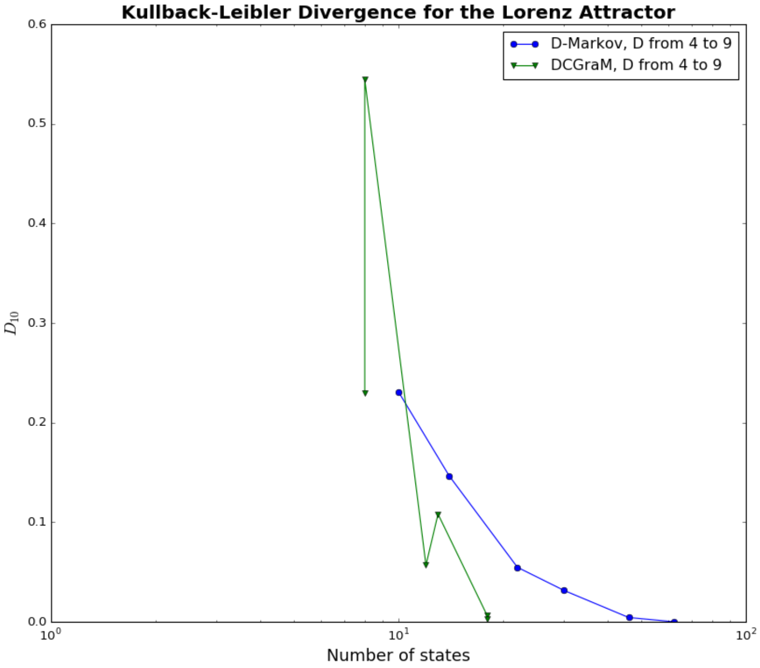

**Figure 5.13:** *Kullback-Leibler divergence $D_{10}$ versus the number of states of sequences generated by D-Markov machines and DCGraM for $K = 5$ compared to the original sequence $S$ of the Lorenz attractor for $D$ ranging from 4 to 9.*

# CHAPTER 6

# CONCLUSION

In this work two algorithms for modeling discrete dynamical systems were presented. Both of them use PFSA to obtain compact representations while presenting reasonable time complexities. The first algorithm is ALEPH, which is more suitable to model synchronizable dynamical systems, while the second one is DCGraM that covers non-synchronizable PFSA. They both start by analyzing the statistics of the output sequence of the system to be modeled and use graph minimization techniques to obtain even more compact results. Furthermore, DCGraM also uses machine learning techniques.

The ALEPH algorithm makes use of synchronization words as starting points for its inner workings. A new algorithm was also developed to find synchronization words more efficiently than the brute force method used in CRISSiS. It creates an RTP and explores it until it finds all the synchronization words. After this, it takes the leaf nodes from the RTP and connect them using the $\Omega$ criterion in order to create a complete graph. From this graph a partition is created by grouping states with similar morphs in equivalence classes. Finally, the graph minimization algorithm is then applied to this partition obtaining a final PFSA.

We applied ALEPH to some synchronizable systems such as the ternary even shift, the tri-shift, a six state PFSA and the maximum entropy $(d, k)$-constrained code. For all these cases, ALEPH was able to recover the original machines. Its results for conditional entropy and Kullback-Leibler divergence were compared to the results obtained by CRISSiS and by D-Markov machines and it was seen that a D-Markov machine that compares to an ALEPH generated PFSA would need to be considerably larger, while CRISSiS and ALEPH show similar performance, but ALEPH presents a lower complexity.

The DCGraM algorithm starts by creating a D-Markov machine for a given $D$ based on the output sequence from the original system. It then uses the modified K-Means algorithm using the Kullback-Leibler divergence as a kind of metric to cluster states together and finally applies a graph minimization algorithm to obtain a final PFSA.

The DCGraM algorithm was applied to a synchronizable system and two non-synchronizable systems: the ternary even shift, the logistic map and the fading channel. As this algorithm is actually a refinement method over D-Markov machines using machine learning techniques its results were compared to the original D-Markov machines. It was shown that although DCGraM does not recover the original PFSA of the ternary even shift, as ALEPH does, it is still able to produce a significantly smaller machine compared to a D-Markov machine for some $D$ and it produces good results. For the logistic map the DCGraM were significantly smaller than the D-Markov machines while retaining similar conditional entropy and Kullback-Leibler divergences for a given $D$. For the fading channel, although the DCGraM PFSA are smaller than the D-Markov machines the difference is not as noticeable as for the logistic map, although this difference gets larger as $D$ increases. Still, the DCGraM PFSA showed similar performance to their D-Markov counterparts. The difference in amount of states that generate good machines for the DCGraM varies from producing machines with the same number of states as the D-Markov for lower values of $D$ to $97\%$ when applied to the synchronizable system, but overall managing to produce compact and precise machines.

## 6.1 Future Work

Future work that improves the presented algorithms include using techniques from information theory and statistical mechanics to analyze the given sequence from the original system and determine whether it comes from a synchronizable system or not.

It would also be interesting to explore the possibilities of a dimension reduction, such as the Principal Component Analysis (PCA) [32][33][34], to reduce the computational burden of DCGraM. Besides that, more efficient and modern clustering algorithms such as DBSCAN[35] or OPTICS[36] can also be considered to substitute K-Means in DCGraM.

An improvement over the traditional PFSA would be instead of using fixed probabilities for state transitions in a PFSA, to use a non-deterministic approach to these transitions based on the original sequence statistic.

Finally, it would also be interesting to apply the models obtained by our algorithms to applications such as fault detection in which even smaller and less precise machines are capable of quickly detecting anomalies [27].

# Reference

[1] D. Luenberger, "Introduction to dynamic systems: theory, models, and applications," 1979.

[2] S. Strogatz, "Nonlinear dynamics and chaos: With applications to physics, biology, chemistry and engineering," 2001.

[3] D. Lind and B. Marcus, *An Introduction To Symbolic Dynamics and Codings*. Cambridge University Press, 1995.

[4] V. Rajagopalan and A. Ray, "Symbolic time series analysis via wavelet-based partitioning," *Signal Processing*, vol. 86, no. 11, pp. 3309–3320, 2006.

[5] H. Li, W.-Y. Yin, K. Banerjee, and J.-F. Mao, "Circuit modeling and performance analysis of multi-walled carbon nanotube interconnects," *IEEE Transactions on electron devices*, vol. 55, no. 6, pp. 1328–1337, 2008.

[6] P. M. Nørgård, O. Ravn, N. K. Poulsen, and L. K. Hansen, "Neural networks for modelling and control of dynamic systems-a practitioner's handbook," 2000.

[7] T. Boulard, R. Haxaire, M. Lamrani, J. Roy, and A. Jaffrin, "Characterization and modelling of the air fluxes induced by natural ventilation in a greenhouse," *Journal of Agricultural Engineering Research*, vol. 74, no. 2, pp. 135–144, 1999.

[8] R. Temam, *Infinite-dimensional dynamical systems in mechanics and physics*, vol. 68. Springer Science & Business Media, 2012.

[9] M. D. Lewis, "Bridging emotion theory and neurobiology through dynamic systems modeling," *Behavioral and brain sciences*, vol. 28, no. 02, pp. 169–194, 2005.

[10] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. Academic press, 2000.

[11] A. S. Willsky, “A survey of design methods for failure detection in dynamic systems,” *Automatica*, vol. 12, no. 6, pp. 601–611, 1976.

[12] D. Heckerman, D. Geiger, and D. M. Chickering, “Learning bayesian networks: The combination of knowledge and statistical data,” *Machine learning*, vol. 20, no. 3, pp. 197–243, 1995.

[13] A. Corazza and G. Satta, “Probabilistic context-free grammars estimated from infinite distributions,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 29, no. 8, pp. 1379–1393, 2007.

[14] L. R. Rabiner, “A tutorial on hidden markov models and selected applications in speech recognition,” *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[15] I. Chattopadhyay, Y. Wen, A. Ray, and S. Phoha, “Unsupervised inductive learning in symbolic sequences via recursive identification of self-similar semantics,” *American Control Conference*, June 2011.

[16] E. Vidal, F. Thollard, C. de la Higuera, F. Casacuberta, and R. C. Carrasco, “Probabilistic finite-state machines - part I,” *IEEE Transactions On Pattern Analysis And Machine Intelligence*, vol. 27, pp. 1013–1025, July 2005.

[17] K. P. Murphy *et al.*, “Passively learning finite automata,” Santa Fe Institute, 1995.

[18] A. Ray, “Symbolic dynamic analysis of complex systems for anomaly detection,” *Signal Processing*, vol. 84, no. 7, pp. 1115–1130, 2004.

[19] C. R. Shalizi and K. L. Shalizi, “Blind construction of optimal nonlinear recursive predictors for discrete sequences,” in *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pp. 504–511, AUAI Press, 2004.

[20] G. Lallement, *Semigroups and combinatorial applications*. John Wiley & Sons, Inc., 1979.

[21] J. Berstel, L. Boasson, O. Carton, and I. Fagnot, “Minimization of automata,” *arXiv:1010.5318*, December 2010.

[22] E. F. Moore, “Gedanken-experiments on sequential machines,” *Automata studies*, vol. 34, pp. 129–153, 1956.

[23] D. J. MacKay, *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[24] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, “An efficient k-means clustering algorithm: Analysis and implementation,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 24, no. 7, pp. 881–892, 2002.

[25] G. Hamerly and C. Elkan, “Alternatives to the k-means algorithm that find better clusterings,” in *Proceedings of the eleventh international conference on Information and knowledge management*, pp. 600–607, ACM, 2002.

[26] D. Arthur, B. Manthey, and H. Röglin, “k-means has polynomial smoothed complexity,” in *Foundations of Computer Science, 2009. FOCS’09. 50th Annual IEEE Symposium on*, pp. 405–414, IEEE, 2009.

[27] K. Mukherjee and A. Ray, “State splitting and merging in probabilistic finite state automata for signal representation and analysis,” *Signal Processing*, vol. 104, pp. 105–119, April 2014.

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, vol. 6. MIT press Cambridge, 2001.

[29] T. M. Cover and J. A. Thomas, *Elements of information theory*. John Wiley & Sons, 2012.

[30] K. A. Schouhamer, P. H. Siegel, and J. K. Wolf, “Codes for digital recorders,” *IEEE Transactions on Information Theory*, vol. 44, no. 6, pp. 2260–2299, October 1998.

[31] R. Clarke, “A statistical theory of mobile-radio reception,” *Bell Labs Technical Journal*, vol. 47, no. 6, pp. 957–1000, 1968.

[32] I. K. Fodor, “A survey of dimension reduction techniques,” *Center for Applied Scientific Computing, Lawrence Livermore National Laboratory*, vol. 9, pp. 1–18, 2002.

[33] L. I. Smith *et al.*, “A tutorial on principal components analysis,” *Cornell University, USA*, vol. 51, no. 52, p. 65, 2002.

[34] B. C. Geiger and G. Kubin, “Relative information loss in the PCA,” in *Information Theory Workshop (ITW), 2012 IEEE*, pp. 562–566, IEEE, 2012.

[35] D. Arlia and M. Coppola, “Experiments in parallel clustering with DBSCAN,” in *European Conference on Parallel Processing*, pp. 326–331, Springer, 2001.

[36] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, “OPTICS: ordering points to identify the clustering structure,” in *ACM Sigmod record*, vol. 28, pp. 49–60, ACM, 1999.

# Appendix A

# Hopcroft Algorithm

THIS appendix presents an alternative graph minimization algorithm that can be used instead of Moore. It has a lower complexity but it is slightly more complicated to implement. The pseudo-code is shown in Algorithm 11.

**Algorithm 11** Hopcroft($G$)

```
$\mathcal{P} \leftarrow InitialPartition(G)$
$\mathcal{W} \leftarrow \emptyset$
for all $\sigma \in \Sigma$ do
    Append(($\min$($F$, $F^c$,$\sigma$),$\mathcal{W}$)
    while $\mathcal{W} \neq \emptyset$ do
        ($W$,$\sigma$) $\leftarrow$ TakeSome($\mathcal{W}$)
        for each $P \in \mathcal{P}$ which is split by $(W, \sigma)$ do
            $P', P'' \leftarrow (W, \sigma)|P$ Replace $P$ by $P$' and $P''$ in $\mathcal{P}$
            for all $\tau \in \Sigma$ do
                if $(P, \tau) \in \mathcal{W}$ then
                    Replace $(P, \tau)$ by $(P', \tau)$ and $(P'', \tau)$ in $\mathcal{W}$
                else
                    Append(($\min(P', P'', \tau), \mathcal{W}$)
```

The notation min($P$, $P'$) indicates the set of smaller size of the two sets $P$ and $P'$ or any of them when both have the same size. Hopcroft's algorithm computes the coarsest partition that saturates the set $F$ of final states. The algorithm keeps a current partition $\mathcal{P} = \{P_1, \dots, P_n\}$ and a current set $\mathcal{W}$ of splitters (i.e. pairs $(W, \sigma)$ that remain to be processed where $W$ is a class of $\mathcal{P}$ and $\sigma$ is a symbol) which is called the *waiting set*. $\mathcal{P}$ is initialized with the initial partition following the same criteria as described in Moore's algorithm. The waiting set is initialized with all the pairs (min($F$, $F^c$), $\sigma$) for

$\sigma \in \Sigma$.

For each iteration of the loop, one splitter $(W, \sigma)$ is taken from the waiting set. It then checks whether $(W, \sigma)$ splits each class of $P$ of $\mathcal{P}$. If it does not split, nothing is done, but if it does then $P'$ and $P''$ (which are the result of splitting *P* by $(W, \sigma)$) replace $P$ in $\mathcal{P}$. Next, for each letter $\tau \in \Sigma$, if the pair $(P, \tau)$ is present in $\mathcal{W}$ , it is replaced by the two pairs $(P', \tau)$ and $(P''\tau)$. Otherwise, only $(min(P', P''), \tau)$ is added to $\mathcal{W}$.

The previous computation is performed until $\mathcal{W}$ is empty. It is proven that the final partition of the algorithm is the same as the one given by the Nerode equivalence. No specific order of pairs $(W, \sigma)$ is described, which gives rise to different implementations in how the pairs are taken from the waiting set but all of them produce the right partition of states. Hopcroft proved that the running time of any execution of his algorithm is bounded by *O(*$|\Sigma|n \log n$*)*.