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

DANIEL KUDLOWIEZ FRANCH

DYNAMIC SYSTEM MODELING AND FAULT DETECTION WITH PROBABILISTIC FINITE STATE AUTOMATA

Recife
2017

DANIEL KUDLOWIEZ FRANCH

DYNAMIC SYSTEM MODELING AND FAULT DETECTION WITH PROBABILISTIC FINITE STATE AUTOMATA

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Pernambuco como parte dos requisitos para obtenção do grau de **Mestre em Engenharia Elétrica**.

Orientador: Prof. Cecilio José Lins Pimentel.

Co-orientador: Prof. Daniel Pedro Bezerra Chaves.

Área de Concentração: Comunicações

Recife
2017

To my grandmother Zélia

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

To my advisers for all the guidance provided.

RESUMO

TODO

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

ABSTRACT

TODO

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

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

CAPÍTULO 1

INTRODUCTION

TODO.

CAPÍTULO 2

REVISION

THIS chapter revises the concepts needed to develop the algorithms and their application.

2.1 Sequences of Discrete Symbols

This section provides tools to describe sequences of discrete symbols. The length of a sequence u is denoted by $|u|$. The empty sequence ϵ is defined as the sequence with length 0. The set of all possible symbols for a sequence is called its alphabet, represented as Σ. The set of all possible sequences of n, $n\ in\mathbb{Z}$ symbols from Σ is Σ^n and the set of all sequences of symbols from Σ with all possible lengths, including the empty sequence sequence ϵ, is Σ^*.

Two sequences u and $v \in \Sigma^*$ can be concatenated to form a sequence uv. For example, using a binary alphabet $\Sigma = \{0, 1\}$ and u = 1010 and v = 111, they can be concatenated to form uv = 1010111.

Note that $|uv| = |u| + |v|$. Concatenation is associative, which means $u(vw) = (uv)w = uvw$, but it is not commutative, as uv is not necessarily equal to vu. This means that Σ^* with the operation of concatenation is a Monoid, as it is a set with an associative operation with an identity element (the empty string).

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

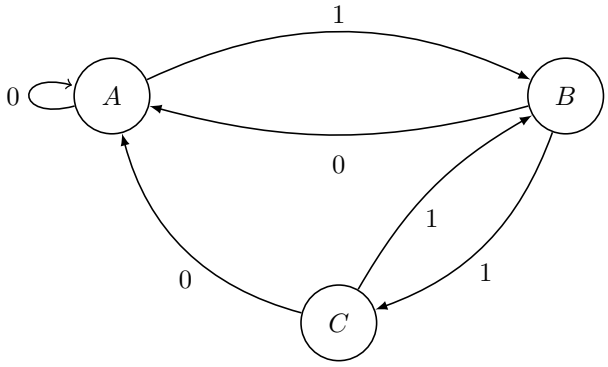

Figura 2.1: *Example of a graph with Q = {A, B, C} and* $\Sigma = \{0, 1\}$.

2.2 Graphs

Definition 2.1 – Graph

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

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

Each state q from Q has at most $|\Sigma|$ outgoing edges. Each outgoing edge from a state is labeled with a unique symbol from Σ and it arrives at only one state $q' \in Q$. This behavior is described by the function $\delta : Q \times \Sigma \rightarrow Q$, which is the transition function. For example, leaving state q with the edge labeled with a and arriving at state q' is represented by $\delta(q, 0) = q'$. Figure 2.1 shows an example of a graph over a binary alphabet.

A graph G can also be represented as a triple (Q, E, L) where Q is the set of states, E is the set of edges connecting the states and L is the set of edges' labels. A walk in a graph G is the sequence of labels $l \in L$ formed by starting at a state $q \in Q$ and a string s that starts as the empty string and going to a next state connected to it by a vertex and appending the vertex's label to s. This process can be repeated and when it stops, s defines a walk over G starting at q. Calling the graph of Figure 2.1 as G, the following is an example of a walk: start at state A and go to A, A, B, C, B, A. This forms the string $s = 001110$.

Definition 2.2 – Follower Set

The follower set of the graph G rooted at state $q \in Q$ *is defined as the set of all possible walks that can be formed by starting at state q. That is:* □

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

where $q{\cdot}\omega$ denotes the state arrived after starting at q and following the path determined by the word ω.

Definition 2.3 – Language of a Graph

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

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

A word *w* is called a synchronizing word of *G* if all walks in *G* that generate *w* terminate at the same state and *G* is called *synchronizing* if there exists a synchronization word for every state $q \in Q$.

2.3 Graph Minimization

This section explains the two most widely used algorithms for automata minimization: Moore and Hopcroft.

Definition 2.4 – 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 defines an equivalence relation on E and the set of all equivalence classes [x], $x \in E$, of an equivalence relation in E defines a partition of the set. □

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

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

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

Given partitions $\mathcal{P}$ and $\mathcal{P}'$ of disjoint sets *E* and *E'*, the partition of set $E \cup E'$ whose restriction to *E* and *E'* are $\mathcal{P}$ and $\mathcal{P}'$ is denoted by $\mathcal{P} \vee \mathcal{P}'$. It is possible to write $\mathcal{P} = \vee_{P \vee \mathcal{P}}\{P\}$.

Definition 2.5 – Irreducible Graph

A graph G is said to be irreducible if all its states have distinct follower sets (from Definition 2.2), that is F(p) ≠ F(q) for each pair of distinct states p, q ∈ Q. □

From Definition 2.5 it is possible to define an equivalence relation called the Nerode equivalence:

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

A graph is considered minimal if and only if its Nerode equivalence is the identity. The problem of minimizing a graph is that of computing the Nerode equivalence. The quotient graph *G*/≡ obtained by taking for *Q* the set of Nerode equivalence classes. The minimal graph is unique and it accepts the same language as the original graph.

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

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

the partition composed of two non-empty subsets:

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

and

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

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

Proposition 2.1

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

Lemma 2.1

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

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

and consequently

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

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

2.3.1 Moore's Algorithm

Algorithm 1 Moore(G)

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

Given a graph *G = (Q, Σ, δ)*, the set $L_q^{(h)}$ is defined as:

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

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

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

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

Proposition 2.2

For two states p, q ∈ Q and h ≥ 0, one has

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

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

Proposition 2.3

For h ≥ 0, one has

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

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

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

2.3.2 Hopcroft's Algorithm

Algorithm 2 Hopcroft(*G*)

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

The notation min(*P, P'*) indicates the set of smaller size of the two sets *P* and *P'* or any of them when both have the same size. Hopcroft's algorithm computes the coarsest partition that saturates

the set *F* of final states. The algorithm keeps a current partition $\mathcal{P} = \{P_1, \ldots, P_n\}$ and a current set $\mathcal{W}$ of splitters (i.e. pairs (*W,σ*) that remain to be processed where *W* is a class of $\mathcal{P}$ and σ is a letter) which is called the *waiting set*. $\mathcal{P}$ is initialized with the initial partition following the same criteria as described in Moore's algorithm. The waiting set is initialized with all the pairs (min(*F,* F^c), σ) for $\sigma \in \Sigma$.

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

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

2.4 Probabilistic Finite State Automata

Definition 2.6 – Probabilistic Finite State Automata

A Probabilistic Finite State Automaton (PFSA) P is defined as a quadruple $(Q, \Sigma, \delta, \mathcal{V})$*. The first three items are the same as of a graph as defined in Definition 2.1, while* $\mathcal{V}$ *is a probability function,* $\mathcal{V}$*:* $\delta \rightarrow [0, 1)$ *which associates a probability to each edge.* □

The function $\mathcal{V}$ gives the probabilistic factor to the PFSA. It is the associated probability distribution for the outgoing edges of each state. This means that for each state $q \in Q$, there will be a probability $\mathcal{V}(\delta(q, \sigma)), \forall \sigma \in \Sigma$ associated to each edge in such a way that $\Sigma_\sigma \mathcal{V}(\delta(q, \sigma)) = 1$ and $0 \leq \mathcal{V}(\delta(q, \sigma)) \leq 1$. The probability associated to an edge is the probability of taking this path once the system is in the state *q*.

Definition 2.7 – Morph

The probability distribution $\mathcal{V}(q) = \{\mathcal{V}(\delta(q, \sigma)); \forall \sigma \in \Sigma\}$ *of a state is called the state morph.* □

A PFSA can be represented with a graph, in which each $q \in Q$ is represented as a node. The edges are given by function $\delta(q, a) = q'$, indicating there is an edge going from *q* to q' labeled with the symbol *a*. The probability associated with an edge, $\mathcal{V}(\delta(q, a))$ is also in the edge label.

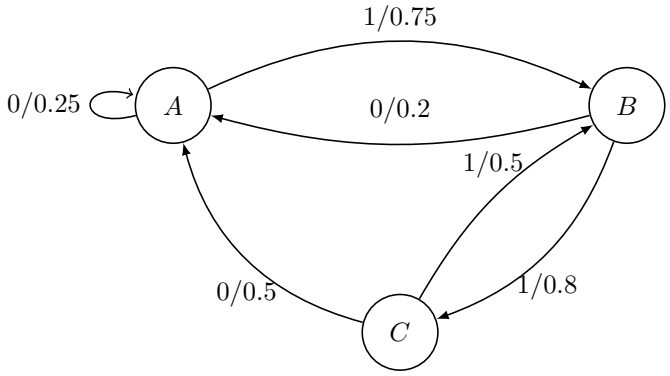

Figura 2.2: *A probabilistic version of the graph of Figure 2.1.*

Definition 2.8 – Deterministic PFSA

A PFSA is called deterministic if the outgoing edges of each state are labeled with distinct symbols. □

An example of a graph of a PFSA is shown in Figure 2.2, for which $Q = \{A, B, C\}$, $\Sigma = \{0, 1\}$ and the functions δ and $\mathcal{V}$ are represented in the edges of the graph.

A sequence can be generated by a PFSA as the sequence formed by starting at a given state $q \in Q$ then following a path with its edges and concatenating the labels for each edge. Its probability is given by multiplying the probability of each edge that was taken.

From Figure 2.2, starting at the state *A*, it is possible to form the sequence $u = 1011001$ 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 $p(u) = 0.75 \times 0.2 \times 0.75 \times 0.8 \times 0.5 \times 0.25 \times 0.75 = 0.0084375$.

It is useful do adapt the concept of synchronization word to the context of PFSA:

Definition 2.9 – PFSA Synchronization Word

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

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

□

Definition 2.9 means that the probability of obtaining any sequence after the synchronization word does not depend on whatever came before w. The main problem with this definition is the fact that is not possible to check (2.1) for all $u \in \Sigma^*$ and for all $v \in \Sigma^*$ as there are an infinite number of sequences.

A solution uses the d-th order derived frequency, which is the probability using u and v from Σ^d,

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

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

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

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

2.4.1 Initial Partition for PFSA

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

Definition 2.10

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

2.5 Consolidated Algorithms

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

2.5.1 D-Markov Machines

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

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

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

Tabela 2.1: *Sequence s subsequence probabilities.*

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

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

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

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

2.5.2 CRISSiS

The Compression via Recursive Identification of Self-Similar Semantics (CRISSiS) algorithm was presented in [ref]. It starts with a stationary symbol sequence *X* of length N which should be generated by a synchronizable and irreducible PFSA. CRISSiS estimates the original PFSA by looking at its output sequence. CRISSiS is shown in Algorithm 3 and it consists of three steps:

Identification of Shortest Synchronization Word

Using the definition of Synchronization Word given by 2.9, CRISSiS uses brute force to find the shortest synchronization word. This is shown in Algorithm 4 where each state's morph is checked

Algorithm 3 CRISSiS

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

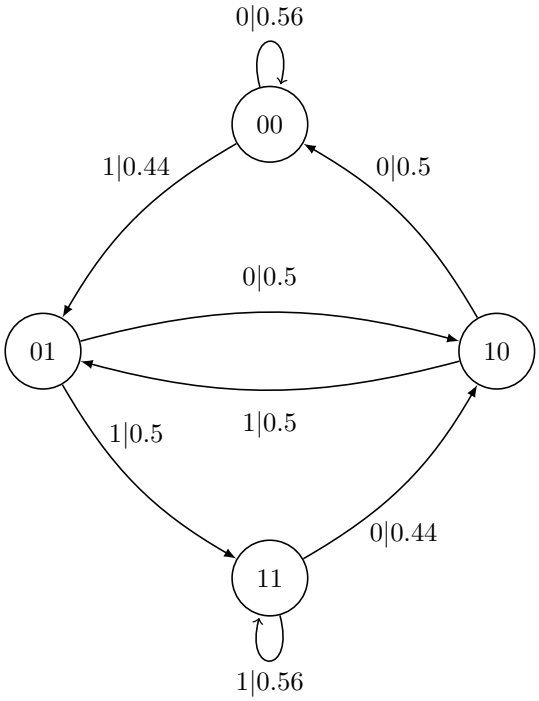

Figura 2.3: *A D-Markov machine for sequence s and D = 2.*

with its extensions' morphs up to a length L_2. If all statistical tests are positive for a given word, it is returned as the synchronization word to be used.

Algorithm 4 isSynString(ω, L_1)

```
Outputs: true or false
for D = 0 to L_1 do
    for all s ∈ Σ^D do
        if V_d(ω) = V_d(sω) fails the statistic test for some d ≤ L_2 then
            return false
return true
```

Recursive Identification of States

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

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

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

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

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

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

The next procedure starts by letting Q be the set of states to be discovered for the PFSA and it is initialized containing only the state defined by the synchronization word ω_{syn} found in the first step. Then, a tree is constructed using ω_{syn} as the root node to $|\Sigma|$ children. Each one of the children nodes is regarded as a candidate states with a representation $\omega_{syn}\sigma$ for $\sigma \in \Sigma$. Each one of them will be tested using Equation 2.5 with each of the states in Q. If a match is found, the child state is removed and its parent σ-transition should be connected to the matching state. If it does not match any state in Q, it is considered a new state and it is then added to Q and it should also be split in $|\Sigma|$ new candidate states. This procedure is to be repeated until no new candidate states have to be visited.

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

Algorithm 5 matchStates(ω, Q, L_2)

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

Estimation of Morph Probabilities

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

Example

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

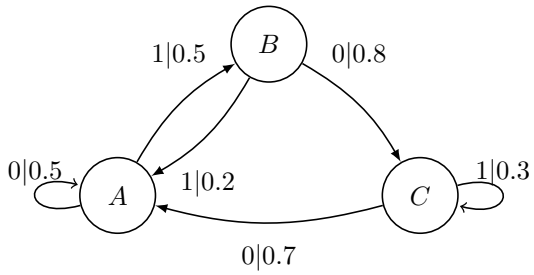

Figura 2.4: *The Tri-Shift PFSA.*

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

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

10000. Table 2.2 gives the frequency count of some subsequences occurring in *X*. In this example, $L_1 = L_2 = 1$.

First, the synchronization word needs to be found. States 0, 1 and so on are checked with Equation 2.2. Starting by 0, neither $\Pr_1(0) = \Pr_1(00)$ nor $\Pr_1(0) = \Pr_1(01)$ pass the χ^2 test. Then the state 1 is tested, which also fails. For state 00, the derived frequencies are relatively close and it passes the test, giving 00 the status of synchronization word.

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

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

is different from both 00 and 001, therefore it is a new state, it is added to Q and split into the new candidates 00100 and 00101. $\mathcal{V}_1(0011) = [0.5000.500]$ passes the test with $\mathcal{V}_1(00)$, which means that 0011 is removed and the edge from 001 to 0011 goes back to 00. This leads to the configuration in Figure 2.7.

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

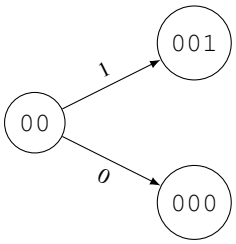

Figura 2.5: *Tree with 00 at its root.*

Time Complexity

As shown in [ref], CRISSiS operates with a time complexity of $O(N) \cdot (|\Sigma|^{O(|Q|^3)+L_1+L_2} + |Q||\Sigma|^{L_2})$, where N is the length of the input sequence, $|\Sigma|$ is the sequence's alphabet size, $|Q|$ is the number of states in the original PFSA and L_1 and L_2 are parameters determining how much of the past and future of a state is needed to determine it. It is stated that as L_1 and L_2 are both usually

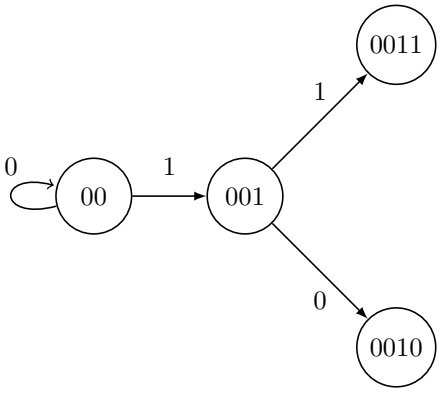

Figura 2.6: *Second iteration of three.*

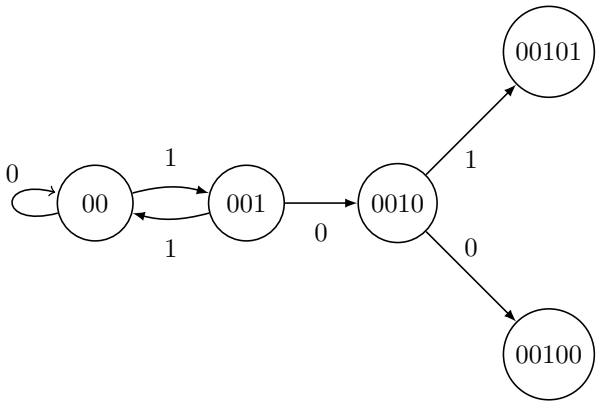

Figura 2.7: *Third iteration of the three.*

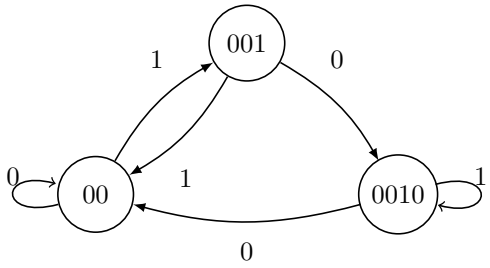

Figura 2.8: *Recovered Tri-Shift topology.*

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

CAPÍTULO 3

ALGORITHMS DESCRIPTIONS

3.1 A New Algorithm for Finding Synchronization Words

Given a sequence X of length L over an alphabet Σ which is the output of a dynamical system, the proposed algorithm is an alternative to find possible synchronization words in X. The typical method would be using Equation 2.2 and testing for all possible values. The proposed algorithm will use data structures in order to speed up the process.

The algorithm uses two rooted tree with probabilities to search for synchronization words. The main tree $\mathcal{T}$ represents the statistics of sub-sequences of the original sequence X and the auxiliary tree $\mathcal{S}$ is used to search the suffixes of those sub-sequences and keep track of their status as valid candidates for synchronization words. The auxiliary tree regulates how the main tree is explored while the search is performed. Once the expansion reaches its end, a list of the most likely synchronization words is returned.

A Rooted Tree with Probabilities $\mathcal{T}$ over $\Sigma = \{0, 1\}$ is presented via an example in figure 3.1. $\mathcal{T}$ consists of a set of branching nodes $\mathcal{B}$ and a set of leaf nodes $\mathcal{L}$. All nodes have exactly one predecessor (with the exception of the root node, which has no predecessors). Leaf nodes have no successors, while each branching node has $|\Sigma|$ successors as each element of Σ labels one of the outgoing branches. Those branches are also labeled with the probability of leaving the node with that symbol. Each node is labeled with the string formed from concatenating the symbols in the branches in the path from the root to the current node. The root node is labeled with the empty string ϵ. The

probability of reaching a node is given by multiplying the probabilities labeling the branches in the path from the root node to the current node.

For example, the leaf node *10* is labeled as so because to reach it, the path taken from the root node ϵ is first *1* and then *0*. The probability of reaching this node is $P(1) \times P(0|1)$, that is the probability of leaving the root node with 1 (which is $P(1)$) multiplied by the probability of leaving the node 1 with 0 (that is, $P(0|1)$.

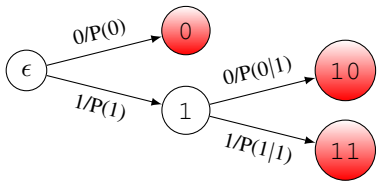

Figura 3.1: *Example of a rooted tree with probabilities.*

The two rooted tree with probabilities used in the algorithm are $\mathcal{T}$ and $\mathcal{S}$. The primary tree $\mathcal{T}$ has branch probabilities taken from the conditional probabilities from the sequence. The auxiliary tree $\mathcal{S}$ is constructed from $\mathcal{T}$. Each node has a label that is the inverse from the node label in $\mathcal{T}$, but keeping the same branch probabilities. This is done in order to use the nodes in $\mathcal{S}$ to verify the suffixes of nodes in $\mathcal{T}$ as it is explained in an example.

From these two trees, two dynamic lists are created. The list Δ is initialized with the root node from $\mathcal{T}$ and all of its children nodes. The elements in the second list Γ are triples *(s, candidacy, tested)*. *s* is a state from $\mathcal{S}$ and the other two elements are binary flags. The *candidacy flag* is checked true if the state *s* is a valid candidate for synchronization word. The *test flag* is checked true if *s* have been through all the statistical tests to check its candidacy for synchronization word status.

Definition 3.1

A state s from a rooted tree with probabilities $\mathcal{S}$ is called a valid state if the triple (s, candidacy, tested) in list Γ has the candidacy flag set to true and the tested flag set to false. □

From the this definition, a state is called valid when it is a valid candidate for a synchronization word, but its status is yet to be tested. Whenever a new element is added to Γ it is added as a valid state. Γ is initialized with the triple *(ϵ, True, False)*, where ϵ is the root of $\mathcal{S}$ and, as stated before, it is initialized as a valid state. The function *nextValidState* returns the valid state in Γ with the shortest label.

The algorithm receives the parameter *W* which indicates the maximum length of subsequences it will take into consideration, i.e. the depth of the tree $\mathcal{T}$. The depth of $\mathcal{S}$ is $W - 1$.

Figures 3.2 and 3.3 are used as examples. They are both taken over binary alphabet and with branch probabilities taken from a computer generated sequence. W is taken to be equal to 3. $\mathcal{S}$ is constructed from $\mathcal{T}$ by taking the labels from each state and reversing them, while keeping the branch probabilities. To show how this is useful to find suffixes, take the node *110* from $\mathcal{T}$, reverse it (obtaining *011*) and walk through $\mathcal{S}$. Starting at ϵ, the walk goes through *0* and then *01*. Reverting each visited state, it is seen that ϵ, *0* and *10* are suffixes of *110*, as it was expected.

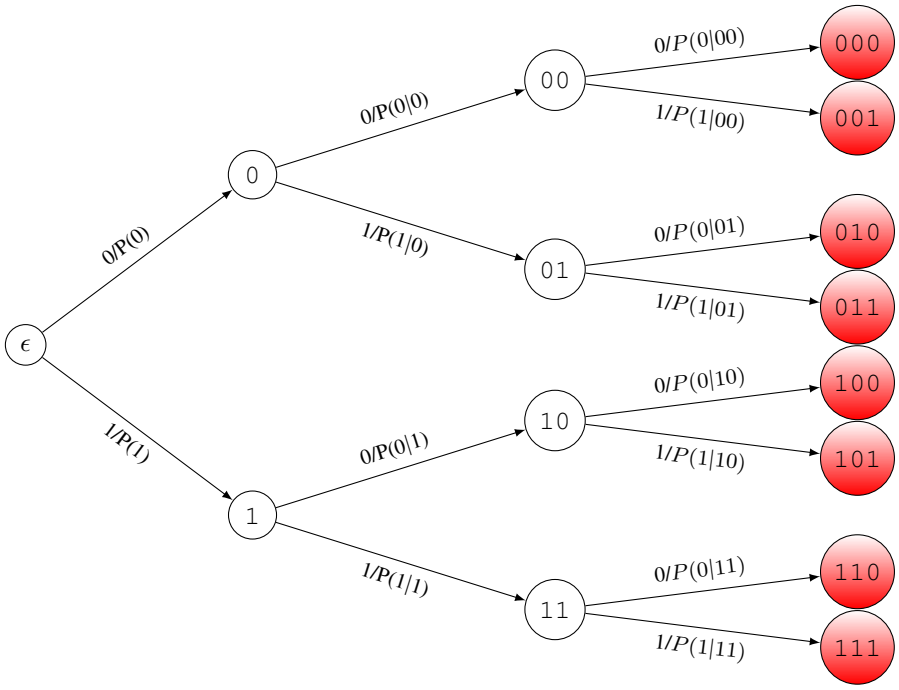

Figura 3.2: *Example of binary* $\mathcal{T}$

When two morphs are compared as in $\mathcal{V}(q) = \mathcal{V}(p)$ it means that both of these distributions are being compared via an appropriate statistical test and this operation returns a true or false whether the test was successful or not for a predetermined confidence level α.

To find the synchronization words, algorithm 6 is used. It receives as parameters both trees and *W*. The lists Γ and Δ are then created as described above. Besides that, the list Ω_{syn}, which will receive the results from the algorithm, is initialized as an empty list.

At the start of each iteration, the function *nextValidState* is applied to Γ and returns the shortest valid candidate for synchronization word c. If no valid states are found, Ω_{syn} will receive the labels of all elements in Γ for which the candidacy flags are set to true. Given the tuple $\omega = (x_1, x_2, \ldots, x_t)$, the notation $\omega.x_i$ is used to denote the element x_i from ω. In other words, Ω_{syn} receives all elements γ from Γ for which $\gamma.candidacy$ is true. This list is then returned. As there are no valid states, this

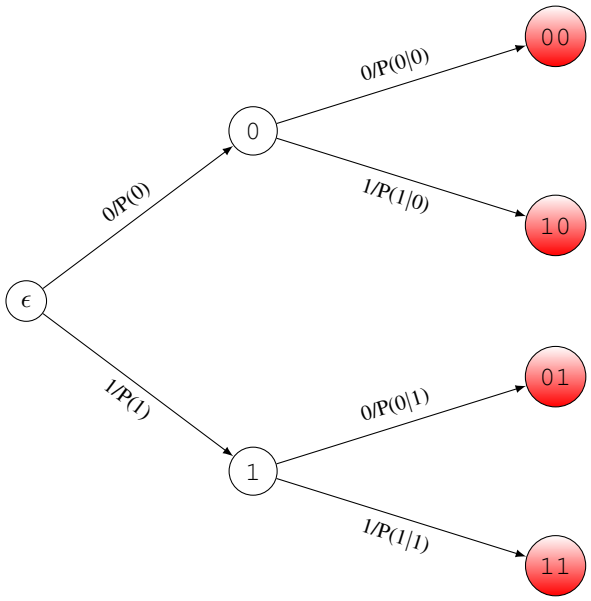

Figura 3.3: *Example of binary $\mathcal{S}$*

means that all possible states have been tested and the states which still have the candidacy flag set to true are the synchronization words to be returned.

If there is a valid state c. If the label of c is shorter than W, another list Λ is created with all the elements from Δ whose lengths are less than the length of c. On the other hand, if it larger than the window size, this candidate is not taken into consideration.

Each element λ of Λ is then compared to c. First, it is checked if *c.label* is a suffix of λ. In the affirmative case, the label of c and λ are compared via a statistical test (usually the χ^2 test as noted before).

If the test result is positive, the current candidate keeps its candidacy status and the algorithm keeps iterating. After c is tested against all states from Λ for which it is a suffix, the candidate is marked as tested.

On the other hand, if the test fails, Γ and Δ are expanded via Algorithm 7 and all states from Γ are marked as untested as they will have to go through a new phase of testing with the new states that were added to the lists Γ and Δ from the expansion.

This is a way of implementing Equation 2.2. Each word is tested against all its possible suffixes in each iteration and if it still keeps the candidacy flag set to true, it is a synchronization word. The tree structure improves on the brute force method as no unnecessary tests will be performed for longer words if a shorter one is still valid.

Algorithm 7 will properly expand Γ and Δ after a statistical test fails. First, the candidate for

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

1: **procedure** INITIALIZATION
2: $\Gamma \leftarrow \{(\epsilon, True, False)\}$
3: $\Delta \leftarrow \{\epsilon, \delta(\epsilon_T, \sigma)\ \forall \sigma \in \Sigma\}$
4: $\Omega_{syn} \leftarrow \{\emptyset\}$
5: **procedure** MAINLOOP
6: $c \leftarrow nextValidState(\Gamma)$
7: **if** $\nexists$ valid states in Γ **then**
8: $\Omega_{syn} \leftarrow \{\gamma \in \Gamma : \gamma.candidacy = True\}$
9: **return** Ω_{syn}
10: **else**
11: $l \leftarrow length(c.label)$
12: **if** $l < W$ **then**
13: $\Lambda \leftarrow \{s \in \Delta : length(s) > l\}$
14: **for** each $\lambda \in \Lambda$ **do**
15: **if** *c.label* is a suffix of λ **then**
16: $p \leftarrow \mathcal{V}(c(label)) = \mathcal{V}(h)$
17: **if** $p = False$ **then**
18: c.candidacy = False
19: $expandTrees(c.label, \mathcal{S}, \mathcal{T}, \Gamma, \Delta, \Sigma)$
20: $\gamma.tested = False\ \forall \gamma \in \Gamma$
21: **break**
22: **else**
23: **if** all elements in Λ were tested **then**
24: $c.tested = True$
25: **goto** *MAINLOOP*.

which the test failed is expanded in Δ: its children are stored in a list called Φ and Δ is updated appending these new states. As the nodes in $\mathcal{S}$ have inverted labels in regard to $\mathcal{T}$, the components of Φ have their names inverted and stored in Υ. For each element υ in Υ, its shortest valid suffix in $\mathcal{S}$ is found and stored in η. To find the shortest valid suffix, the procedure described before to find a suffix is used and it stops when a state in $\mathcal{S}$ is found for which its *candidacy* flag in Γ is true. If the label of υ and η are the same state, the list Π receives all of the children of υ. After repeating this process for all elements in Υ, the updated Π is appended to the list Δ.

When Algorithm 7 is called, the lists of tree elements are updated with the new nodes that will be needed for synchronization worda analysis.

Algorithm 7 expandTrees($c, \mathcal{S}, \mathcal{T}, \Gamma, \Delta, \Sigma$)

procedure EXPAND TREES
  $\Phi \leftarrow \{(\delta(c,\sigma), True, False),\ \forall\sigma \in \Sigma\}$
  $\Gamma \leftarrow \Gamma \cup \Phi$
  $\Upsilon \leftarrow \{invert(\phi.label), \forall\phi \in \Phi\}$
  $\Pi \leftarrow \{\emptyset\}$
  for each υ in Υ **do**
    $\eta \leftarrow$ shortest valid suffix of υ in $\mathcal{S}$
    if $\eta = \upsilon$ **then**
      $\Pi \leftarrow \Pi \cup \{\delta(\upsilon,\sigma)\ \forall\sigma \in \Sigma\}$
  $\Delta \leftarrow \Delta \cup \Pi$
  return

3.1.1 An Example

3.2 Tree Termination

3.3 Graph Construction

CAPÍTULO 4

RESULTS

In this chapter, some examples of dynamic systems are used to compare the efficiency of the previously described algorithms. First, from the original system a discrete sequence *S* over the alphabet Σ of length $N = 10^7$ is generated. The frequencies of subsequences occurring in *S* up to a length L_{max}. After this, three PFSA are created using the D-Markov Machine, CRISSiS and our algorihm with different values of D, L and L_1 and L_2. Sequences of length *N* are generated from each of those PFSA and using the metrics explained in Section 4.1 their performances are estimated. The results comparing the performances of the three algorithms' PFSA are then shown in Section 4.2.

4.1 Evaluation Metrics

4.1.1 Number of States

This is the simplest comparison parameter. Each PFSA generated by the algorithms will have a set of states *Q* of size |*Q*|. When the original PFSA is known, it is useful to compare if the generated PFSA has a number of states close to the original. When that is not the case and all that is known is the original sequence, ideally it would be better to have a smaller PFSA to model the system. But there might be a trade-off where the smaller PFSA will result in the other metrics described in the following sections are deteriorated, while a bigger and more complex model achieves better performance. But given similar performances, the best model is the one with less states.

4.1.2 Entropy Rate

In Information Theory, entropy is a measure of the average information contained in a system. Using the definition in [3] and [1], the Conditional Entropy is given by:

$$H(X|Y) \triangleq -\sum_{x \in X} P(x|Y) \log P(x|Y). \tag{4.1}$$

The Entropy Rate is a measure of how dependent the current symbol of a sequence is on all the symbols that came before it and it is defined as:

$$h \triangleq \lim_{n \to \infty} H(X_n|X_1X_2 \ldots X_{n-1}) = -\sum_{x \in X} P(x_n|X_1X_2 \ldots X_{n-1}) \log P(x_n|X_1X_2 \ldots X_{n-1}) \tag{4.2}$$

where x_n is the n-th output symbol in the sequence *S*. When Equation 4.2 converges for a certain *n*, this indicates that the system that generated the sequence has a memory of *n*, that is, knowing the *n* previous symbols is enough to estimate the next one.

As it is not possible to compute Equation 4.2 up to infinity, we use the ℓ-order Entropy Rate defined as:

$$h_\ell \triangleq H(X_\ell|X_1X_2 \ldots X_{\ell-1}), \tag{4.3}$$

where ℓ should be chosen around a value where the system converges so the memory can be correctly estimated. If the entropy rate does not converge, it should be chosen to be at an inflection point. Comparing the values of ℓ-order entropy rate of the generated PFSA with the one from the original system is useful to test if the generated one correctly captures the system memory.

4.1.3 Kullback-Leibler Divergence

The Kullback-Leibler Divergence is a method to compare the distance between two probability distributions *P* and *Q*. Its formula is given by:

$$D(P||Q) = \sum_i P(i) \log(\frac{P(i)}{Q(i)}). \tag{4.4}$$

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. The smaller it is, the closer they are.

For the purpose of comparing the algorithms, consider two PFSA $K_1 = (\Sigma, Q_1, \delta_1, \mathcal{V}_1)$ and $K_2 = (\Sigma, Q_2, \delta_2, \mathcal{V}_2)$ over a common alphabet Σ. $P_1(\Sigma^\ell)$ and $P_2(\Sigma^\ell)$ are the steady state probability vectors of generating sequences of length ℓ from PFSA K_1 and K_2 respectively. For a given ℓ we take the ℓ-order Kullback-Leibler Divergence as:

$$D_\ell(K_1||K_2) = \sum_{\sigma \in \Sigma^\ell} P_1(\sigma) \log(\frac{P_1(\sigma)}{P_2(\sigma)}). \tag{4.5}$$

Instead of comparing the PFSA directly, D_ℓ can be used to compare the probability distribution of sub-sequences of length ℓ of a sequence generated by K_1 and another generated by K_2. A small divergence will indicate that the sequence generated by the algorithm is statistically close to the original sequence, which shows that the PFSA is a good estimate for the original system.

4.1.4 Φ-Metric

The Φ-Metric was presented in [1] as a way to compare to different PFSA. Given two PFSA $K_1 = (\Sigma, Q_1, \delta_1, \mathcal{V}_1)$ and $K_2 = (\Sigma, Q_2, \delta_2, \mathcal{V}_2)$ over a common alphabet Σ and $P_1(\Sigma^\ell)$ and $P_2(\Sigma^\ell)$ defined as in Section 4.1.3. The Φ-Metric is then defined as:

$$\Phi(K1, K2) \triangleq \lim_{n \to \infty} \sum_{j=1}^{n} \frac{\|P_1(\Sigma^j) - P_2(\Sigma^j)\|_{\ell_1}}{2^{j+1}}, \tag{4.6}$$

where $\| \star \|_{\ell_1}$ indicates the sum of absolute values of the elements in the vector $\star$. As Equation 4.6 puts more weight in shorter words, it can be truncated with a relatively small ℓ by the ℓ-order Φ-Metric:

$$\Phi_\ell(K1, K2) \triangleq \sum_{j=1}^{\ell} \frac{\|P_1(\Sigma^j) - P_2(\Sigma^j)\|_{\ell_1}}{2^{j+1}}. \tag{4.7}$$

As with the Kullback-Leibler Divergence, the Φ-Metric can be applied to compare a PFSA and a sequence generated by a PFSA or two PFSA-generated sequences by using the distribution of sub-sequences of length ℓ. A small Φ indicates that the PFSA are similar to each other. For the performance comparison, each PFSA generated by the algorithms is compared with the original sequence using Equation 4.7 and the smaller the result, the better the algorithm models the original system.

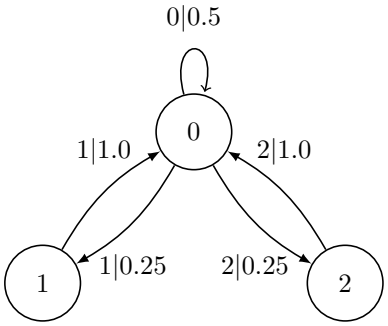

Figura 4.1: *The graph of a Ternary Even-Shift.*

	D-Markov		**Algo/CRISSiS**
D/L	6	7	2
# of States	169	339	3
h_{10}	1.0084	1.0058	1.0058
D_{10}	$2.7 \cdot 10^{-3}$	$4.16 \cdot 10^{-5}$	$9.55 \cdot 10^{-5}$
Φ_{10}	$2.1 \cdot 10^{-3}$	$1.2 \cdot 10^{-3}$	$2.3 \cdot 10^{-3}$

Tabela 4.1: *Results for Ternary Even Shift.*

4.2 Test Cases

4.2.1 Ternary Even Shift

The Ternary Even Shift is a symbolic dynamic system with a ternary alphabet where there can be no odd-numbered succession of non-zero symbols between zeros. This means that there must be an even number of 1's or two's between 0's. This is represented by the graph shown in Figure 4.1. For this system, 0 is a synchronization word.

The results of our algorithm are compared to D-Markov and CRISSiS in Table 4.1. Our algorithm used the Omega termination and handed the same results for any L greater than 2. D-Markov machines of D = 6 and 7 were used. CRISSiS was tested using $L_1 = L_2 = 1$. It is possible to see that in this case, both CRISSiS and our algorithm reconstruct the same PFSA (shown in Figure 4.2) and are a good estimate to the original PFSA while a large D-Markov machine of at least 169 states is needed to obtain approximately the same performance. Even though D = 6 and 7, these D-Markov machines do not have 64 and 128 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 $h_{10} = 1.0013$, which is close to the value found by all the algorithms.

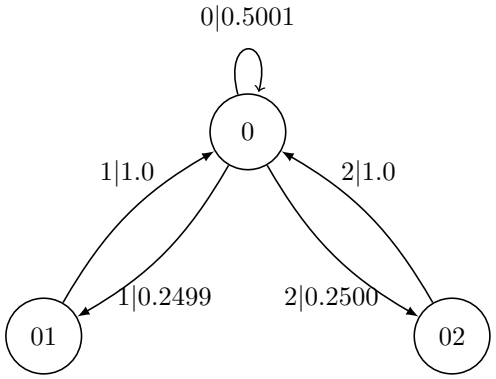

Figura 4.2: *The graph of a Ternary Even-Shift generated by out algorithm and by CRISSiS.*

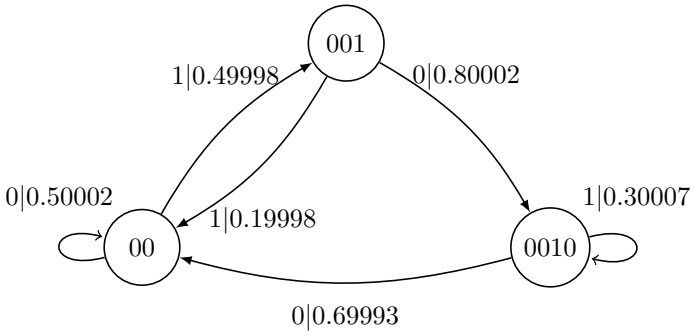

Figura 4.3: *The Tri-Shift PFSA generated by our algorithm and by CRISSiS.*

4.2.2 Tri-Shift

The Tri-Shift was previously discussed in Section 2.5.2 and its graph is shown in Figure 2.4. It was shown that its synchronization word is 00. The comparative results are shown in Table 4.2. Once again this is an example where our algorithm and CRISSiS are able to recover the three states from the original PFSA with a good estimate for the morphs as seen in Figure 2.8. To obtain a similar performance with a D-Markov machine, 128 or 256 states might be needed. The original system presented has $h_{10} = 0.9005$, showing that our algorithm, CRISSiS and the 7-Markov Machine are able to capture the system's memory.

	D-Markov		**Algo/CRISSiS**
D/L	7	8	2
# of States	128	256	3
h_{10}	0.9016	0.9005	0.9001
D_{10}	$4.1 \cdot 10^{-3}$	$1.65 \cdot 10^{-3}$	$1.16 \cdot 10^{-3}$
Φ_{10}	$2.1 \cdot 10^{-3}$	$7.2 \cdot 10^{-4}$	$8.2 \cdot 10^{-4}$

Tabela 4.2: *Results for the Tri-Shift.*

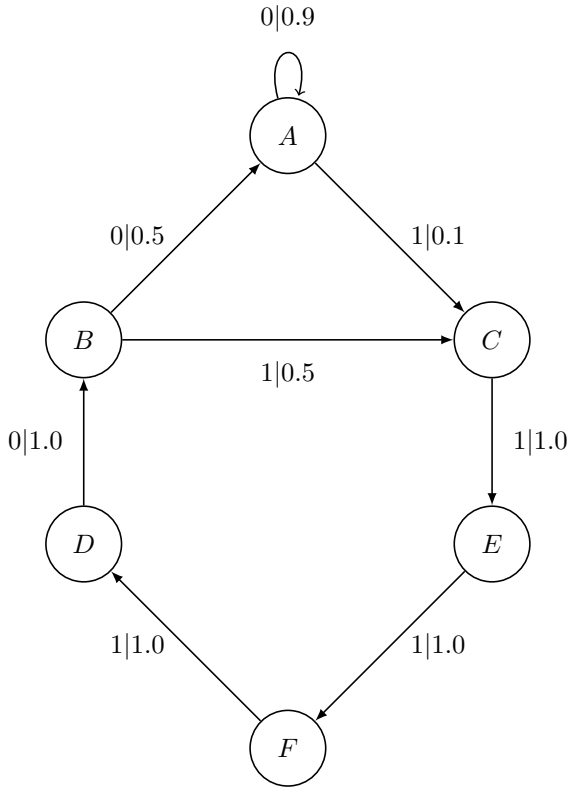

Figura 4.4: *A Six-State PFSA.*

4.2.3 A Six State PFSA

Figure 4.4 shows a PFSA with six states that elucidates the differences between our algorithm and CRISSiS. This system has 4 synchronization words: 00, 01, 10 and 1111. The comparative results between the algorithms is show in Table 4.3.

In this example, CRISSiS using $L_1 = L_2 = 1$ is not able to recover the original machine . It creates a PFSA with 2 states (Figure 4.5) which generates sequences fairly different from the original, as after a small transient, it outputs sequences of just 1's. On the other hand, by using any L larger than 4, our algorithm is capable of reconstructing 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 = 3 or 4, obtaining a PFSA with 7 and 11 states respectively. Once again, some sequences do not occur, therefore the D-Markov Machine in those cases will not have 2^D states.

The results for this system show a type of system where CRISSiS performs badly. Starting from a single synchronization word and as many states have equal morphs and their paths will only be different after 3 steps, L_2 needs to be at least 3 to be able to recover the original PFSA and as CRISSiS' complexity is exponential on L_2 this means a hindrance in performance. As our algorithm uses all

	D-Markov		**Algo**	**CRISSiS**
D/L/ $_1$ and L_2	3	4	4	1/1
# of States	7	11	6	2
h_{10}	0.5341	0.3344	0.3344	$1.4427 \cdot 10^{-7}$
D_{10}	1.1980	$4.0499 \cdot 10^{-6}$	$5.6969 \cdot 10^{-5}$	43.6556
Φ_{10}	$2.0005 \cdot 10^{-3}$	$4.6072 \cdot 10^{-4}$	$9.3745 \cdot 10^{-4}$	2.6505

Tabela 4.3: *Results for the Six-State PFSA.*

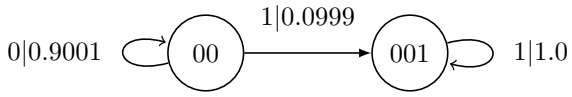

Figura 4.5: *The Recovered Six-State PFSA by CRISSiS with* $L_1 = L_2 = 1$.

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

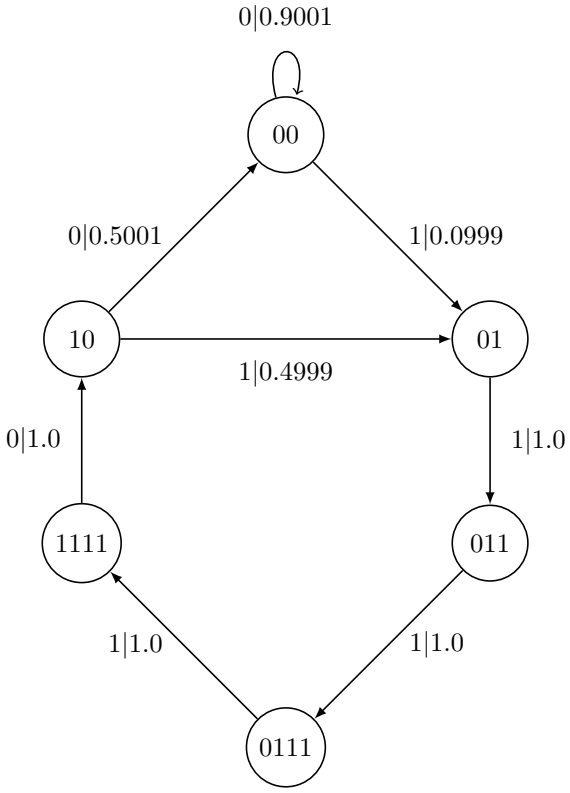

Figura 4.6: *The Recovered Six-State PFSA by our algorithm.*

Capítulo 5

TODO

TODO

Capítulo 6

TODO

TODO

APÊNDICE A

TODO

TODO

APÊNDICE B

TODO

SOBRE O AUTOR

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

Endereço: Endereço

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

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

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

BIBLIOGRAFIA

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

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

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

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

[5] E. Vidal, F. Thollard, C. de la Higuera, F. Casacuberta, and R. C. Carrasco. Probabilistic finite-state machines - part i. *IEEE Transactions On Pattern Analysis And Machine INtelligence*, 27(7):1013–1025, July 2005.

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