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

To my grandmother Zélia

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

To my advisers for all the guidance provided.

RESUMO

Sistemas dinâmicos discretos são amplamente aplicados em uma variedade de aplicações de engenharia e cientifícas. Modelagem de sistemas envolve aproximar parâmetros de um sistema analisando suas estatísticas e as utilizando para criar um modelo que se comporte de maneira similar ao sistema original. Esses modelos podem ser usados para simulação, referência ou detecção de falhas. Este trabalho apresenta um algoritma para modelar sistemas dinâmicos discretos por meio de Autômatos Finitos Probabilísticos analisando sequências geradas pelo sistema original e aplicando métodos estatísticos e técnicas de redução de grafos para obter modelos compactos e eficients. Sua performance e complexidade temporal são comparadas com algoritmos presentes na literatura que buscam atingir o mesmo objetivo.

Palavras-chaves: Probabilistic finite state automata, dynamic systems, system modeling, fault detection, conditional entropy, Kullback-Leibler divergence.

ABSTRACT

Discrete dynamical systems are widely applied in a variety of engineering and scientific applications. System modeling involves approximating system parameters by analyzing its statistics and using them to create a model that behaves similarly to the original system. These models can be used for simulation, reference or fault detection. The current work presents an algorithm to model discrete dynamical systems using Probabilistic Finite State Automata by analyzing sequences generated by the original system and applying statistical methods and graph reduction techniques to obtain compact and efficient models. Its performance and time complexity are compared with other algorithms present in literature that aim to achieve the same goal.

Keywords: Probabilistic finite state automata, dynamic systems, system modeling, fault detection, conditional entropy, Kullback-Leibler divergence.

LIST OF FIGURES

LIST OF TABLES

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. 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 [4]. Discrete dynamical systems can be obtained by sampling a continuous one and can only produce symbols from a discrete set called an alphabet and their behavior can be described by difference equations or discrete state transform relations [5] [6].

These systems provide a useful framework for analyzing phenomena in several fields of Engineering and Science such as meteorology [1], mechanics [2] and neurobiology [3]. These systems often lead to chaotic behavior, which means that given 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 [4].

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.

In order to obtain models for dynamical systems and pattern classification, there are methods and frameworks such as belief networks [9], probabilistic context free grammars [10] and hidden Markov models [11] but they tend do be complex and require large sampling times. [7]. An alternative method, which is the one chosen in this work, is to use Probabilistic Finite State Machines (PFSA), which can solve this issue 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 [7], 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 [8] in which the symbol generation probabilities are not specified and there is a distribution over the possible end states. The advantages of using PFSA are that they are simple and the sample time required for learning them is easy to characterize [7] and it is also an efficient framework for learning the causal structure of observed dynamical behavior [12].

The algorithms that construct PFSA include D-Markov machines [13], 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) [14], which starts by assuming that the process is 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 χ^2 or Kolmogorov-Smirnov) and then they are checked to keep them to be deterministic. There is also the Compression via Recursive Identification of Self-Similar Semantics (CRISSiS) [7] which firsts find 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 [7] CRISSiS outperforms CSSR.

1.2 Objectives and Contributions

In the current work, we are interested in modeling discrete dynamical systems having only their outputs. To achieve this goal, we developed an algorithm that analyzes the statistics of output sequences from dynamical systems and model them via Probabilistic Finite State Automata (PFSA).

The goal of the algorithm developed in this dissertation is to take a sequence generated by a discrete dynamical system and create a PFSA model of it by analyzing the statistics and structure inherent to the sequence. In order to obtain models that are less memory consuming, our algorithm applies techniques of graph minimization to obtain smaller PFSA. The algorithm is first applied to sequences generated by other PFSA to determine how well it recovers the original systems and then it is applied to dynamic system sequences whose original system might not have a PFSA representation. The results are compared to other algorithms in the literature that seek similar goals.

As CRISSIS, ALPHA 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.

1.3 Work Structure

This dissertation is organized in six chapters: Chapter 2 reviews the theoretical background discussing discrete sequences, PFSA and graph minimization algorithm while also showing the CRISSiS and D-Markov Machine algorithms used in the literature. Chapter 3 then presents the ALEPH algorithm and then analyzes the time complexity of running it. Chapter 4 presents some dynamical systems represented with PFSA and shows the comparative results of ALEPH, CRISSiS and D-Markov when recovering the original PFSA. Chapter 5 shows applications whose PFSA representation is not known beforehand and an alternative algorithm to be applied in such situation and how this algorithm and the ones present in literature perform to model such systems. Finally, in Chapter 6 a conclusion is discussed and plans for future works to improve the ALEPH algorithm are presented.

CHAPTER 2

PRELIMINARIES ON GRAPHS AND PROBABILISTIC FINITE STATE AUTOMATA

IN this chapter we revise concepts from graphs and PFSA [5][8] that will be required in the subsequent chapters. The concept of graph minimization is presented and two mainstream algorithms to achieve this are described, Moore and Hopcroft. Finally, two well known algorithms to model dynamic systems with PFSA are presented, 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 Σ is called a word and its length is denoted by $|u|$. The empty word ε is defined as the sequence with length 0. The set of all possible words of length n symbols from Σ is Σ^n and the set of all sequences of symbols from Σ with all possible lengths, including the empty sequence sequence ε, is Σ^*.

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 $|uv| = |u| + |v|$. Concatenation 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 ε is a neutral element for concatenation, that is, $\varepsilon u = u\varepsilon = u$. This means that Σ^* with the operation of concatenation is a Monoid, as it is a set with an associative operation with an identity element.

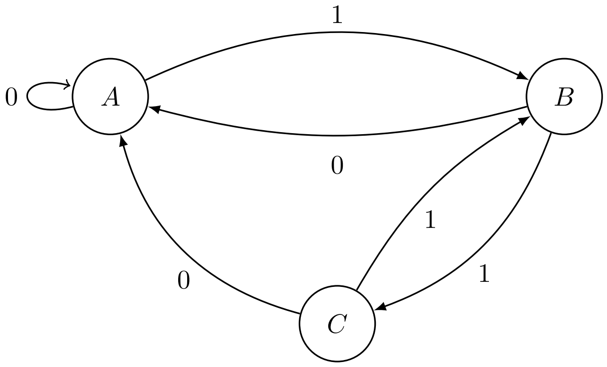

Figura 2.1: *A three-state graph with Q = {A, B, C} and* $\Sigma = \{0, 1\}$.

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

Definition 2.1 – Graph

A graph G over the alphabet Σ consists of a triple (Q, Σ, δ):

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

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

▷ *δ is the state transition function $Q \times \Sigma \to Q$;* □

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 σ. This realization of the transition function is called the outgoing edge from q to q' with symbol σ. Figure 2.1 shows an example of a three-state graph over a binary alphabet from where it 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 ω 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 ω = 001110.

Definition 2.2 – Follower Set

The follower set 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: □

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

Definition 2.3 – Language of a Graph

The language $\mathcal{L}$ of a graph G is the the set of follower sets for each state $q \in Q$: □

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

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

2.3 Graph Minimization

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. It is desirable to use a graph with fewer states as it can be represented with a lower memory requirement. This is called a minimal graph.

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 distinct follower set. □

If a graph has two distinct states q_1 and q_2 with the same follower set, a new graph that generates the same language can be obtained by merging these states, i.e. replace q_1 and q_2 by a new state q such that if any state $q\prime$ has an outgoing edge to either q_1 or q_2 ($\delta(q\prime, \sigma) = q_1$ or q_2 for some $\sigma \in \Sigma$) it will now be an outgoing edge to q ($\delta(q\prime, \sigma) = q$) and copying all of the outgoing edges of both q_1 and q_2 to q. When all the states have distinct follower sets, none of them can be excluded without affecting the generated language.

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

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

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 [15]. 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 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 said that F is saturated by $\mathcal{P}$. Given $\mathcal{Q}$, another partition of E, it 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 by 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 $\mathcal{U} = \mathcal{P}_1 \wedge \mathcal{P}_2 \wedge \ldots \wedge \mathcal{P}_n = \bigwedge_{j=1}^{n} \mathcal{P}_j$. When $n = 0$, $\mathcal{P}$ is the universal partition comprised of just E and it is the neutral element for the $\wedge$-operation.

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 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, σ) 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, σ) 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, σ), 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 $\sigma \in \Sigma$. □

2.3.1 Moore Algorithm

The first minimization algorithm is the Moore algorithm [16]. 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.

Given a graph *G = (Q,* Σ, δ*)* 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

Algorithm 1 Moore(G)

```
$\mathcal{P} \leftarrow InitialPartition(G)$
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 $\mathcal{P} = \mathcal{P}'$
```

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 equations hold:

Proposition 2.3

For $h \geq 0$, one has

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

This computation means that for each symbol $\sigma \in \Sigma$ and for each equivalence class $P \in \mathcal{M}_h$ (where $mathcalM_h$ is the partition of the previous iteration) a splitter (P, σ) 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 σ 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 σ. 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 Σ. 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).

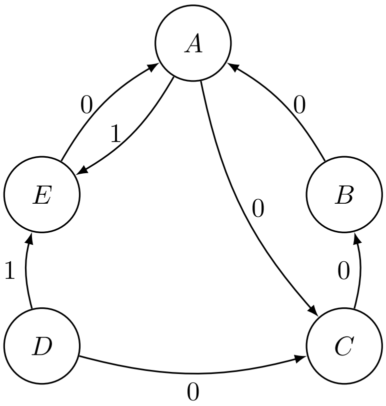

Figura 2.2: *An example of a graph that is not minimal.*

This previous computation is performed in Algorithm 1 in which the loop refines the current partition. As 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 (see 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 based on the criteria 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):

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

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

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

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.

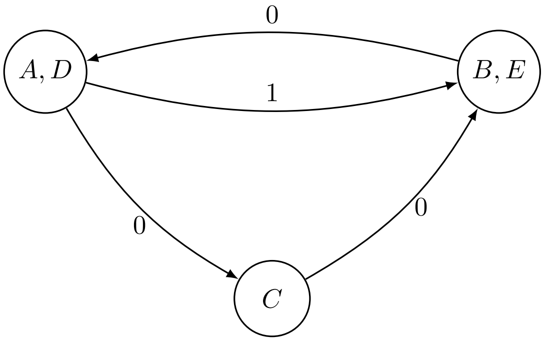

Figura 2.3: *Application of Moore algorithm to the initial partition.*

2.3.2 Hopcroft's Algorithm

Algorithm 2 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 $\mathcal{P} = \{P_1, \ldots, P_n\}$ and a current set $\mathcal{W}$ of splitters (i.e. pairs (W, σ) that remain to be processed where W is a class of $\mathcal{P}$ and σ is a letter) 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), σ) for $\sigma \in \Sigma$.

For each iteration of the loop, one splitter (W, σ) is taken from the waiting set. It then checks whether (W, σ) 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, σ) replace P in $\mathcal{P}$. Next, for each letter $\tau \in \Sigma$, if the pair (P, τ) is present in $\mathcal{W}$ is replaced by the two pairs (P', τ) and $(P''\tau)$. Otherwise, only

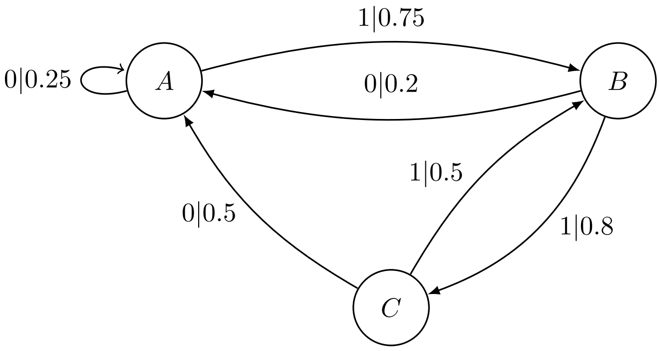

Figura 2.4: *A PFSA with the same graph of Figure 2.1.*

$(min(P', P''), \tau)$ 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, σ) 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(*$|\Sigma|n \log n$*)*.

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* π *associated to each of its outgoing edges, i.e.* (G, π)*. 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.* □

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 do adapt the concept of synchronization word to the context of PFSA as defined in [7].

Definition 2.8 – PFSA Synchronization Word

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

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

□

Definition 2.8 means that the probability of obtaining any sequence after the synchronization word 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 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 $|\Sigma|^{L_1+1}$ previous words and $|\Sigma|^{L_2+1}$ continuation words.

A synchronization words 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 PFSA graph, the following criterion is used to create the initial partition:

Definition 2.9

Given a PFSA $\{G, \pi\}$, 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 α.

Tabela 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

2.5 Consolidated Algorithms

In this section, two algorithms that construct a PFSA from a sequence S of length N over an alphabet Σ are presented: D-Markov Machines and CRISSiS.

2.5.1 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 all symbol blocks of length D are taken as the states in the set Q and their transition probabilities can be computed as follows. Consider 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:

$$\Pr(\tau | q) = \frac{\Pr(q\tau)}{\Pr(q)}, \tag{2.7}$$

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

For example, considering 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 (2.7), the D-Markov machine shown in Figure 2.5 is built.

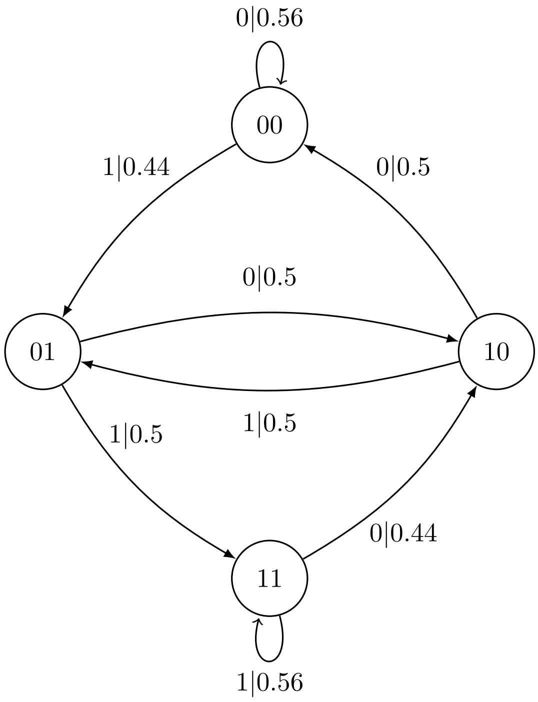

Figura 2.5: *A D-Markov machine with sequence S and D = 2.*

2.5.2 CRISSiS

The Compression via Recursive Identification of Self-Similar Semantics (CRISSiS) algorithm is presented in [7]. It assumes that a sequence *S* over an alphabet Σ of length *N* which 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 ω, it is returned as the synchronization word ω_{syn}.

The auxiliar function *hypothesisTest* is used to check (2.6) for a given value of α. It can be implemented either as the χ^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:

$$\Pr(v|q_1) = \Pr(v|q_2), \forall v \in \Sigma^* \tag{2.8}$$

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:

$$\Pr(v|q_1) = \Pr(v|q_2), \forall v \in \Sigma^d, d = 1, \ldots, L_2. \tag{2.9}$$

If two states pass the statistical test using (2.9), they are considered to be statistically the same. Strings q_1 and q_2 need to be synchronizing in order to use (2.9). If ω 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 Q̃ 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 ω_{syn}. The function δ for the machine is initialized with $\delta(\omega_{syn}, \sigma)$ equal to $\omega_{syn}\sigma$ for all $\sigma \in \Sigma$. This is represent by a tree using ω_{syn} as the root node to $|\Sigma|$ children, which are the states in Q̃. 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 α with each of the states in Q. If a match between some $\omega \in$ Q̃ and some $\omega^* \in Q$ is found, the child state is removed and all the transitions to ω are redirected to ω^* (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 Q̃. This procedure is repeated until no new candidate states have

Algorithm 4 isSynString(ω, 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(ω, 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
```

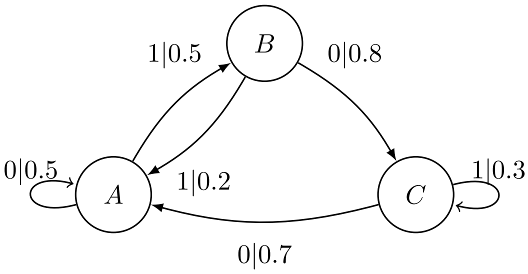

Figura 2.6: *The Tri-Shift PFSA.*

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 of occurrence of ω_{syn}) is fed to the PFSA starting at state ω_{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 [7]. 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 χ^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.

Tabela 2.2: *Probabilities of words generated by the Tri-Shift up to length 10000.*

$\ell = 1$	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 (2.9). $\mathcal{V}(000) = [0.494, 0.506]$ 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 [7], 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

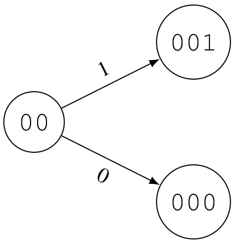

Figura 2.7: *Tree with 00 at its root, Q = {00}.*

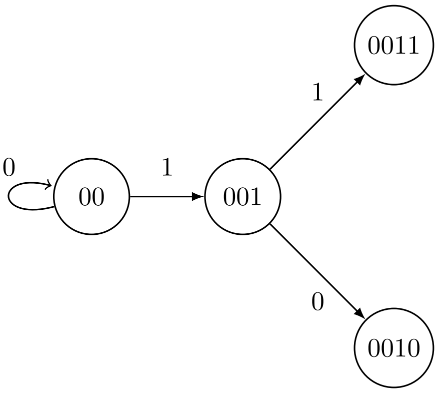

Figura 2.8: *Second iteration of three, Q = {00, 001}.*

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.

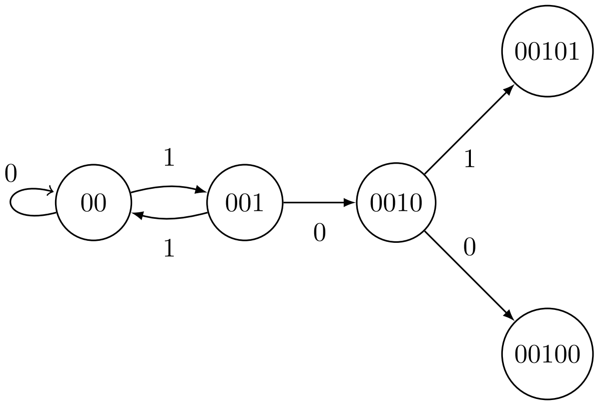

Figura 2.9: *Third iteration of the three, Q = {00, 001, 0010}.*

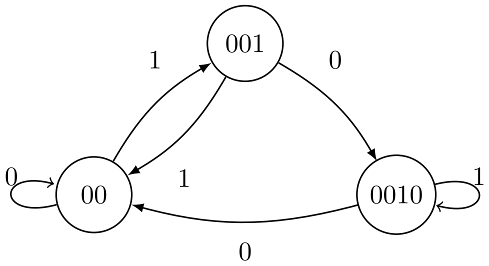

Figura 2.10: *Recovered Tri-Shift topology.*

Chapter 3

Algorithm Description

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 $\mathcal{S}$ 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 Σ 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.

The proposed algorithm uses a rooted tree with probabilities $\mathcal{S}$ over an alphabet Σ to search for synchronization words. At the beginning of the algorithm, all states of $\mathcal{S}$ are considered valid

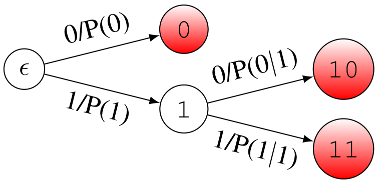

Figura 3.1: *Example of a rooted tree with probabilities.*

candidates to be synchronization words. A search is performed in $\mathcal{S}$ starting by its root using a statistical test (which compares two state morphs via a test such as χ^2 or Kolmogorov-Smirnov for a given confidence level α) 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) $\mathcal{S}$ 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 ϵ, 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 Σ 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 ϵ 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 $\mathcal{S}$ 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) [17]. A dictionary d is a mapping between two sets $d : X \rightarrow Y$. 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 $\mathcal{S}$ 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 state from $\mathcal{S}$ labeled with the shortest word that has ω as a suffix and it is still a valid candidate for synchronization word. The function *shortestValidSuffix* receives the word $\omega = \sigma_1\sigma_2 \dots \sigma_n \in \Sigma^*$, the tree $\mathcal{S}$ and the dictionary *candidacy* as inputs. First $\omega = \sigma_1\sigma_2 \dots \sigma_n$ is reversed, $\omega_{rev} = \sigma_n\sigma_{n-1} \dots \sigma_1$. Then, the tree $\mathcal{S}$ is traversed according to ω_{rev}, starting at the root ϵ. *candidacy[ϵ]* is checked and if it is true, ϵ is returned. If not, the state $\delta(\sigma_n, \epsilon)$ is evaluated. At each level k of $\mathcal{S}$, the current state is $c = \delta^*(\sigma_n \dots \sigma_{n-k}, \epsilon)$. Let c_{rev} be the reversed label of the current candidate (i.e. if $c = \tau_1\tau_2 \dots \tau_m$, $c_{rev} = \tau_m\tau_{m-1} \dots \tau_1$). If *candidacy*$[c_{rev}]$ is True, c_{rev} is returned. If not the next iteration is processed until ω_{rev} is reached, which means that ω 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 $\mathcal{S}$ 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 $\mathcal{S}$ 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 probabilities $\mathcal{S}$ 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 Γ which

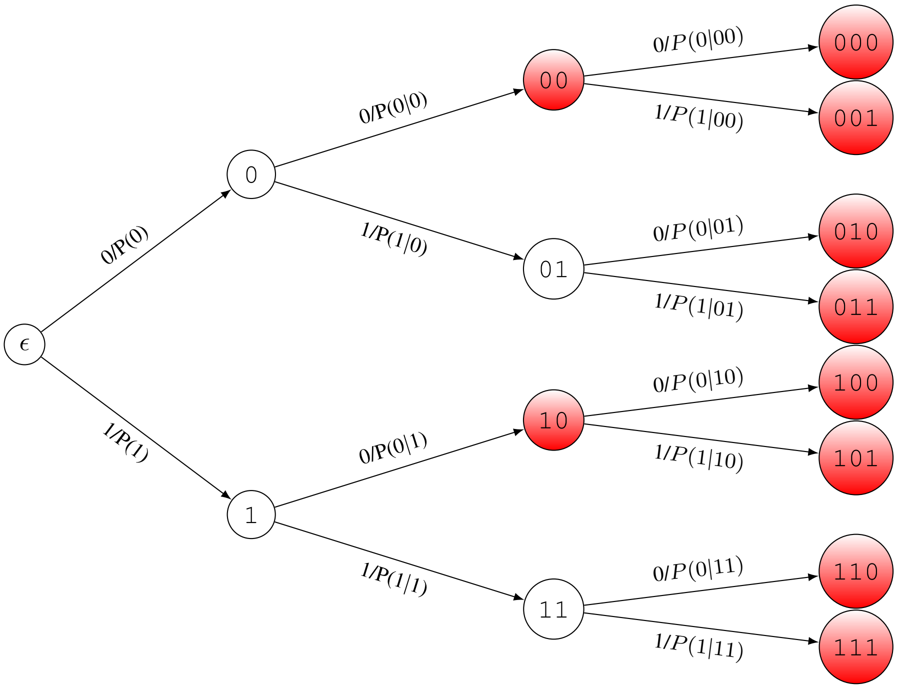

Figura 3.2: *Example of binary RTP with* $L = 3$.

contains states from $\mathcal{S}$ that are not fully tested for the synchronization word hypothesis during the current iteration. As ϵ 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 ϵ as their shortest valid suffix. . A list Θ is created and initialized empty. It receives the states from $\mathcal{S}$ which currently have passed the statistical tests. The statistical tests are implemented as either χ^2 or Kolmogorov-Smirnov tests for a given confidence level α. This is implemented as the auxiliary function *statisticalTest* which takes as inputs two states and a significance level α 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 Γ via dequeueing (as Γ 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 Γ 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 λ of *suffixes*$[c]$ goes through the statistical test with c in order to check (2.6) and p receives the result of *statisticalTest*(c, λ, α). 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 Θ . 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 Γ 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 Θ are queued at the end of Γ (using the auxiliary function queue) and then Θ is set to the empty set. This procedure is repeated until either the queue Γ is empty or if all the elements in Γ have labels longer than W. After one of these conditions is met, it stores Θ in the list Ω_{syn}, which contains all the elements that passed in all their statistical tests, meaning that they are synchronization words according to (2.6). Ω_{syn} is then returned as the final value.

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

```
procedure INITIALIZATION
    $\Gamma \leftarrow \{\epsilon \in \mathcal{S}\}$
    suffixes$[\epsilon] \leftarrow \{\delta(\sigma, \epsilon) \forall \sigma \in \Sigma\}$
    $V \leftarrow$ empty dictionary
    for $s \in \mathcal{S}$ 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($c, V, \mathcal{S}, \Gamma$, 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 Γ 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 Γ and in turn take their descendants, find their SVS and append them to their SVS *suffixes* dictionary entry. There

Algorithm 7 expand($c, V, \mathcal{S}, \Gamma$, 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($\mathcal{S}$, $d$, candidacy)
        if $\zeta = d$ then
            queue($\Gamma$, $\zeta$)
            for $t \in \{\delta(\sigma, \zeta) \forall \sigma \in \Sigma\}$ do
                $\tau \leftarrow$ shortestValidSuffix($\mathcal{S}$, $t$, candidacy)
                suffixes$[\tau] \leftarrow$ suffixes$[\tau] \cup t$
        else
            $V[\zeta] \leftarrow V[\zeta] \cup \{d\}$
```

are some caveats: for an element to be queued into Γ, it needs to be its own SVS, otherwise it means that there are shorter states that need to be checked first. Given an element d that is a descendant of c is not its own SVS (call it ζ), it is appended to the list $V[\zeta]$. This is done so if in a later iteration ζ (which should be in Γ) fails its test, d has the opportunity to check again if it became its own SVS so it might be queued into Γ.

First, a list Ψ 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 Γ. They will be queued if they are their own SVS. The dictionary V uses states of $\mathcal{S}$ 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 Γ 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 Ψ. 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 Ψ are their own SVS using the *shortestValidSuffix* function. This function will return a state ζ which is the SVS of d. If $d = \zeta$, d is queued at the end of Γ. After this, for each descendant t of d, t has its SVS τ found and *suffixes*$[\tau]$ has t appended to it, as now t has to be checked against τ.

When $\zeta \neq d$, $V[\zeta]$ has d appended to it. Later on, if ζ 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 Tri-Shift as it can be compared to CRISSiS in Section 2.5.2. All statistical tests in this section use the χ^2 test with $\alpha = 0.95$. The initial RTP with $L = 4$ is shown in Figure 3.3. We consider $W = 3$. The queue Γ is initialized with the root of $\mathcal{S}$. The dictionary *suffixes* is initialized with *suffixes*[ϵ] = {0, 1}. V is initialized as an empty dictionary, Θ is initialized as an empty list and all the states start with their candidacy set to True.

As Γ 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*[ϵ] = $\{0, 1\}$ and p is set to true. It first compares *statisticalTest*$(\epsilon, 0, \alpha)$. As the morphs are [0.6276, 0.3274] and [0.5615, 0.4385], the test fails, which means ϵ candidacy is set to False and the expansion algorithm is called.

The list Ψ 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 Ψ are their own shortest valid suffixes after ϵ 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.

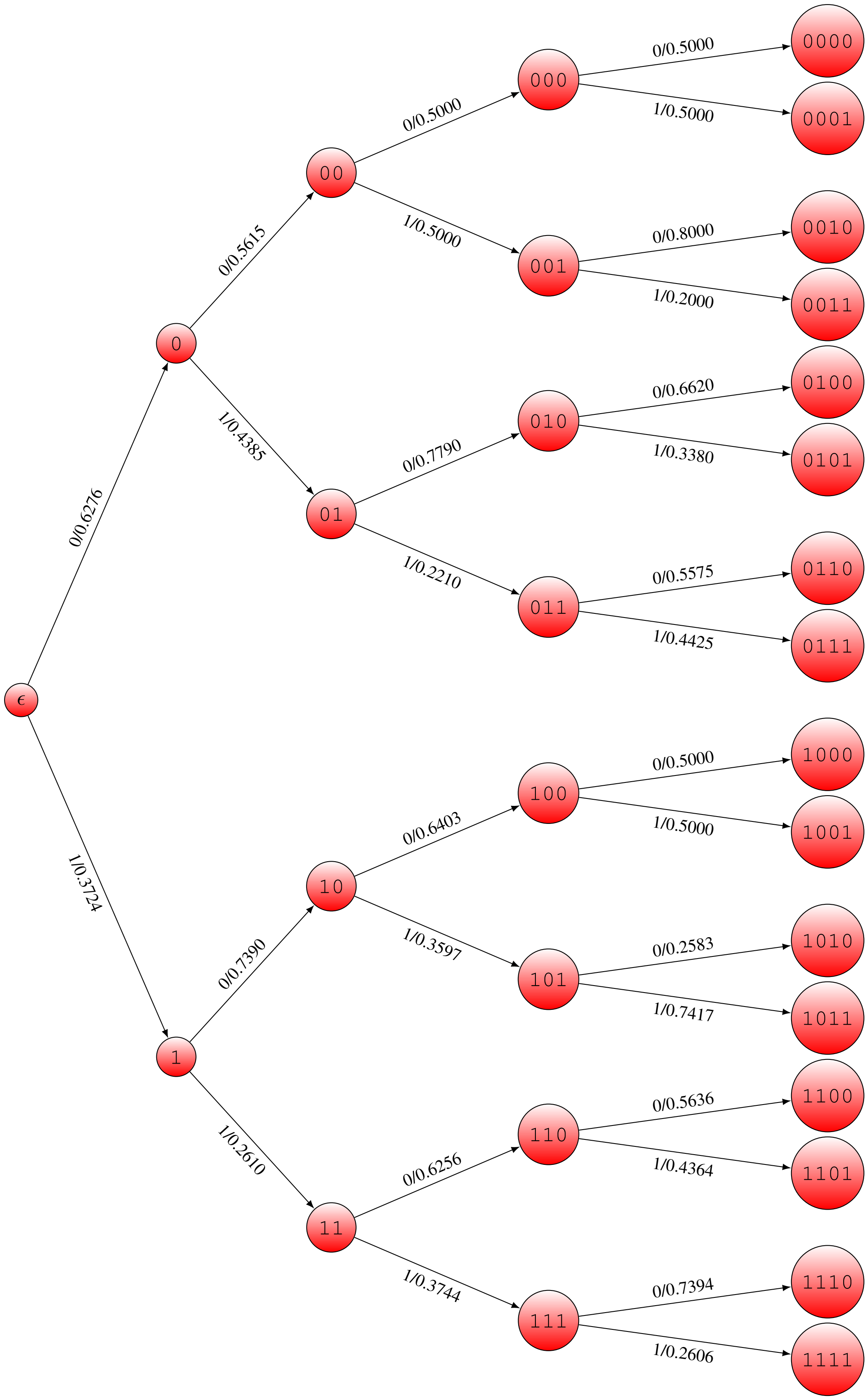

Figura 3.3: *Input Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example.*

At the beginning of the next iteration $\mathcal{S}$ 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 Ψ. 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 Γ. 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 Θ is once again appended to Γ 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 $\mathcal{S}$ is shown in Figure 3.5. Γ 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 Θ is appended in Γ and emptied.

The fourth iteration has $c = 00$, *suffixes*[c]= $\{000, 100\}$ and $mathcalS$ 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 Θ.

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 $mathcalS$ 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 Γ 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 Γ.

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 $mathcalS$ 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 Γ and $V[00] = \{100\}$. 101 is its own shortest valid suffix so it is queued into Γ 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 $mathcalS$ 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 Γ 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 $mathcalS$ 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 test. This keeps 00 candidacy as True and it is once again added to Θ. The rest of the elements in $\Gamma = \{101, 110, 111, 011\}$ have labels equal to than W so they are all skipped and the algorithm returns $\Theta = \{00\}$. This result is the same as the one found by CRISSiS.

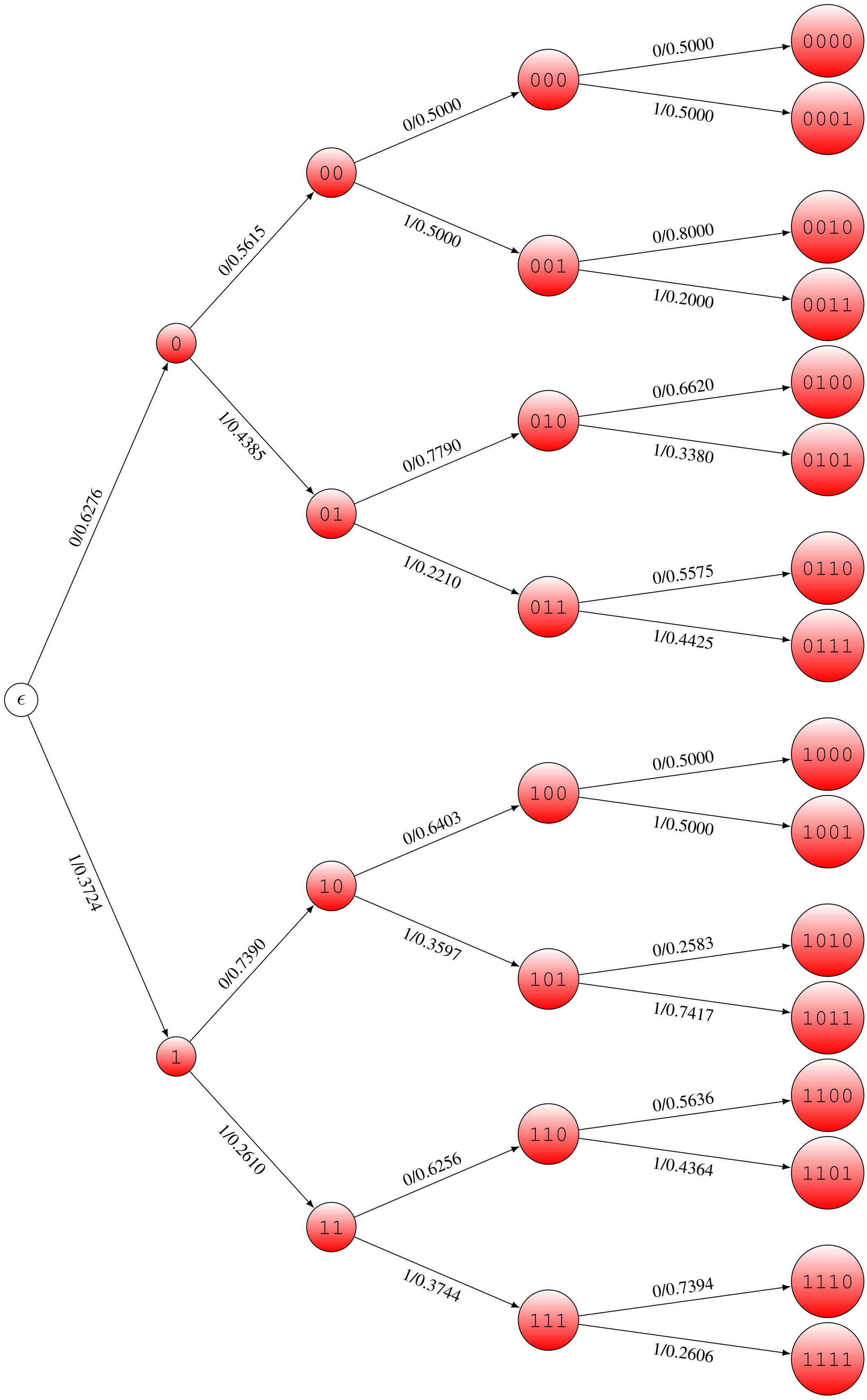

Figura 3.4: *Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example after the first iteration.*

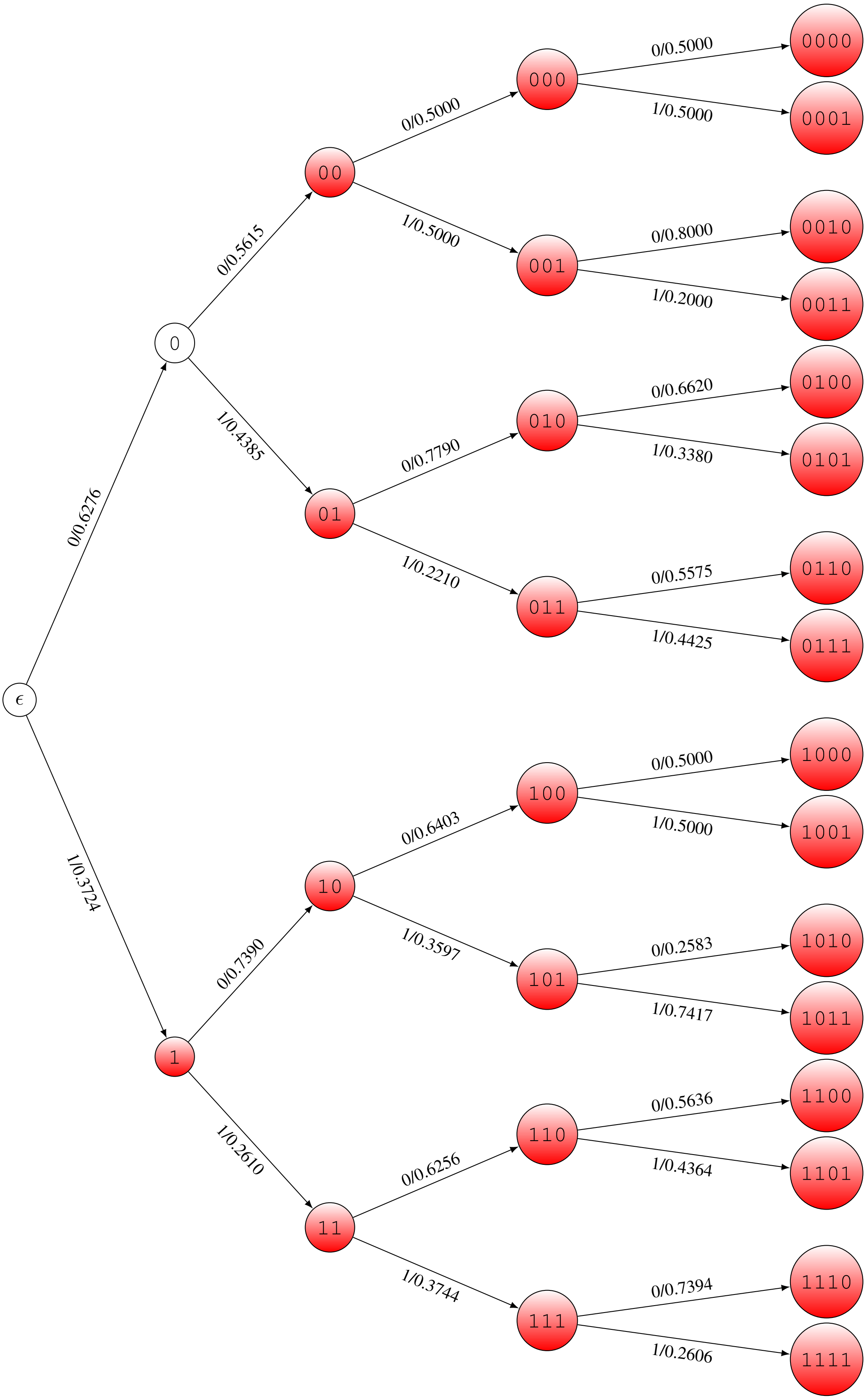

Figura 3.5: *Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example after the second iteration.*

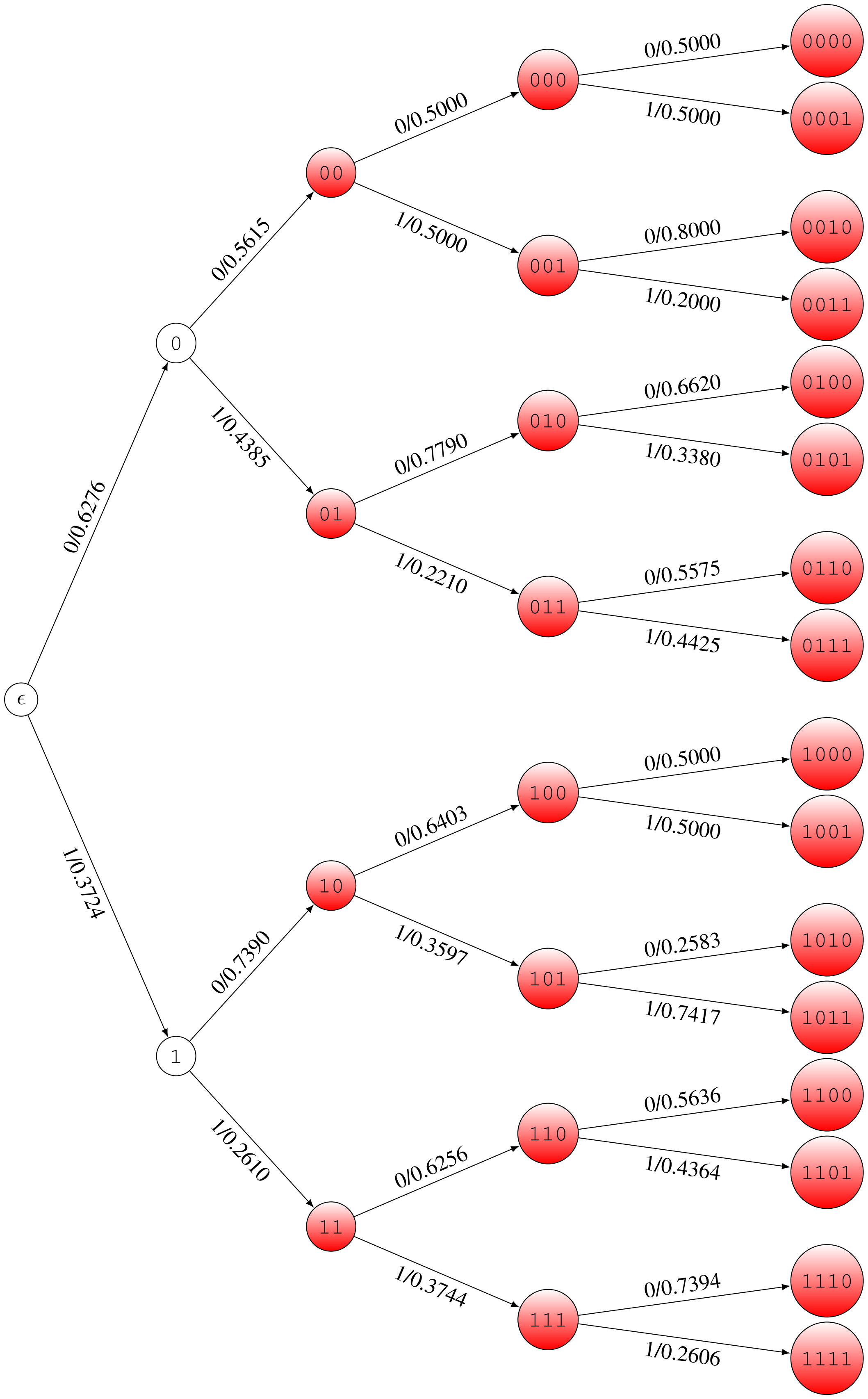

Figura 3.6: *Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example after the third and fourth iterations.*

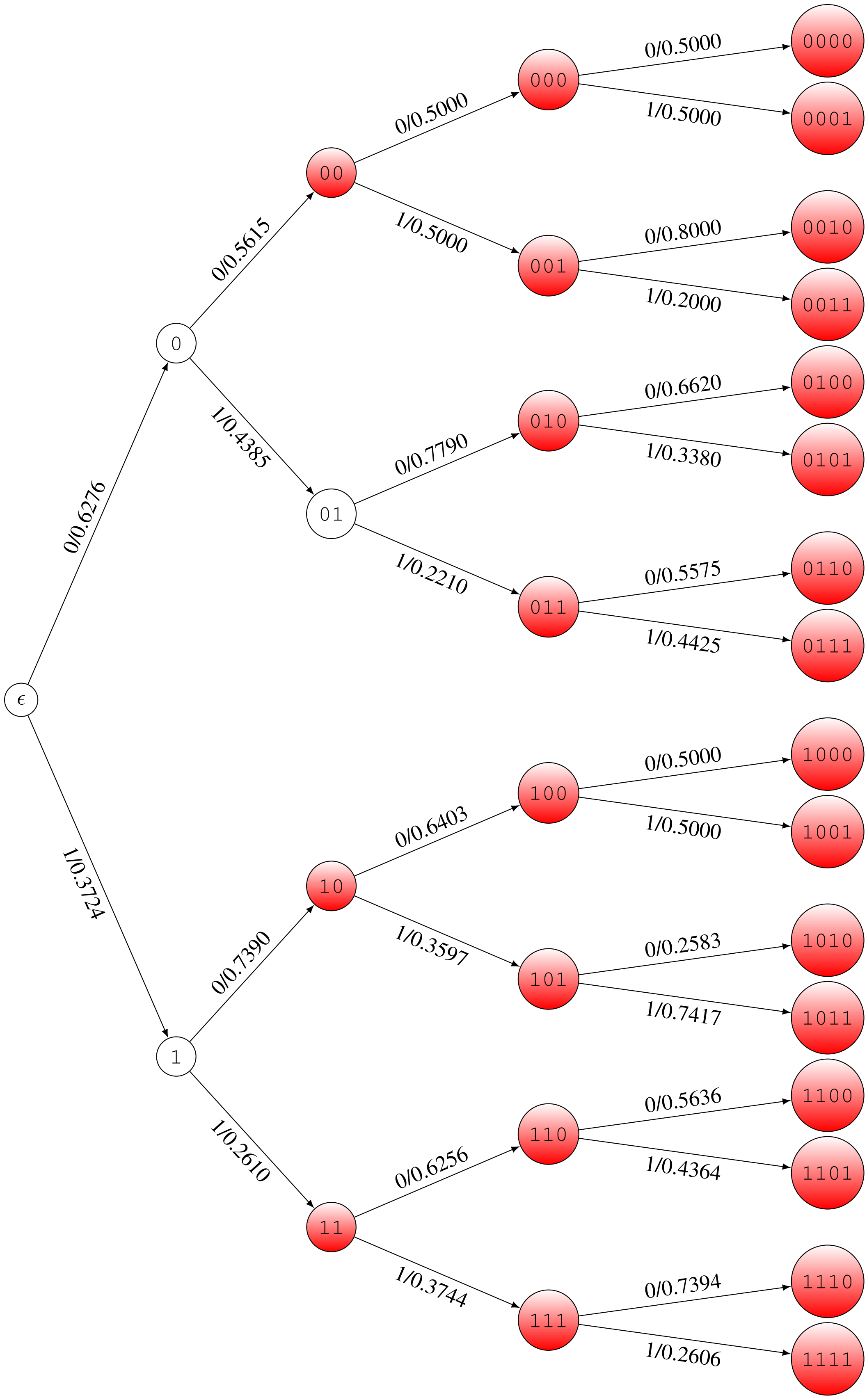

Figura 3.7: *Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example after the fifth iteration.*

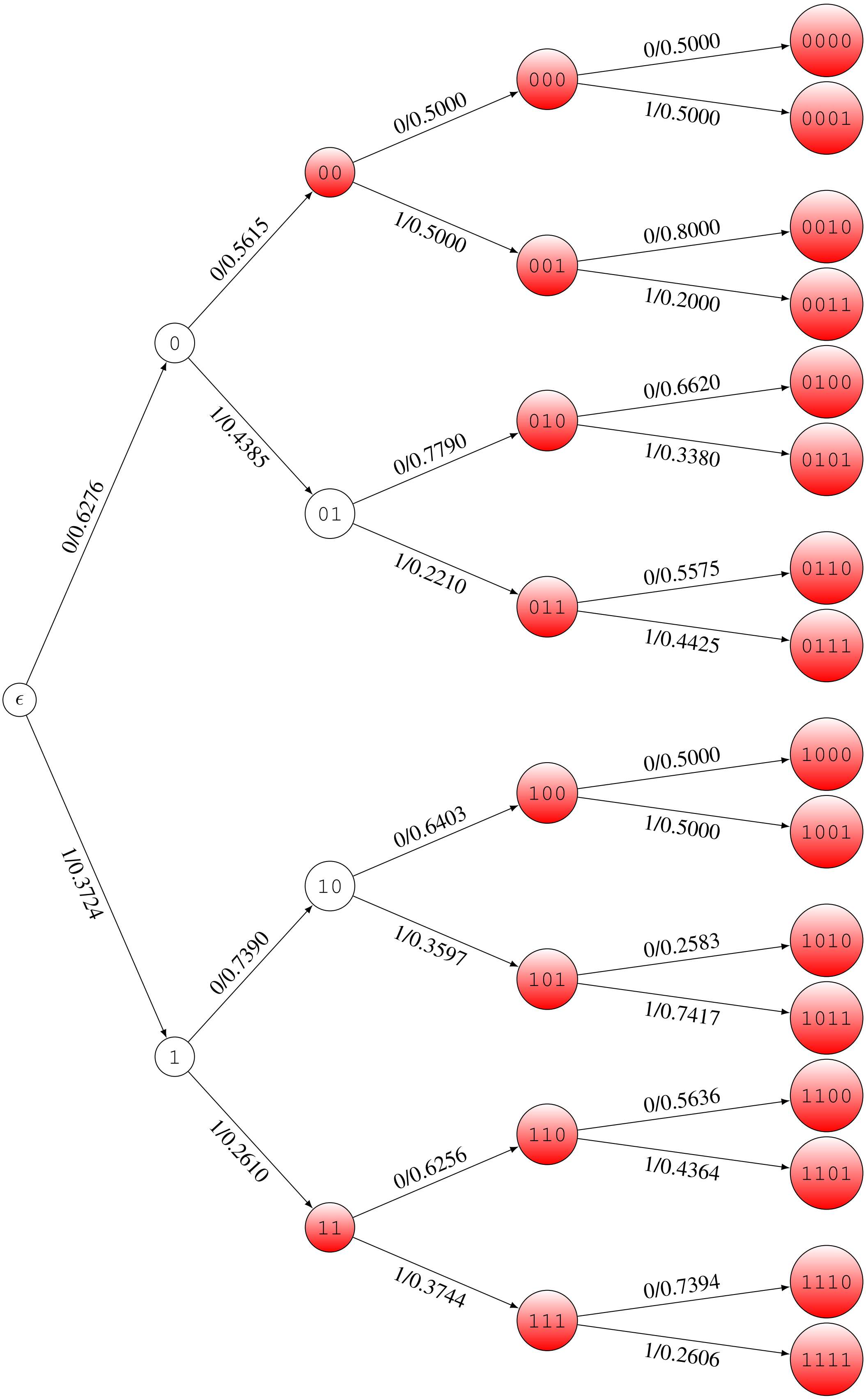

Figura 3.8: *Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example after the sixth iteration.*

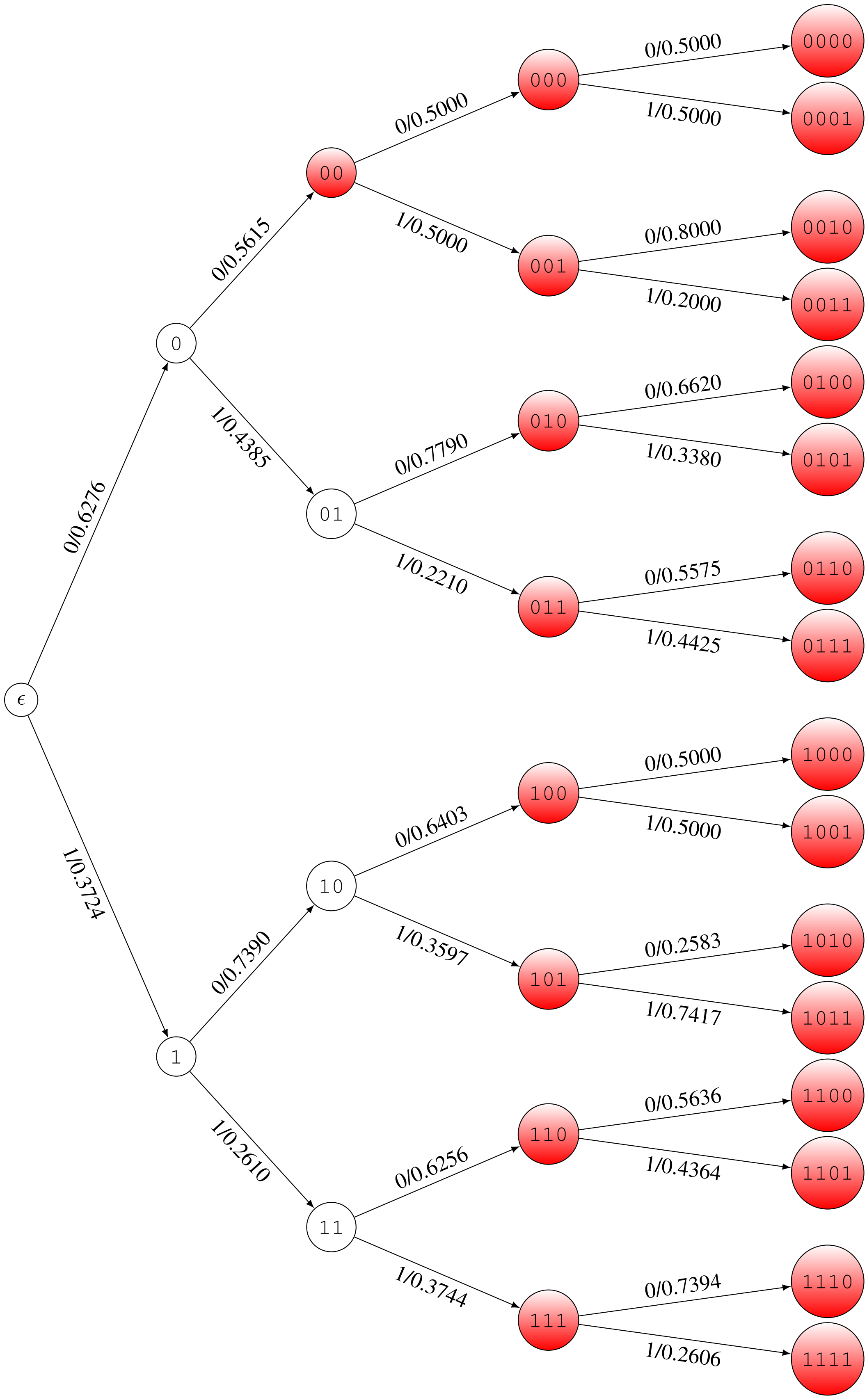

Figura 3.9: *Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example after the seventh iteration.*

3.2 PFSA Construction

In this section, we discuss the ALEPH algorithm that constructs a PFSA from the RTP $\mathcal{S}$. The first step is to transform $\mathcal{S}$ 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 will groups states in equivalence classes of states that have statistically similar morphs (checked by χ^2 or Kolmogorov-Smirnov for a given confidence level α 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 Ω criterion is used to connect the leaf states of the RTP $\mathcal{S}$ 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.

Ω Connection

For each state p in level $L+1$ of $\mathcal{S}$, this criterion checks via statistical test if p has similar morph to any of the synchronization words states. If it is 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 $\mathcal{S}$, the desired last level L and a list of synchronization words Ω_{syn}.

3.2.2 ALEPH Algorithm

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

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

```
procedure CONNECT
    $\Psi \leftarrow \{p \in \mathcal{S}$ 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 Termination

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

3.2.3 Step 2: State Reduction

After the RTP connection process is finished a technique is used to discard some branches of $\mathcal{S}$ in order to obtain fewer starting states in the next two steps, which implies in a smaller complexity. Each state $q' \in \mathcal{S}$ 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 ω as suffix. In the affirmative case, the outgoing edge of q' is reassigned to ω. 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 Ω_{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 $\mathcal{S}$ 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 $\tilde{\mathrm{Q}}$ is created containing the descendants of the head states in each of the initial equivalence classes.

For each element q of $\tilde{\mathrm{Q}}$, 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 $\mathcal{S}$ is present in one equivalence class of the partition $\mathcal{P}$.

Step 4: Graph Minimization

Once every state of $\mathcal{S}$ 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 final result is a PFSA 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 α.

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 [7], CRISSiS complexity depends on the number of states of the original system which (seen in Section 2.5.2), 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. 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 $O(N)$.

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

```
procedure
    ## Step 1: RTP Leaf State Connection
    $\mathcal{S} \leftarrow$ omegaConnection($\mathcal{S}, L$)
    ## 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)
    $G \leftarrow$ GraphMin($\mathcal{P}$)
    return $G$
```

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 ϵ. 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 $\mathcal{S}$, 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 Ω 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 Ω_{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 $\mathcal{S}$ 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 [15], 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 Σ 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 metrics explained in Section 4.1 and the comparison results are explained in Section 4.2.

4.1 Evaluation Metrics

This section presents the three metrics that will be 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 two other metrics, the Kullback-Leibler Divergence and Φ, compare sequences generated by the models with the original one and estimate how similar they are. The lower these metrics are, 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 Σ. Its entropy rate is defined as [18]:

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

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 [18]. As it is not feasible to compute (4.1) up to infinity when the distribution is know only up to L_{max}, we use the ℓ-order conditional entropy 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_ℓ given the previous ℓ samples. The comparison of h_ℓ 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 Σ. 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 ℓ and $P_1(\omega)$ and $P_2(\omega)$ are the probabilities of occurrence of ω in S_1 and S_2 respectively. For a given ℓ we take the ℓ-order Kullback-Leibler Divergence as:

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

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.

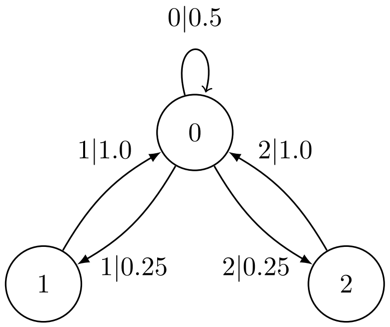

Figura 4.1: *A PFSA of a Ternary Even-Shift.*

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 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 χ^2 test with $\alpha = 0.95$ and for the synchronization words, two values of α (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 α (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 of 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 $D = 8$ 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) and 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

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

	α	
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

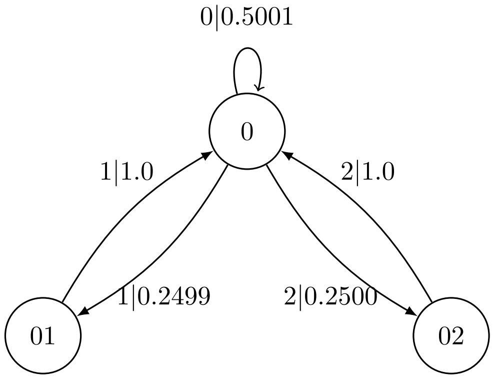

Figura 4.2: *PFSA of a Ternary Even-Shift generated by the ALEPH algorithm and by CRISSiS.*

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 2.5.2 and a PFSA that represents it is shown in Figure 2.6. The synchronization words found by the algorithm are shown in Table 4.3 and 00 appeared, as expected (see Section 2.5.2), 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

Tabela 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}	$2.7 \cdot 10^{-3}$	$4.16 \cdot 10^{-5}$	$9.55 \cdot 10^{-5}$

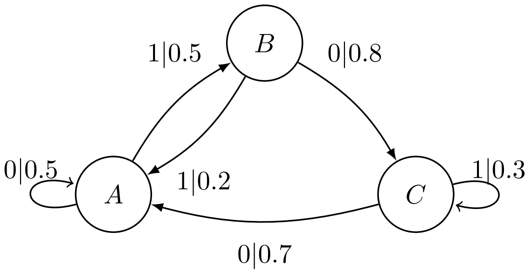

Figura 4.3: *The Tri-Shift PFSA.*

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

	α	
W	0.95	0.99
2	None	None
3	00	00
4	00	00
5	$00, 0110$	$00, 0110$
6	$00, 0110$	$00, 0110$

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 4.4. 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 8-Markov Machine are able to capture the system memory.

4.2.3 A Six-State PFSA

Figure 4.5 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.

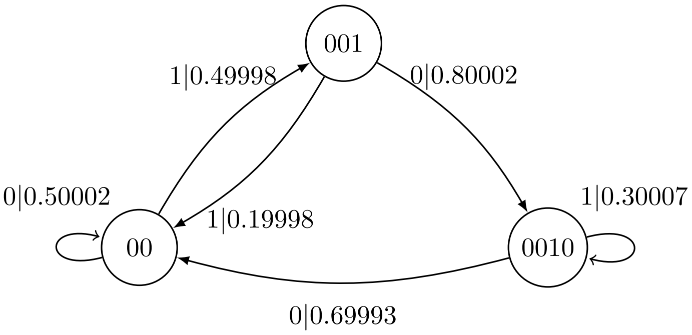

Figura 4.4: *The Tri-Shift PFSA generated by our algorithm and by CRISSiS.*

Tabela 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}	$4.1 \cdot 10^{-3}$	$1.65 \cdot 10^{-3}$	$1.16 \cdot 10^{-3}$

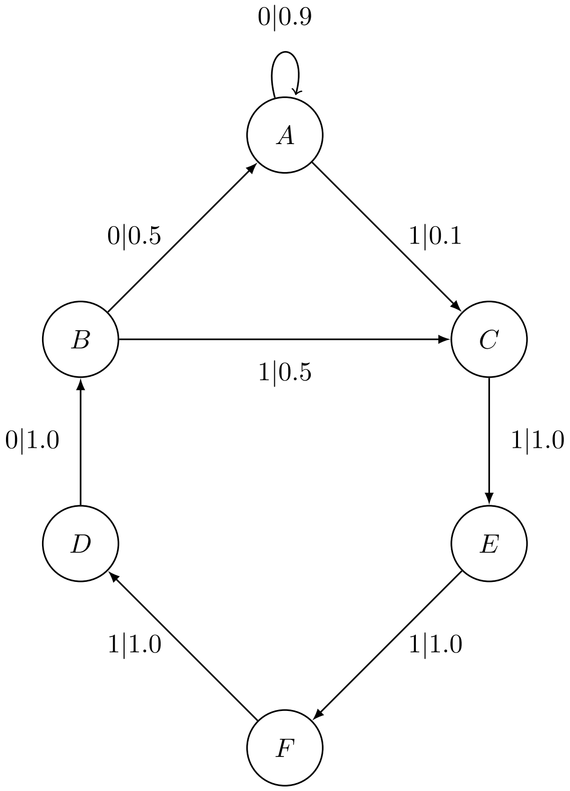

Figura 4.5: *A Six-State PFSA.*

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

	α	
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

Tabela 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	$4.0499 \cdot 10^{-6}$	$5.6969 \cdot 10^{-5}$

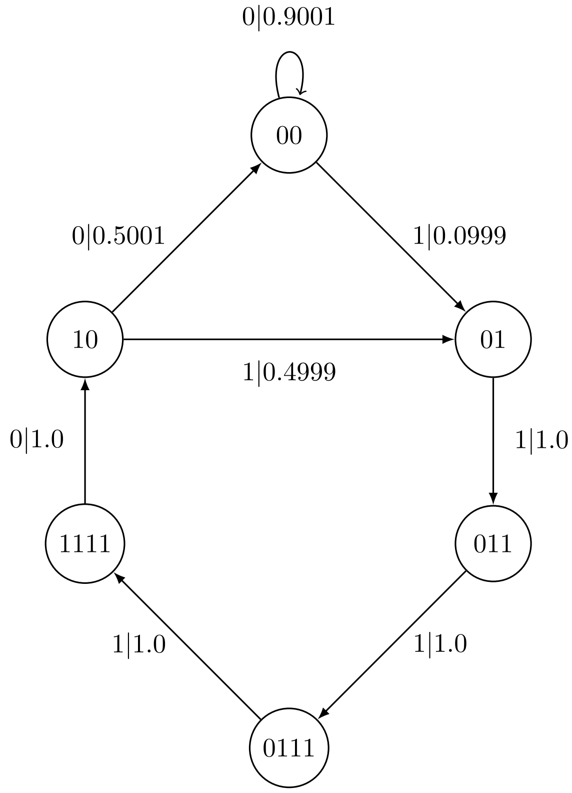

Figura 4.6: *The Recovered Six-State PFSA by our algorithm.*

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 4.6. 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 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.

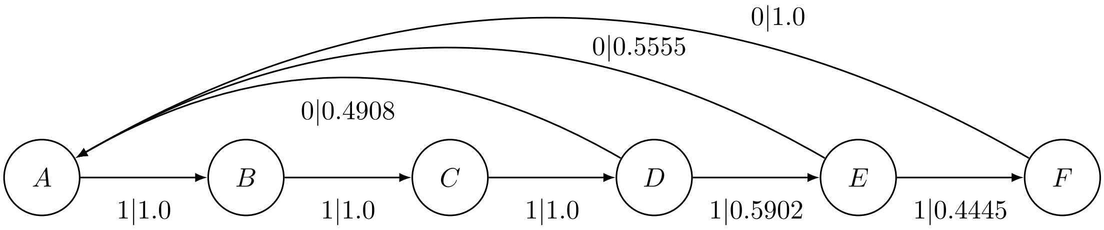

Figura 4.7: *The Maximum Entropy (3,5)-Constrained Code PFSA.*

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

	α	
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

4.2.4 Maximum Entropy (d, k)-Constrained Code

As seen in [20], 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 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 4.7. 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 4.8). 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.

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

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

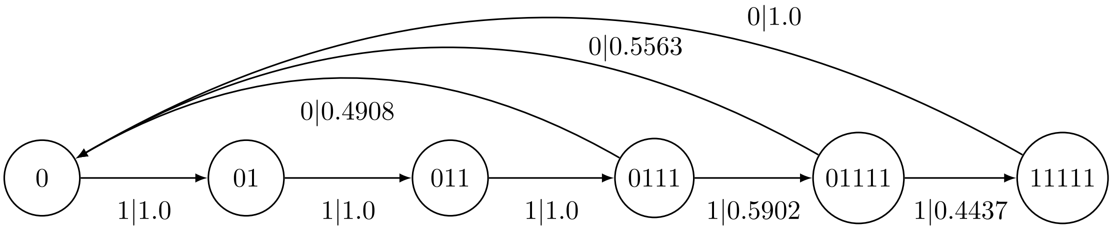

Figura 4.8: *The Maximum Entropy (3,5)-Constrained Code PFSA recovered by ALEPH algorithm and by CRISSiS.*

CHAPTER 5

APPLICATIONS

TODO

5.0.1 Logistic Map

The next two examples show how the algorithms fare when modeling a system whose PFSA is unknown or non-existent. A sequence from these systems will be analyzed and a PFSA model will be created and its output will be compared to the original sequence to see how well this Markovian model approximates a dynamic system which might not even be Markovian.

The first of these examples is the Logistic Map, a symbolic dynamic system whose outputs is given by the difference equation [19]:

$$x_{k+1} \triangleq rx_k(1 - x_k), \tag{5.1}$$

which shows chaotic behavior when the *r* parameter is approximately 3.57. As in [19], the initial x is set to 0.5 and *r* = 3.75. A sequence of length 10^{-7} was generated from this equation and then it was quantized with a ternary alphabet: values x$_k$ ≤ 0.67 were mapped to 0; when 0.67 < x$_k$ ≤ 0.79, it was mapped to 1 and when x$_k$ > 0.79 it was mapped to 2. A part of that sequence and the specified threshold are shown in Figure 5.1.

From this ternary sequence, the three algorithms were applied in order to obtain a Markovian model for the logistic map. As seen in Table 5.1, two sets of synchronization words were found for the sequence, one for each of the confidence levels used in the algorithm. For $\alpha = 0.95$, the synchronization words are 2, 00, 01, 10 and 11. On the other hand, for $\alpha = 0.99$, the synchronization

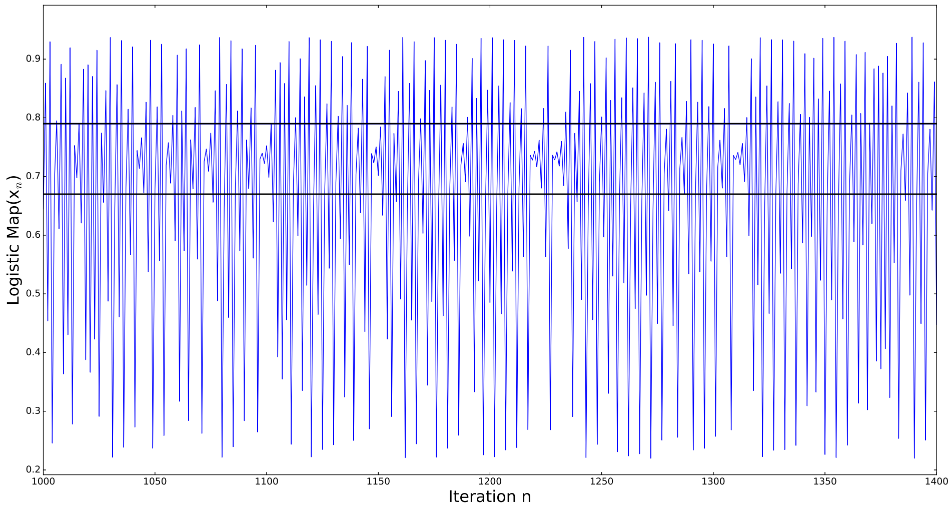

Figura 5.1: *Part of the Logistic Map generated by Equation 5.1 with x_0 = 0.5 and r = 3.75.*

Tabela 5.1: *Synchronization Words for the Logistic Map Ternary Sequence.*

	α	
W	0.95	0.99
2	2	2
3	2, 00, 01, 10, 11	2, 00, 01, 10
4	2, 00, 01, 10, 11	2, 00, 01, 10, 111
5	2, 00, 01, 10, 11	2, 00, 01, 10, 111
6	2, 00, 01, 10, 11	2, 00, 01, 10, 111
7	2, 00, 01, 10, 11	2, 00, 01, 10, 111

words are 2, 00, 01, 10 and 111. The higher value of α made 11 be discarded as synchronization word candidate and allowed 111 to be tested. With the lower value, 11 was never discarded and 111 could not achieve candidate status.

5.0.2 Gilbert-Elliot Channel

The Gilbert-Elliot Channel (GEC) is used to model digital communication channels that suffer with burst errors, i.e. a channel that usually has low probability of error but that has moments where many sequential errors occur. As described in [21], Figure 5.2 is the GEC model. It operates in two states, *0* (the "good channel") and *1* (the "bad channel"). While it is in *0*, it works as a Binary Symmetric Channel (BSC) with error probability of p_0, which is usually very small, indicating a state in which the channel does not produce too many errors. When it is in state *1*, it is a BSC with error

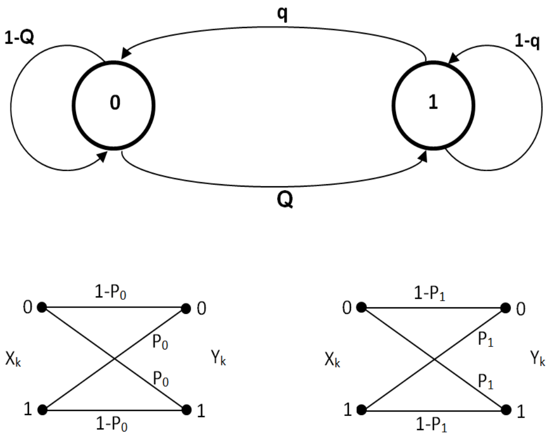

Figura 5.2: *The Gilbert-Elliott Channel.*

probability p_1 which is higher than p_0, indicating a state where it is more probable for an error to occur. When in state *0*, it has a probability q of transitioning to state *1* and $1-q$ to stay in *0*. Similarly, when in *1*, it transitions to *0* with probability q and stays with probability $1-q$. This indicates that there is a chance from going to one situation to the other.

Other important parameters of the GEC that need to be evaluated are its memory μ and the Bit Error Rate (BER), which is a percentage of errors in the transmission. The memory μ is defined as:

$$\mu = 1 - q - Q. \tag{5.2}$$

which reduces to a memoryless BSC when $\mu = 0$. This parameter is called memory because, as seen in [21], the GEC's autocorrelation function is:

$$R_{GEC}[m] = (\text{BER})^2 + \frac{Qq(p_1 - p_0)^2}{(q+Q)^2}(1 - q - Q)^m, \tag{5.3}$$

which, without getting into much detail, shows that μ influences how a symbol is related to another one that is m symbols apart. The BER is given by:

$$\text{BER} = \frac{q}{q+Q}p_0 + \frac{Q}{q+Q}p_1. \tag{5.4}$$

The GEC can be designed to obtain specific values of μ and BER and then it is possible to compare how close to the design parameters the generated PFSA are able to get in order to evaluate their performance.

A binary sequence going through this channel would be output in instant k the following way:

$$y_k = x_k \oplus z_k, \tag{5.5}$$

in which x_k is the input symbol at instant k, z_k is the error symbol at instant k and $\oplus$ is binary addition operation. When z_k is 0, y_k will be equal to x_k, which means that no error occurred. On the other hand, when it 1, y_k will be $x_k \oplus 1 = \neg x_k$, indicating the occurrence of an error. The symbol z_k has a probability p_e of being 1 and $1 - p_e$ of being 0 and p_e is equal to p_0 if the channel is in state *0* and equal to p_1 when it is in state *1*. Following this rule, an error sequence z can be generated to model how the channel and how it affects an input sequence. An error sequence of length 10^7 is generated and used as input to the algorithms. The GEC is strictly not Markovian and this is an example to show how the algorithms fare in modeling such a system in a Markovian fashion.

CHAPTER 6
TODO

TODO

APÊNDICE A

TODO

TODO

Apêndice B

TODO

About the Author

The author was born in Brasília, Brasil, on the 6^{th} of August of 1991. He graduated in Electronic Engineering in the Federal University of Technology of Paraná (UTFPR) in Curitiba, Brazil, in 2014. His research interests include Information Theory, Error Correcting Codes, Data Science, Cryptography, Digital Communications and Digital Signal Processing.

Address: Endereço

e-mail: `daniel.k.br@ieee.org`

Esta dissertação foi diagramada usando LaTeX 2_ε[1] pelo autor.

[1] LaTeX 2_ε é uma extensão do LaTeX. LaTeX é uma coleção de macros criadas por Leslie Lamport para o sistema TeX, que foi desenvolvido por Donald E. Knuth. TeX é uma marca registrada da Sociedade Americana de Matemática ($\mathcal{AMS}$). O estilo usado na formatação desta dissertação foi escrito por Dinesh Das, Universidade do Texas. Modificado por Renato José de Sobral Cintra (2001) e por Andrei Leite Wanderley (2005), ambos da Universidade Federal de Pernambuco. Sua úlltima modificação ocorreu em 2010 realizada por José Sampaio de Lemos Neto, também da Universidade Federal de Pernambuco.

BIBLIOGRAFIA

[1] 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.

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

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

[4] S. Strogatz, “Nonlinear dynamics and chaos: With applications to physics, biology, chemistry and engineering,” 2001.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

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

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

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

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

[19] 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.

[20] 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.

[21] M. Mushkin and I. Bar-David, “Capacity and coding for the gilbert-elliot channels,” *IEEE Transactions on Information Theory*, vol. 35, no. 6, pp. 1277–1290, November 1989.