Homing and Synchronizing Sequences

Yuren Huang
yurhuang@cs.stonybrook.edu

Department of Computer Science
Stony Brook University

October, 2016

Abstract

Finite state machines (FSMs) have been extensively used to model systems, such as sequential circuits, programs in lexical analysis and communication protocals. The testing of an FSM means applying certain input sequences to it and getting desired information from the output. Testing is critical to ensuring a machine's normal functioning and helpful for detecting the behavioral pattern of a machine.

In this report, we focus on the theoretical aspects of a fundamental testing problem: determining the final state after the test when the initial state is unknown. We look into details of the solutions to this problem: homing sequences and synchronizing sequences. Algorithms and complexity of computing them are discussed, as well as related problems.

Contents

Chapter 1

Introduction

Over decades, hardware and software systems have been getting larger to handle more complicated tasks. Their growing scales demand testing to maintain reliability. In a variety range, finite state machines (FSMs) have been in use to model those systems, such as sequential circuits, programs in lexical analysis and communication protocals. Testing an FSM ensures its normal functioning, and meanwhile could detect its behavioral pattern. Therefore this topic is both practically important and theorectically interesting.

An FSM accepts inputs and produces outputs while transiting between a finite set of states. Testing is basically feeding a sequence of input symbols into the FSM and observing its outputs, from which we can deduce the information we are looking for. One of the most fundamental testing problems is to determine the final state after the test, when the description of the FSM is given but the initial state is unknown. The solution to this problem is homing and synchronizing sequences. Simply speaking, a homing sequence is an input sequence that derives ‘‘final-state-specified’’ outputs; a synchronizing sequence leads machine from any state to a particular one state.

The report is organized as follows. In Chapter 2 we introduce basic concepts of this topic. Chapters 3 and 4 discuss the fundamental theorectical problems and approaches. Then in Chapter 5 we talk about some of the applications and related problems.

Chapter 2

Preliminaries

2.1 Mealy Machines

There are several classes of FSMs, such as recognizers and transducers. Recognizers, including the commonly known deterministic finite automata (DFAs), only respond with accept or reject when applied inputs. Transducers are more general in the sense that they are allowed to have an output alphabet. Mealy machines and Moore machines are in this category, with the difference that in Mealy machines outputs depend on both input symbol and current state, while in Moore machines outputs only depend on current state. (Thus the latter is a subset of the former.) We will focus on Mealy machines in this report.

Definition 2.1.1. A **Mealy machine** is a 5-tuple $\mathcal{M} = \langle I, O, S, \delta, \lambda \rangle$, where

- I is a finite set called input alphabet,
- O is a finite set called output alphabet,
- S is a finite set of states,
- $\delta : S \times I \to S$ is the transition function,
- $\lambda : S \times I \to O$ is the output function.

(In some versions of definition, a Mealy machine also has a starting state. But since we don't use it here the definition is simplified.)

As shown in Figure 2.1, a Mealy machine could be described by a state diagram. Labels on the edges are in the form ''input/output'' and the directions of the edges represent that of transitions. For simplicity, we extend the functions δ and λ in the following way: Suppose $x = a_1...a_m$ is the input sequence, $y = b_1...b_m$ is the corresponding output and $s_0, ..., s_m$ being the states passed through, then $\delta(s_0, x) = s_m$ and $\lambda(s_0, x) = y$. Also, we define $\delta(Q, x) = \{\delta(s, x) \mid s \in Q\}, \forall Q \subset S, \forall x \in I^*$.

Note that for the testing problem of finding homing or synchronizing sequences, the description of the machine (i.e. I, O, S, δ and λ) is always given.

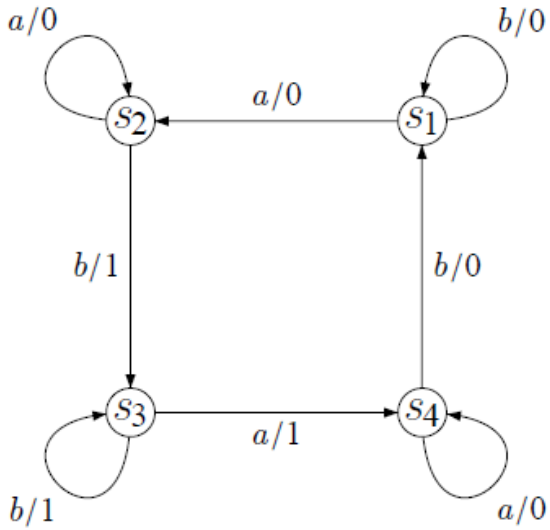

Figure 2.1: [9] A sample Mealy machine state diagram

2.2 Homing Sequences

Definition 2.2.1. Given a Mealy machine $\mathcal{M} = \langle I, O, S, \delta, \lambda \rangle$, a sequence $x \in I^*$ is said to be **homing** if $\forall (s,t) \in S \times S, \lambda(s,x) = \lambda(t,x) \implies \delta(s,x) = \delta(t,x)$.

Homing sequences, as the name indicates, take every state in the machine to a state that can be determined by the outputs. (Different states may be taken to the same state. So this can not decide an unknown initial state.) In the case of testing a machine with no reset function, homing sequences can be applied to get to a known state, serving as a reset.

Conformance testing [4] is such an example: Given a specification A of a machine i.e. its complete state diagram and an implementation B of it, which is a black box (only I/O behavior visible), the task is to decide whether the implementation is correct. A homing sequence is computed from A and applied to B to set it to a known state before the test.

2.3 Synchronizing Sequences

Definition 2.3.1. Given a Mealy machine $\mathcal{M} = \langle I, O, S, \delta, \lambda \rangle$, a sequence $x \in I^*$ is said to be **synchronizing** if $|\delta(S,x)| = 1$.

Synchronizing sequences send the machine from any state to the same final state. (Note that this state is not decided merely by the machine, it depends on the sequence too.) Thus they are a special type of homing sequences. But they also have a remarkable difference from homing sequences: They don't require observing outputs. This advantage makes them practical in some real life situation, like guiding. Suppose you have a map (i.e. the state diagram) and a destination to go (final state) but don't know your current location (initial state), a synchronizing sequence guarantees to bring you there. See Figure 2.2 for an illustration.

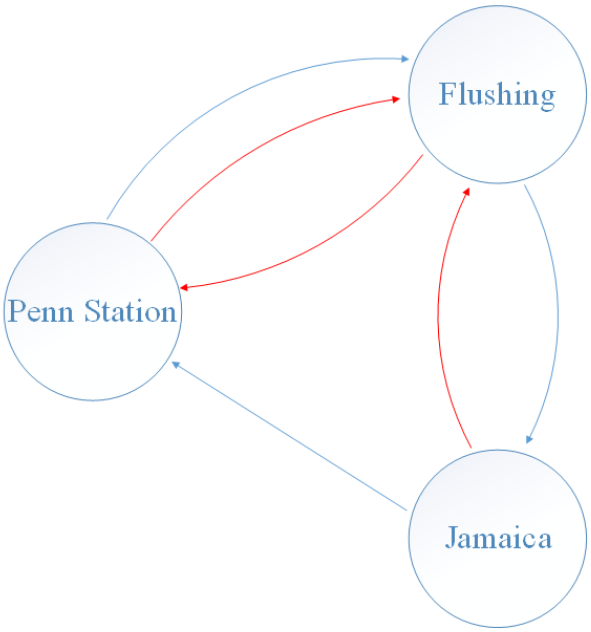

Figure 2.2: A map in which nodes are cities and edges are routes. A universal direction to Penn Station is $(\text{rbb})^2$.

Another application of synchronizing sequences is manipulating algebraic parts in the plane[6]. Manufacturers often use robots to efficiently fetch objects. The robots can be orientation-sensitive, meaning they only grasp objects oriented in some certain way. (Installing orientation sensors can be costly.) If the objects are randomly placed, they may need to be rotated, as shown in Figure 2.3. This is essentially a problem of finding a synchronizing sequence, for a machine whose states represent object orientations and input alphabet being directions of pushing.

2.4 Initial and Current State Uncertainty

Definition 2.4.1. Given a Mealy machine $\mathcal{M} = \langle I, O, S, \delta, \lambda \rangle$ and a sequence $x \in I^*$, the **initial state uncertainty** after x being applied is a partition of S: $\pi(x) = \{B_1, ..., B_r\}$ s.t. any

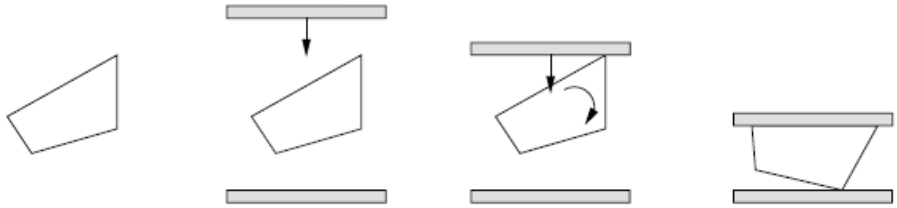

Figure 2.3: Rotating an object using two pushing walls. An input symbol means pushing in that direction until the object no longer rotates.

two states s and t belong to the same block B_i iff $\lambda(s, x) = \lambda(t, x)$. The index r here is the number of blocks. (For simplicity we write $\pi(x)$ instead of $\pi(\mathcal{M}, x)$, as the machine usually won't change in the context.)

Definition 2.4.2. Given a Mealy machine $\mathcal{M} = \langle I, O, S, \delta, \lambda \rangle$ and a sequence $x \in I^*$, the **current state uncertainty** after x being applied is $\sigma(x) = \{\delta(B_i, x) \mid B_i \in \pi(x)\, 1 \leq i \leq r\}$. An element $B \in \sigma(x)$ is also called a block.

It can be easily seen that the blocks in an initial state uncertainty are equivalance classes. However, those blocks in a current state uncertainty not necessarily are. For an example, see the following. Besides, a sequence x is homing iff every block in $\sigma(x)$ is a singleton.

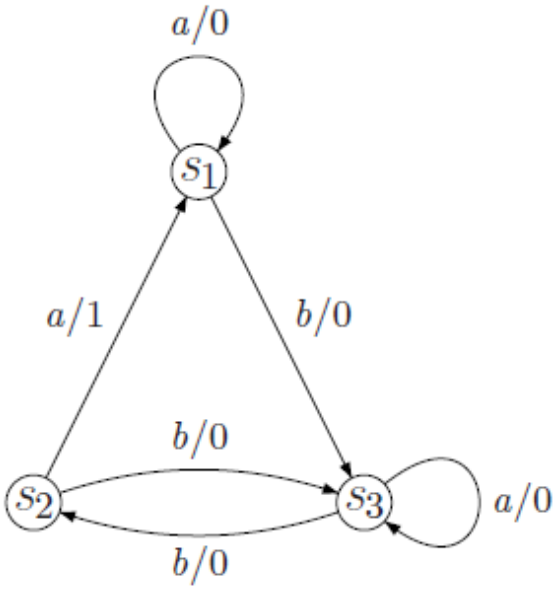

Figure 2.4: [9] The evolvement of current state uncertainty: $\sigma(\epsilon) = \{S\}, \sigma(a) = \{\{s_1\}_1, \{s_1, s_3\}_0\}, \sigma(ab) = \{\{s_3\}_{10}, \{s_2, s_3\}_{00}\}, \sigma(aba) = \{\{s_3\}_{100}, \{s_3\}_{000}, \{\{s_1\}_{001}\}$. Note that the subscripts are corresponding outputs.

Chapter 3

Algorithms

We talk in this chapter about how to compute homing and synchronizing sequences. We present the intuitive algorithms which are polynomial but not necessarily producing the shortest sequences, as well as a straightforward exponential time algorithm that gives the shortest sequences.

3.1 Computing Synchronizing Sequences

An algorithm to compute a sequence is actually a constructive proof of existence. Before we introduce the algorithm we shall realize that synchronizing sequences do not always exist. A counterexample is provided in Figure 2.1. Consider the two ''diagonal'' pairs: $Q_1 = \{s_1, s_3\}$ and $Q_2 = \{s_2, s_4\}$, note that $\delta(Q_1, a) = \delta(Q_2, a) = Q_2$ and $\delta(Q_1, b) = \delta(Q_2, b) = Q_1$. Thus $\forall x \in I^*, |\delta(Q_1, x)| = |\delta(Q_2, x)| = 2$, proving that no synchronizing sequence exist for this machine.

We show an algorithm constructing a synchronizing sequence by concatenating merging sequences.

Definition 3.1.1. Given a Mealy machine $\mathcal{M} = \langle I, O, S, \delta, \lambda \rangle$. A sequence $x \in I^*$ is said to be **merging** for two states $s, t \in S$ if $\delta(s, x) = \delta(t, x)$.

To see the correctness of this algorithm, we notice that $|\delta(S, x)|$ is decreasing in each iteration and will be 1 if the algorithm terminates. This means that the sequence x returned is indeed a synchronizing sequence. On the other hand, if $\mathcal{M}$ has a synchronizing sequence, then it is surely a merging sequence for every pair of states and Step 5 can always work.

Algorithm 1 Build a synchronizing sequence

```
function SYNCHRONIZING(M)
    x ← ϵ
    while |δ(S, x)| > 1 do
        Pick s, t ∈ δ(S, x) s.t. s ≠ t
        Find a merging sequence y for s and t
        if No such y exists then
            return FAILURE
        x ← xy
    return x
```

Finding Merging Sequences

Next we describe in details how to find a merging sequence. We make use of an auxiliary ''product'' machine $\mathcal{M}' = \langle I, S', \delta' \rangle$, where $S' = \{\{s\}, \{t, u\} \mid s, t, u \in S\}$, i.e. the collection of all singletons and pairs. $\delta' : S' \times I \to S'$ is a natural extension of δ, s.t. $\delta'(\{s, t\}, a) = \{u, v\}$ iff $\delta(s, a) = u$ and $\delta(t, a) = v$. See Figure 3.1 for an example. Since synchronizing sequences are output independent, we omit the output alphabet and function.

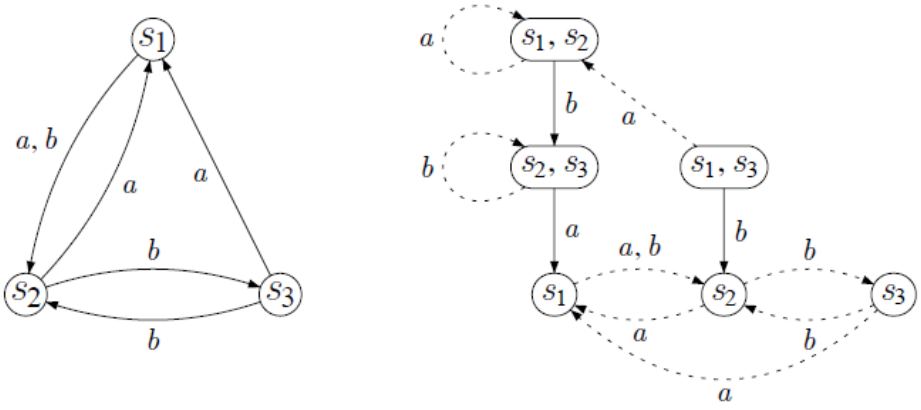

Figure 3.1: [9] Illustration of product machine: a mealy machine $\mathcal{M}$ (left) and the corresponding $\mathcal{M}'$ (right). Solid edges constitute shortest merging paths.

To find a merging sequence for s and t, perform BFS starting from $\{s, t\}$ until a singleton is reached. Then backtrack, concatenating the symbols along the path to form a merging sequence. As a consequence, a merging sequence has length no larger than the number of pairs in S', which is $n(n-1)/2$. And thus, a synchronizing sequence is no longer than $n(n-1)^2/2$.

3.2 Computing Homing Sequences

3.2.1 For Minimized Machines

Similarly, we construct a homing sequence by concatenating separating sequences.

Definition 3.2.1. Given a Mealy machine $\mathcal{M} = \langle I, O, S, \delta, \lambda \rangle$. A sequence $x \in I^*$ is said to be **separating** for two states $s, t \in S$ if $\lambda(s, x) \neq \lambda(t, x)$.

An easy observation is that any pair of different states in a minimized Mealy machine has a separating sequence. Otherwise the pair would be equivalent by definition, contradicting that the machine is minimized.

Algorithm 2 Build a homing sequence for a minimized Mealy machine

Require: $\mathcal{M}$ is a minimized Mealy machine

1: **function** HOMING-MIN($\mathcal{M}$)
2: $x \leftarrow \epsilon$
3: **while** $\exists B \in \sigma(x)$ s.t. $|B| > 1$ **do**
4: Pick $s, t \in B$ s.t. $s \neq t$
5: Find a seperating sequence for s and t
6: $x \leftarrow xy$
7: **return** x

This algorithm serves as a constructive proof that homing sequences exist for every minimized Mealy machine. We show its correctness as below.

First, if the algorithm terminates then the resulting sequence is indeed homing. This follows from the observation at the end of Section 2.4. In each iteration, Step 5 sends the two states s and t to different blocks of the next current state uncertainty. Thus, in the end every block will be reduce to a singleton.

Second, the algorithm could terminate. Our earlier observation shows that the separating sequence in Step 5 could always be found. Thus in every iteration the size of some block in the current uncertainty will drop. After at most $n - 1$ iterations the terminating condition will be satisfied.

Finding Seperating Sequences

Again we need auxiliary variables. They are a nested sequence of partitions: $\{\rho_i\}_{i=0}^{\infty}$, where ρ_i is induced by the equivalence $s \equiv t$ iff $\forall x \in I^*$ with $|x| \leq i,\ \lambda(s, x) = \lambda(t, x)$.

Proposition 3.2.2. *$\{\rho_i\}_{i=0}^{\infty}$ has the following properties:*

1. $\rho_0 = \{S\}$.
2. ρ_{i+1} *refines* ρ_i, $\forall i \in \mathbb{N}$.
3. *If* $\rho_{i+1} = \rho_i$ *then* $\rho_j = \rho_i$, $\forall j > i$.

Proof. The first two properties are obvious. We prove the third here. If we could show that $\rho_{i-1} = \rho_i$ implies $\rho_i = \rho_{i+1}$ then the conclusion will follow. We show this by contradiction. Suppose that $\exists i$ s.t. $\rho_{i-1} = \rho_i$ but $\rho_i \neq \rho_{i+1}$. From the latter we know that there must be some $s, t \in S$ with a shortest separating sequence ax of length $i+1$. Since a cannot separate s and t, x must be a shortest separating sequence for $u = \delta(s, a)$ and $v = \delta(t, a)$, i.e. u and v must belong to different blocks of ρ_i but to the same block in ρ_{i-1}, a contradiction. □

Now we see that $\{\rho_i\}$ is essentially a finite sequence and computable. We show how to compute them as in Algorithm 3. This procedure will be conducted as a preprocessing in Algorithm 2. Its time complexity is $O(pn^2)$, where $p = |I|$. The two layers of loops contribute a factor of $O(n^2)$. We could achieve $O(p)$ efficiency in Step 12 if we record a vector $(B_1, B_2, \dots)$ for every state s where $s \in B_i$ in ρ_i.

Algorithm 3 Build the partition sequence $\{\rho_i\}$

```
1:  function PARTITIONING(M)
2:      ∀s ∈ S, ∀a ∈ I, compute λ(s,a) and δ(s,a)        ▷ stored externally
3:      ρ_0 ← S                                          ▷ stored externally
4:      i ← 0
5:      while ∃B ∈ ρ_i s.t. |B| > 1 do
6:          ρ_{i+1} ← ρ_i                                ▷ stored externally
7:          for each such block B do
8:              Scan through states in B, divide them as below
9:              if i = 0 then
10:                 Divide s and t if some a ∈ I makes λ(s,a) ≠ λ(t,a)
11:             else
12:                 Divide s and t if some a ∈ I makes δ(s,a) ≢ δ(t,a) in ρ_{i-1}
13:             Replace B with the new blocks in ρ_{i+1}  ▷ updated externally
14:         i ← i + 1
```

Then it's time to compute all the seperating sequences. We describe the method in Algorithm 4. Its time complexity is $O(n^2 + pn)$, using the same trick as above. Thus, Algorithm 2 runs in time $O(n^3 + pn^2)$ in total.

Note that the correctness of these two algorithms easily follows from our proof of Proposition 3.2.2. Besides, since there can be at most n nested partitions, a seperating sequence is no longer than $n - 1$, and thus a homing sequence is no longer than $(n-1)^2$.

Algorithm 4 Find a seperating sequence for two states

Require: $\{\rho_i\}$ has been computed

function SEPERATING(s, t)
  $i \leftarrow$ smallest k s.t. $s \not\equiv t$ in ρ_k
  $x \leftarrow \epsilon$
  while $i > 0$ **do**
    if $i = 1$ **then**
      Find $a \in I$ s.t. $\lambda(s, a) \neq \lambda(t, a)$
    else
      Find $a \in I$ s.t. $\delta(s, a) \not\equiv \delta(t, a)$ in ρ_{i-1}
    $x \leftarrow xa$
  return x

3.2.2 For General Machines

While every minimized Mealy machine has a homing sequence, general machines not always do. An example is given in Figure 3.2.

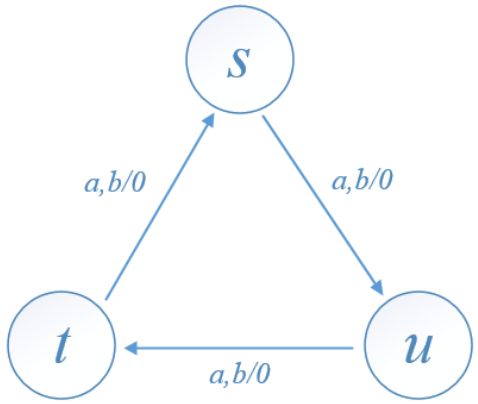

Figure 3.2: A machine with no homing sequence

Algorithm 5 Build a homing sequence for a general Mealy machine

function HOMING($\mathcal{M}$)
  $x \leftarrow \epsilon$
  while $\exists B \in \sigma(x)$ s.t. $|B| > 1$ **do**
    Pick $s, t \in B$ s.t. $s \neq t$
    Find a seperating or merging sequence y for s and t
    if No such y exists **then**
      return FAILURE
    $x \leftarrow xy$
  return x

The definition of homing sequence tells us that it's either separating or merging. Thus we can easily solve this problem by combining the previous two core algorithms. Its correctness is straightforward. And its time complexity is no worse than the sum of the previous two.

3.3 Computing Shortest Homing/Synchronizing Sequences

The algorithms we have presented are simple and efficient, but not necessarily producing the shortest sequences. To be convinced of this, look back at Figure 3.1, where the shortest synchronizing sequence is ba. But Algorithm 1 could have found the merging sequence of s_2 and s_3 first, which is a. Then the resulting synchronizing sequence is of length at least 3, because $\delta(S, a) = \{s_1, s_2\}$, which has a shortest merging sequence of length 2.

Practically, we wish to calculate the shortest sequence because sometimes the bottleneck might be the cost of applying the sequence, rather than the cost of computing it. Think of the case in Figure 2.2.

We present worst-case exponential time algorithms for this problem.

Definition 3.3.1. Given a Mealy machine $\mathcal{M} = \langle I, O, S, \delta, \lambda \rangle$, a synchronizing tree is a rooted tree with the following properties:

1. Every edge represents an input symbol.
2. Every node represents a set of states.
3. Every non-leaf node has $p = |I|$ children, connected with edges representing exactly the input alphabet I.
4. Every node is labeled with $\delta(S, x)$, x being the string along the path from root to it.
5. Such node is defined as a leaf:
   (a) Either its label is a singleton;
   (b) Or its label has appeared in a smaller depth.

To compute the shortest synchronizing sequence, first compute the synchronizing tree. Then perform BFS from the root. By definition, the sequence we seek for is formed along the path from the root to the first leaf found.

Homing tree is defined essentially the same except that the node label is replaced with $\sigma(x)$, i.e. current uncertainty. The first leaf condition is correspondingly replaced with that ‘‘the label consists of only singleton sets’’.

Chapter 4

Complexity

Throughout this chapter we talk about computational complexity of several problems: shortest homing/synchronizing sequences, existence of local homing/synchronizing sequences, existence of homing/synchronizing sequences for nondeterministic machines and so on.

4.1 NP-completeness of the Shortest Homing/Synchronizing Sequences Problem

At the end of last chapter we gave exponential time algorithms for computing shortest homing/synchronizing sequences. Here we show this problem to be NP-complete, and therefore polynomial time algorithms are not likely to exist.

In the following discussion, we talk about machines without output. Or to be more consistent, we consider machines whose output alphabet is a singleton. In this setting, a homing sequence has to be also a synchronizing sequence because output doesn't give us any help on determining the final state. Thus we only need to look at one of them. More specifically, we look at the decision problem:

$$\text{SHORT-SYNC} = \{\langle \mathcal{M}, k\rangle \mid \mathcal{M} \text{ has a synchronizing sequence of length } \leq k\}.$$

Theorem 4.1.1 ([3])**.** SHORT-SYNC *is NP-complete.*

Proof. First, the problem is obviously in NP, since a certificate is a sequence and we can verify whether it is synchronizing and of length $\leq k$, in polynomial time.

For NP-hardness, we show that 3SAT $\leq_{\text{P}}$ SHORT-SINK. Given a CNF formula $\phi = \wedge_{i=1}^{m}(l_1^i \vee l_2^i \vee l_3^i)$ over variables $v_1, \ldots, v_n$, we build a machine $\mathcal{A} = \langle I, S, \delta\rangle$ with $m(n+1)+1$ states

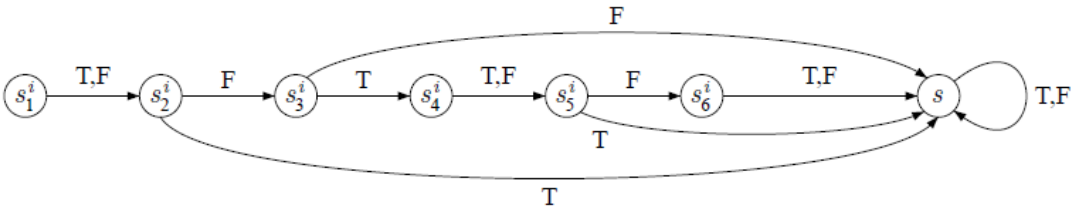

Figure 4.1: [9] The corresponding state sequence for the clause $v_2 \vee \neg v_3 \vee v_5$

(and thus in polynomial time), where $I = \{\text{T,F}\}$. For every clause $C_i = l_1^i \vee l_2^i \vee l_3^i$ there are $n+1$ states $s_1^i, \ldots, s_{n+1}^i$. Apart from that, a special state s is present. The transition function is illustrated in Figure 4.1. In general, s_j^i goes to s_{j+1}^i for both T and F when v_j is not present in C_i. Otherwise $\delta(s_j^i, \text{T}) = s$ or $\delta(s_j^i, \text{F}) = s$, whichever symbol satisfying C_i will be the shortcut label. s_{n+1}^i and s always transits to s.

Then we observe that $\phi \in$ 3SAT iff $\langle \mathcal{A}, n \rangle \in$ SHORT-SYNC. Indeed, if ϕ is satisfiable, so is each C_i, a satisfying assignment must contain labels of shortcut from some s_j^i with $j \leq n$ to s, for each i. On the other hand, if $\mathcal{A}$ has a synchronizing sequence of length $\leq n$, then the final state must be s since this is the only state connecting to every $\{s_j^i\}$ sequence. $\mathcal{A}$ must have traveled along shortcuts in every $\{s_j^i\}$ sequence to s, i.e. every C_i has a satisfying assignment. □

4.2 PSPACE-completeness of Q-Synchronizing Problem

We consider a more general setting. Sometimes we have *a priori* knowledge about the initial state, say, it is in some $Q \subset S$.

Definition 4.2.1. Given a Mealy machine $\mathcal{M} = \langle I, O, S, \delta, \lambda \rangle$ and $Q \subset S$, a sequence $x \in I^*$ is said to be ***Q*-synchronizing** if $|\delta(Q, x)| = 1$.

Then the fundamental synchronizing problem becomes:

$$Q\text{-SYNC} = \{\langle \mathcal{M}, Q \rangle \mid \mathcal{M} \text{ has a } Q\text{-synchronizing sequence}\}.$$

Theorem 4.2.2 ([8])**.** Q-SYNC *is PSPACE-complete.*

We skip the proof here, which is done in [8] by a reduction from a PSPACE-complete problem:

$$\text{DFA-INTERSECTION} = \{\langle \mathcal{D}_1, \ldots, \mathcal{D}_k \rangle \mid L(\mathcal{D}_1 \cap \cdots \cap \mathcal{D}_k) \neq \emptyset\}.$$

The idea is, given DFAs $\mathcal{D}_1, \ldots, \mathcal{D}_k$, to build a Mealy machine $\mathcal{M}$ copying all the DFA states, while adding two ''sink'' states, one has incoming transitions from all accepting states and the other on the contrary. Then every word in the intersection language is mapped to a Q-synchronizing sequence of $\mathcal{M}$, where Q consists of initial states of $\mathcal{D}_1, \ldots, \mathcal{D}_k$.

4.3 PSPACE-completeness of Synchronizing Problem for Nondeterministic Mealy Machines

So far we are discussing within the deterministic setting, and the existence problem of homing/synchronizing sequences has been shown in P. However, for nondeterministic Mealy machines, the existence problem becomes PSPACE-hard. To show this, we first introduce nondeterministic Mealy machines.

Definition 4.3.1. A **nondeterministic Mealy machine** is a 5-tuple $\mathcal{M} = \langle I, O, S, \delta, \lambda \rangle$, where

- I is a finite set called input alphabet,
- O is a finite set called output alphabet,
- S is a finite set of states,
- $\delta : S \times I \to \mathcal{P}(S)$ is the transition function,
- $\lambda : S \times I \to \mathcal{P}(O)$ is the output function.

Note that for the same reason in Section 4.1, we will ignore output, though we include it in the definition for consistency. The essential difference with the deterministic Mealy machines is that here we allow multiple (or zero) transitions from one state on one input symbol. Formally, $\forall Q \subset S$, $\forall a \in I$, $\delta(Q, a) = \bigcup_{s \in Q} \delta(s, a)$. The definition of synchronizing sequences for nondeterministic Mealy machines is exactly the same as Definition 2.3.1.

We could now formalize the decision problem as follows:

$$\text{N-SYNC} = \{\langle \mathcal{M} \rangle \mid \mathcal{M} \text{ is nondeterministic and has a synchronizing sequence}\}.$$

Theorem 4.3.2. N-SYNC *is PSPACE-complete.*

Proof. First, to show that the problem is in PSPACE, we create a nondeterministic polynomial space Turing machine for it, as follows.

$T =$ ''On input $\langle \mathcal{M} \rangle$, where $\mathcal{M}$ is a nondeterministic Mealy machine:

1. Let $n = |S|$ and $Q_0 = S$. If $n = 1$, accept.

2. For $i = 1$ to n:
3. Nondeterministically select $a_i \in I$ and compute $Q_i = \delta(Q_{i-1}, a_i)$. If $|Q_i| = 1$, accept.
4. Reject.''

Since T only stores $Q_i \subset S$, it uses $O(n)$ space. So N-SYNC $\in$ NPSPACE $=$ PSPACE.

Next, we show that Q-SYNC $\leq_{\mathrm{P}}$ N-SYNC. For any given pair $(\mathcal{M}, Q)$ where $\mathcal{M} = \langle I, O, S, \delta, \lambda \rangle$ is a deterministic Mealy machine and Q is a subset of its states, we construct a nondeterministic Mealy machine $\mathcal{M}' = \langle I', O', S', \delta', \lambda' \rangle$ s.t. $I' = I \cup \{z\}$, where z is a newly added symbol, $O' = O$ and $S' = S$. As for the structure (state diagram) of $\mathcal{M}'$, we first copy all that of $\mathcal{M}$, and then add z-transitions. Formally, $\forall Q \subset S$ and $\forall a \in I$ where $\delta(Q, a)$ is defined, let $\delta'(Q, a) = \delta(Q, a)$. The extension is, $\forall s \in S \setminus Q,\ \delta(s, z) = Q$, and $\forall s \in Q,\ \delta(s, z) = \{s\}$. The construction takes linear time in the number of states and transitions in $\mathcal{M}$.

It is easy to observe that if $\mathcal{M}'$ has a synchronizing sequence, then $\mathcal{M}$ has a Q-synchronizing sequence. Suppose $w \in I'^*$ is synchronizing for $\mathcal{M}'$. Since z-transitions are just self loops for states in Q, if we remove all z's from w, getting a new string $w' \in I^*$, we would have $\delta'(s, w') = \delta(s, w),\ \forall s \in Q$. Thus, $|\delta'(Q, w')| = |\delta(Q, w)| = 1$ and w is a Q-synchronizing sequence for $\mathcal{M}$.

On the other hand, if $\mathcal{M}$ has a Q-synchronizing sequence $x \in I^*$, then, $x' = zx$ is a synchronizing sequence for $\mathcal{M}'$.

□

Chapter 5

Applications and Related Problems

5.1 Machine Identification

Suppose we are given a ‘‘black box’’ implementation machine $\mathcal{B}$ with visible I/O behavior, the task is to identify its structure, i.e. determining its state transition diagram. This problem is of interest in reverse engineering [4].

To make this problem solvable we make some reasonable assumptions. $\mathcal{B}$ should be strongly connected so that we could always reach every part of it. It should be minimized so that we can uniquely identify it rather than simply an equivalent machine. Besides its I/O alphabets I, O and upper bound n on number of states should be known.

The algorithm is as follows [5]:

1. Enumerate all Mealy machines with n states, p inputs and q outputs. There are in total $N = (qn)^{pn}/n!$ of them. We then minimize them and extract a subset of strongly connected, minimized, mutually inequivalent machines $\{\mathcal{M}_i = \langle I, O, S_i, \delta_i, \lambda_i \rangle\}$. According to **??**, the size of this subset is (super?) exponential in n.
2. Take a direct sum $\mathcal{M} = \langle I, O, S, \delta, \lambda \rangle$ of them, where S is the union of all S_i's. $\delta(s, a) = \delta_i(s, a)$ and $\lambda(s, a) = \lambda_i(s, a)$ for $s \in S_i$, are simply natural extensions.
3. Compute a homing sequence h for $\mathcal{M}$ and apply it to $\mathcal{B}$. Observe the output y, then we can tell which state $s_k \in S$ $\mathcal{B}$ is in. And the machine $\mathcal{M}_k$ containing s_k must be identical to $\mathcal{B}$.

Remark

To some reader the direct sum here may seem odd, as it is disconnected. And it is suspected not to have a homing sequence. Actually, the original definition of Mealy machines (Definition

2.1.1) never mentions connectivity. Neither does the existence of homing sequence require that (Algorithm 2). The only thing that remains to prove is that the direct sum $\mathcal{M}$ is a minimal Mealy machine.

Proof. By definition of $\mathcal{M}$, any component machine $\mathcal{M}_k$ within it is minimal. So we only need to check whether it is possible for two states from two different component to be equivalent.

Suppose there are two such equivalent states, s_i and s_j, from $\mathcal{M}_i$ and $\mathcal{M}_j$, respectively. Then by definition of equivalence, $\forall x \in I^*,\ \lambda(s_i, x) = \lambda(s_j, x)$. Since $\mathcal{M}_i$ is strongly connected, $\forall t_i \in S_i,\ \exists y$ s.t. $t_i = \delta(s_i, y)$. Let $t_j = \delta(s_j, y) \in S_j$, then $t_i \equiv t_j$, because $\forall x \in I^*,\ \lambda(t_i, x) = \lambda(s_i, yx) = \lambda(s_j, yx) = \lambda(t_j, x)$. This is to say, every state in $\mathcal{M}_i$ has an equivalent state in $\mathcal{M}_j$. By symmetry, we can also deduce that every state in $\mathcal{M}_j$ has an equivalent state in $\mathcal{M}_i$. Thus, $\mathcal{M}_i$ and $\mathcal{M}_j$ are equivalent, a contradiciton.

Therefore, no equivalent states exist in the direct sum $\mathcal{M}$. It is minimal. □

5.2 Learning

Similar to the problem above, the goal of learning (or inferring) a machine is also determining its state transition diagram. The difference is that it doesn't use enumeration. Instead, provided an oracle or with team work, it seeks for polynomial algorithm with high probability of correctness.

In [7] the inferring of an automaton is based on Angluin's algorithm [2], which infers a regular set but requires reset function. A homing sequence is used as a reset function so that requirement is removed. The homing sequence can be learned through the inferring process.

5.3 The Road Coloring Theorem

Given a finite, directed, strongly connected graph where all vertex outdegrees are constant and the gcd of all cycle lengths being 1, it is conjectured that it has a synchronizing coloring (a certain transition diagram s.t. a synchronizing sequence exists), as illustrated in Figure 5.1.It was first claimed by Roy Adler and Benjamin Weiss[1] and proved by Avraham Trahtmanin 2009 [10].

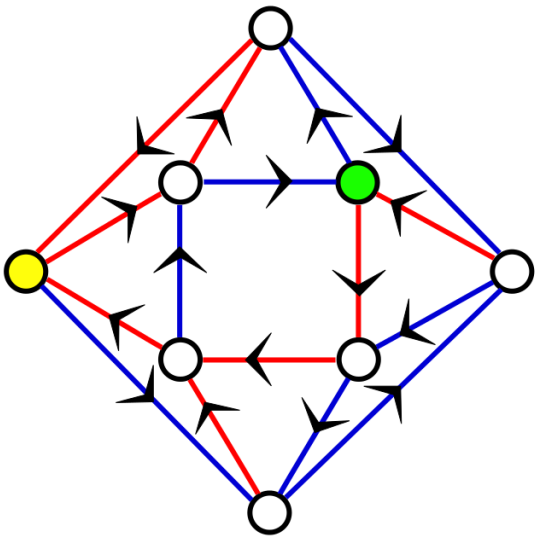

Figure 5.1: A synchronizing coloring. The sequence ‘‘brrbrrbrr’’ leads to the yellow node, and ‘‘bbrbbrbbr’’ to the green one.

5.4 The Černý Conjecture

In previous chapters we conclude that synchronizing sequences, if existing, could be as short as $O(n^3)$. Černý conjectured that the upper bound of the length of a shortest synchronizing sequence for an n-state machine is $(n-1)^2$. If this is true, then it will be a tight bound since an example of machine requiring $(n-1)^2$ has been found[9].

Chapter 6

Summary

In this report we surveyed about a fundamental FSM testing problem, which is to take a machine from an unknown state to a known one, the solutions being homing sequences and synchronizing sequences. We discussed several problems regarding the two kinds of sequences: existence, computation, length, etc. plus some generalizations. The applications and related problems are briefly reviewed.

Despite a rather complete theory on this area, there remains work to be done in the future:

- Developing PTAS for shortest homing sequence, and possibly other hard problems in terms of complexity;
- Proving the Černý Conjecture for more general classes of machines;
- Exploring variant interpretations and the corresponding theory, like the rank of a sequence and different types of randomizations;
- Finding more real scenarios where homing sequences could be applied.

References

[1] R. Adler. *Similarity of automorphisms of the torus*, pages 63--64. Springer Berlin Heidelberg, Berlin, Heidelberg, 1971. ISBN 978-3-540-36662-1. doi: 10.1007/BFb0070153. URL http://dx.doi.org/10.1007/BFb0070153.

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75 (2):87--106, Nov. 1987. ISSN 0890-5401. doi: 10.1016/0890-5401(87)90052-6. URL http://dx.doi.org/10.1016/0890-5401(87)90052-6.

[3] D. Eppstein. Reset sequences for monotonic automata. *SIAM Journal on Computing*, 19 (3):500--510, 1990. doi: 10.1137/0219033. URL http://dx.doi.org/10.1137/0219033.

[4] J. Esch. Prolog to principles and methods of testing finite state machines - a survey. *Proceedings of the IEEE*, 84(8):1089--, Aug 1996. ISSN 0018-9219. doi: 10.1109/JPROC.1996.533955.

[5] E. F. Moore. Gedanken Experiments on Sequential Machines. In *Automata Studies*, pages 129--153. Princeton U., 1956.

[6] A. S. Rao and K. Y. Goldberg. Manipulating algebraic parts in the plane. *IEEE Transactions on Robotics and Automation*, 11(4):598--602, Aug 1995. ISSN 1042-296X. doi: 10.1109/70.406944.

[7] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 411--420, New York, NY, USA, 1989. ACM. ISBN 0-89791-307-8. doi: 10.1145/73007.73047. URL http://doi.acm.org/10.1145/73007.73047.

[8] I. Rystsov. Polynomial complete problems in automata theory. *Information Processing Letters*, 16(3):147 -- 151, 1983. ISSN 0020-0190. doi: http://dx.doi.org/10.1016/0020-0190(83)90067-4. URL http://www.sciencedirect.com/science/article/pii/0020019083900674.

[9] S. Sandberg. *1 Homing and Synchronizing Sequences*, pages 5--33. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-32037-1. doi: 10.1007/11498490_2. URL http://dx.doi.org/10.1007/11498490_2.

[10] A. N. Trahtman. The road coloring problem. *Israel Journal of Mathematics*, 172 (1):51--60, 2009. ISSN 1565-8511. doi: 10.1007/s11856-009-0062-5. URL http://dx.doi.org/10.1007/s11856-009-0062-5.