UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE TECNOLOGIA E GEOCIÊNCIAS
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

DANIEL KUDLOWIEZ FRANCH

DYNAMIC SYSTEM MODELING AND FAULT DETECTION WITH PROBABILISTIC FINITE STATE AUTOMATA

Recife
2017

DANIEL KUDLOWIEZ FRANCH

DYNAMIC SYSTEM MODELING AND FAULT DETECTION 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.

Co-orientador: 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

TODO

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

ABSTRACT

TODO

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

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

CAPÍTULO 1

INTRODUCTION

TODO.

CAPÍTULO 2

PRELIMINARIES ON GRAPHS AND PROBABILISTIC FINITE STATE AUTOMATA

IN this chapter we revise concepts from graphs and Probabilistic Finite State Automata (PFSA) 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's and Hopcroft's. Finally, two algorithms to model dynamic systems with PFSA are presented in the last session, 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 sequence 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 sequences u and $v \in \Sigma^*$ can be concatenated to form a sequence uv. For example, using a binary alphabet $\Sigma = \{0, 1\}$, the sequences u = 1010 and v = 111, they can be concatenated to form 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.

A sequence $v \in \Sigma^*$ is called a suffix of a sequence $w \in \Sigma^*$ ($|w| > |v|$) 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 coming out of state q and pointing to state q'. The arrow is labeled with the symbol σ. This realization of the transition function can be called the outgoing edge from q to q' with symbol σ. Figure 2.1 shows an example of 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, representing $\delta(A, 1) = B$.

It is possible to extend the transition function so it accepts words and not just symbols. Given $\omega \in \Sigma^n$ and $\omega = \sigma_1\sigma_2 \ldots \sigma_n$ with $\sigma_m \in \Sigma$ for $m = 1 \ldots n$. Also, 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 states $q_1, q_2 \in Q$ it is said there is a path between q_1 and q_2 and that ω is generated by G. Calling the graph of Figure 2.1 as G, the following is an example of a path: start at state A and go to A, A, B, C, B, A. This generates the string s = 001110.

Definition 2.2 – Follower Set

The follower set of a state $q \in Q$ is defined as the set of all possible 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 $q \in Q$: □

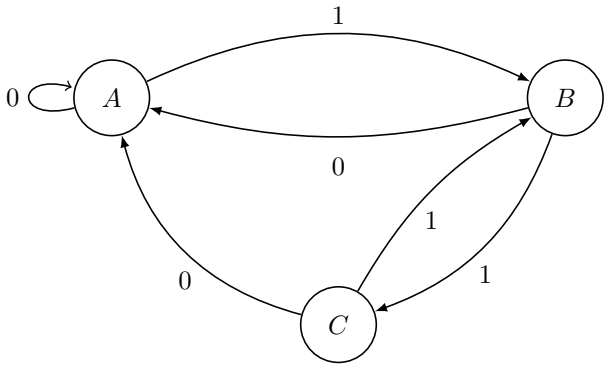

Figura 2.1: *A graph with Q = {A, B, C} and* $\Sigma = \{0, 1\}$.

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

A word $\omega \in \Sigma^*$ is called a synchronizing word of G if starting from any state $q \in Q$ and following the path defined by ω the same state $q_{syn} \in Q$ is reached. That is, if ω is a synchronizing word, $\delta^*(q, \omega) = q_{syn}, \forall q \in Q$. q_{syn} is called a synchronizing state for ω. In the example of Figure 2.1, 0 is a synchronizing word that synchronizes to state A.

2.3 Graph Minimization

There are times when 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 in a computer with less memory consumption. And for all graphs that have certain language, there is always one with the least number of states, which is called the 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. □

It is elementary to see that if a graph has two distinct states with the same follower set, a new graph can be obtained by excluding one of them and the graph will still be able to generate the same language. When all the states have distinct follower sets, none of them can be excluded without affecting the generated language.

Given some graph G there are two main algorithms used to obtain a minimal graph from it, Moore's and Hopcroft's. 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 $P\cap Q$, $P\in \mathcal{P}$ and $Q\in \mathcal{Q}$. The notation is extended for multiple sets as $\mathcal{U} = \mathcal{P}_1 \wedge \mathcal{P}_2 \wedge \ldots \wedge \mathcal{P}_n$. 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 partitions $\mathcal{P}$ and $\mathcal{P}'$ of disjoint sets E and E', the partition of set $E \cup E'$ whose restriction to E and E' are $\mathcal{P}$ and $\mathcal{P}'$ is denoted by $\mathcal{P} \vee \mathcal{P}'$. It is possible to write $\mathcal{P} = \vee_{P\vee\mathcal{P}}\{P\}$.

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 quotient graph $G/\equiv$ obtained by taking for Q the set of Nerode equivalence classes. The minimal graph is unique and it accepts the same language as the original graph.

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

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

the partition composed of two non-empty subsets:

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

and

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

The pair (P, σ) is called a splitter. Observe that (P, σ)|R = R if either $\delta(R,\sigma) \subset P$ or $\delta(R,\sigma) \cap P = \emptyset$ and (P, σ)|R is composed of two classes if both $\delta(R,\sigma) \cap P \neq \emptyset$ and $\delta(R,\sigma) \cap P^c \neq \emptyset$ or equivalently if $\delta(R,\sigma) \not\subset P$ and $\delta(R,\sigma) \not\subset P^c$. If (P, σ)|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 ∈ $\mathcal{P}$ and σ ∈ Σ, splits a class in $\mathcal{P}$, that is such that (P,σ)|R = R for all P, R ∈ $\mathcal{P}$ and σ ∈ Σ. □

Lemma 2.1

Let P be a set of states and $\mathcal{P} = P_1, P_2$ a partition of P. For any symbol σ and for any set of states R, one has: □

$$(P,\sigma)|R \wedge (P_1,\sigma)|R = (P,\sigma)|R \wedge (P_2,\sigma)|R = (P_1,\sigma)|R \wedge (P_2,\sigma)|R,$$

and consequently

$$(P,\sigma)|R \geqslant (P_1,\sigma)|R \wedge (P_2,\sigma)|R,$$

$$(P_1,\sigma)|R \geqslant (P,\sigma)|R \wedge (P_2,\sigma)|R.$$

2.3.1 Moore's Algorithm

Algorithm 1 Moore(G)

```
1: P ← InitialPartition(G)
2: repeat
3:     P' ← P
4:     for all σ ∈ Σ do
5:         P_σ ← ⋀_{P∈P} (P,σ)|Q
6:     P ← P ∧ ⋀_{σ∈Σ} P_σ
7: until P = P'
```

Given a graph $G = (Q, \Sigma, \delta)$, the set $L_q^{(h)}$ is defined as:

$$L_q^{(h)}(G) = \{w \in \Sigma^* || w| \leq h \, qw \in G\}.$$

The Moore equivalence of order *h* (denoted by $\equiv_h$) is defined by:

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

The depth of Moore's algorithm on a graph *G* is the integer *h* such that the Moore equivalence $\equiv_h$ becomes equal to the Nerode equivalence $\equiv$ and it is dependent only on the graph's language. 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 ∈ Q and h ≥ 0, one has

$$p \equiv_{h+1} q \Longleftrightarrow p \equiv_h qandp \cdot \sigma \equiv_h q \cdot \sigma forall\sigma \in \Sigma.$$

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

Proposition 2.3

For h ≥ 0, one has

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

This previous computation is done 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 the deterministic case, it is done by grouping together states in *Q* which have outgoing edges with the same labels, but another criterion will be used in the probabilistic case (Section 2.4.1).

Moore's 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(lkn)*, where *l* is the total number of refinement steps needed to compute the Nerode equivalence.

Algorithm 2 Hopcroft(G)

```
1: P ← InitialPartition(G)
2: W ← ∅
3: for all σ ∈ Σ do
4:     Append((min(F, F^c,σ),W)
5:     while W ≠ ∅ do
6:         (W,σ) ← TakeSome(W)
7:         for each P∈ P which is split by (W, σ) do
8:             P', P'' ← (W, σ)|P Replace P by P' and P'' in P
9:             for all τ ∈ Σ do
10:                if (P, τ) ∈ W then
11:                    Replace (P, τ) by (P', τ) and (P", τ) in W
12:                else
13:                    Append((min(P', P",τ),W)
```

2.3.2 Hopcroft's Algorithm

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'',*τ). Otherwise, only (*min(P',P''),*τ) 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 $O(|\Sigma| n \log n)$.

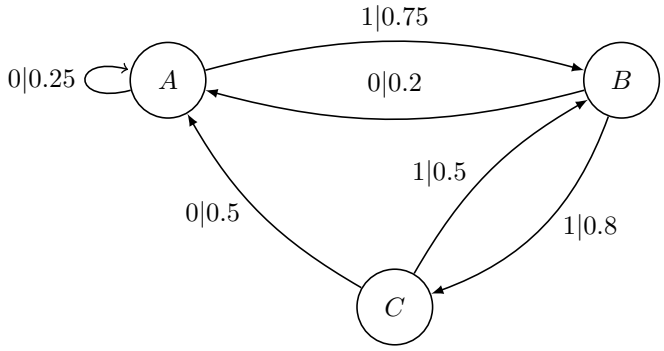

Figura 2.2: *A probabilistic version of the graph of Figure 2.1.*

2.4 Probabilistic Finite State Automata

Definition 2.6 – Probabilistic Finite State Automata

A Probabilistic Finite State Automaton (PFSA) P 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 $q \in Q$, $\sum_{\sigma \in \Sigma} \pi(q, \sigma) = 1$ which defines a probability distribution associated with each state of G. □

Definition 2.7 – Morph

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

A PFSA can be drawn with its graph with each outgoing edge labeled not only with a symbol, but the probability $\pi(q, \sigma)$ associated with it. An example of a PFSA P is shown in Figure 2.2. for which $Q = \{A, B, C\}$, $\Sigma = \{0, 1\}$. It is the same graph from Figure 2.1 but now probabilities have been associated to its edges to create a PFSA.

Given a PFSA $P = \{Q, \Sigma, \delta, \pi\}$, there is a probability associated with each word $\omega \in \Sigma^*$ that can be generated from its graph $G = \{Q, \Sigma, \delta\}$. From Figure 2.2, 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 seen in [**?**]:

Definition 2.8 – PFSA Synchronization Word

For a state $q \in Q$, w is a synchronization word if, $\forall u \in \Sigma^*$ and $\forall v \in \Sigma^*$:

$$\Pr(u|w) = \Pr(u|vw). \tag{2.1}$$

□

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 (2.1) for all $u \in \Sigma^*$ and for all $v \in \Sigma^*$ as there are an infinite number of sequences.

A solution uses the *d*-th order derived frequency, which is the probability using *u* and *v* from Σ^d, $d \in \mathbb{Z}$, instead of taking them from Σ^*. Calling $\Pr_d(\omega)$ the d-th order derived frequency of ω, a statistical test (such as the Chi-Squared or Kolmogorov-Smirnov) with significance level α has to be performed with the following null hypothesis for *w* being a synchronization word:

$$\Pr_d(w) = \Pr_d(uw), \forall u \in \cup_{i=1}^{L_1}\Sigma^i, \forall d = 1, 2, \ldots, L_2, \tag{2.2}$$

where L_1 and L_2 are precision parameters. This means that the statistical test compares the probabilities of words *w* with length from 0 to L_2 with the probabilities of words *uw*, where *u* is a prefix of *w* with lengths from 0 to L_1. This limits the number of tests to be realized.

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 came 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 Reduction algorithm (such as Moore's (Algorithm 1) or Hopcroft's (Algorithm 2) on a PFSA's graph, the following criterion will be used to create the initial partition:

Definition 2.9

Given a PFSA G = (Q, $\Sigma, \delta, \mathcal{V}$), two states p, q ∈ Q will be grouped together in the initial partition if their morphs are equivalent via a statistical test, i.e. $\mathcal{V}(p) = \mathcal{V}(q)$.

2.5 Consolidated Algorithms

In this section, other algorithms that achieve the same goal as the current work are described. In later sections, it will be pointed out how the proposed algorithm performs better than the ones detailed in this section.

Tabela 2.1: *Sequence s subsequence probabilities.*

L = 1	Prob.	L = 2	Prob.	L = 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

2.5.1 D-Markov Machines

A D-Markov machine is a PFSA that generates symbols that depend only on the history of at most *D* symbols in the sequence, in which *D* is the machine's *depth*. It is equivalent to stochastic process where the probability of a symbol depends only on the last *D* symbols:

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

To construct a D-Markov Machine, first all symbol blocks of length *D* of a given sequence *S* are taken as the states in the set *Q* and their transition probabilities can be computed by frequency counting. The transition from a state q with symbol σ will have probability:

$$P(\sigma|q) = \frac{P(q\cdot\sigma)}{P(q)}, \tag{2.3}$$

and, considering that $q = \tau \cdot q'$, in which $\tau \in \Sigma$ and $q' \in \Sigma^{D-1}$, this transition will go to state $p = q' \cdot \sigma$, i.e. $\delta(q,\sigma) = p = q'\sigma$.

For example, using the following sequence *S* over a binary alphabet: 10000000111101000100011 11010100001011000110101011111010000111010011111011100011110100010110011001000. To build a 2-Markov Machine, the states will be 00, 01, 10 and 11. Table 2.1 shows the frequency-counting probability of sub-sequences up to length 3. Using this table and Equation 2.3, the D-Markov machine shown in Figure 2.3 is built.

2.5.2 CRISSiS

The Compression via Recursive Identification of Self-Similar Semantics (CRISSiS) algorithm was presented in [ref]. It starts with a stationary symbol sequence *X* of length N which should

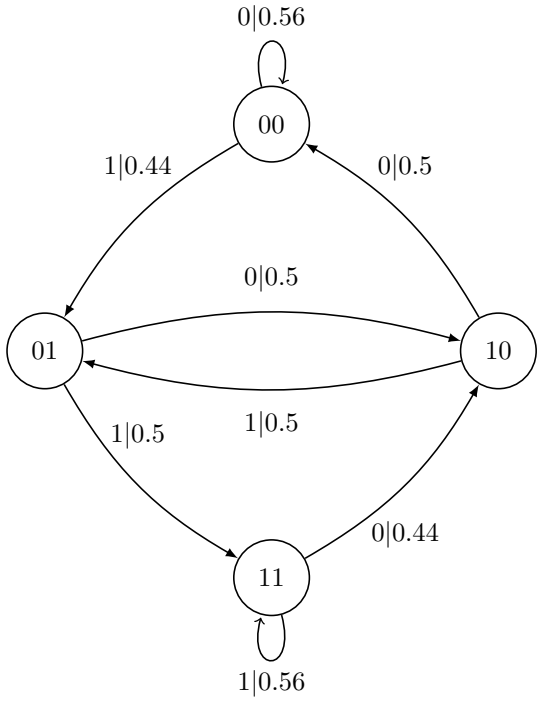

Figura 2.3: *A D-Markov machine for sequence s and D = 2.*

be generated by a synchronizable and irreducible PFSA. CRISSiS estimates the original PFSA by looking at its output sequence. CRISSiS is shown in Algorithm 3 and it consists of three steps:

Identification of Shortest Synchronization Word

Using the definition of Synchronization Word given by 2.8, CRISSiS uses brute force to find the shortest synchronization word. This is shown in Algorithm 4 where each state's morph is checked with its extensions' morphs up to a length L_2. If all statistical tests are positive for a given word, it is returned as the synchronization word to be used.

Recursive Identification of States

States are equivalence class of strings under Nerode equivalence class. For any two strings ω_1 and ω_2 in a state q,

$$\Pr(\omega|\omega_1) = \Pr(\omega|\omega_2). \tag{2.4}$$

These future conditional probabilities uniquely identify each state and Equation 2.4 can be used to check whether two states q_1 and q_2 are the same given $\omega_1 \in q_1$ and $\omega_1 \in q_2$. Once again, the

Algorithm 3 CRISSiS

Inputs: Symbolics string X, Σ, L_1, L_2, significance level α
Outputs: PFSA $\hat{\mathrm{G}} = \{\mathrm{Q}, \Sigma, \delta, \pi\}$
$\omega_{syn} \leftarrow$ null
d $\leftarrow 0$
while ω_{syn} is null **do**
    $\Omega \leftarrow \Sigma^d$
    for all $\omega \in \Omega$ **do**
        if (isSynString(ω, L_1)) **then**
            $\omega_{syn} \leftarrow \omega$
            break
    d $\leftarrow$ d + 1
Q $\leftarrow \{\omega_{syn}\}$
$\tilde{\mathrm{Q}} \leftarrow \{\}$
Add $\omega_{syn}\sigma_i$ to $\tilde{\mathrm{Q}}$ and $\delta(\omega_{syn}, \sigma_i) = \omega_{syn}\sigma_i \forall \sigma \in \Sigma$
for all $\omega \in \tilde{\mathrm{Q}}$ **do**
    if ω occurs in *X* **then**
        $\omega^* \leftarrow$ matchStates(ω, Q, L_2)
        if ω^* is null **then**
            Add ω to Q
            Add $\omega\sigma_i$ to $\tilde{\mathrm{Q}}$ and $\delta(\omega_{syn}, \sigma_i) = \omega_{syn}\sigma_i \forall \sigma \in \Sigma$
        else
Find *k* such that X_k is the symbol after the first occurrence of ω_{syn} in *X*
Initialize π to zero
state $\leftarrow \omega_{syn}$
for all i $\geq k$ in *X* **do**
    π(*state*, X_i) $\leftarrow \pi$(*state*,X_i) + 1
    state $\leftarrow \delta$(*state*,X_i)
Normalize π for each state

Algorithm 4 isSynString(ω, L_1)

Outputs: true or false
for D = 0 to L_1 **do**
    for all s $\in \Sigma^D$ **do**
        if $\mathcal{V}_d(\omega) = \mathcal{V}_d(\mathrm{s}\omega)$ fails the statistic test for some d $\leq L_2$ **then**
            return false
return true

problem of checking all possible strings can not be done in finite time, so only L_2-steps ahead are to be checked, giving:

$$\mathcal{V}_d(\omega_1) = \mathcal{V}_d(\omega_2), \forall d = 1, 2, \ldots, L_2. \tag{2.5}$$

If two states pass the statistical test using Equation 2.5, they are considered to be statistically the same. Strings ω_1 and ω_2 need to be synchronizing in order to use Equation 2.5. If ω is a synchronization word for $\mathrm{q}_i \in \mathrm{Q}$, then $\omega\tau$ is also a synchronization word for $\mathrm{q}_j = \delta(q_i, \tau)$.

The next procedure starts by letting Q be the set of states to be discovered for the PFSA and it is initialized containing only the state defined by the synchronization word ω_{syn} found in the first step. Then, a tree is constructed using ω_{syn} as the root node to $|\Sigma|$ children. Each one of the children nodes is regarded as a candidate states with a representation $\omega_{syn}\sigma$ for $\sigma \in \Sigma$. Each one of them will be tested using Equation 2.5 with each of the states in Q. If a match is found, the child state is removed and its parent σ-transition should be connected to the matching state. 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. This procedure is to be repeated until no new candidate states have to be visited.

As CRISSiS should be applied to estimate finite PFSA G, this procedure will terminate. The edges of the created tree correspond to the PFSA's δ.

Algorithm 5 matchStates(ω, Q, L_2)

```
for all i ∈ Q do
    if V_d(ω) = V_d(Q(i)) fails the statistic test for all d then
        return Q(i), the i-th element of Q
return null
```

Estimation of Morph Probabilities

To recover the morphs of each state in Q found in the last step, the sequence *X* is fed to the PFSA starting at state ω_{syn} and transition following the first symbol after the first occurunce of ω_{syn} in *X*. Each transition is counted and then normalized in order to recover an estimation of the morph.

Example

The PFSA in Figure 2.4, which will be called Tri-Shift in this work, was presented in [ref]. It is synchronizable and works over a binary alphabet. It is used to generate a string *X* of length

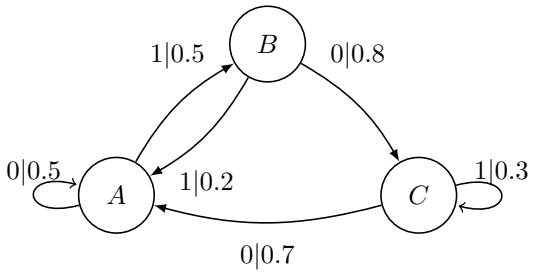

Figura 2.4: *The Tri-Shift PFSA.*

Tabela 2.2: *Subsequence frequencies of a sequence generated by the Tri-Shift.*

L = 1	Freq.	L = 2	Freq.	L = 3	Freq.	L = 4	Freq.	L $\geq$ 5	Freq.
0	62711	00	35164	000	17565	0000	8673	00100	9881
1	37291	01	27546	001	17599	0001	8892	00101	4181
		10	27546	010	21451	0010	14062	001000	4990
		11	9745	011	6094	0011	3536	001001	4891
				100	17599	0100	14206	001010	2926
				101	9946	0101	7245	001011	1255
				110	6094	1000	8892		
				111	3651	1001	8707		
						1100	3393		
						1101	2701		

10000. Table 2.2 gives the frequency count of some subsequences occurring in *X*. 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 Equation 2.2. Starting by 0, neither $\Pr_1(0) = \Pr_1(00)$ nor $\Pr_1(0) = \Pr_1(01)$ pass the χ^2 test. Then the state 1 is tested, which also fails. For state 00, the derived frequencies are relatively close and it passes the test, giving 00 the status of synchronization word.

The second step starts by defining the synchronization word's state 00 and split it into two candidates states, 000 and 001 (Figure 2.5. State 00 is added to Q. Each candidate has its derived frequencies compared to 00, which is the only state in Q, with Equation 2.5. $\mathcal{V}_1(000) = [0.4940.506]$ is considerably close to $\mathcal{V}_1(00) = [0.5000.500]$, so they pass the statistical test and 00 and 000 are considered to be the same state. 000 is removed and the edge going from 00 to 000 becomes a self-loop from 00 to itself. On the other hand, $\mathcal{V}_1(001) = [0.8000.200]$ is considerably different from 00's morph, therefore it is considered a state and added to Q and then it is split into two new candidates (Figure 2.6).

The same procedure is then repeated for the candidates 0010 and 0011. $\mathcal{V}_1(0010) = [0.7030.297]$

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. $\mathcal{V}_1(0011) = [0.5000.500]$ passes the test with $\mathcal{V}_1(00)$, which means that 0011 is removed and the edge from 001 to 0011 goes back to 00. This leads to the configuration in Figure 2.7.

The next candidates are similar to two states in Q ($\mathcal{V}_1(00100) = [0.5050.495]$ passes with 00 and $\mathcal{V}_1(00101) = [0.7000.300]$ passes with 0010), so both are removed and its edges rearranged to the configuration in Figure 2.8, which is the same topology as the original Tri-Shift, showing that CRISSiS already recovered the PFSA's topology. All that is left is to run step 3, feeding the input sequence to the graph and computing the morph probabilities, which will recover an accurate Tri-Shift PFSA.

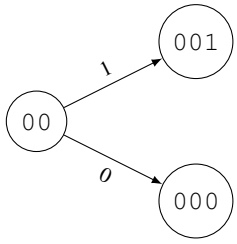

Figura 2.5: *Tree with 00 at its root.*

Time Complexity

As shown in [ref], 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 sequence's alphabet size, $|Q|$ is the number 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

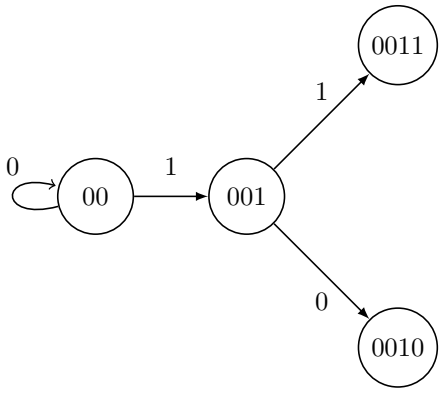

Figura 2.6: *Second iteration of three.*

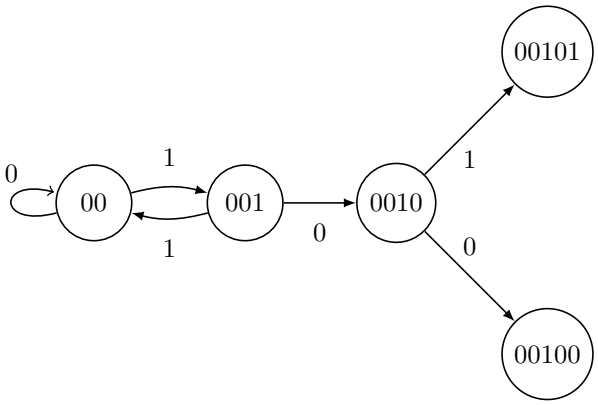

Figura 2.7: *Third iteration of the three.*

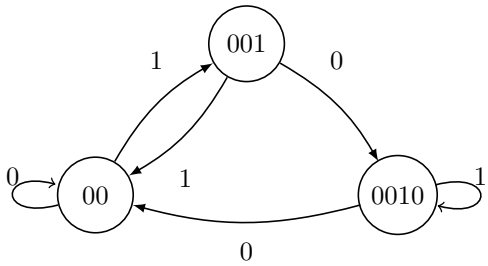

Figura 2.8: *Recovered Tri-Shift topology.*

small, it does not affect the performance greatly, even though the algorithm is exponential in these parameters. The biggest burden lies in finding the synchronization word, which can be very time consuming when it is very large.

CAPÍTULO 3

ALGORITHM DESCRIPTION

In this chapter, the proposed algorithm to model a system by its 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 PFSA completion, which will make sure that all states have outgoing edges and the full $\aleph$ algorithm which will create the most compact 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 termination criterion for the rooted tree with probabilities $\mathcal{S}$ based on S;

   a) Use D-Markov Termination if no synchronization words were found;

   b) Use Ω if synchronization words were found;

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 (2.2) 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.

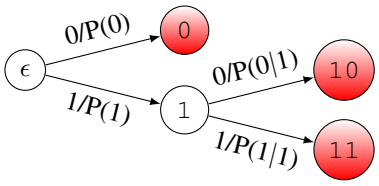

Figura 3.1: *Example of a rooted tree with probabilities.*

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 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 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 of *S*. If *S* is infinite, $\mathcal{S}$ can also have infinite *L*. In practice, that is not possible and a user defined maximum depth has to be used. It is good to remind that as the chance of sub-sequences occurring gets smaller as their length increases, the statistics of really large sub-sequences might be really poor. 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) [**?**]. A

dictionary d is a mapping between two sets $d : X \rightarrow Y$. The elements from X are called the dictionary's 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.

The concept of a *shortest valid suffix* (SVS) also needs to be explained as it is important in one of the algorithm's 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 $\omega = \sigma_1\sigma_2 \dots \sigma_n \in \Sigma^*$ and the tree$\mathcal{S}$ as inputs. A third input is the dictionary *candidacy*, which indicates whether a state is a valid candidate for synchronization word or not. For a given state x, the entry *candidacy*$[x]$ has the value *True* if x is a valid candidate and *False* if not. First ω 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*[ϵ] 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.

To find the synchronization words, Algorithm 6 is used. Its input are the rooted tree with probabilities $\mathcal{S}$, 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 contains states from $\mathcal{S}$ that are not fully tested for the synchronization word hypothesis during the current iteration. Γ is initialized only with ϵ. A list Θ is created and initialized empty. It receives the states from $\mathcal{S}$ which currently have passed the statistical test. When a test fails, Θ is appended to Γ and emptied again, as the states in Θ have to be checked along with other states that might now have them as suffixes. Once all the tests are performed, Θ is returned as it contains the list of synchronization words by the

end of the algorithm.

Three dictionaries are also created. The first one, *candidacy* has already been discussed before. The states from $\mathcal{S}$ are the keys and to each one a boolean value is associated. When a state is still a valid candidate for synchronization word (which means it either had not been tested against its suffixes yet or has passed all the tests up to the current iteration), it has the *True* value associated with it. Once it fails a test and can no longer be considered a valid candidate to be a synchronization word, the associated value becomes *False*. A state for which the *candidacy* value is *True* is called a valid state. At the beginning of the algorithm, all states are valid as they have not been tested yet and their *candidacy* is initialized accordingly.

The second dictionary is called *suffixes* and 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. 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.

The last dictionary, V stores lists of states associated with their shortest valid suffix as key. states should only be queued into Γ if they are their own shortest valid suffix. During a call to Algorithm 7, which occurs every time a statistical test fails, if a new state to be added in Γ is not its own shortest valid suffix, its shortest valid suffix is used as a key in V and this element is associated with it. If later this shortest valid suffix fails a statistical test, all elements associated with it in V have to be checked again to see if now they are their own shortest valid suffixes.

When two states $p, q \in \mathcal{S}$ are compared (with the notation $\mathcal{V}(p) = \mathcal{V}(q)$ it means that their morphs are compared via an appropriate statistical test (such as the χ^2 test or Kolmogorov-Smirnov test) with a predetermined confidence level α.

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). If the label of c (denoted by c.label) is longer than W, the algorithm stops and returns the list Θ. If it is not, a flag p is set to True and it will be used to store the result of the statistical tests. If *suffixes*$[c]$ is empty, p will stay True. On the other hand, if there are states for which c is a suffix, *suffixes*$[c]$ will be iterated. Each element of *suffixes*$[c]$ goes through the statistical test with c. This is done to check if the states that have c as suffix have morphs statistically equal to the morph of c. For each of these tests, p is updated. If all tests are true, p is True by the end of it and c gets to keep its status as a valid candidate for synchronization word and it is appended at Θ as it currently is a valid candidate

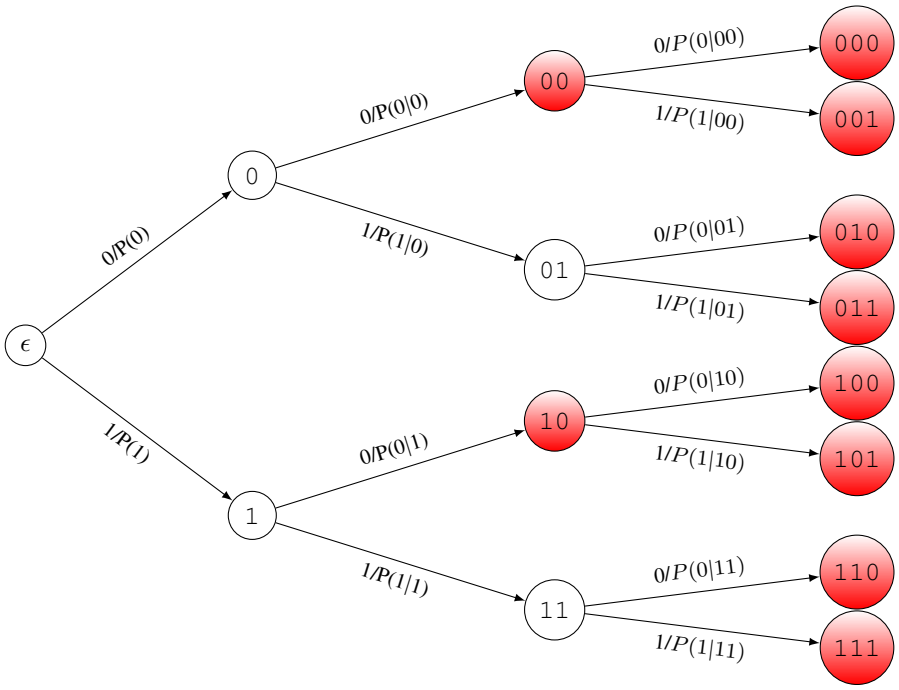

Figura 3.2: *Example of binary $\mathcal{S}$*

synchronization word and it passed in all its tests. If one of the tests fails, p is set to False and no more tests need to be done for c. The candidacy of c will be set to False, the list Γ and the dictionaries will be expanded according to Algorithm 7 (which will be explained later) and for each element $\theta \in \Theta$ will need to be tested again for the new elements appended to *suffixes*[θ] after the expansion. This means that each element of Θ is concatenated at the end of Γ and then it will be set to the empty set again. This procedure is repeated until either the queue Γ is empty or if all the elements in Γ have labels longer than W. It will return Θ, with all the elements that passed in all their statistical tests, meaning that they are synchronization words according to (2.2).

Algorithm 7 updates Γ and the dictionaries *suffixes* and V after a statistical test fails. 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. If there is a list associated to c in V, all its elements are appended to Ψ. Those are the elements that instead of being their own SVS, had c as the shortest valid suffix. The entry $V[c]$ is then deleted as it no longer has a use.

The next step is to iterate for each element d in Ψ and check if they are their own SVS using the *shortestValidSuffix* function and comparing if the returned state is d. If they are not, this means that there are shorter words that need to be checked before them. In this case, the entry $V[\zeta]$ is created,

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(Γ)
        if length$(c$.label$) < W$ **then**
            $p \leftarrow$ True
            if $\Lambda \neq \emptyset$ **then**
                for every $\lambda \in$ *suffixes*$[c]$ **do**
                    $p \leftarrow$ *statisticalTest*$(\mathcal{V}(c), \mathcal{V}(\lambda), \alpha)$
                    if $p =$ False **then**
                        candidacy$[c] \leftarrow$ False
                        expand$(c, V, \mathcal{S}, \Gamma,$ candidacy, suffixes$)$
                        for every $\theta \in \Theta$ **do**
                            Γ.queue(θ)
                        $\Theta \leftarrow \emptyset$
                        break
            if $p =$ True **then**
                Θ.*append*(c)
    return Θ

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**
      Γ.queue(ζ)
      for $t \in \{\delta(\sigma, \zeta) \forall \sigma \in \Sigma\}$ **do**
        short $\leftarrow$ *shortestValidSuffix*($\mathcal{S}, t$, candidacy)
        suffixes[short].*append*(t)
    else
      if $V[\zeta] = \emptyset$ **then**
        $V[\zeta] \leftarrow \{d\}$
      else
        $V[\zeta]$.*append*(d)

where ζ is the SVS of d, and d is stored in this list. Once this ζ is tested in Algorithm 6 and if it fails its statistical test, this d will be checked again to see if it is its SVS.

On the other hand, if $\zeta = d$, this element is then added to the end of the queue Γ and it will be later dequeued and tested. The last step is to update the suffix dictionary so there are new elements to be compared with their suffixes. The list ofall the descendants of ζ is iterated. The shortest valid suffix *short* of each of its elements t is found and used as an entry of *suffixes*, such that *suffixes*[short] appends t.

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 for $W = 3$ and $L = 4$ is shown in Figure 3.3. 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 $\mathcal{V}(\epsilon) = \mathcal{V}(0)$. 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 (seen in Figure 3.4). 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.

At the beginning of the next iteration, $\Gamma = \{0, 1\}$ 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 $\mathcal{V}(0) = \mathcal{V}(00)$ 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 re-emptied.

In the beginning of the next iteration, we have $\Gamma = \{1, 000\}$, $V[1] = \{01\}$, $\Theta = \emptyset$, suffixes[1] = $\{01, 11, 001\}$ and suffixes[00] = $\{000\}$. Γ is dequeued and $c = 1$, *suffixes*[1] = $\{01, 11, 001\}$ is iterated through. First, $\mathcal{V}(1) = \mathcal{V}(01)$ 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 $c = 00$ and *suffixes*[c]= $\{000, 100\}$. 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] = $\{011, 111\}$. After dequeueing, $c = 01$ and *suffixes*[c]= $\{001, 101\}$. The test $\mathcal{V}(01) = \mathcal{V}(001)$ 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] = $\{000, 100, 0100\}$. $c = 10$, *suffixes*[c]= $\{010, 110\}$ and $\mathcal{V}(10) = \mathcal{V}(010)$ 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] = $\{0101\}$. $c = 11$ and *suffixes*[c]= $\{011, 111, 1011\}$. The test $\mathcal{V}(11) = \mathcal{V}(011)$ 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] = $\{1011\}$. $c = 010$ which is now equal in length to $W = 3$, which means it is no longer tested.

In the ninth iteration, $c = 00$ and $\Lambda = \{000, 100, 0100, 1100\}$. 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. Although this Algorithm seems more contrived, less statistical tests were performed and the search was more thorough than CRISSiS.

3.2 PFSA Construction

In this section we discuss the $\aleph$ algorithm that constructs a PFSA from the RTP $\mathcal{S}$. Whether synchronization words were found or not will influence the operations of the algorithm. 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 transformation criteria described in Section 3.2.1 that will reconnect states in such a way that all remaining states are non-leaf states. Different criteria are used when synchronization words are found and when there are none.

Once $\mathcal{S}$ has been turned into a graph, $\aleph$ will apply a procedure to reduce the number of states if there are synchronization words. If there are none, this step is skiped. The final procedure will group states in equivalence classes of states that have statistically similar morphs (checked by χ^2 or Kolmogorov-Smirnov for a given confidence level α) and the partition given by these equivalence classes is used as an initial partition for a graph reduction algorihm (such as Moore or Hopcroft) to obtain the final reduced PFSA.

3.2.1 Termination 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. Two criteria are used to terminate the last levels of the RTP $\mathcal{S}$ and turn it into a PFSA. The first one, the D-Markov termination, is more straightforward while the Ω termination depends on the system having synchronization words and is able to create connections that better represent the original system.

D-Markov Termination

This is the simplest termination criterion. It will aim to form a D-Markov Machine with the states in the last level. This means that a state labeled with $\omega = \sigma_0\sigma_1 \dots \sigma_L \in \Sigma^L$ will have its $\tau \in \Sigma$ labeled edge connected to the state labeled with $\sigma_1\sigma_2 \dots \sigma_L\tau$ for each $\tau \in \Sigma$. This is shown in Algorithm 8.

This termination does not rely on the system's memory, preferring to count on the amount of cases that are captured in a D-Markov machine and applying the subsequent algorithms to reduce its size. It is a better option when the system to be modeled does not synchronize.

Ω Termination

This termination criteria relies more on using synchronization words, which means it is more suitable to systems that synchronize and that have some memory. For each state n in level $L+1$, it checks via statistical test if n has similar morph to any of the synchronization words states. If it is

Algorithm 8 dmarkov-termination($\mathcal{S}, L$)

procedure TERMINATE
  $\Psi \leftarrow \{n \in \mathcal{S}$ if n in level $L\}$
  for $p \in \Psi$ **do**
    Given that p.label is $\sigma_0\sigma_1 \dots \sigma_L$
    for $\tau \in \Sigma$ **do**
      $\delta(\tau, p) \leftarrow \sigma_1\sigma_2 \dots \sigma_L\tau$

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 m in level L that has $\delta(\tau, m) = n$ for $\tau \in \Sigma$ will have this edge reassigned for the state with which the test was successful. In case no test passes, the D-Markov criteria is used for m. This is shown in Algorithm 9 whose inputs are the rooted tree with probabilities $\mathcal{S}$, the desired last level L and a list of synchronization words Ω_{syn}.

Algorithm 9 Ω-termination($\mathcal{S}, L, \Omega_{syn}$)

procedure TERMINATE
  $\Psi \leftarrow \{p \in \mathcal{S}$ if n in level $L\}$
  for $m \in \Psi$ **do**
    next = NULL
    for $\tau \in \Sigma$ **do**
      $n = \delta(\tau, m)$
      for $\omega \in \Omega_{syn}$ **do**
        $r \leftarrow \mathcal{V}(n) = \mathcal{V}(\omega)$
        if $r = True$ **then**
          next $\leftarrow \omega$
          break
      if next = NULL **then**
        $\eta \leftarrow \{$All extensions of ω up to length $L, \forall\omega \in \Omega_{syn}\}$
        for $e \in \eta$ **do**
          $r \leftarrow$ *statisticalTest*(n, e)
          if $r = True$ **then**
            next $\leftarrow e$
            break
      if next = NULL **then**
        Given that m.label = $\sigma_0 \dots \sigma_L$
        next = $\sigma_1 \dots \sigma_L\tau$
      $\delta(\tau, m) \leftarrow$ next

3.2.2 ℵ Algorithm

The full ℵ algorithm is shown in Algorithm 10. When there are no synchronization words (i.e. the Ω_{syn} list is empty), the D-Markov termination is applied to $\mathcal{S}$ and an equivalence class is created with just the root state ϵ. When Ω_{syn} is not empty, the Ω termination is used in $\mathcal{S}$. In this case, each state q' 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. If it has, this outgoing edge is reassigned to point to ω. This is done because a state that has a synchronization word as suffix is bound to have a morph similar to the synchronization word and generate the same sequences, so they can safely be discarded. As the rest of the algorithm depends on the number of states of the original PFSA, rapidly discarding some states guarantees more efficiency in the following steps. The case where Ω_{syn} is not empty ends by creating an equivalence class for each of the synchronization words. In both cases, all equivalence classes are stored in the list $\mathcal{P}$.

A list Q is created containing the descendants of the states in each of the initial equivalence classes. This list is iterated and each element q from it is compared via statistical test with each of the equivalence classes head states. If q passes in a test, it is added to the equivalence class for each the test was sucessful. If no test is positive, a new equivalence class is created for q and this class is subsequently added to $\mathcal{P}$. Once every state of $\mathcal{S}$ is in one class of $\mathcal{P}$, a graph reduction algorithm (either Moore or Hopcroft) is applied using $\mathcal{P}$ as the initial partition. This initial partition guarantees that the elements in the equivalence class have the same morph and the reduction algorithm will break this class if they eventually point to states with different morphs. The final result is the minimal PFSA that represents the original system for the given parameters.

3.3 Time Complexity

The main improvements of the ℵ 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 ℵ will generate a PFSA that approximates it) and the lower time complexity to run the algorithm. As seen in [**?**], CRISSiS complexity depends on the number of states of the original system which, in practical applications, remains unknown until the end of the algorithm. As it is discussed in this section, the complexity of α depends only on paramters known prior the the algorithm execution. The complexity of each part of the algorithm is discussed individually and a final complexity is given in the end.

Algorithm 10 $\aleph(\mathcal{S}, \Omega_{syn})$

procedure
  if $\Omega_{syn} \neq \emptyset$ **then**
    $\mathcal{S} \leftarrow$ *omegaTermination*($\mathcal{S}$)
    $Q_0 \leftarrow \Omega_{syn}$
    $Q_1 \leftarrow \emptyset$
    while $Q_0 \neq \emptyset$ **do**
      $q_0 \leftarrow Q_0.pop()$
      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 $q_0' \notin Q_1$ **then**
            $Q_0 \leftarrow Q_0 \bigcup \{q_0'\}$
        $Q_1 \leftarrow Q_1 \bigcup \{q_0\}$
    $Q \leftarrow \{\delta(\sigma, \omega), \forall \sigma \in \Sigma, \forall \omega \in \Omega_{syn}\}$
  else
    $\mathcal{S} \leftarrow$ *dmarkovTermination*($\mathcal{S}$)
    $Q_1 \leftarrow \{\epsilon\}$
    $Q \leftarrow \{\delta(\sigma, \epsilon), \forall \sigma \in \Sigma\}$
  $\mathcal{P} \leftarrow \{Q_1\}$
  for $q \in Q$ **do**
    $r \leftarrow$ False
    for $p \in \mathcal{P}$ **do**
      $r \leftarrow$ *statisticalTest*($q, p[0]$)
      if $r =$ True **then**
        $p \leftarrow p \bigcup \{q\}$
        break
    if $r =$ False **then**
      $R \leftarrow \{q\}$
      $\mathcal{P} \leftarrow \mathcal{P} \bigcup \{R\}$
    $Q \leftarrow Q \bigcup \{\delta(\sigma, q), \forall \sigma \in \Sigma | \delta(\sigma, q)$ not in any $p \in \mathcal{P}\}$
  ## The following function can be either Moore or Hopcroft
  $G \leftarrow$ GraphReduction($\mathcal{P}$)
  return G

3.3.1 Synchronization Word Search

For a given state with length $n < W$, the maximum amount of statistical tests it will go through is n for each of its suffixes, starting by ϵ. For each of these tests, one search for SVS is performed. This search has complexity $O(m)$ for a SVS of length m. Thus, for the given state of length n, searches 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.

Looking at $\mathcal{S}$, for a given level d, $|\Sigma|^d$ tests and searches of length d are performed, giving a complexity of $O(|\Sigma|^d d^3)$ per level. The total complexity is the sum of all levels from 1 to W. Therefore, it is $O(\sum_{d=1}^{W} |\Sigma|^d d^3)$ and as usually $W > |\Sigma|$, the final complexity for the synchronization word search is $O(\frac{|\Sigma|^{W+1} W^3}{|\Sigma|-1})$.

3.3.2 Termination

In the D-Markov termination, 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 termination 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 $O(|\Sigma|^{L+1}|\Omega_{syn}|L^2)$.

3.3.3 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.

To separate all the $|\Sigma|^{L+1}$ states of $\mathcal{S}$ (worst case scenario) in equivalence classes, each one of them has to be tested against the

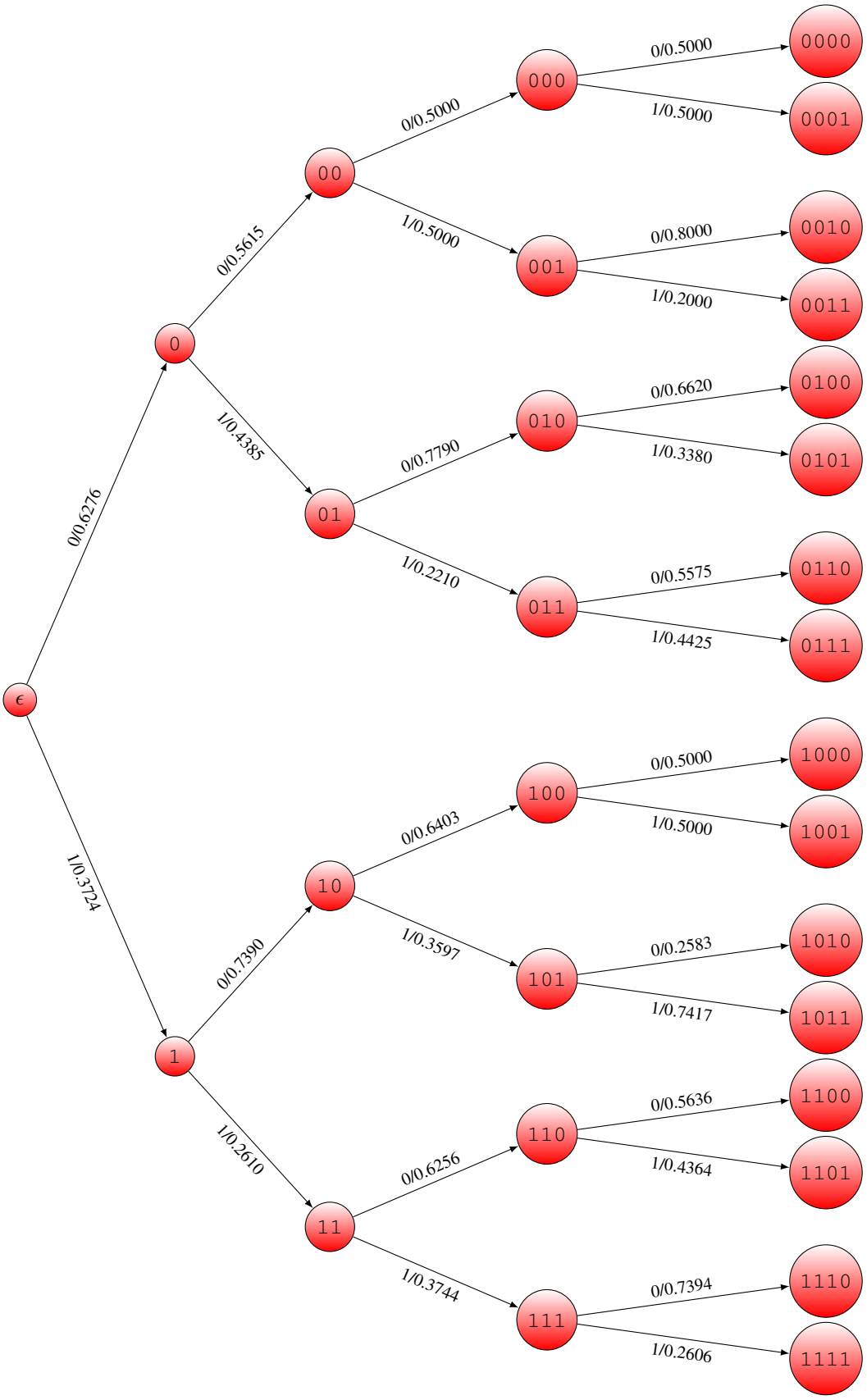

Figura 3.3: *Input Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example.*

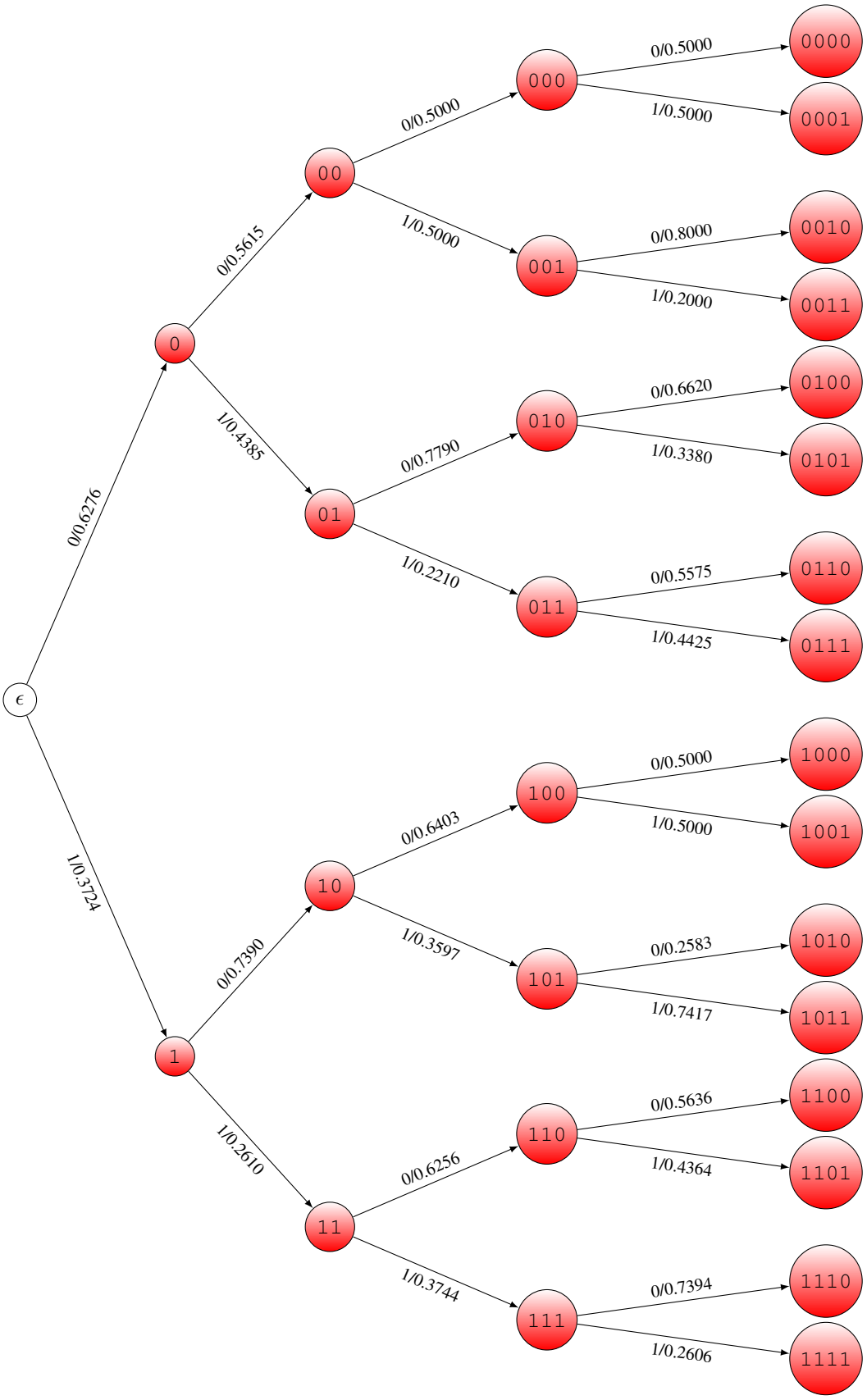

Figura 3.4: *Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example after the first iteration.*

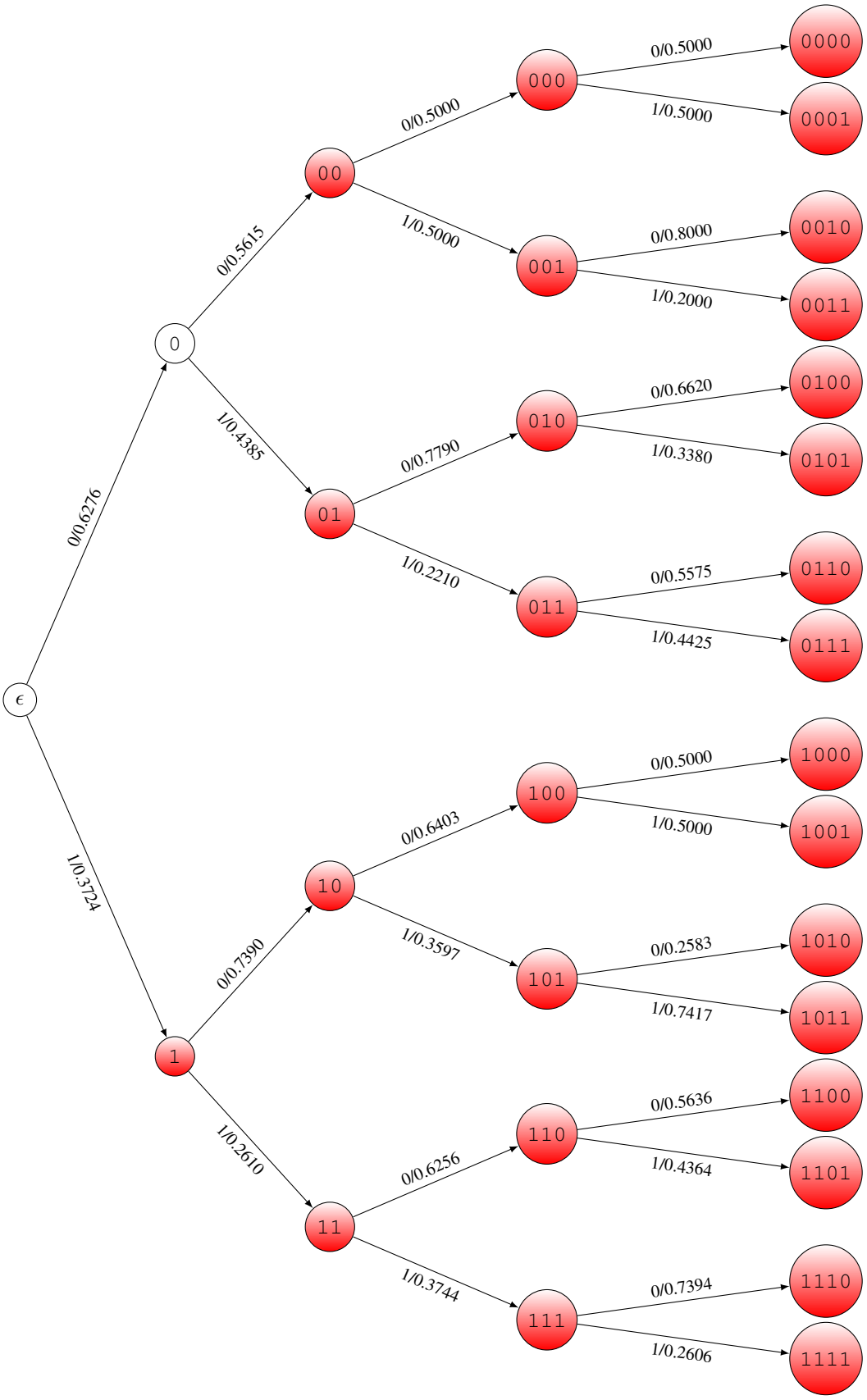

Figura 3.5: *Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example after the second iteration.*

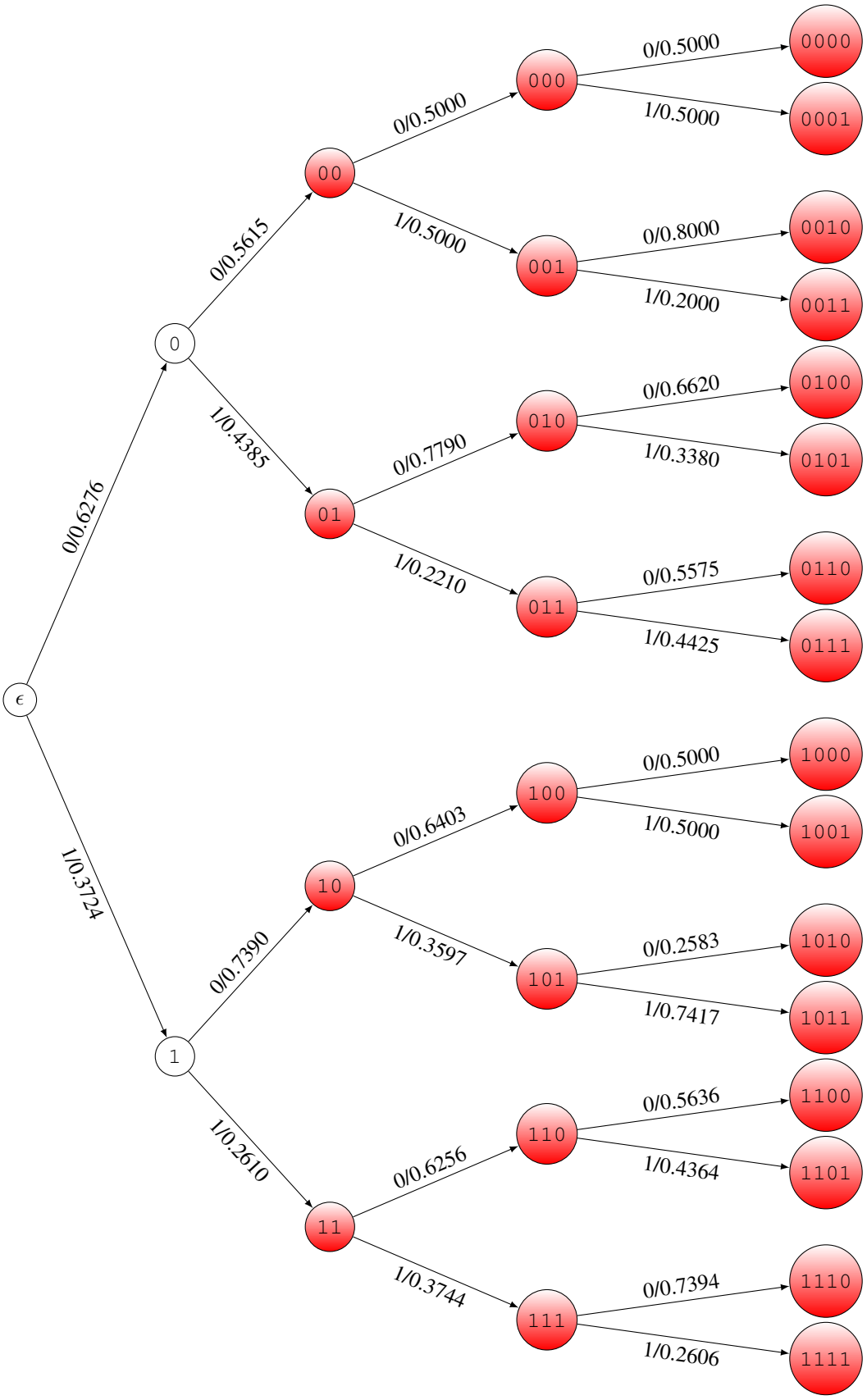

Figura 3.6: *Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example after the third and fourth iterations.*

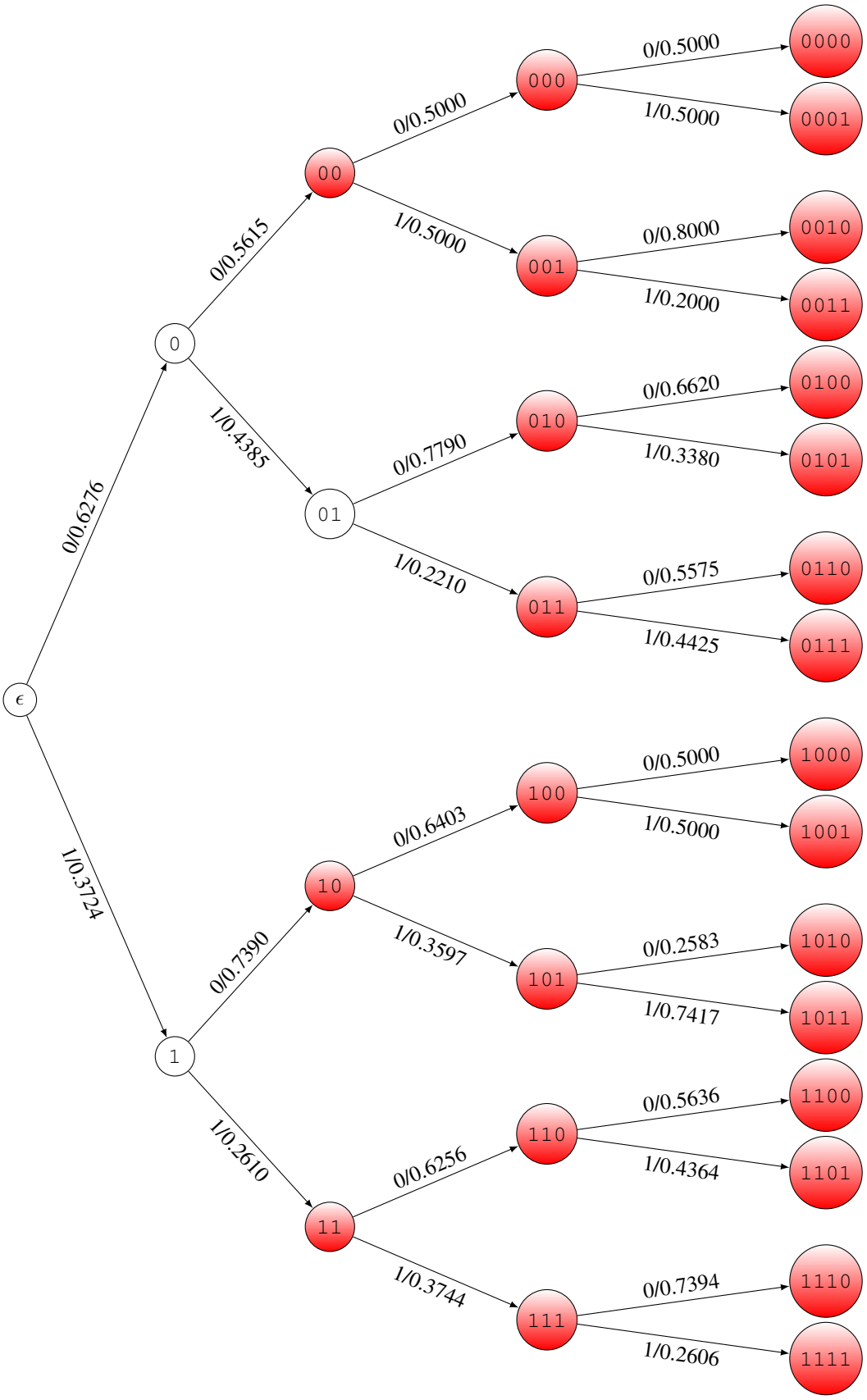

Figura 3.7: *Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example after the fifth iteration.*

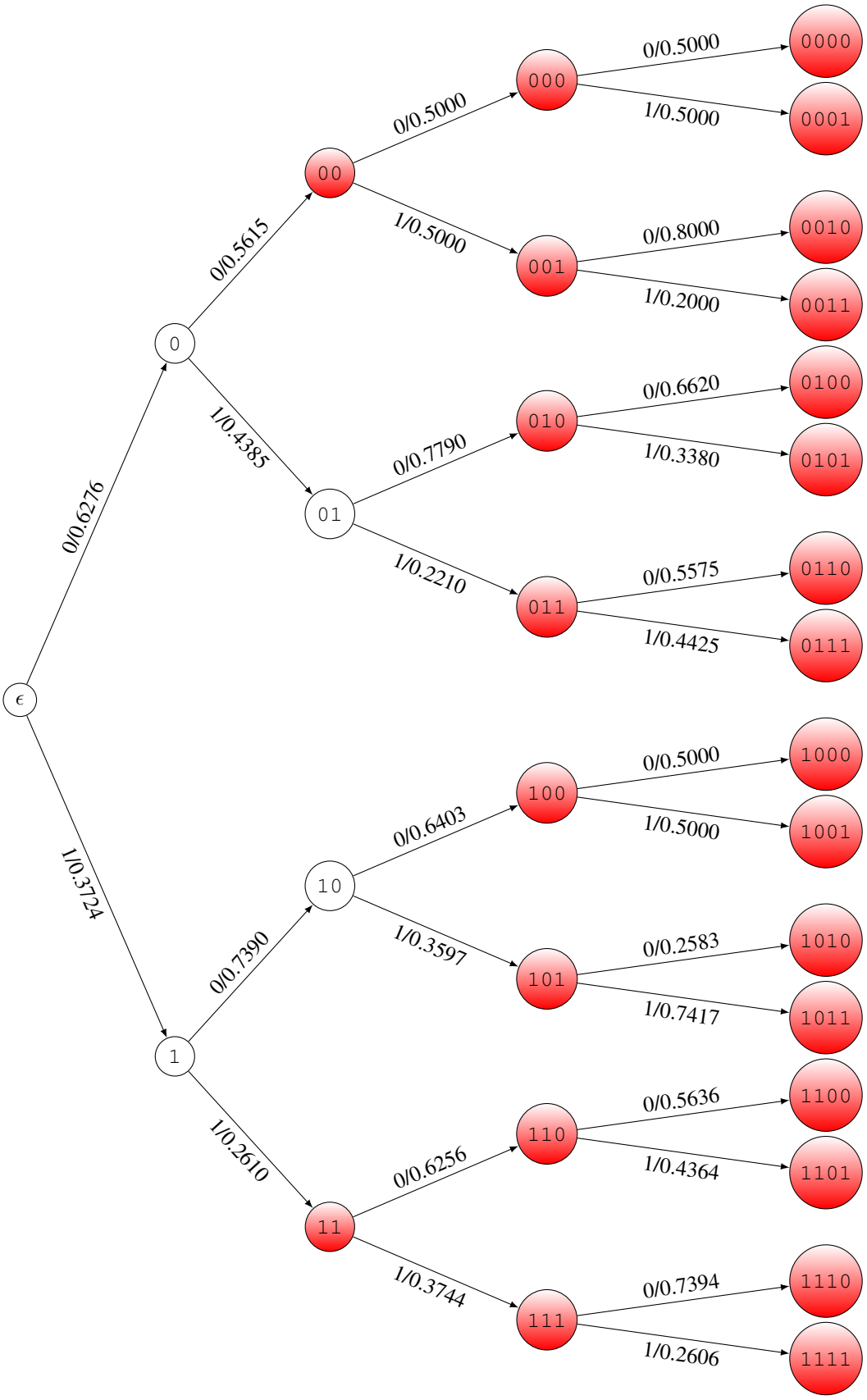

Figura 3.8: *Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example after the sixth iteration.*

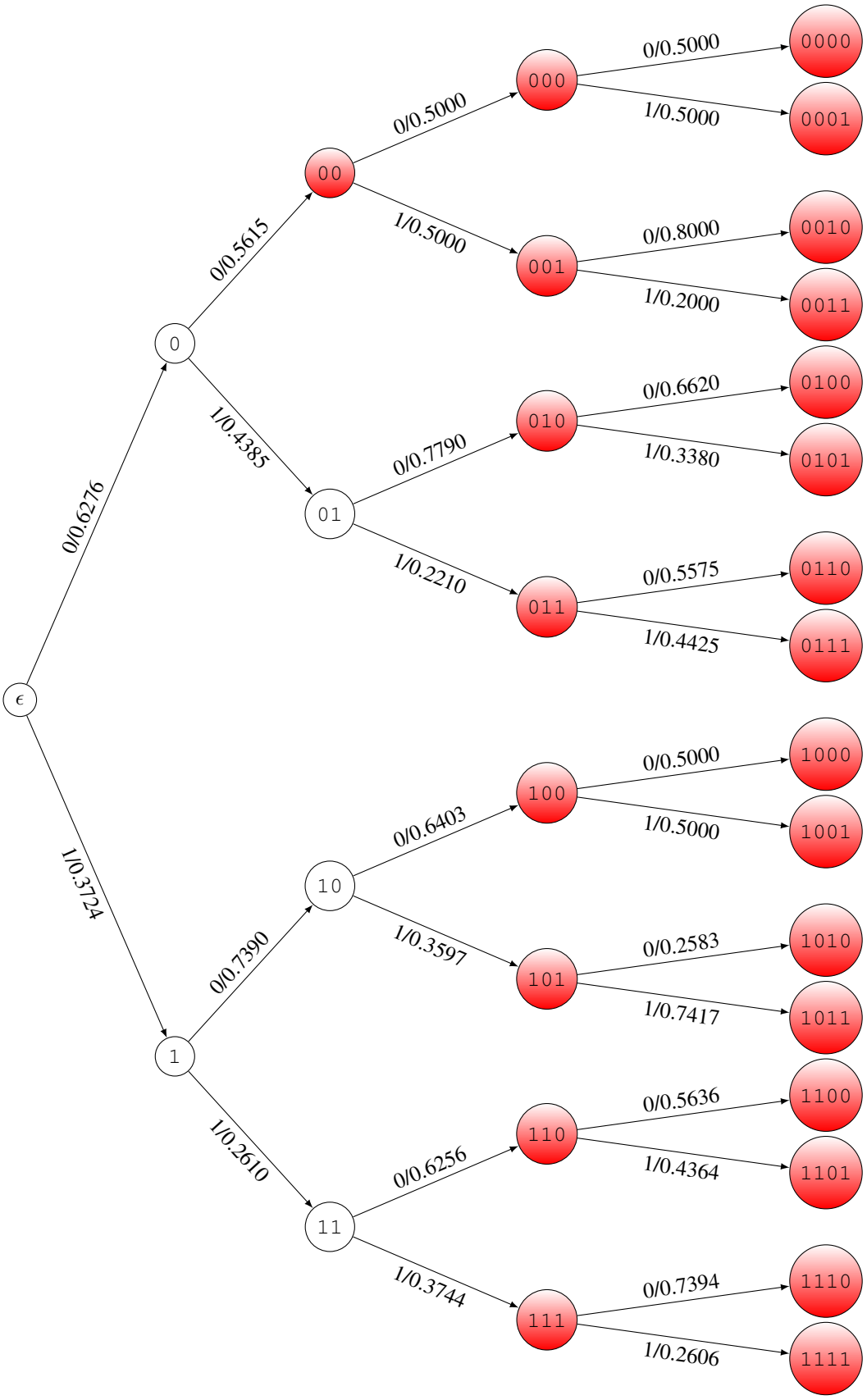

Figura 3.9: *Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example after the seventh iteration.*

CAPÍTULO 4

RESULTS

In this chapter, the efficiency of the algorithms proposed in Chapter 3 to construct a PFSA is verified for some examples of simple dynamic systems. 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}. After this, a series of PFSA are created using the D-Markov Machine, CRISSiS and our algorihm with different values of their parameters 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 of the results are explained in Section 4.2.

4.1 Evaluation Metrics

4.1.1 Entropy Rate

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

$$h \triangleq \lim_{l \to \infty} H(X_k|X_1X_2 \dots X_{k-1}) = -\sum_{x \in X} P(x_l|X_1X_2 \dots X_{k-1}) \log P(x_k|X_1X_2 \dots X_{k-1}). \tag{4.1}$$

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 [**?**]. 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:

$$h_\ell \triangleq H(X_\ell | X_1 X_2 \dots X_{\ell-1}), \tag{4.2}$$

which measures the uncertainty of a random process given the previous ℓ samples. Comparing the values of ℓ-order entropy rate of the generated PFSA with the one from the original system is useful to test if the generated one correctly captures the system memory, in a sense of outputting reasonable symbols given the ℓ previous ones.

4.1.2 Kullback-Leibler Divergence

For the purpose of comparing the algorithms, consider two PFSA $\mathrm{K}_1 = (\Sigma, Q_1, \delta_1, \mathcal{V}_1)$ and $\mathrm{K}_2 = (\Sigma, Q_2, \delta_2, \mathcal{V}_2)$ over a common alphabet Σ. Let $\omega \in \Sigma^\ell$ be a sequence of length ℓ and $P_1(\omega)$ and $P_2(\omega)$ are the probabilities of K_1 and K_2 generating ω respectively. For a given ℓ we take the ℓ-order Kullback-Leibler Divergence as:

$$D_\ell(K_1 || K_2) = \sum_{\omega \in \Sigma^\ell} P_1(\omega) \log \Big(\frac{P_1(\omega)}{P_2(\omega)} \Big). \tag{4.3}$$

Although it is technically not a distance, as it does not obey the triangle inequality nor is necessarily commutative, the Kullback-Leibler Divergence is useful to give an idea of how similar two distributions are. A small divergence will indicate 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.

4.1.3 Φ-Metric

The Φ-Metric is presented in [**?**] as a way to compare to different PFSA. Given two PFSA $\mathrm{K}_1 = (\Sigma, Q_1, \delta_1, \mathcal{V}_1)$ and $\mathrm{K}_2 = (\Sigma, Q_2, \delta_2, \mathcal{V}_2)$ over a common alphabet Σ and $P_1(\Sigma^\ell)$ and $P_2(\Sigma^\ell)$ defined as in Section 4.1.2. The Φ-Metric is then defined as:

$$\Phi(K1, K2) \triangleq \lim_{n \to \infty} \sum_{j=1}^{n} \frac{\| P_1(\Sigma^j) - P_2(\Sigma^j) \|_{\ell_1}}{2^{j+1}}, \tag{4.4}$$

where $\| \star \|_{\ell_1}$ indicates the sum of absolute values of the elements in the vector $\star$. As (4.4) puts more weight in shorter words, it can be truncated with a relatively small ℓ. The ℓ-order Φ-Metric is:

$$\Phi_\ell(K1, K2) \triangleq \sum_{j=1}^{\ell} \frac{\| P_1(\Sigma^j) - P_2(\Sigma^j) \|_{\ell_1}}{2^{j+1}}. \tag{4.5}$$

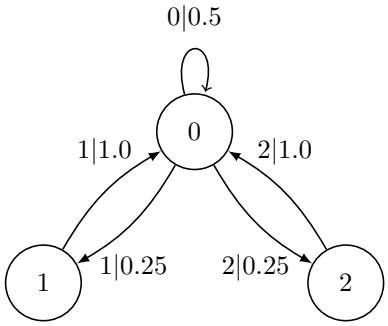

Figura 4.1: *A PFSA of a Ternary Even-Shift.*

As with the Kullback-Leibler Divergence, the Φ-Metric compares two PFSA using sequences generated by them, taking the distribution of sub-sequences of length ℓ. A small Φ indicates that the PFSA are similar to each other. For the performance comparison, each PFSA generated by the algorithms is compared with the original sequence using (4.5) and the smaller the result, the better the algorithm models the original system.

4.2 Construction of PFSA for Dynamic Systems

The following cases are examples of dynamic systems with known representations as PFSA. The goal is to apply the D-Markov Machine, $\aleph$ and CRISSiS algorithm to try to recover the original PFSA or at least a good PFSA with the smallest number of states possible.

In all examples, the $\aleph$ algorithms are able to recover the original PFSA for some value of L. When CRISSiS recovers the same PFSA as $\aleph$ for some L_2, its results are shown alongside with the $\aleph$ results. When different values of L_2 produce different PFSA, two results for CRISSiS are shown: one with a lower L_2, which generates a smaller PFSA but with worse performance, and one for a higher L_2 which is capable to retrieve the original PFSA. A similar procedure is done for D-Markov Machines: first we show the result for a D that is capable to produce a smaller but worse result, and D that achieves performance closer to $\aleph$, but in exchange for a much higher number of states.

In the following cases we consider $\ell = 10$.

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 non-zero symbols between zeroes. A PFSA that satisfies this restriction is shown in Figure 4.1.

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

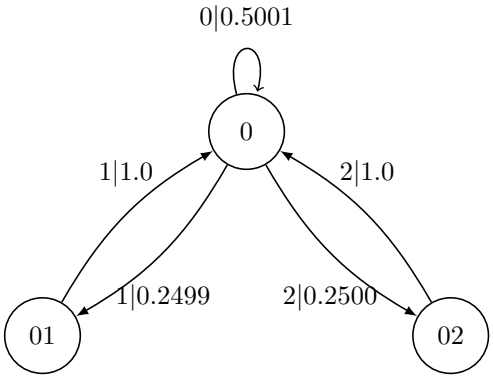

Figura 4.2: *PFSA of a Ternary Even-Shift generated by the $\aleph_1$ algorithm and by CRISSiS.*

The synchronization words found by our algorithm are shown in Table 4.1. It is possible to check in the graph that all found synchronization words are indeed valid and comprise the three states of the graph. They can all be used as starting points for the algorithm.

The results of our algorithm are compared to D-Markov and CRISSiS in Table 4.2. Our algorithm used the Omega termination and handed the same results for any L greater than 2. D-Markov machines of $D = 8$ and $D = 9$ were considered. CRISSiS was tested using $L_2 = 1$. Both CRISSiS and $\aleph_1$ 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 $D = 9$ of at least 339 states is needed to obtain approximately the same performance. Even though D = 6 and 7, these D-Markov machines do not have 2187 and 19683 states respectively because there are forbidden words in the original system, which results in some states being non-existent in the RTP. The original system had $h_{10} = 1.0003$, which is close to the value found by all the algorithms.

4.2.2 Tri-Shift

The Tri-Shift was previously discussed in Section 2.5.2 and a PFSA that represents it is shown in Figure 2.4. The synchronization words found by the algorithm are shown in Table 4.3 and 00

Tabela 4.2: *Results for Ternary Even Shift.*

	D-Markov		$\aleph_1$**/CRISSiS**
	D = 8	D = 9	L = 2/L_2 = 4
# of States	169	339	3
h_{10}	1.0084	1.0058	1.0003
D_{10}	$2.7 \cdot 10^{-3}$	$4.16 \cdot 10^{-5}$	$9.55 \cdot 10^{-5}$
Φ_{10}	$2.1 \cdot 10^{-3}$	$1.2 \cdot 10^{-3}$	$2.3 \cdot 10^{-3}$

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

appeared, as expected, and 0110 synchronizes to the same state as 00. The comparative results are shown in Table 4.4. Once again this is an example where our algorithm and CRISSiS are able to recover the three states from the original PFSA with a good estimate for the morphs as seen in Figure 2.8. 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's memory.

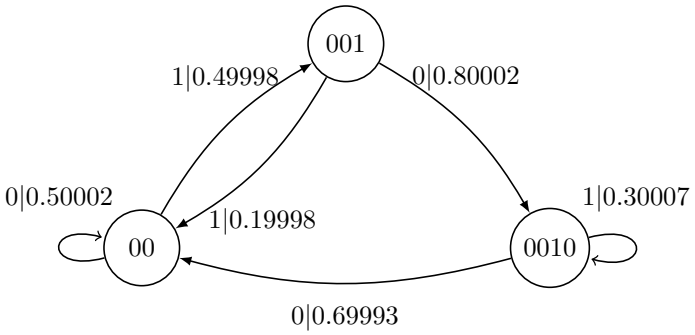

Figura 4.3: *The Tri-Shift PFSA generated by our algorithm and by CRISSiS.*

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

	D-Markov		$aleph_1$**/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}	$4.1 \cdot 10^{-3}$	$1.65 \cdot 10^{-3}$	$1.16 \cdot 10^{-3}$
Φ_{10}	$2.1 \cdot 10^{-3}$	$7.2 \cdot 10^{-4}$	$8.2 \cdot 10^{-4}$

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

4.2.3 A Six State PFSA

Figure 4.4 shows a PFSA with six states that elucidates the differences between our algorithm and CRISSiS. This system has 4 synchronization words: 00, 01, 10 and 1111, as shown in Table 4.5. The comparative results between the algorithms is show in Table 4.6.

In this example, CRISSiS using $L_2 = 1$ is not able to recover the original machine . It creates a PFSA with 2 states (Figure 4.5) which generates sequences fairly different from the original, as after a small transient, it outputs sequences of just 1's. On the other hand, by using CRISSiS with L_2 larger than 3 and $\aleph_1$ 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.

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

	D-Markov		**CRISSiS**	$\aleph_1$**/CRISSiS**
	$D = 3$	$D = 4$	$L_2 = 1$	$L = 4/L_2 = 3$
# of States	7	11	2	6
h_{10}	0.5341	0.3344	$1.4427 \cdot 10^{-7}$	0.3344
D_{10}	1.1980	$4.0499 \cdot 10^{-6}$	43.6556	$5.6969 \cdot 10^{-5}$
Φ_{10}	$2.0005 \cdot 10^{-3}$	$4.6072 \cdot 10^{-4}$	2.6505	$9.3745 \cdot 10^{-4}$

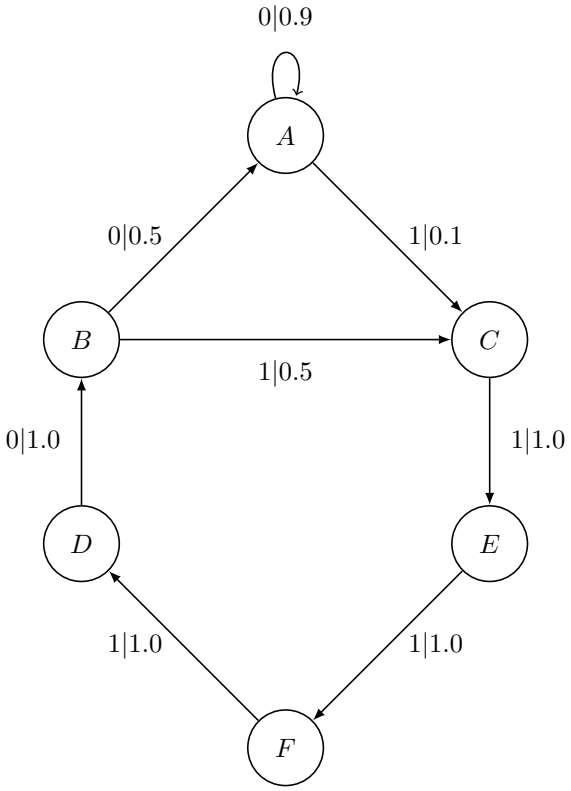

Figura 4.4: *A Six-State PFSA.*

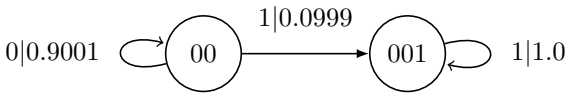

Figura 4.5: *The Recovered Six-State PFSA by CRISSiS with* $L_2 = 1$.

The results for this system show a type of system where CRISSiS might nor perform so well. Starting from a single synchronization word and as many states have equal morphs and their paths will only be different after 3 steps, L_2 needs to be at least 3 to be able to recover the original PFSA and as CRISSiS' complexity is exponential on L_2 this means a considerable penalty in performance. As $\aleph_1$ 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 and our algorithm are able to estimate the PFSA with good precision.

4.2.4 Maximum Entropy *(d,k)*-Constrained Code

As seen in [**?**], 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

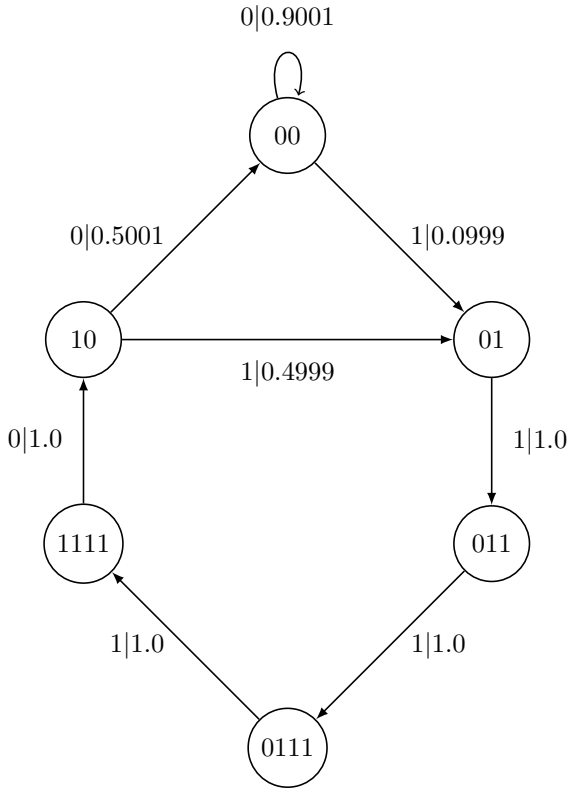

Figura 4.6: *The Recovered Six-State PFSA by our algorithm.*

that at least *d* 1's are generated between occurrences of 0's and that after *k* consecutive 1's, a 0 has to appear. 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 a higher L_2 to obtain a correct estimate, implying a cubic complexity. When L_2 is 3, CRISSiS recovers the same PFSA as our algorithm (shown in Figure 4.8) and have a similar performance. On the other

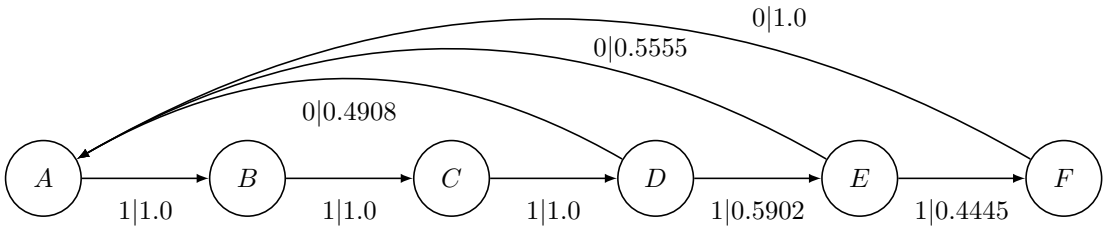

Figura 4.7: *The Maximum Entropy (3,5)-Constrained Code PFSA.*

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

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

	D-Markov		**CRISSiS**		$\aleph_1$
D/L/L_1 and L_2	$D = 4$	$D = 5$	$L_2 = 1$	$L_2 = 3$	$L = 6$
# of States	5	7	1	6	6
h_{10}	0.3575	0.3218	$1.4427 \cdot 10^{-7}$	0.3218	0.3218
D_{10}	0.1793	$7.0139 \cdot 10^{-7}$	45.5434	$5.9715 \cdot 10^{-7}$	$2.3766\dot{1}0^{-6}$
Φ_{10}	$5.0521 \cdot 10^{-3}$	$2.8001 \cdot 10^{-4}$	1.5165	$9.3656 \cdot 10^{-5}$	$1.5380 \cdot 10^{-4}$

hand, when L_2 is lower than 3, CRISSiS creates a PFSA with one state that continually generates the symbol 1, which performs poorly. 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.

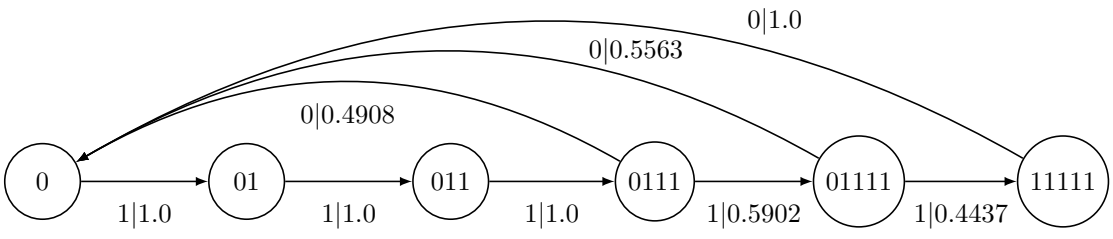

Figura 4.8: *The Maximum Entropy (3,5)-Constrained Code PFSA recovered by our algoritum and by CRISSiS.*

CAPÍTULO 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 [**?**]:

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

which shows chaotic behavior when the *r* parameter is approximately 3.57. As in [**?**], 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 $x_k \leq 0.67$ were mapped to 0; when $0.67 < x_k \leq 0.79$, 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

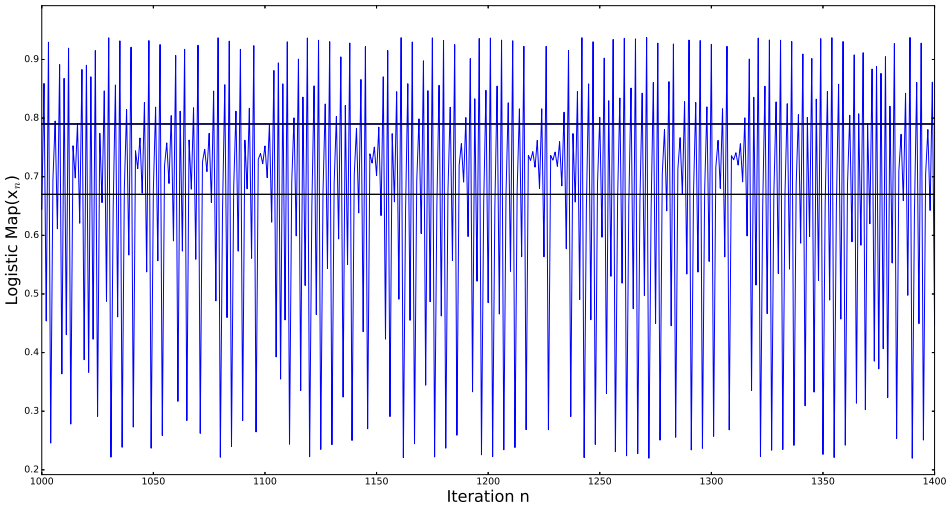

Figura 5.1: *Part of the Logistic Map generated by Equation 5.1 with x_0 = 0.5 and r = 3.75.*

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 [**?**], 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

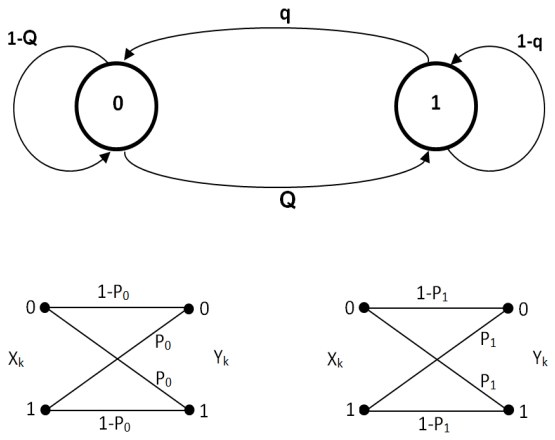

Figura 5.2: *The Gilbert-Elliott Channel.*

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 [**?**], 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.

Capítulo 6

TODO

TODO

APÊNDICE A

TODO

TODO

Apêndice B

TODO

SOBRE O AUTOR

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.

Endereço: 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] S.H. Strogatz. *Nonlinear Dynamics and Chaos with Applications to Physics, Biology, Chemistry, and Engineering*. Studies in Nonlinearity Series. Westview Press, 2001.

[2] K.T. Alligood, T.D. Sauer, and J.A. Yorke. *Chaos: An Introduction to Dynamical Systems*. New York, NY, 1997.

[3] S. Hayes, C. Grebogi, and E. Ott. Communicating with chaos. *Phys. Rev. Lett.*, 70:3031–3034, May 1993.

[4] C. Jianyong, Z Junwei, and Kwok-Wo W. A modified chaos-based joint compression and encryption scheme. *Circuits and Systems II: Express Briefs, IEEE Transactions on Circuit and Systems*, 58(2):110–114, February 2011.

[5] A. Masmoudi and W. Puech. Lossless chaos-based crypto-compression scheme for image protection. *IET Image Processing*, 8(12):671–686, 2014.

[6] F.C.M. Lau and C.K. Tse. *Chaos-Based Digital Communication Systems*. Engineering online library. Springer, 2010.

[7] M. Eisencraft, R. Attux, and R. Suyama. *Chaotic Signals in Digital Communications*. Electrical Engineering & Applied Signal Processing Series. Taylor & Francis, 2013.

[8] P. Stavroulakis. *Chaos Applications in Telecommunications*. Taylor & Francis, 2005.

[9] L. Kocarev and S. Lian. *Chaos-based Cryptography: Theory, Algorithms and Applications*. Studies in Computational Intelligence. Springer, 2011.

[10] F. Dachselt and W. Schwarz. Chaos and cryptography. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 48(12):1498–1509, February 2001.

[11] L. Kocarev, J. Makraduli, and P. Amato. Public-key encryption based on chebyshev polynomials. *Circuits, Systems and Signal Processing*, 24(5):497–517, October 2005.

[12] T. Stojanovski and L. Kocarev. Chaos-based random number generators-part I: analysis [cryptography]. *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on Communication*, 48(3):281–288, March 2001.

[13] L. De Micco, H. A. Larrondo, A. Plastino, and O. A. Rosso. Quantifiers for randomness of chaotic pseudo-random number generators. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 367(1901):3281–3296, August 2009.

[14] L. De Micco, C.M. González, H.A. Larrondo, M.T. Martin, A. Plastino, and O.A. Rosso. Randomizing nonlinear maps via symbolic dynamics. *Physica A: Statistical Mechanics and its Applications*, 387(14):3373 – 3383, June 2008.

[15] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. John Wiley & Sons, 2006.

[16] A. Beirami and H. Nejati. A framework for investigating the performance of chaotic-map truly random number generators. *IEEE Transactions on ciucuits and systems -II: Express Briefs*, 60(7):446 – 450, July 2013.

[17] H. Poincare. *The Value of Science: Essential Writings of Henri Poincare*. Modern Library Science, October 2001.

[18] J. Gleick. *Chaos: Making a New Science*. Penguin Books, 1987.

[19] R.M. May. Biological populations with nonoverlapping generations: Stable points, stable cycles and chaos. *Science*, 186(4164):645–647, November 1974.

[20] J. Wisdom and J. Stanton. The chaotic rotation of hyperion. *Physics Letters A*, 58(2):137–152, May 1984.

[21] A. Babloyantz and J.M. Salazar. Evidence of chaotic dynamics of brain activity during the sleep cycle. *Physics Letters A*, 111(3):152–156, September 1985.

[22] M. Kennedy, R. Rovatti, and G. Setti. *Chaotic Electronics In Telecommunications*. CRC Press, FL (USA), June 2000.

[23] M.P. Kennedy and L.O. Chua. Van der pol and chaos. *IEEE Trans. on Circuits and Systems,*, 33(10):974–980, October 1986.

[24] L.O. Chua. The genesis of chua's circuit. *Archiv fur Elektronik und Ubertragungstechnik*, 46(4):250–257, 1992.

[25] L.M. Pecora and T. L. Carroll. Synchronization in chaotic systems. *Physical Review Letters*, 64(8):821–825, February 1990.

[26] E. Ott, C. Grebogi, and J.A. Yorke. Controlling chaos. *Physical Review Letters*, 64:1196–1199, March 1990.

[27] G.S. Vernam. Cipher printing telegraph systems for secret wire and radio telegraphic communications. *Transactions of the American Institute of Electrical Engineers*, pages 295–301, 1926.

[28] D. Chaves, C. Souza, and C. Pimentel. A new map for chaotic communication. *International telecommunication Symposium (ITS 2014)*, pages 1 – 5, August 2014.

[29] L.A. Aguirre. *Introdução à Identificação de Sistemas: Técnicas Lineares e Não-Lineares Aplicadas a Sistemas Reais*. UFMG, 2007.

[30] P. Glendinning. *Stability, Instability and Chaos: An Introduction to the Theory of Nonlinear Differential Equations.* Cambridge Texts in Applied Mathematics, December 1994.

[31] D.O. Pederson and K. Mayaram. *Analog Integrated Circuits for Communication: Principles, Simulation, and Design*. Kluwer Academic Publishers, 1991.

[32] P. Dudek and V.D. Juncu. Compact discrete-time chaos generator circuit. *Electronics Letters*, 39(20):1431–1432, October 2003.

[33] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo. Statistical test suite for random and pseudo random number generators for cryptographic applications. *Special Publication 800-22 Revision 1a, National Institute of Standards and Technology*, April 2010.

[34] G. Marsaglia. Diehard statistical tests. 1995.

[35] D.E. Knuth. *The Art of Computer Programming: Seminumerical algorithms*. Addison-Wesley series in computer science and information processing. Addison-Wesley, 1981.

[36] W.G. Solomon and G Guang. *Signal Design for Good Correlation: For Wireless Communication, Cryptography, and Radar*. Cambridge University Press, July 2004.

[37] C.M. Lua and C.K Tse. *Chaos-Based Digital Communication Systems Operating Principles, Analysis Methods, and Performance Evaluation*. Signals and Communication Technology. Springer; 2003 edition, June 2003.

[38] C. Paar and J. Pelzl. *Understanding Cryptography, A Textbook for Students and Practitioners*. Springer, 2010.

[39] J.L. Massey. *Cryptography: Fundamentals and Applications. Copies of transparencies*. Advances Technology Seminars, 1997.