

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

2.1	A graph with $Q = \{A, B, C\}$ and $\Sigma = \{0, 1\}$ . . . . .	12
2.2	An example of a graph that is not minimal. . . . .	17
2.3	Initial partition of the graph in Figure 2.2. . . . .	17
2.4	Application of Moore algorithm to the initial partition in Figure 2.3. . . . .	18
2.5	A PFSA with the same graph of Figure 2.1. . . . .	20
2.6	A D-Markov machine with sequence $S$ and $D = 2$ . . . . .	22
2.7	The Tri-Shift PFSA. . . . .	25
2.8	Tree with 00 at its root, $Q = \{00\}$ . . . . .	27
2.9	Second iteration of three, $Q = \{00, 001\}$ . . . . .	27
2.10	Third iteration of the three, $Q = \{00, 001, 0010\}$ . . . . .	27
2.11	Recovered Tri-Shift topology. . . . .	28
3.1	Example of a rooted tree with probabilities. . . . .	30
3.2	Example of binary RTP with $L = 3$ . . . . .	32
3.3	Input Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example. . . . .	43
3.4	Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example after the first iteration. . . . .	44
3.5	Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example after the second iteration. . . . .	45
3.6	Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example after the third and fourth iterations. . . . .	46
3.7	Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example after the fifth iteration. . . . .	47
3.8	Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example after the sixth iteration. . . . .	48
3.9	Rooted Tree with Probabilities $\mathcal{S}$ for the Tri-Shift Example after the seventh iteration. . . . .	49
4.1	A PFSA of a Ternary Even-Shift. . . . .	53
4.2	PFSA of a Ternary Even-Shift generated by the $\aleph_1$ algorithm and by CRISSiS. . . . .	54
4.3	The Tri-Shift PFSA. . . . .	54
4.4	The Tri-Shift PFSA generated by our algorithm and by CRISSiS. . . . .	55
4.5	A Six-State PFSA. . . . .	56
4.6	The Recovered Six-State PFSA by our algorithm. . . . .	58
4.7	The Maximum Entropy (3,5)-Constrained Code PFSA. . . . .	58
4.8	The Maximum Entropy (3,5)-Constrained Code PFSA recovered by $\aleph$ algorithm and by CRISSiS. . . . .	59

5.1	Part of the Logistic Map generated by Equation 5.1 with $x_0 = 0.5$ and $r = 3.75$ . . . .	61
5.2	The Gilbert-Elliott Channel. . . . .	62

# LISTA DE TABELAS

2.1	Probabilities of words of length up to 3 for the binary sequence $S$ .	22
2.2	Probabilities of words generated by the Tri-Shift up to length 10000.	26
4.1	Synchronization Words for Ternary Even Shift.	53
4.2	Results for Ternary Even Shift.	54
4.3	Synchronization Words for Tri-Shift.	55
4.4	Results for the Tri-Shift.	55
4.5	Synchronization Words for the Six-State PFSA.	56
4.6	Results for the Six-State PFSA.	57
4.7	Synchronization Words for the Maximum Entropy (3,5)-Constrained Code.	57
4.8	Results for the Maximum Entropy (3,5)-Constrained Code PFSA.	58
5.1	Synchronization Words for the Logistic Map Ternary Sequence.	61



# SUMÁRIO

<b>1</b>	<b>INTRODUCTION</b>	<b>10</b>
<b>2</b>	<b>PRELIMINARIES ON GRAPHS AND PROBABILISTIC FINITE STATE AUTOMATA</b>	<b>11</b>
<b>2.1</b>	<b>Sequences of Discrete Symbols . . . . .</b>	<b>11</b>
<b>2.2</b>	<b>Graphs . . . . .</b>	<b>12</b>
<b>2.3</b>	<b>Graph Minimization . . . . .</b>	<b>13</b>
2.3.1	Moore Algorithm . . . . .	15
2.3.2	Hopcroft's Algorithm . . . . .	18
<b>2.4</b>	<b>Probabilistic Finite State Automata . . . . .</b>	<b>19</b>
2.4.1	Initial Partition for PFSA . . . . .	20
<b>2.5</b>	<b>Consolidated Algorithms . . . . .</b>	<b>21</b>
2.5.1	D-Markov Machines . . . . .	21
2.5.2	CRISSiS . . . . .	21
<b>3</b>	<b>ALGORITHM DESCRIPTION</b>	<b>29</b>
<b>3.1</b>	<b>A New Algorithm for Finding Synchronization Words . . . . .</b>	<b>29</b>
3.1.1	An Example . . . . .	34
<b>3.2</b>	<b>PFSA Construction . . . . .</b>	<b>37</b>
3.2.1	Termination Criteria . . . . .	37
3.2.2	$\aleph$ Algorithm . . . . .	38
<b>3.3</b>	<b>Time Complexity . . . . .</b>	<b>40</b>
3.3.1	RTP Construction . . . . .	40
3.3.2	Synchronization Word Search . . . . .	40
3.3.3	Termination . . . . .	42
3.3.4	PFSA Construction . . . . .	42
<b>4</b>	<b>RESULTS</b>	<b>50</b>
<b>4.1</b>	<b>Evaluation Metrics . . . . .</b>	<b>50</b>
4.1.1	Entropy Rate . . . . .	50
4.1.2	Kullback-Leibler Divergence . . . . .	51
4.1.3	$\Phi$ -Metric . . . . .	51
<b>4.2</b>	<b>Construction of PFSA for Dynamic Systems . . . . .</b>	<b>52</b>

4.2.1	Ternary Even Shift . . . . .	52
4.2.2	Tri-Shift . . . . .	53
4.2.3	A Six State PFSA . . . . .	54
4.2.4	Maximum Entropy $(d,k)$ -Constrained Code . . . . .	57
<b>5</b>	<b>APPLICATIONS</b>	<b>60</b>
5.0.1	Logistic Map . . . . .	60
5.0.2	Gilbert-Elliot Channel . . . . .	61
<b>6</b>	<b>TODO</b>	<b>64</b>
<b>Apêndice A</b>	<b>TODO</b>	<b>65</b>
<b>Apêndice B</b>	<b>TODO</b>	<b>66</b>

# **CAPÍTULO 1**

## **INTRODUCTION**

TODO.

## CAPÍTULO 2

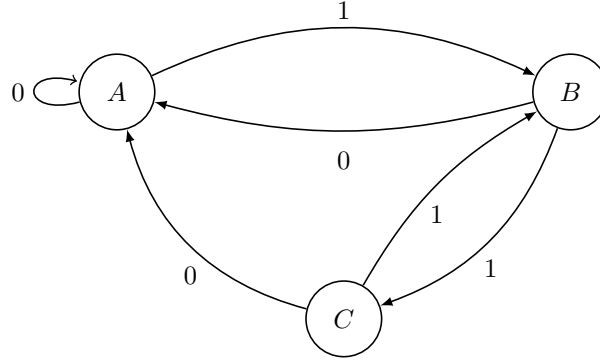
# PRELIMINARIES ON GRAPHS AND PROBABILISTIC FINITE STATE AUTOMATA

**I**N this chapter we revise concepts from graphs and 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 well known 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  $\Sigma$  is called a word and its length is denoted by  $|u|$ . The empty word  $\varepsilon$  is defined as the sequence with length 0. The set of all possible words of length  $n$  symbols from  $\Sigma$  is  $\Sigma^n$  and the set of all sequences of symbols from  $\Sigma$  with all possible lengths, including the empty sequence  $\varepsilon$ , is  $\Sigma^*$ .

Two words  $u$  and  $v \in \Sigma^*$  can be concatenated to form a sequence  $uv$ . For example, using a binary alphabet  $\Sigma = \{0, 1\}$ , the concatenation of  $u = 1010$  and  $v = 111$  is  $uv = 1010111$ . Note that  $|uv| = |u| + |v|$ . 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  $\varepsilon$  is a neutral element for concatenation. That is,  $\varepsilon u = u\varepsilon = u$ . This means that  $\Sigma^*$  with the operation of concatenation is a Monoid, as it is a set with an associative operation with an identity element.



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

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

## 2.2 Graphs

### Definition 2.1 – Graph

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

- ▷  $Q$  is a finite set of states with cardinality  $|Q|$ ;
- ▷  $\Sigma$  is a finite alphabet with cardinality  $|\Sigma|$ ;
- ▷  $\delta$  is the state transition function  $Q \times \Sigma \rightarrow Q$ ;

□

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

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

**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  $Q$ . □

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

**Definition 2.3 – Language of a Graph**

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

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

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

**2.3 Graph Minimization**

Suppose that two graphs  $G_1 = \{Q_1, \Sigma, \delta_1\}$  and  $G_2 = \{Q_2, \Sigma, \delta_2\}$  with  $|Q_1| \neq |Q_2|$  and are capable of generating the same language. It is desirable to use a graph with fewer states as it can be represented with a lower memory requirement. This is called a minimal graph.

**Definition 2.4 – Minimal Graph**

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

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

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

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

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

Given a graph  $G$  there are two main algorithms used to obtain a minimal graph from it: Moore and Hopcroft [? ]. 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.*  $\square$

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

Given partitions  $\mathcal{P}$  and  $\mathcal{Q}$  of  $E$ ,  $\mathcal{U} = \mathcal{P} \wedge \mathcal{Q}$  denotes the coarsest partition which refines  $\mathcal{P}$  and  $\mathcal{Q}$ . The elements of  $\mathcal{U}$  are non-empty sets  $P \cap Q$ , such that  $P \in \mathcal{P}$  and  $Q \in \mathcal{Q}$ . The notation is extended for multiple sets as  $\mathcal{U} = \mathcal{P}_1 \wedge \mathcal{P}_2 \wedge \dots \wedge \mathcal{P}_n = \bigwedge_{j=1}^n \mathcal{P}_j$ . When  $n = 0$ ,  $\mathcal{P}$  is the universal partition comprised of just  $E$  and it is the neutral element for the  $\wedge$ -operation.

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

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

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

is the partition composed of two non-empty subsets:

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

and

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

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

### Proposition 2.1

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

#### 2.3.1 Moore Algorithm

The first minimization algorithm is the Moore algorithm. It is based on the idea of taking an initial partition with a very wide criteria and then refining it until the Nerode equivalence classes are obtained. The outline of the algorithm is shown in Algorithm 1.

Given a graph  $G = (Q, \Sigma, \delta)$  and the initial partition  $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_n\}$ , the set  $L_q^{(h)}$  is defined as:

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

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

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

This equivalence relation states that two states are equivalent if they generate the same words of length up to  $h$  that reach a state in  $\mathcal{P}_j$ . The depth of the Moore algorithm on a graph  $G$  is

---

#### Algorithm 1 Moore( $G$ )

---

- 1:  $\mathcal{P} \leftarrow \text{InitialPartition}(G)$
  - 2: **repeat**
  - 3:    $\mathcal{P}' \leftarrow \mathcal{P}$
  - 4:   **for all**  $\sigma \in \Sigma$  **do**
  - 5:      $\mathcal{P}_\sigma \leftarrow \bigwedge_{P \in \mathcal{P}} (P, \sigma)|Q$
  - 6:    $\mathcal{P} \leftarrow \mathcal{P} \wedge \bigwedge_{\sigma \in \Sigma} \mathcal{P}_\sigma$
  - 7: **until**  $\mathcal{P} = \mathcal{P}'$
-



the integer  $h$  such that the Moore equivalence  $\equiv_h$  becomes equal to the Nerode equivalence  $\equiv$  and it is dependent only on the language of the graph. The depth is the smallest  $h$  such that  $\equiv_h$  equals  $\equiv_{h+1}$ , which leads to an algorithm that computes successive Moore equivalences until it finds two consecutive equivalences that are equal, making it halt.

**Proposition 2.2**

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

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

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

**Proposition 2.3**

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

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

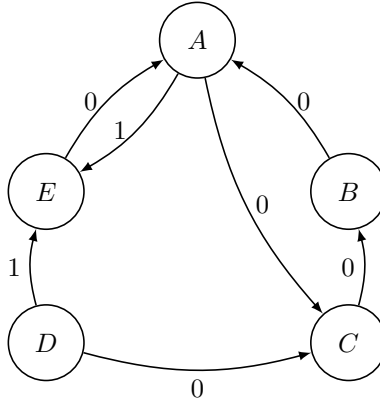
This previous computation is done in Algorithm 1 in which the loop refines the current partition. As will be explored in this work, the initial partition can be created with different criteria. For a graph, it is done by grouping together states in  $Q$  which have outgoing edges with the same labels, but another criterion is used in the probabilistic case (Section 2.4.1).

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

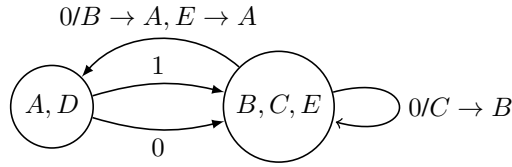
**An Example**

To illustrate how the Moore algorithm works, this example will apply it to the graph of Figure 2.2, which has  $Q = \{A, B, C, D, E\}$ ,  $\Sigma = \{0, 1\}$  and it is not minimal. The first step is to create the initial partition based on the criteria of grouping states together in equivalence classes if they have the same amount of outgoing edges with the same labels. In Figure 2.2, states  $A$  and  $D$  have outgoing edges labeled with 0 and 1 while  $B, C$  and  $E$  have only an outgoing edge labeled with 0. Thus, the initial partition has two equivalence classes,  $\mathcal{P} = \{\{A, D\}, \{B, C, E\}\}$  shown in Figure 2.3.

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



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



**Figure 2.3:** Initial partition of the graph in Figure 2.2.

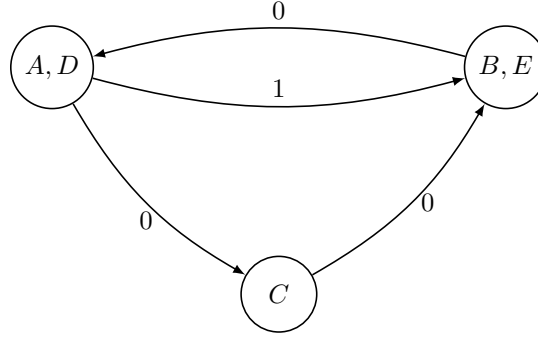
to  $Q$ . First, take  $(\{A, D\}, 0)|Q$ :  $Q \cap 0^{-1}\{A, D\} = \{B, E\}$  and  $Q \setminus 0^{-1}\{A, D\} = \{A, C, D\}$ . The same process is repeated for the splitter  $(\{B, C, E\}, 0)$ :  $Q \cap 0^{-1}\{B, C, E\} = \{A, C, D\}$  and  $Q \setminus 0^{-1}\{B, C, E\} = \{B, E\}$ .  $\mathcal{P}_0$  is then the coarsest partition between  $(\{A, D\}, 0)|Q = \{\{B, E\}, \{A, C, D\}\}$  and  $(\{B, C, E\}, 0)|Q = \{\{A, C, D\}, \{B, E\}\}$ . Thus  $\mathcal{P}_0 = \{\{A, C, D\}, \{B, E\}\}$ .

This process is repeated to obtain  $\mathcal{P}_1$ .  $(\{A, D\}, 1)|Q$ :  $Q \cap 1^{-1}\{A, D\} = \emptyset$  and  $Q \setminus 1^{-1}\{A, D\} = \{A, B, C, D, E\}$  and  $(\{B, C, E\}, 1)|Q$ :  $Q \cap 1^{-1}\{B, C, E\} = \{A, D\}$  and  $Q \setminus 1^{-1}\{B, C, E\} = \{B, C, E\}$ . The coarsest partition between  $\{\{A, B, C, D, E\}\}$  and  $\{\{A, D\}, \{B, C, E\}\}$  is  $\{\{A, D\}, \{B, C, E\}\} = \mathcal{P}_1$ .

The next step is to take the coarsest partition between  $\mathcal{P}_0$  and  $\mathcal{P}_1$ , which is  $\{\{A, D\}, \{C\}, \{B, E\}\}$ . Then the coarsest partition between this result and the current  $\mathcal{P}$  is taken, which leaves it unchanged. This result is then stored as the new partition  $\mathcal{P}$  and it is shown in Figure 2.4.

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

First, the result of the splitters for 0 are  $(\{A, D\}, 0)|Q = \{\{B, E\}, \{A, C, D\}\}$ ,  $(\{C\}, 0)|Q = \{\{A, D\}, \{B, C, E\}\}$  and  $(\{B, E\}, 0)|Q = \{\{C\}, \{A, B, D, E\}\}$  which results in  $\mathcal{P}_0 = \{\{B, E\}, \{A, D\}, \{C\}\}$ . Similarly,  $(\{A, D\}, 1)|Q = \{\{A, B, C, D, E\}\}$ ,  $(\{C\}, 1)|Q = \{\{A, B, C, D, E\}\}$  and  $(\{B, E\}, 1)|Q = \{\{A, D\}, \{B, C, E\}\}$  and results in  $\mathcal{P}_1 = \{\{A, D\}, \{B, C, E\}\}$ . The coarsest partition between  $\mathcal{P}_0$  and  $\mathcal{P}_1$  is  $\{\{A, D\}, \{C\}, \{B, E\}\}$  which remains unchanged when its coarsest partition is taken with



**Figure 2.4:** Application of Moore algorithm to the initial partition in Figure 2.3.

$\mathcal{P}$ . This result is then stored in  $\mathcal{P}$  and it is equal to  $\mathcal{P}'$ , which means the algorithm has converged and the minimal graph is shown in Figure 2.4.

### 2.3.2 Hopcroft's Algorithm

---

**Algorithm 2** Hopcroft( $G$ )

---

```

1:  $\mathcal{P} \leftarrow \text{InitialPartition}(G)$ 
2:  $\mathcal{W} \leftarrow \emptyset$ 
3: for all  $\sigma \in \Sigma$  do
4:   Append( $(\min(F, F^c, \sigma), \mathcal{W})$ )
5:   while  $\mathcal{W} \neq \emptyset$  do
6:      $(W, \sigma) \leftarrow \text{TakeSome}(\mathcal{W})$ 
7:     for each  $P \in \mathcal{P}$  which is split by  $(W, \sigma)$  do
8:        $P', P'' \leftarrow (W, \sigma) \mid P$  Replace  $P$  by  $P'$  and  $P''$  in  $\mathcal{P}$ 
9:       for all  $\tau \in \Sigma$  do
10:        if  $(P, \tau) \in \mathcal{W}$  then
11:          Replace  $(P, \tau)$  by  $(P', \tau)$  and  $(P'', \tau)$  in  $\mathcal{W}$ 
12:        else
13:          Append( $(\min(P', P'', \tau), \mathcal{W})$ )

```

---

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

For each iteration of the loop, one splitter  $(W, \sigma)$  is taken from the waiting set. It then checks

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

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

## 2.4 Probabilistic Finite State Automata

### Definition 2.6 – Probabilistic Finite State Automata

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

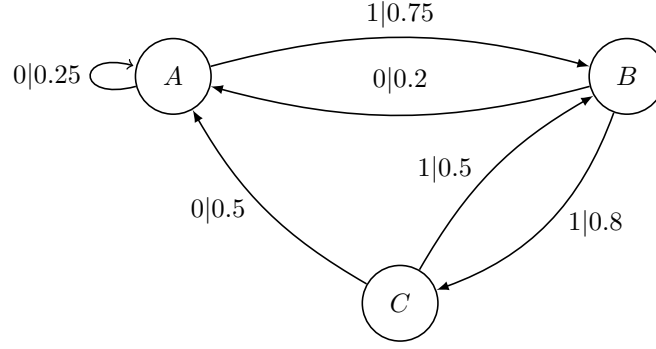
### Definition 2.7 – Morph

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

A PFSA can be drawn with its graph with each outgoing edge labeled with a symbol and the probability  $\pi(q, \sigma)$  associated with that transition. An example of a PFSA is shown in Figure 2.5. for which  $Q = \{A, B, C\}$ ,  $\Sigma = \{0, 1\}$ . It is the same graph from Figure 2.1 with probabilities associated to its edges to create a PFSA.

Given a PFSA  $\{G, \pi\}$ , there is a probability associated with each word  $\omega \in \Sigma^*$  that can be generated from each state of  $G$ . From Figure 2.5, starting at the state  $A$ , it is possible to generate the word  $\omega = 1011001$  (as  $\delta^*(A, \omega) = B$ ) by taking a path going to states  $B, A, B, C, A, A$  and  $B$  and concatenating the labels of the path from each of these transitions. By multiplying the probabilities of these edges, it is seen that  $\Pr(\omega|A) = 0.75 \times 0.2 \times 0.75 \times 0.8 \times 0.5 \times 0.25 \times 0.75 = 0.0084375$ .

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



**Figure 2.5:** A PFSA with the same graph of Figure 2.1.

### Definition 2.8 – PFSA Synchronization Word

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

$$\Pr(u|w) = \Pr(u|vw). \quad (2.1)$$

□

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.

The solution is to use (2.2):

$$\Pr(wv) = \Pr(uwv), \forall u \in \cup_{i=1}^{L_1} \Sigma^i, \forall v \in \cup_{j=1}^{L_2} \Sigma^j \quad (2.2)$$

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

A synchronization words is a good starting point to model a system from its output sequence because the probability of its occurrence does not depend on what 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 minimization algorithm (such as Moore or Hopcroft) on a PFSA graph, the following criterion is used to create the initial partition:

### Definition 2.9

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

## 2.5 Consolidated Algorithms

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

### 2.5.1 D-Markov Machines

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

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

To construct a D-Markov Machine, first all symbol blocks of length  $D$  are taken as the states in the set  $Q$  and their transition probabilities can be computed by frequency counting. Consider state  $q$  labeled as  $q = \sigma_1 \sigma_2 \dots \sigma_D$  with  $\sigma_n \in \Sigma$  for  $n = 1, 2, \dots, D$ . There is a transition from  $q$  to  $\sigma_2 \dots \sigma_D \tau$  for  $\tau \in \Sigma$  ( $\delta(q, \tau) = q'$ ) with probability:

$$\Pr(\tau|q) = \frac{\Pr(q')}{\Pr(q)}, \quad (2.3)$$

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

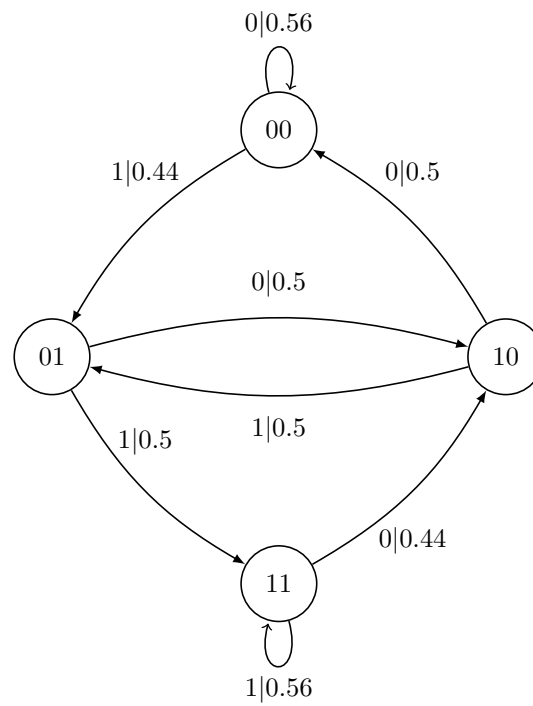
For example, considering a binary sequence  $S$  of length 100 with the probabilities of words of length  $\ell \leq 3$  shown in Table 2.1. To build a 2-Markov Machine, the states are 00, 01, 10 and 11. Using Equation (2.3), the D-Markov machine shown in Figure 2.6 is built.

### 2.5.2 CRISSiS

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

**Tabela 2.1:** Probabilities of words of length up to 3 for the binary sequence  $S$ .

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

**Figura 2.6:** A  $D$ -Markov machine with sequence  $S$  and  $D = 2$ .

---

**Algorithm 3** CRISSiS
 

---

```

1: Inputs: Symbolic string  $S, \Sigma, L_1, L_2$ , significance level  $\alpha$ 
2: Outputs: PFSA  $\hat{P} = \{G, \pi\}$ 
3: ## Identification of Shortest Synchronization Word:
4:  $\omega_{syn} \leftarrow \text{null}$ 
5:  $d \leftarrow 0$ 
6: while  $\omega_{syn}$  is null do
7:    $\Omega \leftarrow \Sigma^d$ 
8:   for all  $\omega \in \Omega$  do
9:     if (isSynString( $\omega, L_1$ )) then
10:        $\omega_{syn} \leftarrow \omega$ 
11:       break
12:    $d \leftarrow d + 1$ 
13: ## Recursive Identification of States:
14:  $Q \leftarrow \{\omega_{syn}\}$ 
15:  $\tilde{Q} \leftarrow \{\}$ 
16: Add  $\omega_{syn}\sigma_i$  to  $\tilde{Q}$  and  $\delta(\omega_{syn}, \sigma_i) = \omega_{syn}\sigma_i \forall \sigma \in \Sigma$ 
17: for all  $\omega \in \tilde{Q}$  do
18:   if  $\omega$  occurs in  $X$  then
19:      $\omega^* \leftarrow \text{matchStates}(\omega, Q, L_2)$ 
20:     if  $\omega^*$  is null then
21:       Add  $\omega$  to  $Q$ 
22:       Add  $\omega\sigma_i$  to  $\tilde{Q}$  and  $\delta(\omega_{syn}, \sigma_i) = \omega_{syn}\sigma_i \forall \sigma \in \Sigma$ 
23:     else
24: ## Estimation of Morph Probabilities:
25: Find  $k$  such that  $S[k]$  is the symbol after the first occurrence of  $\omega_{syn}$  in  $S$ 
26: Initialize  $\pi$  to zero
27:  $state \leftarrow \omega_{syn}$ 
28: for all  $i \geq k$  in  $S$  do
29:    $\pi(state, S[i]) \leftarrow \pi(state, S[i]) + 1$ 
30:    $state \leftarrow \delta(state, S[i])$ 
31: Normalize  $\pi$  for each state

```

---



### Identification of Shortest Synchronization Word

Using the definition of a synchronization word given in (2.2), CRISSiS uses brute force to find the shortest synchronization word with fixed parameters  $L_1$  and  $L_2$ . This is shown in Algorithm 4 where each state morph is checked with the morph of its extensions up to a length  $L_2$ . If all statistical tests are positive for a given word  $\omega$ , it is returned as the synchronization word  $\omega_{syn}$  to be used.

### Recursive Identification of States

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

$$\Pr(v|q_1) = \Pr(v|q_2), \forall v \in \Sigma^* \quad (2.4)$$

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

$$\Pr(v|q_1) = \Pr(v|q_2), \forall v \in \Sigma^d, d = 1, \dots, L_2. \quad (2.5)$$

If two states pass the statistical test using (2.5), they are considered to be statistically the same. Strings  $q_1$  and  $q_2$  need to be synchronizing in order to use (2.5). If  $\omega$  is a synchronization word for some  $q_i \in Q$ , then  $\omega\tau$  is also a synchronization word for  $q_j = \delta(q_i, \tau)$ .

The next procedure starts by letting  $Q$  be the set of states to be discovered for the PFSA and it is initialized containing only the state defined by the synchronization word  $\omega_{syn}$  found in the first step. Then, a tree is constructed using  $\omega_{syn}$  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 is tested using a statistical test with confidence level  $\alpha$  with each of the states in  $Q$ . If a match is found, the child state is removed and its parent  $\sigma$ -transition should be connected to the matching state. If it does

---

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

---

```

1: Outputs: true or false
2: for  $D = 0$  to  $L_1$  do
3:   for all  $u \in \Sigma^D$  do
4:     for  $d = 0$  to  $L_2$  do
5:       for all  $v \in \Sigma^d$  do
6:         if  $statisticalTest(\omega v, u\omega v, \alpha) = \text{False}$  then
7:           return False
8: return true

```

---

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

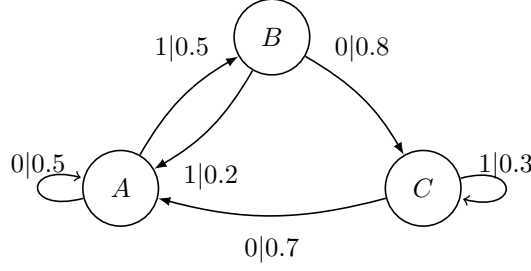
---

```

1: for all  $q \in Q$  do
2:   for  $d = 0$  to  $L_2$  do
3:     for all  $v \in \Sigma^d$  do
4:       if  $\text{statisticalTest}(\omega v, qv, \alpha) = \text{False}$  then
5:         return  $q$ 
6: return null

```

---

**Figure 2.7:** The Tri-Shift PFSA.

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 a finite PFSA, this procedure is guaranteed to terminate.

**Estimation of Morph Probabilities**

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

**Example**

The PFSA in Figure 2.7, which is called Tri-Shift in this work, is presented in [? ]. It is synchronizable and works over a binary alphabet. It is used in this example to generate a string  $S$  of length 10000. Table 2.2 gives the estimated probabilities of subsequences occurring in  $S$ . In this example,  $L_1 = L_2 = 1$ .

First, the synchronization word needs to be found. States 0, 1 and so on are checked with (2.2). Starting by 0,  $\Pr(0|0) = 0.5607$  is not equal to  $\Pr(1|0) = \Pr(01) = 0.4393$  which means they do not pass the  $\chi^2$  test. Then, the state 1 is tested, which also fails ( $\Pr(0|1) = 0.7387 \neq 0.2613 = \Pr(1|1)$ ).

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

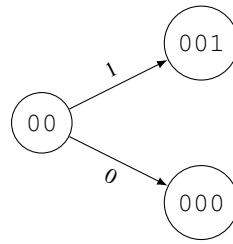
$\ell = 1$	Prob.	$\ell = 2$	Prob.	$\ell = 3$	Prob.	$\ell = 4$	Prob.	$\ell \geq 5$	Prob.
0	0.62711	00	0.35164	000	0.17565	0000	0.08673	00100	0.09881
1	0.37291	01	0.27546	001	0.17599	0001	0.08892	00101	0.04181
		10	0.27546	010	0.21451	0010	0.14062	001000	0.0499
		11	0.09745	011	0.06094	0011	0.03536	001001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

For state 00, the probabilities are relatively close ( $\Pr(0|00) = 0.5 = \Pr(001) = \Pr(1|00)$ ) and it passes the test, giving 00 the status of synchronization word.

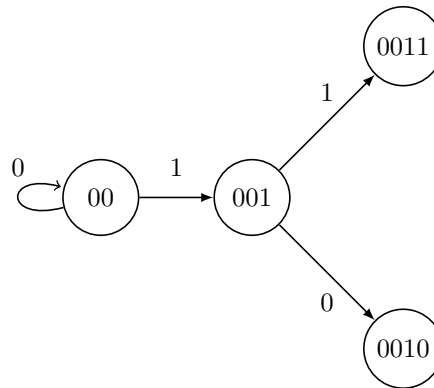
The second step starts by defining the synchronization word state 00 adding it to  $Q$  and splitting it into two candidate states, 000 and 001 (Figure 2.8). Each candidate has its morphs compared to that of 00, which is the only state in  $Q$ , with (2.5).  $\mathcal{V}(000) = [0.494, 0.506]$  is considerably close to  $\mathcal{V}(00) = [0.500, 0.500]$ , so they pass the statistical test and 00 and 000 are considered to be 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,  $\mathcal{V}(001) = [0.800, 0.200]$  is considerably different from  $\mathcal{V}(00)$ , therefore it is considered a state and added to  $Q$  (which now becomes  $\{00, 001\}$ ) and then it is split into two new candidates (Figure 2.9).

The same procedure is then repeated for the candidates 0010 and 0011.  $\mathcal{V}(0010) = [0.703, 0.297]$  is different from both 00 and 001, therefore it is a new state, it is added to  $Q$  and split into the new candidates 00100 and 00101.  $\mathcal{V}(0011) = [0.500, 0.500]$  passes the test with  $\mathcal{V}(00)$ , which means that 0011 is removed and the edge from 001 to 0011 goes back to 00. This leads to the configuration in Figure 2.10, with  $Q = \{00, 001, 0010\}$ .

The next candidates are similar to two states in  $Q$  ( $\mathcal{V}(00100) = [0.505, 0.495]$  passes with 00 and  $\mathcal{V}(00101) = [0.700, 0.300]$  passes with  $\mathcal{V}(0010)$ ), so both are removed and its edges rearranged to the configuration in Figure 2.11, which is the same graph as the original Tri-Shift, showing that CRISSiS recovered the PFSA graph. All that is left is to run step 3, feeding the input sequence to the graph and computing the morph probabilities, which recovers an accurate Tri-Shift PFSA.



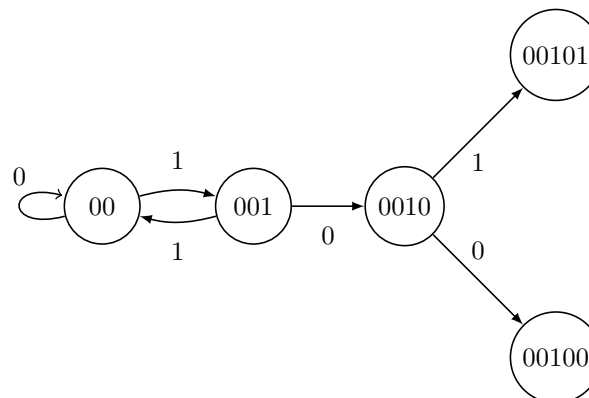
**Figure 2.8:** Tree with 00 at its root,  $Q = \{00\}$ .



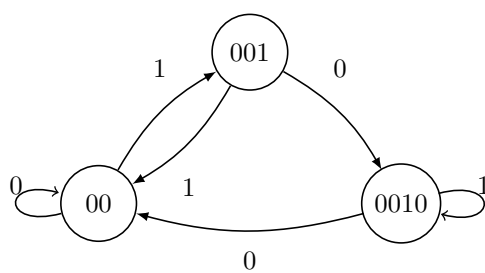
**Figure 2.9:** Second iteration of three,  $Q = \{00, 001\}$ .

### Time Complexity

As shown in [? ], CRISSiS operates with a time complexity of  $O(N) \cdot (|\Sigma|^{O(|Q|^3)+L_1+L_2} + |Q||\Sigma|^{L_2})$ , where  $N$  is the length of the input sequence,  $|\Sigma|$  is the alphabet size,  $|Q|$  is the number of states in the original PFSA and  $L_1$  and  $L_2$  are parameters determining how much of the past and future of a state is needed to determine it. It is stated that as  $L_1$  and  $L_2$  are both usually small, it does not affect the performance greatly, even though the algorithm is exponential in these parameters.



**Figure 2.10:** Third iteration of the three,  $Q = \{00, 001, 0010\}$ .



**Figura 2.11:** *Recovered Tri-Shift topology.*

## CAPÍTULO 3

# ALGORITHM DESCRIPTION

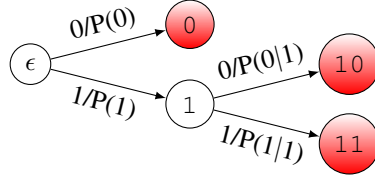
IN this chapter, the proposed algorithm to model a system by its sequence  $S$  is presented. The first section discusses an algorithm to find synchronization words which has lower complexity than the brute force method used by CRISSiS. Later, the PFSA construction algorithm is shown. It is further divided 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  $\Omega$  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  $\Sigma$  generated by a dynamical system, we introduce in this section an algorithm to find possible synchronization words in  $S$ . The CRISSiS method uses (2.2) 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  $\Sigma$  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  $\chi^2$  or Kolmogorov-Smirnov for a given confidence level  $\alpha$ ) to determine whether a state should be expanded. The way the tree is explored guarantees that a state is only tested against other states that have it as a suffix. When a test fails, an expansion algorithm is used to determine how the next states are to be tested. On the other hand, when the test is successful, the 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  $\epsilon$ , which has no predecessors). Leaf states (0, 10 and 11 in the example) have no successors, while the other states have  $|\Sigma|$  successors as each element of  $\Sigma$  labels its outgoing edges. Those edges are also labeled with the probability of leaving the state with that symbol. Each state is labeled with the string formed from concatenating the symbols in the branches in the path from the root to the current state. The probability of reaching a state is given by multiplying the probabilities labeling the branches in the path from the root state to the current state. For example, consider the leaf state 10. The path taken from the root state  $\epsilon$  is first 1 and then 0. The probability of reaching this state is  $P(1) \times P(0|1)$ , that is the probability of leaving the root state with 1 (which is  $P(1)$ ) multiplied by the probability of leaving the state 1 with 0 (that is,  $P(0|1)$ ). The edge probabilities of  $\mathcal{S}$  are taken from the conditional probabilities of sub-sequences of  $S$ .

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

Another data structure used in the algorithm is a dictionary (also called a hash table) [? ]. A dictionary  $d$  is a mapping between two sets  $d : X \rightarrow Y$ . The elements from  $X$  are called the

dictionary keys. An entry in the dictionary is the element  $y \in Y$  associated to the key  $x \in X$  and is denoted by  $d[x]$ , which is also called the *value* of  $x$  in  $d$ . An entry  $d[x]$  might be updated and even deleted from  $d$ .

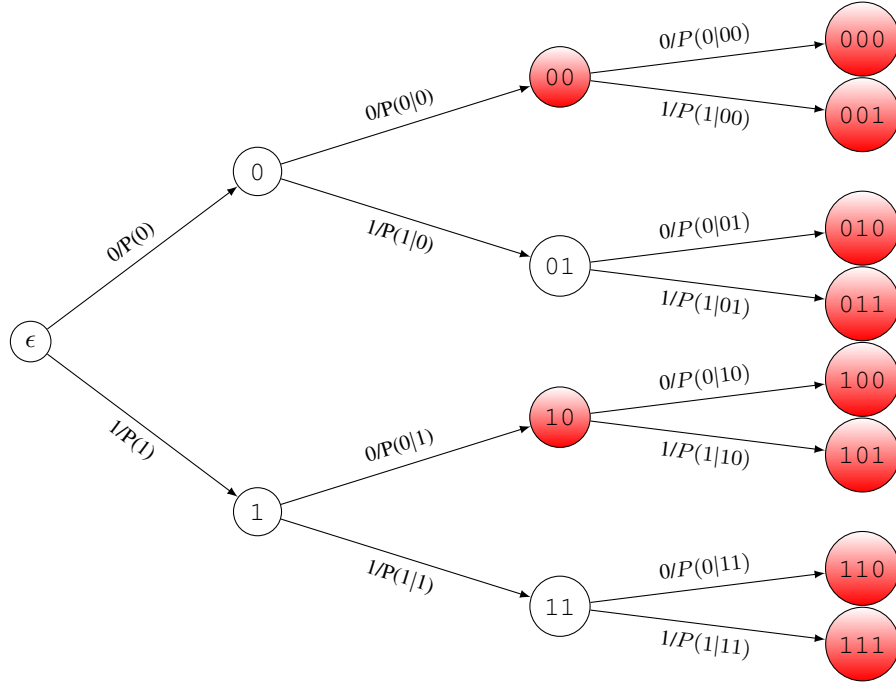
As mentioned before, all states of  $\mathcal{S}$  start as possible candidates for synchronization words. This is represented by a dictionary called *candidacy* which takes states as keys. For a given key  $k$ , *candidacy*[ $k$ ] is a boolean value: it is True when  $k$  is still a possible synchronization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

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

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

To find the synchronization words, Algorithm 6 is used. Its input are the rooted tree with probabilities  $\mathcal{S}$  with maximum depth  $L$ , the maximum window size  $W$ , which is a parameter that determines how deep in the tree the algorithm searches. The algorithm starts by creating the queue  $\Gamma$  which contains states from  $\mathcal{S}$  that are not fully tested for the synchronization word hypothesis during the current iteration.  $\Gamma$  is initialized only with  $\epsilon$ . A list  $\Theta$  is created and initialized empty. It receives the states from  $\mathcal{S}$  which currently have passed the statistical test.





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

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

The main loop then begins. At the start of each iteration, the variable  $c$  receives the first element of  $\Gamma$  via dequeuing (as  $\Gamma$  is a queue, the first element to be inserted into it is the first to be removed). If the label of  $c$  is longer than  $W$ , the algorithm stops and returns the list  $\Theta$ . 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  $\text{suffixes}[c]$  is empty,  $p$  will stay *True*. On the other hand, if there are states for which  $c$  is a suffix,  $\text{suffixes}[c]$  will be iterated. Each element of  $\text{suffixes}[c]$  goes through the statistical test with  $c$  in order to check (2.2). 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  $\Theta$  as it currently is a valid candidate 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  $\Gamma$  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  $\text{suffixes}[\theta]$  after the expansion. This means that each element of  $\Theta$  is concatenated at the end of  $\Gamma$  and then it will be set to the empty set again. This procedure is repeated until either the queue  $\Gamma$  is empty or if all the elements in  $\Gamma$  have labels longer than  $W$ . Once all tests are performed, it returns  $\Theta$ , with all the elements that passed in all their statistical tests, meaning that they are synchronization words according to (2.2).

---

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

---

```

1: procedure INITIALIZATION
2:    $\Gamma \leftarrow \{\epsilon \in \mathcal{S}\}$ 
3:    $\text{suffixes}[\epsilon] \leftarrow \{\delta(\sigma, \epsilon) \mid \forall \sigma \in \Sigma\}$ 
4:    $V \leftarrow$  empty dictionary
5:   for  $s \in \mathcal{S}$  do
6:      $\text{candidacy}[s] = \text{True}$ 
7:    $\Theta \leftarrow \emptyset$ 
8: procedure MAINLOOP
9:   while  $\Gamma \neq \emptyset$  do
10:     $c \leftarrow \text{dequeue}(\Gamma)$ 
11:    if  $\text{length}(c.\text{label}) < W$  then
12:       $p \leftarrow \text{True}$ 
13:      if  $\Lambda \neq \emptyset$  then
14:        for every  $\lambda \in \text{suffixes}[c]$  do
15:           $p \leftarrow \text{statisticalTest}(\mathcal{V}(c), \mathcal{V}(\lambda), \alpha)$ 
16:          if  $p = \text{False}$  then
17:             $\text{candidacy}[c] \leftarrow \text{False}$ 
18:             $\text{expand}(c, V, \mathcal{S}, \Gamma, \text{candidacy}, \text{suffixes})$ 
19:            for every  $\theta \in \Theta$  do
20:               $\Gamma.\text{queue}(\theta)$ 
21:             $\Theta \leftarrow \emptyset$ 
22:            break
23:      if  $p = \text{True}$  then
24:         $\Theta.\text{append}(c)$ 
25:   return  $\Theta$ 

```

---

Algorithm 7 updates  $\Gamma$  and the dictionaries  $\text{suffixes}$  and  $V$  after a statistical test fails. First, a list  $\Psi$  with all the descendants of the state  $c$  is created. This list holds the elements that need to be checked if they can be queued into  $\Gamma$ . They will be queued if they are their own SVS. The dictionary  $V$  uses states of  $\mathcal{S}$  as keys. For a given key  $k$   $V[k]$  is a list of states that have  $k$  as SVS. When an element is not its own SVS it cannot be added to  $\Gamma$  and so it is added to a list in  $V$ . In a later call to

---

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


---

```

1: procedure EXPAND  $\Gamma$ 
2:    $\Psi \leftarrow \{\delta(\sigma, c), \forall \sigma \in \Sigma\}$ 
3:   if  $c$  is a key of  $V$  then
4:      $\Psi \leftarrow \Psi \cup V[c]$ 
5:     delete  $V[c]$ 
6:   for every  $d \in \Psi$  do
7:      $\zeta \leftarrow \text{shortestValidSuffix}(\mathcal{S}, d, \text{candidacy})$ 
8:     if  $\zeta = d$  then
9:        $\Gamma.\text{queue}(\zeta)$ 
10:      for  $t \in \{\delta(\sigma, \zeta) \mid \forall \sigma \in \Sigma\}$  do
11:         $\tau \leftarrow \text{shortestValidSuffix}(\mathcal{S}, t, \text{candidacy})$ 
12:         $\text{suffixes}[\tau].\text{append}(t)$ 
13:      else
14:        if  $V[\zeta] = \emptyset$  then
15:           $V[\zeta] \leftarrow \{d\}$ 
16:        else
17:           $V[\zeta].\text{append}(d)$ 

```

---

Algorithm 7 it might have become its own SVS and so it has to be checked again. If there is a list associated to  $c$  in  $V$ , all its elements are appended to  $\Psi$ . The entry  $V[c]$  is then deleted as it no longer has a use.

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

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

### 3.1.1 An Example

To illustrate how the algorithm works, the Tri-Shift as it can be compared to CRISSiS in Section 2.5.2. All statistical tests in this section use the  $\chi^2$  test with  $\alpha = 0.95$ . The initial RTP for  $W = 3$  and  $L = 4$  is shown in Figure 3.3. Figures 3.4 to 3.9 show the progression of the candidacy status for each iteration. The queue  $\Gamma$  is initialized with the root of  $\mathcal{S}$ . The dictionary *suffixes* is initialized with  $\text{suffixes}[\epsilon] = \{0, 1\}$ .  $V$  is initialized as an empty dictionary,  $\Theta$  is initialized as an empty list and all the states start with their candidacy set to True.

As  $\Gamma$  is not empty, it is dequeued and  $c = \epsilon$ , which has a label length of zero and is shorter than  $W = 3$ . It then proceeds to iterate through  $\text{suffixes}[c] = \text{suffixes}[\epsilon] = \{0, 1\}$  and  $p$  is set to true. It first compares  $\text{statisticalTest}(\epsilon, 0)$ . As the morphs are  $[0.6276, 0.3274]$  and  $[0.5615, 0.4385]$ , the test fails, which means  $\epsilon$  candidacy is set to False and the expansion algorithm is called.

The list  $\Psi$  is initialized with the direct descendants of  $c = \epsilon$ , that is  $\Psi = \{0, 1\}$ .  $V[\epsilon]$  is empty and can be disregarded. It is easy to check that all elements in  $\Psi$  are their own shortest valid suffixes after  $\epsilon$  candidacy becomes false (seen in Figure 3.4). This means both of them are queued into  $\Gamma$ , so that  $\Gamma = \{0, 1\}$ . For both 0 and 1, they are their direct descendants' shortest valid suffixes, which means that  $\text{suffixes}[0] = \{00, 10\}$  and  $\text{suffixes}[1] = \{01, 11\}$ . The expansion algorithm returns to the synchronization algorithm. The list  $\Theta$  is appended to the end of  $\Gamma$ , but as it is currently empty it does not change  $\Gamma$ . 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  $\text{suffixes}[0] = \{00, 10\}$  has each of its elements tested. First to be tested is 00 and  $\text{statisticalTest}(0, 00)$  returns False as  $\mathcal{V}(0) = [0.5615, 0.4385]$  diverges significantly from  $\mathcal{V}(00) = [0.5, 0.5]$ . This means that  $\text{candidacy}[0]$  is set to False and the expansion algorithm is called.

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

In the beginning of the next iteration, we have  $\Gamma = \{1, 000\}$ ,  $V[1] = \{01\}$ ,  $\Theta = \emptyset$ ,  $\text{suffixes}[1] = \{01, 11, 001\}$  and  $\text{suffixes}[00] = \{000\}$ .  $\Gamma$  is dequeued and  $c = 1$ ,  $\text{suffixes}[1] = \{01, 11, 001\}$  is iterated through. First,  $\text{statisticalTest}(1, 01)$  is checked to be false ( $[0.779, 0.221]$  against  $[0.739, 0.261]$ ) making  $\text{candidacy}[1] = \text{False}$  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  $\text{candidacy}[0] = \text{candidacy}[1] = \text{False}$ , 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  $\text{suffixes}[00] = \{000, 100\}$ ,  $\text{suffixes}[01] = \{001, 101\}$ ,  $\text{suffixes}[10] = \{010, 110\}$  and  $\text{suffixes}[11] = \{011, 111\}$ . Once again  $\Theta$  is appended in  $\Gamma$  and emptied.

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

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

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

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

In the eighth iteration,  $\Gamma = \{010, 00, 101, 110, 111, 011\}$ ,  $V[00] = \{100\}$ ,  $\text{suffixes}[010] = \{1010\}$ ,  $\text{suffixes}[00] = \{000, 100, 0100, 1100\}$ ,  $\text{suffixes}[101] = \{0101, 1101\}$ ,  $\text{suffixes}[110] = \{1110, 0110\}$ ,  $\text{suffixes}[111] = \{1111, 0111\}$  and  $\text{suffixes}[011] = \{1011\}$ .  $c = 010$  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  $\Theta$ . 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 skipped. The final procedure will group states in equivalence classes of states that have statistically similar morphs (checked by  $\chi^2$  or Kolmogorov-Smirnov for a given confidence level  $\alpha$ ) and the partition given by these equivalence classes is used as an initial partition for a graph reduction algorithm (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  $\Omega$  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.

---

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

---

```

1: procedure TERMINATE
2:    $\Psi \leftarrow \{n \in \mathcal{S} \text{ if } n \text{ in level } L\}$ 
3:   for  $p \in \Psi$  do
4:     Given that  $p.\text{label}$  is  $\sigma_0\sigma_1 \dots \sigma_L$ 
5:     for  $\tau \in \Sigma$  do
6:        $\delta(\tau, p) \leftarrow \sigma_1\sigma_2 \dots \sigma_L\tau$ 

```

---

 **$\Omega$  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 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  $\Omega_{syn}$ .

### 3.2.2 $\aleph$ Algorithm

The full  $\aleph$  algorithm is shown in Algorithm 10. When there are no synchronization words (i.e. the  $\Omega_{syn}$  list is empty), the D-Markov termination is applied to  $\mathcal{S}$  and an equivalence class is created with just the root state  $\epsilon$ . When  $\Omega_{syn}$  is not empty, the  $\Omega$  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  $\omega$  as suffix. If it has, this outgoing edge is reassigned to point to  $\omega$ . 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  $\Omega_{syn}$  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 successful. If no test is positive, a new equivalence class is created for  $q$  and this class

---

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

---

```

1: procedure TERMINATE
2:    $\Psi \leftarrow \{p \in \mathcal{S} \text{ if } n \text{ in level } L\}$ 
3:   for  $m \in \Psi$  do
4:     next = NULL
5:     for  $\tau \in \Sigma$  do
6:        $n = \delta(\tau, m)$ 
7:       for  $\omega \in \Omega_{syn}$  do
8:          $r \leftarrow \mathcal{V}(n) = \mathcal{V}(\omega)$ 
9:         if  $r = True$  then
10:          next  $\leftarrow \omega$ 
11:          break
12:       if next = NULL then
13:          $\eta \leftarrow \{\text{All extensions of } \omega \text{ up to length } L, \forall \omega \in \Omega_{syn}\}$ 
14:         for  $e \in \eta$  do
15:            $r \leftarrow \text{statisticalTest}(n, e)$ 
16:           if  $r = True$  then
17:             next  $\leftarrow e$ 
18:             break
19:       if next = NULL then
20:         Given that  $m.\text{label} = \sigma_0 \dots \sigma_L$ 
21:         next =  $\sigma_1 \dots \sigma_L \tau$ 
22:        $\delta(\tau, m) \leftarrow \text{next}$ 

```

---



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  $\aleph$  algorithm are that, unlike CRISiS, it does not depend on the original system being synchronizable (the original system does not even need to be represented by a PFSA and  $\aleph$  will generate a PFSA that approximates it) and the lower time complexity to run the algorithm. As seen in [? ], CRISiS 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  $\alpha$  depends only on parameters known prior the algorithm execution. The complexity of each part of the algorithm is discussed individually and a final complexity is given in the end.

#### 3.3.1 RTP Construction

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

#### 3.3.2 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  $\epsilon$ . 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 CRISiS, 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  $O(\frac{|\Sigma|^{W+1} W^3}{|\Sigma|-1})$ .

---

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


---

```

1: procedure
2:   if  $\Omega_{syn} \neq \emptyset$  then
3:      $\mathcal{S} \leftarrow \text{omegaTermination}(\mathcal{S})$ 
4:      $Q_0 \leftarrow \Omega_{syn}$ 
5:      $Q_1 \leftarrow \emptyset$ 
6:     while  $Q_0 \neq \emptyset$  do
7:        $q_0 \leftarrow Q_0.pop()$ 
8:       for  $\sigma \in \Sigma$  do
9:          $q'_0 \leftarrow \delta(q_0, \sigma)$ 
10:        if for some  $\omega \in \Omega_{syn}$ ,  $\omega$  is a suffix of  $q'_0$  then
11:           $\delta(\sigma, q_0) \leftarrow \omega$ 
12:        else
13:          if  $q'_0 \notin Q_0$  and  $q'_0 \notin Q_1$  then
14:             $Q_0 \leftarrow Q_0 \cup \{q'_0\}$ 
15:             $Q_1 \leftarrow Q_1 \cup \{q_0\}$ 
16:           $Q \leftarrow \{\delta(\sigma, \omega), \forall \sigma \in \Sigma, \forall \omega \in \Omega_{syn}\}$ 
17:        else
18:           $\mathcal{S} \leftarrow \text{dmarkovTermination}(\mathcal{S})$ 
19:           $Q_1 \leftarrow \{\epsilon\}$ 
20:           $Q \leftarrow \{\delta(\sigma, \epsilon), \forall \sigma \in \Sigma\}$ 
21:         $\mathcal{P} \leftarrow \{Q_1\}$ 
22:        for  $q \in Q$  do
23:           $r \leftarrow \text{False}$ 
24:          for  $p \in \mathcal{P}$  do
25:             $r \leftarrow \text{statisticalTest}(q, p[0])$ 
26:            if  $r = \text{True}$  then
27:               $p \leftarrow p \cup \{q\}$ 
28:              break
29:            if  $r = \text{False}$  then
30:               $R \leftarrow \{q\}$ 
31:               $\mathcal{P} \leftarrow \mathcal{P} \cup \{R\}$ 
32:           $Q \leftarrow Q \cup \{\delta(\sigma, q), \forall \sigma \in \Sigma \mid \delta(\sigma, q) \text{ not in any } p \in \mathcal{P}\}$ 
33:        ## The following function can be either Moore or Hopcroft
34:         $G \leftarrow \text{GraphReduction}(\mathcal{P})$ 
35:      return  $G$ 

```

---

### 3.3.3 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  $O(|\Sigma|^{L+1})$ .

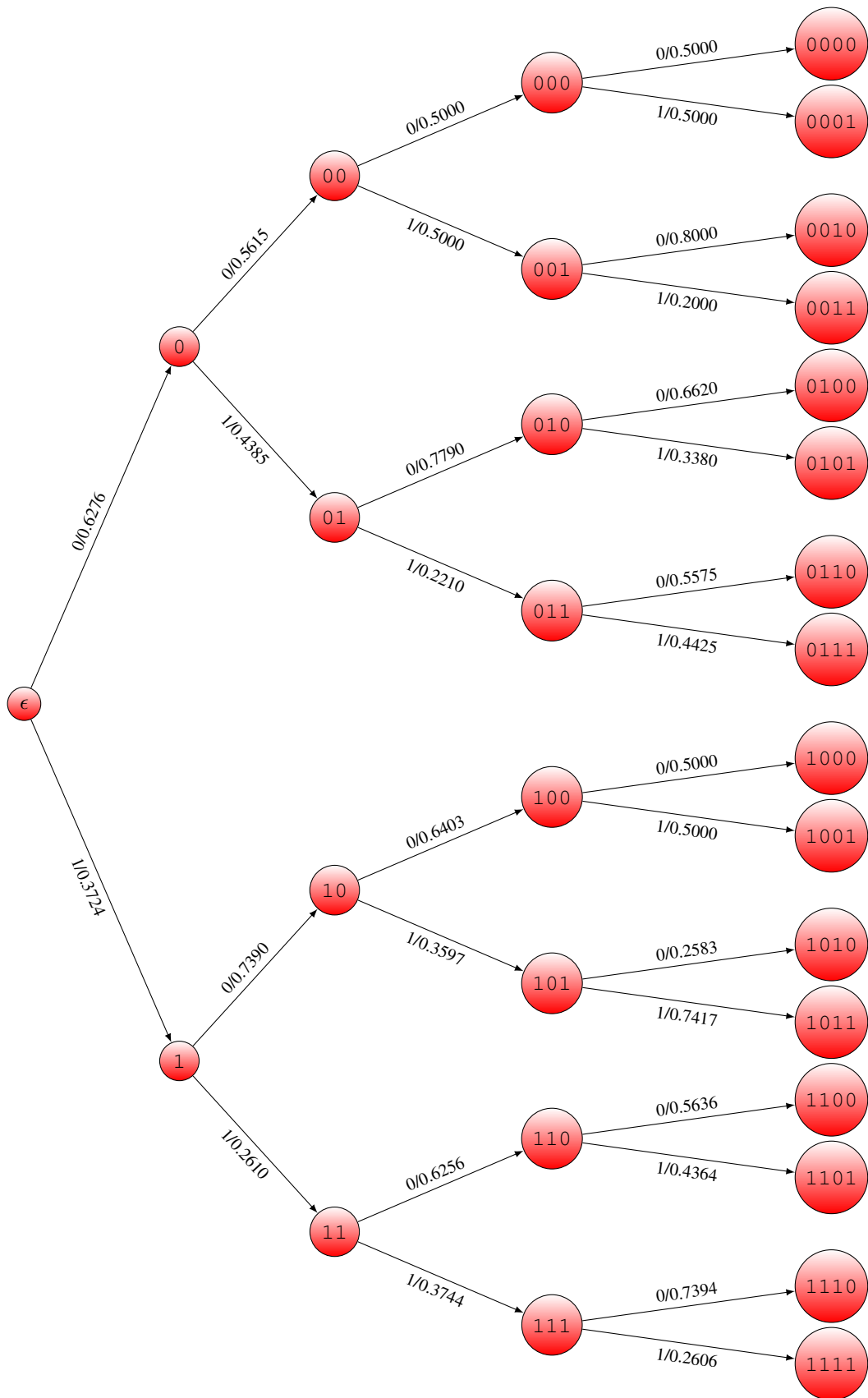
The  $\Omega$  termination is a little more complex to analyze. It also needs to perform operations for each of the  $|\Sigma|$  outgoing edges of each of the  $|\Sigma|^L$  states in level  $L$ , but those operations are not simply reconections of complexity  $O(1)$ . It performs tests with all states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

### 3.3.4 PFSA Construction

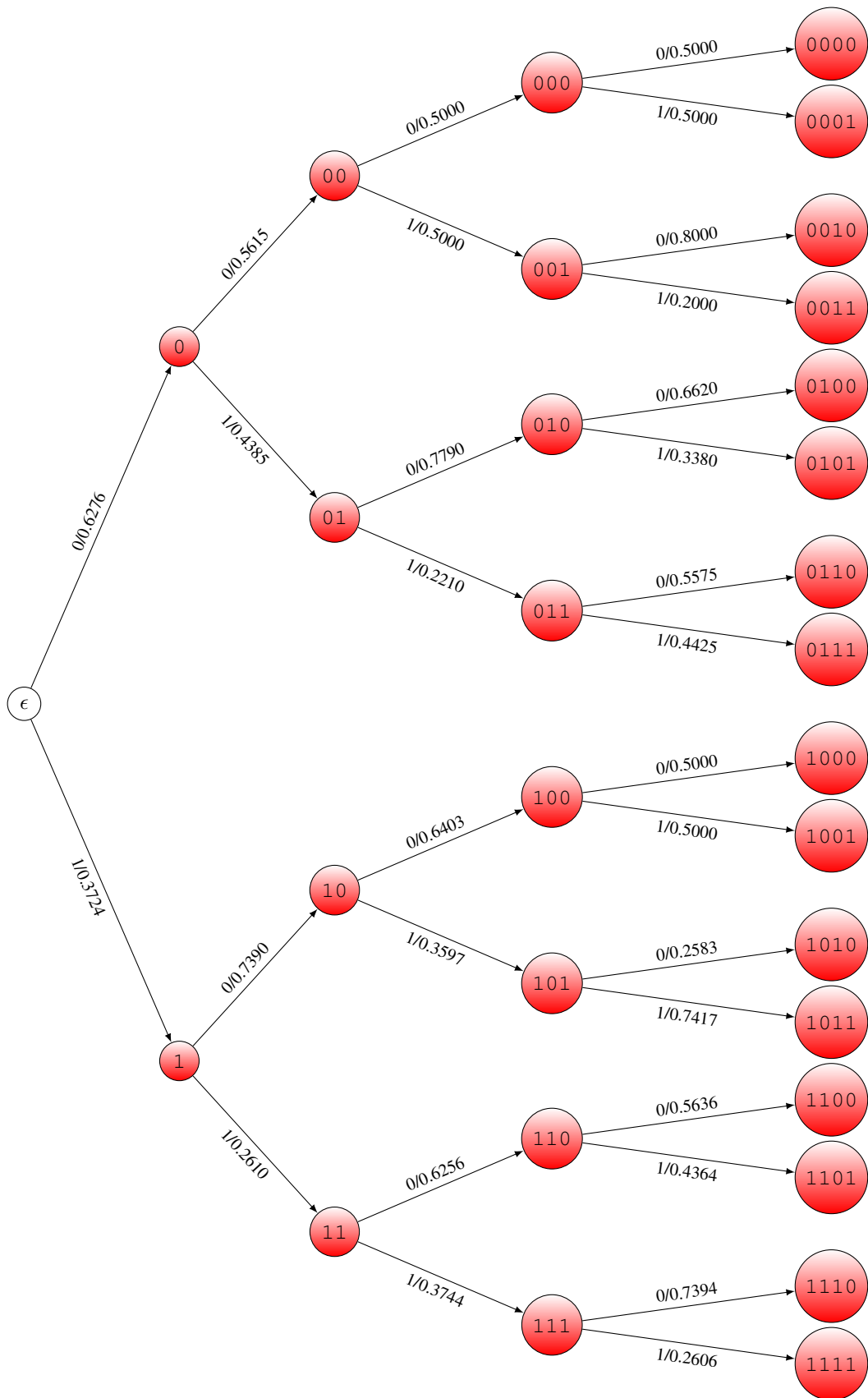
When there are synchronization words, the algorithm starts by checking if all the  $|\Sigma|^L$  states have an outgoing edge that could be substituted by a synchronization word, giving a final complexity of  $O(|\Sigma|^{L+1})$  for this additional step.

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

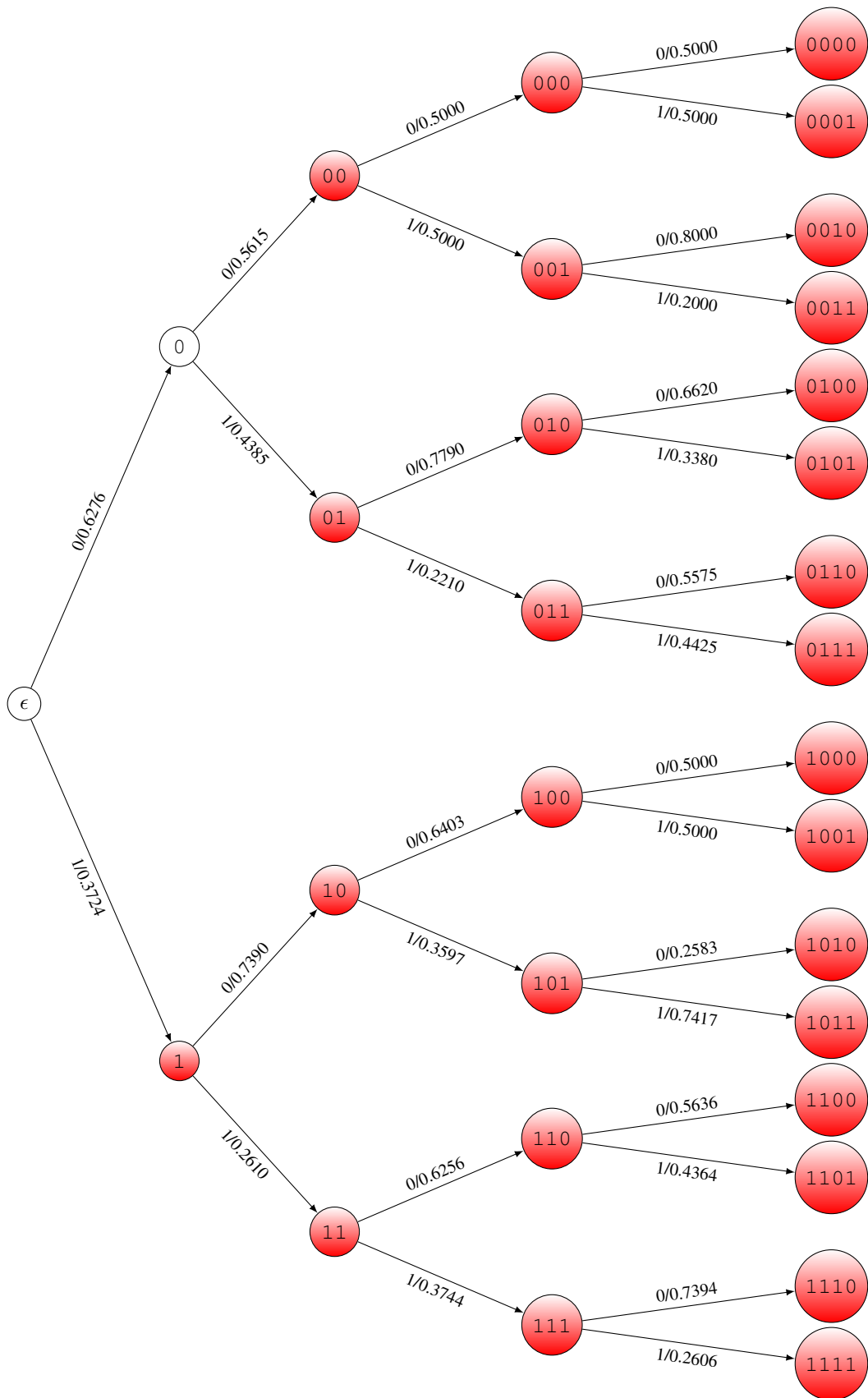
When there are synchronization words and the first step of complexity  $O(|\Sigma|^{L+1})$  is applied, the number of states that will be organized in equivalence classes might be dramatically reduced, which also reduces the complexity of that step. But in the worst case scenario, the  $O(|\Sigma|^{2L+2})$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



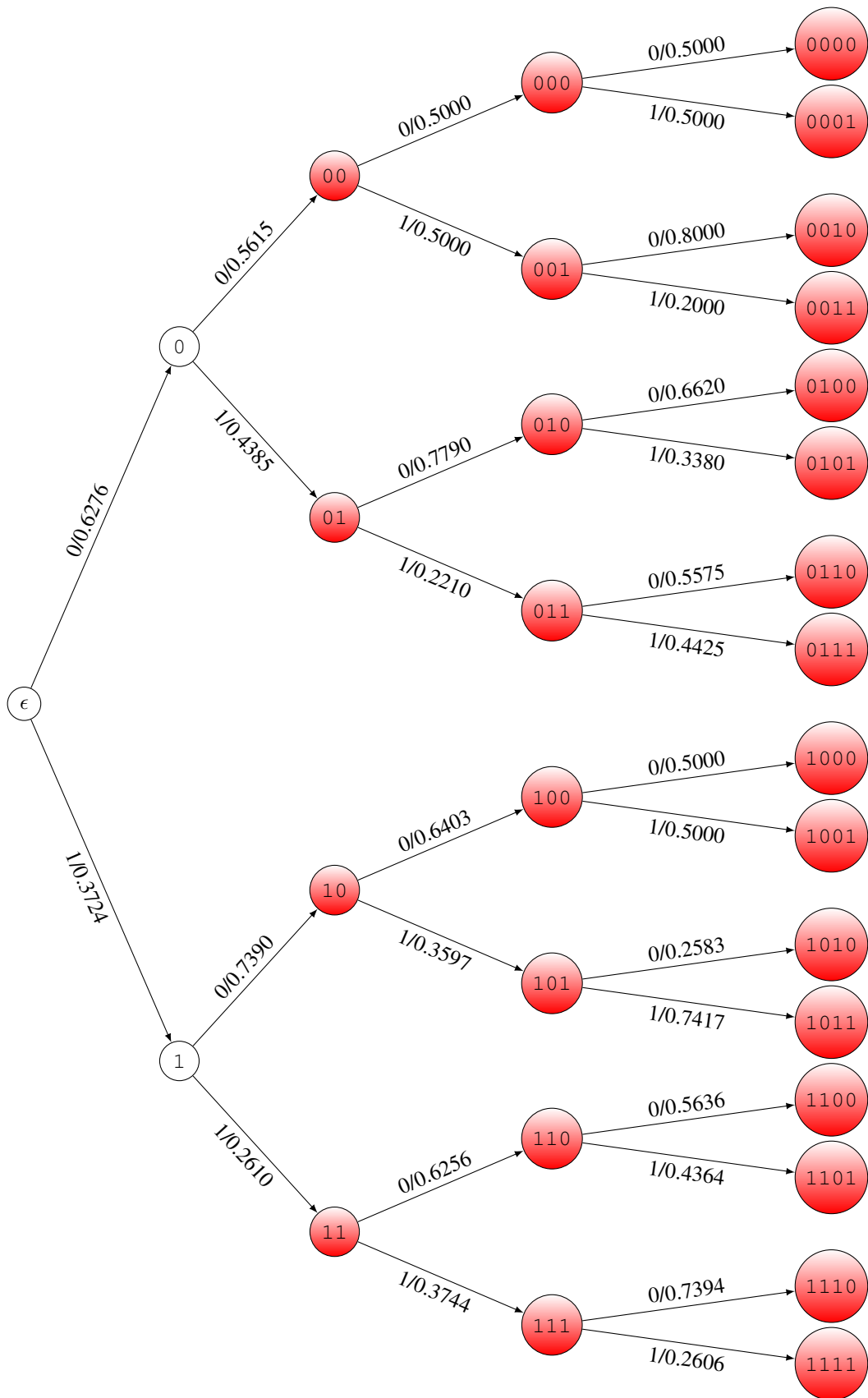
**Figura 3.3:** Input Rooted Tree with Probabilities  $S$  for the Tri-Shift Example.



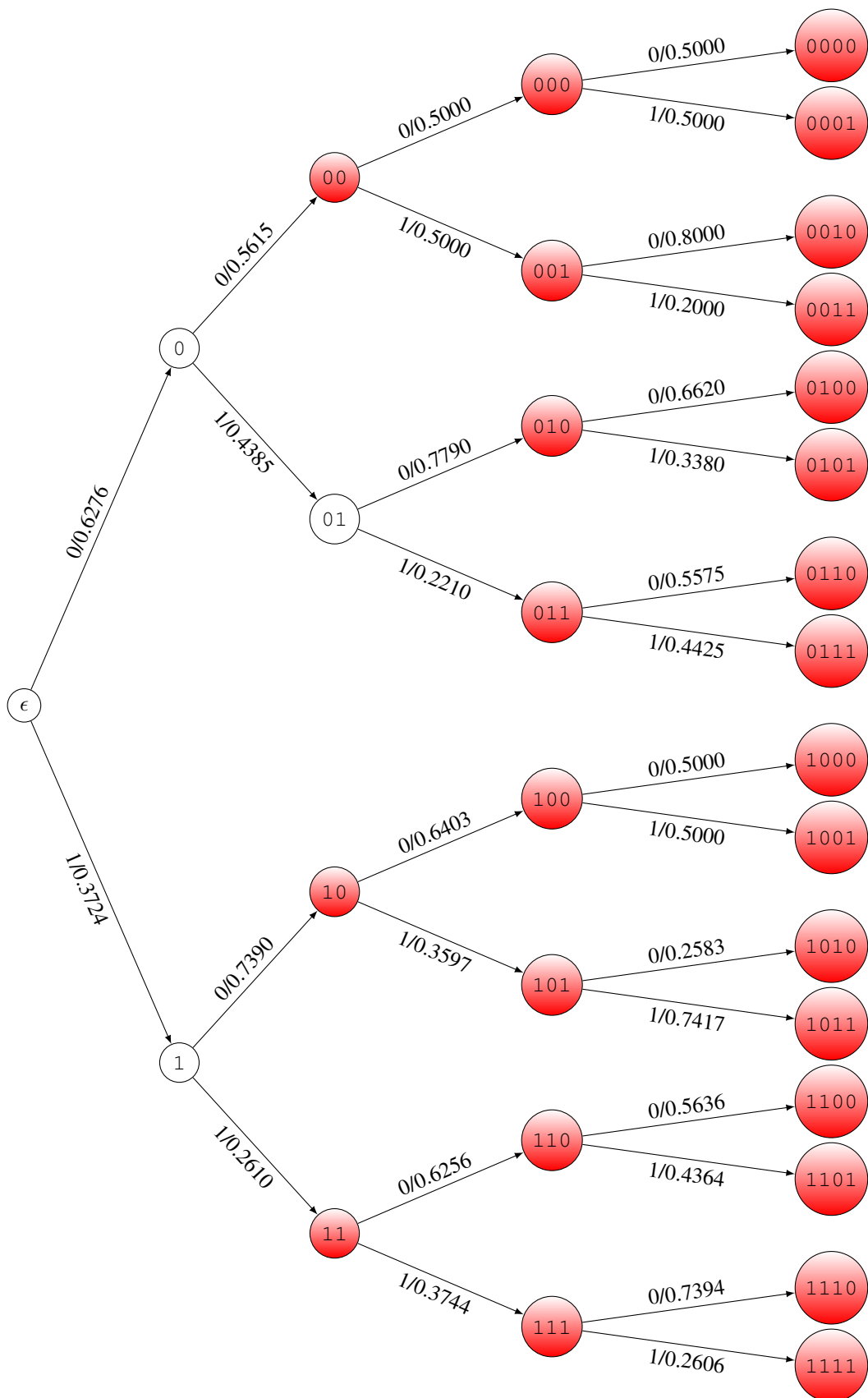
**Figure 3.4:** Rooted Tree with Probabilities  $S$  for the Tri-Shift Example after the first iteration.



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

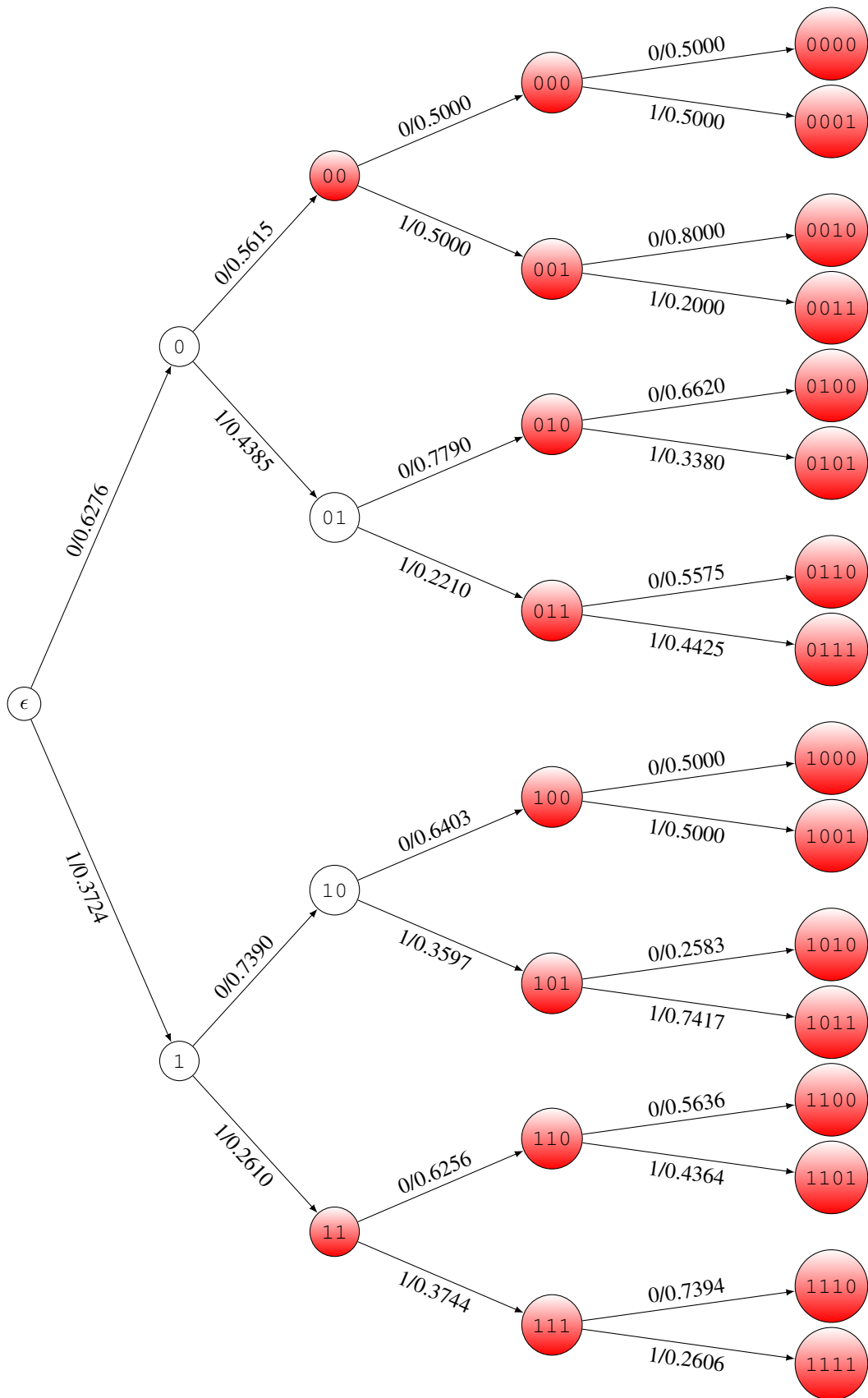


**Figure 3.6:** Rooted Tree with Probabilities  $S$  for the Tri-Shift Example after the third and fourth iterations.

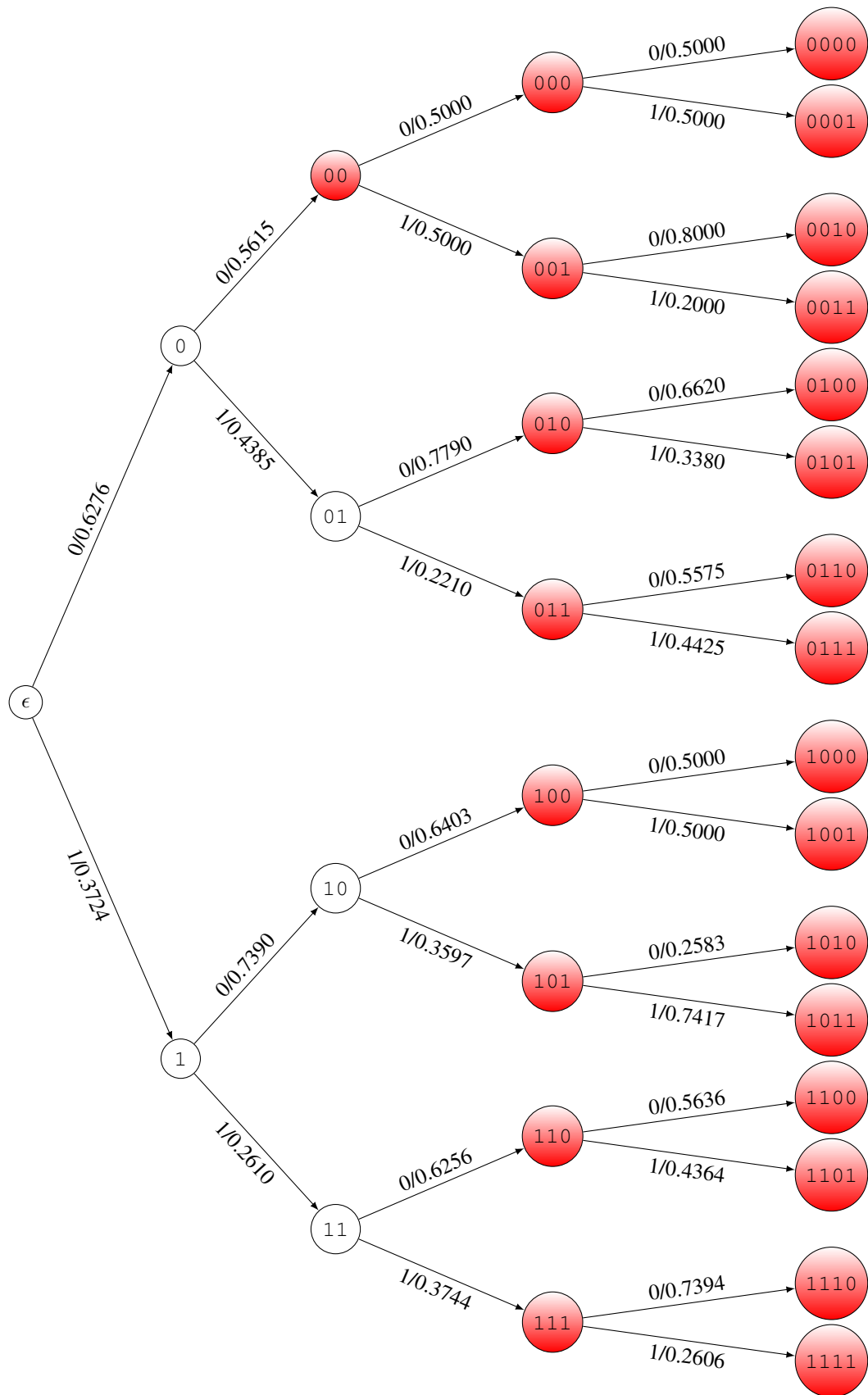


**Figura 3.7:** Rooted Tree with Probabilities  $S$  for the Tri-Shift Example after the fifth iteration.





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



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

# CAPÍTULO 4

## RESULTS

**I**N this chapter, the efficiency of the algorithms proposed in Chapter 3 to construct a PFSA is verified for some examples dynamic systems that can be represented by PFSA. Results for more practical and complex systems are discussed in the next chapter. First, from the original system, a discrete sequence  $S$  over the alphabet  $\Sigma$  of length  $N = 10^7$  is generated. Then, we calculate the probabilities of subsequences occurring in  $S$  up to a length  $L_{max}$  and construct an RTP from these probabilities. After this, a series of PFSA are created using the D-Markov Machine, CRISSiS and the  $\aleph$  algorithm 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 results are explained in Section 4.2.

### 4.1 Evaluation Metrics

This section presents the three metrics that will be used to compare the performance of the models generated by the algorithms. The first one is the conditional entropy that is used to approximate the entropy rate, which gives a sense of the memory of the system. The two other metrics, the Kullback-Leibler Divergence and  $\Phi$ , compare sequences generated by the models with the original one and estimate how similar they are. The lower these metrics are, the more similar to the original system are the models.

#### 4.1.1 Entropy Rate

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

$$h \triangleq \lim_{k \rightarrow \infty} H(X_k | X_1 X_2 \dots X_{k-1}) = - \lim_{k \rightarrow \infty} \sum_{x \in \Sigma^k} \Pr(x) \log \Pr(x_k | x_1 x_2 \dots x_{k-1}). \quad (4.1)$$

For a stationary process, the conditional entropy  $H(X_k | X_1 \dots X_{k-1})$  is non-increasing in  $k$  and converges to  $h$  as  $k$  approaches infinity [? ]. As it is not feasible to compute (4.1) up to infinity when the distribution is known only up to  $L_{max}$ , we use the  $\ell$ -order conditional entropy defined as:

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

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

#### 4.1.2 Kullback-Leibler Divergence

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

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

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 $\Phi$ -Metric

The  $\Phi$ -Metric is presented in [? ] as a way to compare sequences. Given two sequences  $S_1$  and  $S_2$  over a common alphabet  $\Sigma$  (again, these can be either the original sequence or a sequence generated by a PFSA) and  $\mathbf{P}_1(\Sigma^j)$  and  $\mathbf{P}_2(\Sigma^j)$  are the vectors of relative frequencies of occurrences of each  $\sigma^j \in \Sigma^j$  in  $S_1$  and  $S_2$  respectively. For example, for a binary alphabet  $\mathbf{P}_1(\Sigma^2) = [P_1(00), P_1(01), P_1(10), P_1(11)]$  with  $P_1$  defined as in Section 4.1.2. The  $\Phi$ -Metric is then defined as:

$$\Phi(S_1, S_2) \triangleq \lim_{n \rightarrow \infty} \sum_{j=1}^n \frac{\|\mathbf{P}_1(\Sigma^j) - \mathbf{P}_2(\Sigma^j)\|_{\ell_1}}{2^{j+1}}, \quad (4.4)$$

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  $\ell$ . The  $\ell$ -order  $\Phi$ -Metric is:

$$\Phi_\ell(K1, K2) \triangleq \sum_{j=1}^{\ell} \frac{\|\mathbf{P}_1(\Sigma^j) - \mathbf{P}_2(\Sigma^j)\|_{\ell_1}}{2^{j+1}}. \quad (4.5)$$

As with the Kullback-Leibler Divergence, the  $\Phi$ -Metric compares two systems using sequences generated by them, taking the distribution of all sub-sequences up to length  $\ell$ . A small  $\Phi$  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) via their generated sequences 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, CRISSiS and  $\aleph$  algorithm to recover a good PFSA with the smallest number of states.

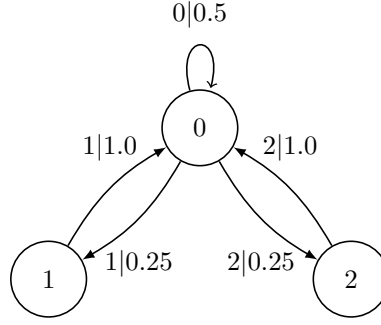
In all examples, the  $\aleph$  algorithms are able to recover the original PFSA for some value of  $L$ . Usually, D-Markov Machines are not capable of retrieving the original PFSA, but by increasing  $D$ , better machines are obtained in expense of an exponential growth in the number of states. The results for two D-Markov Machines are shown for each example: one for the  $D$  that achieves a performance similar to the original PFSA and another for  $D - 1$  to show a more compact alternative and how it performs.

In the following cases we consider  $\ell = 10$ . All the PFSA were constructed using the  $\chi^2$  test with  $\alpha = 0.95$ .

### 4.2.1 Ternary Even Shift

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

The synchronization words found by our algorithm are shown in Table 4.1 for two values of  $\alpha$  (0.95 and 0.99) and the same words ( $\Omega_{syn} = \{0, 12, 21\}$ ) are found for any  $W \geq 3$ . It is possible to



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

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

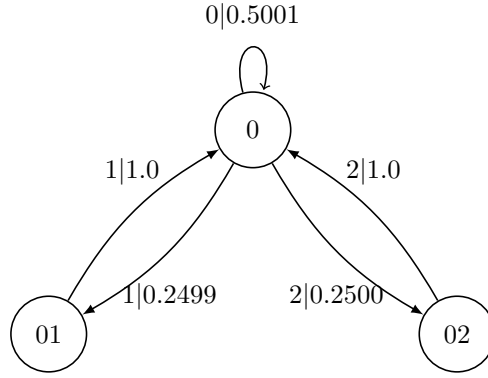
$W$	$\alpha$	
	0.95	0.99
2	0	0
3	0, 12, 21	0, 12, 21
4	0, 12, 21	0, 12, 21
5	0, 12, 21	0, 12, 21
6	0, 12, 21	0, 12, 21

check in the graph of Figure 4.1 that all found synchronization words are indeed valid and each one synchronizes to one of state of the graph. They can all be used as starting points for the  $\aleph$  algorithm.

The results of the  $\aleph$  algorithm are compared to D-Markov and CRISSiS in Table 4.2. The  $\aleph$  algorithm obtained the same results for any  $L$  greater than 2. D-Markov machines of  $D = 8$  and  $D = 9$  were considered. CRISSiS was tested using  $L_2 = 1$ . Both CRISSiS and  $\aleph$  reconstruct the same PFSA (shown in Figure 4.2) and are a good estimate to the original 3-state PFSA while a large D-Markov machine with  $D = 9$  with 339 states is needed to obtain approximately the same performance. Even though  $D = 8$  and 9, these D-Markov machines do not have  $3^8$  and  $3^9$  states respectively because there are forbidden words in the original system, which results in some states being non-existent in the RTP. The original system had  $h_{10} = 1.0003$ , which is close to the value found by all the algorithms.

#### 4.2.2 Tri-Shift

The Tri-Shift was previously discussed in Section 2.5.2 and a PFSA that represents it is shown in Figure 2.7. The synchronization words found by the algorithm are shown in Table 4.3 and 00 appeared, as expected (see Section 2.5.2), and 0110 synchronizes to the same state as 00, thus  $\Omega_{syn} = \{00, 0110\}$ . The comparative results are shown in Table 4.4. Once again this is an example where



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

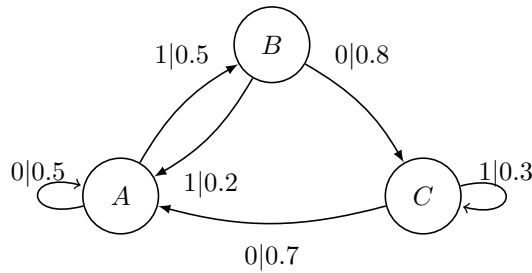
**Tabela 4.2:** Results for Ternary Even Shift.

	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
$D_{10}$	$2.7 \cdot 10^{-3}$	$4.16 \cdot 10^{-5}$	$9.55 \cdot 10^{-5}$
$\Phi_{10}$	$2.1 \cdot 10^{-3}$	$1.2 \cdot 10^{-3}$	$2.3 \cdot 10^{-3}$

our algorithm and CRISSiS are able to recover the three states from the original PFSA with a good estimate for the morphs as seen in Figure 4.4. To obtain a similar performance with a D-Markov machine, 256 states might be needed. The original system presented has  $h_{10} = 0.4873$ , showing that our algorithm, CRISSiS and the 8-Markov Machine are able to capture the system memory.

#### 4.2.3 A Six State PFSA

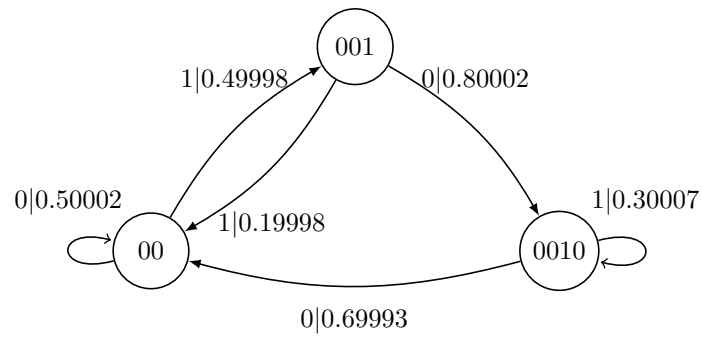
Figure 4.5 shows a PFSA with six states that shows how CRISSiS might need a larger  $L_2$  (an subsequent higher complexity) to retrieve the original machine.. This system has 4 synchronization words: 00, 01, 10 and 1111, as shown in Table 4.5. The comparative results between the algorithms



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

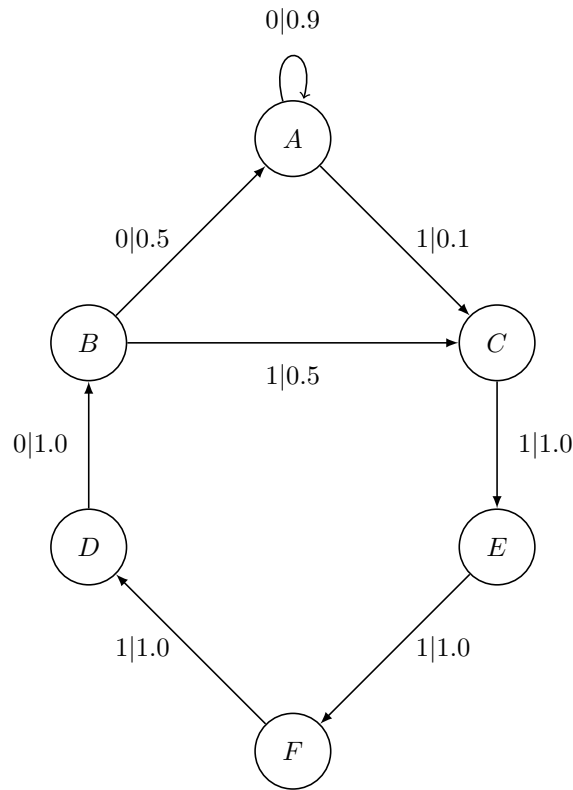
**Tabela 4.3:** *Synchronization Words for Tri-Shift.*

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

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

	D-Markov		$aleph_1$ /CRISSiS
	$D = 7$	$D = 8$	$L = 4/L_2 = 1$
# of States	128	256	3
$h_{10}$	0.4870	0.4867	0.4872
$D_{10}$	$4.1 \cdot 10^{-3}$	$1.65 \cdot 10^{-3}$	$1.16 \cdot 10^{-3}$
$\Phi_{10}$	$2.1 \cdot 10^{-3}$	$7.2 \cdot 10^{-4}$	$8.2 \cdot 10^{-4}$





**Figura 4.5:** A Six-State PFSA.

is shown in Table 4.6.

Using CRISSiS with  $L_2$  larger than 3 and  $\aleph$  with  $L$  larger than 4, it is possible to reconstruct a good estimate to the original system, shown in Figure 4.6. For a D-Markov Machine to perform similarly, it is necessary to use  $D = 4$ , obtaining a PFSA with 11 states. Once again, some sequences do not occur, therefore the D-Markov Machine in those cases will not have  $2^D$  states.

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

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

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

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

	D-Markov		$\aleph_1/\text{CRISSiS}$
	$D = 3$	$D = 4$	$L = 4/L_2 = 3$
# of States	7	11	6
$h_{10}$	0.5341	0.3344	0.3344
$D_{10}$	1.1980	$4.0499 \cdot 10^{-6}$	$5.6969 \cdot 10^{-5}$
$\Phi_{10}$	$2.0005 \cdot 10^{-3}$	$4.6072 \cdot 10^{-4}$	$9.3745 \cdot 10^{-4}$

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

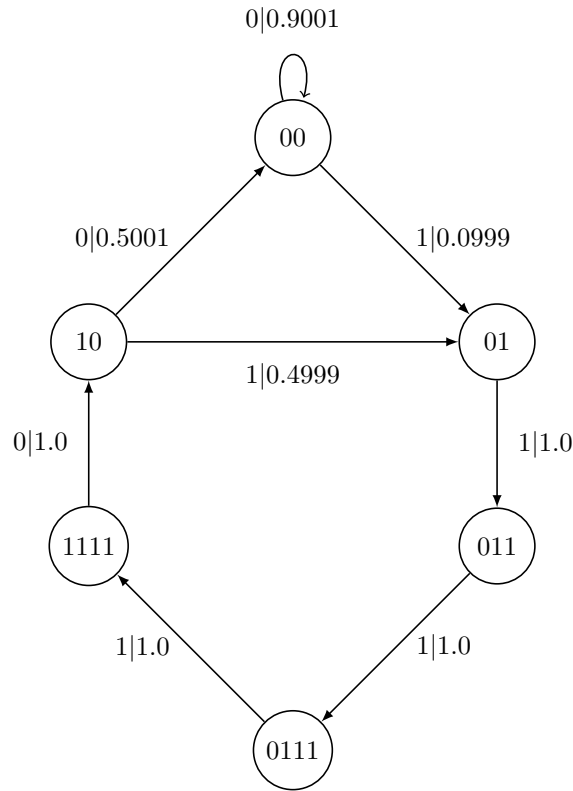
	$\alpha$	
W	0.95	0.99
2	0	0
3	0	0
4	0	0
5	0	0
6	00, 11111	00, 11111
7	00, 11111	00, 11111

CRISSiS are able to estimate the PFSA with good precision.

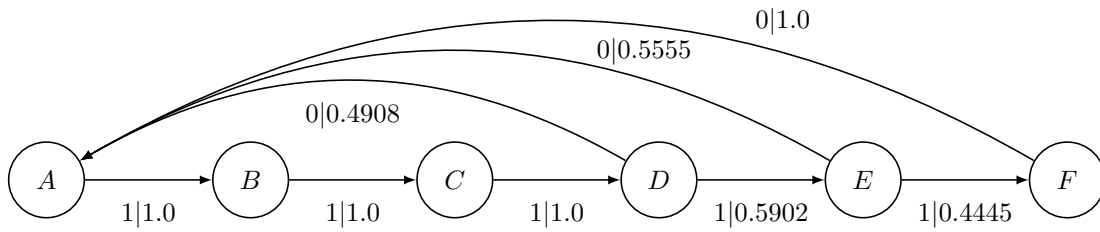
#### 4.2.4 Maximum Entropy $(d,k)$ -Constrained Code

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 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  $L_2 \geq 3$  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. 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.



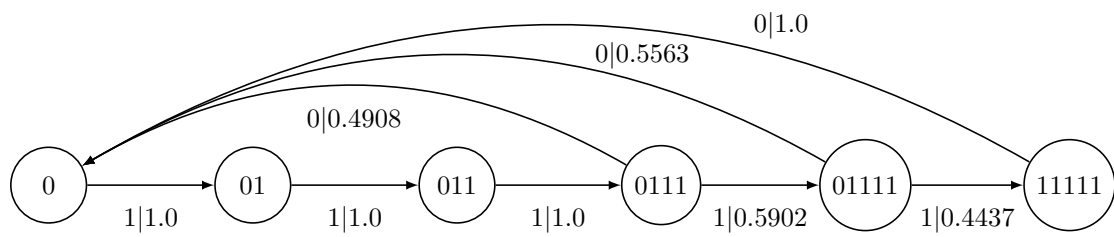
**Figura 4.6:** *The Recovered Six-State PFSA by our algorithm.*



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

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

	D-Markov		CRISSiS	$\aleph$
	$D = 4$	$D = 5$	$L_2 = 3$	$L = 6$
# of States	5	7	6	6
$h_{10}$	0.3575	0.3218	0.3218	0.3218
$D_{10}$	0.1793	$7.0139 \cdot 10^{-7}$	$5.9715 \cdot 10^{-7}$	$2.3766 \cdot 10^{-6}$
$\Phi_{10}$	$5.0521 \cdot 10^{-3}$	$2.8001 \cdot 10^{-4}$	$9.3656 \cdot 10^{-5}$	$1.5380 \cdot 10^{-4}$



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

## CAPÍTULO 5

# APPLICATIONS

**T**<sub>ODO</sub>

### 5.0.1 Logistic Map

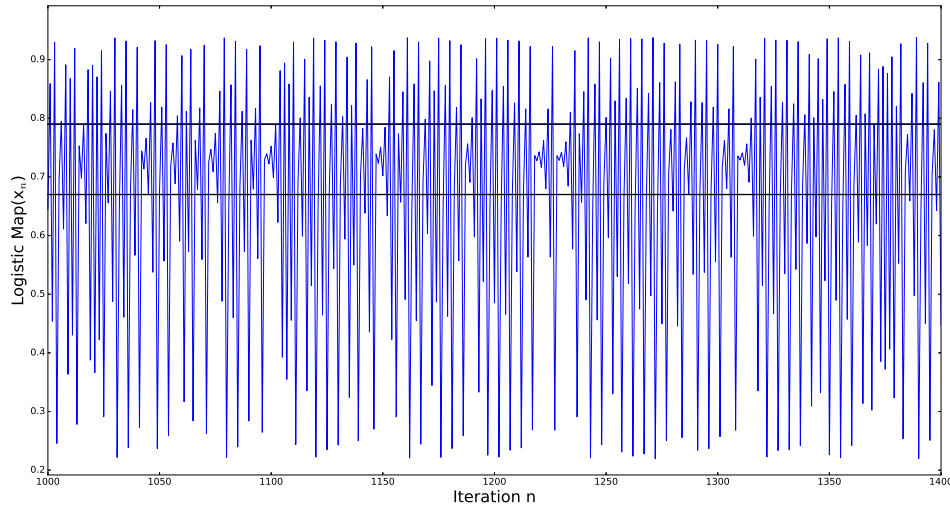
The next two examples show how the algorithms fare when modeling a system whose PFSA is unknown or non-existent. A sequence from these systems will be analyzed and a PFSA model will be created and its output will be compared to the original sequence to see how well this Markovian model approximates a 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 [? ]:

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

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



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

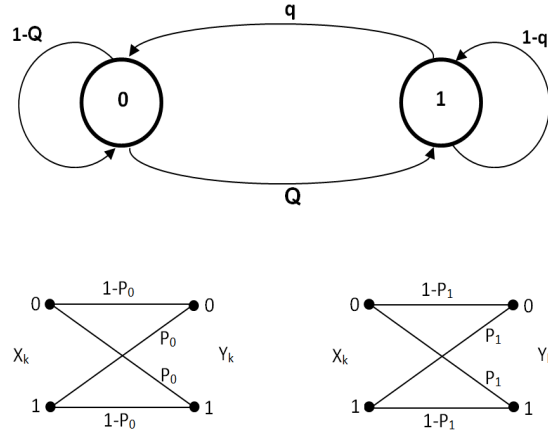
**Tabela 5.1:** Synchronization Words for the Logistic Map Ternary Sequence.

W	$\alpha$	
	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  $\alpha$  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  $\mu$  and the Bit Error Rate (BER), which is a percentage of errors in the transmission. The memory  $\mu$  is defined as:

$$\mu = 1 - q - Q. \quad (5.2)$$

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

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

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

$$\text{BER} = \frac{q}{q + Q} p_0 + \frac{Q}{q + Q} p_1. \quad (5.4)$$

The GEC can be designed to obtain specific values of  $\mu$  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:

$$y_k = x_k \oplus z_k, \quad (5.5)$$

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

T<sub>ODO</sub>

APÊNDICE A

**TODO**

**T**<sub>ODO</sub>

## APÊNDICE B

### **TODO**

# SOBRE O AUTOR

The author was born in Brasília, Brasil, on the 6<sup>th</sup> 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  $\text{\LaTeX 2}_{\epsilon}$ <sup>1</sup> pelo autor.

---

<sup>1</sup> $\text{\LaTeX 2}_{\epsilon}$  é uma extensão do  $\text{\LaTeX}$ .  $\text{\LaTeX}$  é uma coleção de macros criadas por Leslie Lamport para o sistema  $\text{\TeX}$ , que foi desenvolvido por Donald E. Knuth.  $\text{\TeX}$  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 última 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. Dachsel 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.