1 Homing and Synchronizing Sequences

Sven Sandberg

Department of Information Technology
Uppsala University
svens@it.uu.se

1.1 Introduction

1.1.1 Mealy Machines

This chapter considers two fundamental problems for Mealy machines, i.e., finite-state machines with inputs and outputs. The machines will be used in subsequent chapters as models of a system or program to test. We repeat Definition 21.1 of Chapter 21 here: readers already familiar with Mealy machines can safely skip to Section 1.1.2.

Definition 1.1. A Mealy Machine is a 5-tuple $\mathcal{M} = \langle I, O, S, \delta, \lambda \rangle$, where I, O and S are finite nonempty sets, and $\delta : S \times I \to S$ and $\lambda : S \times I \to O$ are total functions.

The interpretation of a Mealy machine is as follows. The set S consists of "states". At any point in time, the machine is in one state $s \in S$. It is possible to give inputs to the machine, by applying an input letter $a \in I$. The machine responds by giving output $\lambda(s, a)$ and transforming itself to the new state $\delta(s, a)$. We depict Mealy machines as directed edge-labeled graphs, where S is the set of vertices. The outgoing edges from a state $s \in S$ lead to $\delta(s, a)$ for all $a \in I$, and they are labeled "a/b", where a is the input symbol and b is the output symbol $\lambda(s, a)$. See Figure 1.1 for an example.

We say that Mealy machines are *completely specified*, because at every state there is a next state for every input (δ and λ are total). They are also *deterministic*, because only one next state is possible.

Applying a string $a_1 a_2 \cdots a_k \in I^*$ of input symbols starting in a state s_1 gives the sequence of states $s_1, s_2, \ldots, s_{k+1}$ with $s_{j+1} = \delta(s_j, a_j)$. We extend the transition function to $\delta(s_1, a_1 a_2 \cdots a_k) \stackrel{\text{def}}{=} s_{k+1}$ and the output function to $\lambda(s_1, a_1 a_2 \cdots a_k) \stackrel{\text{def}}{=} \lambda(s_1, a_1)\lambda(s_2, a_2) \cdots \lambda(s_k, a_k)$, i.e., the concatenation of all outputs. Moreover, if $Q \subseteq S$ is a set of states then $\delta(Q, x) \stackrel{\text{def}}{=} \{\delta(s, x) : s \in Q\}$. We sometimes use the shorthand $s \xrightarrow{a} t$ for $\delta(s, a) = t$, and if in addition we know that $\lambda(s, a) = b$ then we write $s \xrightarrow{a/b} t$. The number $|S|$ of states is denoted n.

Throughout this chapter we will assume that an explicit Mealy machine $\mathcal{M} = \langle I, O, S, \delta, \lambda \rangle$ is given.

M. Broy et al. (Eds.): Model-Based Testing of Reactive Systems, LNCS 3472, pp. 5-33, 2005.

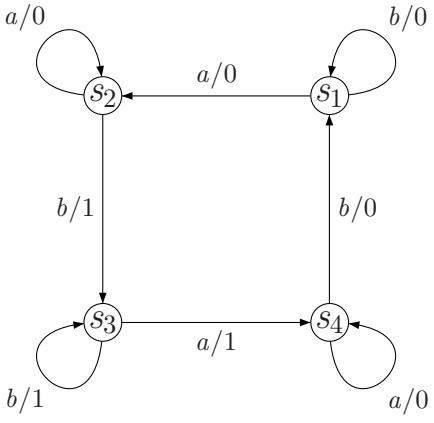

Fig. 1.1. A Mealy machine $\mathcal{M} = \langle I, O, S, \delta, \lambda \rangle$ with states $S = \{s_1, s_2, s_3, s_4\}$, input alphabet $I = \{a, b\}$, and output alphabet $O = \{0, 1\}$. For instance, applying a starting in s produces output $\lambda(s, a) = 0$ and moves to next state $\delta(s, a) = t$.

1.1.2 Synchronizing Sequences

In the problems of this chapter, we do not know the initial state of a Mealy machine and want to apply a sequence of input symbols so that the *final* state becomes known. A **synchronizing sequence** is one that takes the machine to a unique final state, and this state does not depend on where we started. Which particular final state is not specified: it is up to whoever solves the problem to select it. Thus, formally we have:

Definition 1.2. A sequence $x \in I^*$ is **synchronizing** (for a given Mealy machine) if $|\delta(S, x)| = 1$. [1] □

Note that synchronizing sequences are independent of the output. Consequently, when talking about synchronizing sequences we will sometimes omit stating the output of the machine. For the same reason, it is not meaningful to talk about synchronizing sequences "for minimized machines", because if we ignore the output then all machines are equivalent.

Example 1.3. Synchronizing sequences have many surprising and beautiful applications. For instance, robots that grasp and pick up objects, say, in a factory, are often sensitive to the orientation of the object. If objects are fed in a random

[1] The literature uses an amazing amount of synonyms (none of which we will use here), including *synchronizing word* [KRS87], *synchronization sequence* [PJH92], *reset sequence* [Epp90], *reset word* [Rys97], *directing word* [ČPR71], *recurrent word* [Rys92], and *initializing word* [Göh98]. Some texts talk about the machine as being a *synchronized* [CKK02], *synchronizing* [KRS87], *synchronizable* [PS01], *resettable* [PJH92], *reset* [Rys97], *directable* [BIĆP99], *recurrent* [Rys92], *initializable* [Göh98], *cofinal* [ID84] or *collapsible* [Fri90] *automaton*.

orientation, the problem arises of how to rotate them from an initially unknown orientation to a known one. Using sensors for this is expensive and complicated. A simple and elegant solution is depicted in Figure 1.2. Two parallel "pushing walls" are placed around the object, and one is moved toward the other so that it starts pushing the object, rotating it until a stable position between the walls is reached. Given the possible directions of these pushing walls, one has to find a sequence of pushes from different directions that takes the object to a known state. This problem can be reduced to finding a synchronizing sequence in a machine where the states are the possible orientations, the input alphabet is the set of possible directions of the walls, and the transition function is given by how a particular way of pushing rotates the object into a new orientation. This problem has been considered by, e.g., Natarajan [Nat86] and Eppstein [Epp90], who relate the problem to automata but use a slightly different way of pushing. Rao and Goldberg [RG95] use our way of pushing and their method works for more generally shaped objects. □

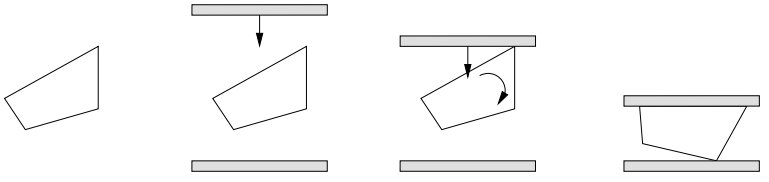

Fig. 1.2. Two pushing walls rotating the object to a new position. (a) The object. (b) One wall moves toward the object until (c) it hits it and starts pushing it, rotating it to the final stable position (d).

An alternative way to formulate the synchronizing sequence problem is as follows. Let S be a finite set, and $f_1, \ldots, f_k : S \to S$ total functions. Find a composition of the functions that is constant. Function f_i corresponds to $\delta(\cdot, a_i)$, where a_i is the i'th input symbol.

Example 1.4. To see that synchronizing sequences do not always exist, consider the Mealy machine in Figure 1.1. If the same sequence of input symbols is applied to two states that are "opposite corners", then the respective final states will be opposite corners too. So in particular no sequence x satisfies $\delta(s_1, x) = \delta(s_3, x)$ or $\delta(s_2, x) = \delta(s_4, x)$. □

Besides the parts orienting problem in Example 1.3, synchronizing sequences have been used to generate test cases for synchronous circuits with no reset [CJSP93], and are also important in theoretical automata theory and structural theory of many-valued functions [Sal02].

1.1.3 Homing Sequences

The second problem of this chapter, **homing sequences**, are sequences of input symbols such that the final state after applying it can be determined by looking at the output:

Definition 1.5. A sequence $x \in I^*$ is **homing** (for a given Mealy machine) if for every pair $s, t \in S$ of states, $\delta(s, x) \neq \delta(t, x) \Rightarrow \lambda(s, x) \neq \lambda(t, x)$. [2] □

Note that every synchronizing sequence is a homing sequence, but the converse is not true. See Figure 1.1 for an example: we saw earlier that it does not have a synchronizing sequence, but it has a homing sequence. After applying ab, the possible outputs are $\lambda(s_1, ab) = 01$, $\lambda(s_2, ab) = 01$, $\lambda(s_3, ab) = 10$, and $\lambda(s_4, ab) = 00$. Hence, if we observe output 00 or 10, the initial and hence also final state becomes known. For output 01 we only know the initial state was s or t, but in both cases the final state is u. Thus the output uniquely determines the final state, and the sequence is homing.

Homing sequences can be either **preset**[3], as in Definition 1.5, or **adaptive**. While preset sequences are completely determined before the experiment starts, adaptive sequences are applied to the machine as they are constructed, and the next symbol in the sequence depends on the previous outputs. Thus, preset sequences can be seen as a special case of adaptive sequences, where this dependence is not utilized. Formally one can define adaptive homing sequences as decision trees, where each node is labeled with an input symbol and each edge is labeled with an output symbol. The test consists in walking from the root of the tree toward a leaf: apply the input on the node, observe the output and walk to the successor through the edge with the corresponding label. When a leaf is reached, the sequence of outputs determines the final state (but unlike preset sequences, the sequence of *inputs* would not necessarily determine the final state if the initial state had been different).

Homing sequences are typically used as building blocks in testing problems with no reset. Here, a reset is a special input symbol that takes every input to the same state, i.e., it is a synchronizing sequence of length one. They have been used in conformance testing (Section 4.5), and in learning (by Rivest and Schapire [RS93]; see also Chapter 19). For machines with output, homing sequences are often preferable to synchronizing sequences: first, they are usually shorter; second, they always exist if the automaton is minimized (cf. Theorem 1.17), a natural criterion that is often required anyway.

1.1.4 Chapter Outline

Section 1.2 introduces the important notion of current state uncertainty, used when computing homing sequences. Section 1.3 presents algorithms for several

[2] Synonyms (not used here) are *homing word* [Rys83], *terminal state experiment* [Hib61], *Identification experiment of the second kind (IE 2)* [Sta72] and *experiment which distinguishes the terminal state of a machine* [Gin58].

[3] A synonym is *uniform* [Gin58].

versions of the homing and synchronizing sequences problems: first an algorithm to compute homing sequences for minimized Mealy machines (Section 1.3.1), then an algorithm to compute synchronizing sequences (Section 1.3.2). Section 1.3.3 unifies these algorithms into one for computing homing sequences for general (not necessarily minimized) machines – this algorithm can be used both to compute homing and synchronizing sequences. The two algorithms for computing homing sequences are then modified to compute *adaptive* homing sequences in Section 1.3.4. Finally, Section 1.3.5 gives exponential algorithms to compute minimal length homing and synchronizing sequences.

Section 1.4 turns from algorithms to complexity. First, Section 1.4.1 shows that it is NP-hard to find the *shortest* homing or synchronizing sequence. Second, Section 1.4.2 shows that it is PSPACE-complete to determine if a machine has a homing or synchronizing sequence, if it is known that the initial state is in a particular subset $Q \subseteq S$. In both cases it means that polynomial algorithms for the problems are unlikely to exist.

Section 1.5 gives an overview of research in the area and mentions some related areas, and Section 1.6 summarizes the most important ideas in the chapter.

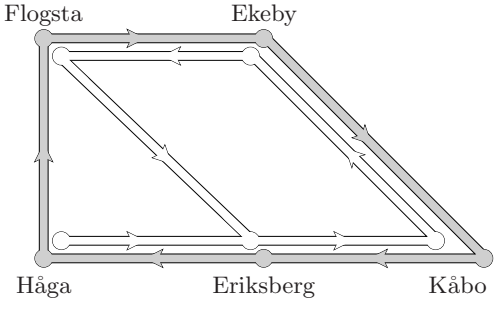

Fig. 1.3. The subway map of Uppsala. The five stations are connected by two one-way lines: white and grey.

Exercise 1.1. The schematic map of the subway in Uppsala looks as in Figure 1.3. You do not know at which station you are, and there are no signs or other characteristics that reveal the current station, but you have to get to Flogsta by moving from station to station, in each step taking either the white or the grey line. What type of sequence does this correspond to? Find a sequence if one exists.

(Hint: use that if you are in Flogsta or Håga, the white line takes you to Eriksberg for sure.)

1.2 Initial and Current State Uncertainty

Consider a Mealy machine to which we apply some input string and receive an output string. Even if the input string was not homing, we may still draw some

partial conclusions from the output. The *initial state uncertainty* describes what we know about the initial state, and the *current state uncertainty* describes what we know about the final state. Current state uncertainty is crucial when computing homing sequences, and the definition relies on initial state uncertainty.

Definition 1.6. The **initial state uncertainty** (with respect to a Mealy machine) after applying input sequence $x \in I^*$ is a partition

$$\pi(x) \stackrel{\text{def}}{=} \{B_1, B_2, \ldots, B_r\} \subset \mathcal{P}(S)$$

of the states. Two states s, t are in the same block B_i if and only if $\lambda(s, x) = \lambda(t, x)$. □

Thus, after applying input x, for a certain output we know the initial state was in B_1, for another output we know it was in B_2, and so on. Although initial state uncertainty will not be used explicitly until Sections 2 and 3.4.3, it provides intuitions that will be useful here, and we also need it to define the current state uncertainty.

The *current* state uncertainty is a data structure that, given an input string, describes for each output string the set of possible final states. Thus, computing a homing sequence means to find an input string for which the current state uncertainty associated with each output string is a singleton.

Definition 1.7. The **current state uncertainty** (with respect to a Mealy machine) after applying input sequence $x \in I^*$ is $\sigma(x) \stackrel{\text{def}}{=} \{\delta(B_i, x) : B_i \in \pi(x)\} \subset \mathcal{P}(S)$. □

The elements of both the initial and the current state uncertainty are called **blocks**. If B is a block of $\pi(x)$ or $\sigma(x)$ then $|B|$ denotes its size, whereas $|\pi(x)|$ and $|\sigma(x)|$ denote the number of blocks. While the initial state uncertainty is a partition (i.e., any two blocks are disjoint, and the union of all blocks is the entire set of states), Example 1.8 will show that the current state uncertainty does not need to be one: a state may belong to several different blocks, and the union of all blocks does not need to be the whole set of states.

We will frequently take the viewpoint that the current state uncertainty *evolves* as more input symbols are applied. Namely, the current state uncertainty $\sigma(xy)$ is obtained by applying $\delta(\cdot, y)$ to each block of $\sigma(x)$, splitting the result if some states gave different outputs on y.

Example 1.8. To see how the current state uncertainty works, consider the machine in Figure 1.4. Initially, we do not know the state, so it may be either s_1, s_2, or s_3. Thus the current state uncertainty is $\sigma(\varepsilon) = \{\{s_1, s_2, s_3\}\}$. Apply the string a to the machine. If we receive the output 1, then we were in state s_2 so the current state is s_1. If we receive the output 0, then we were in either s_1 or s_3 and the current state is either s_1 or s_3. We then describe the current state uncertainty as $\sigma(a) = \{\{s_1\}_1, \{s_1, s_3\}_0\}$ (the subscripts, included only in this example for clarity, show which outputs correspond to each block). Now we

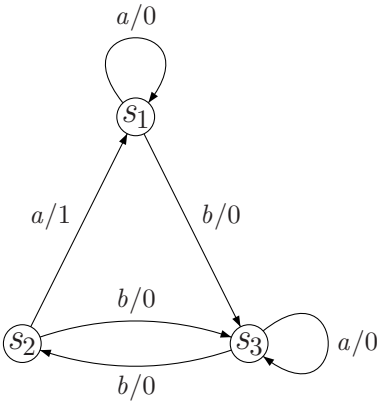

Fig. 1.4. Example illustrating current state uncertainty

additionally apply the letter b. If we were in s_1 then we end up in s_3 and receive output 0, and if we were in either s_3 or s_1 then we end up in either s_2 or s_3, in both cases receiving output 0. Thus the current state uncertainty after applying ab is $\sigma(ab) = \{\{s_3\}_{10}, \{s_2, s_3\}_{00}\}$. Finally, we apply the letter a at this point. If we were in s_3, then we move to s_3 with output 0, and if we were in s_2, then we move to s_1 and receive output 1. Thus the current state uncertainty becomes $\sigma(aba) = \{\{s_3\}_{100 \text{ or } 000},\ \{s_1\}_{001}\}$. We end the example with an important remark: since every set in the current state uncertainty is now a *singleton*, we can determine the current state uniquely, by looking at the output. Thus aba is a homing sequence. (Verify this using Definition 1.5!) □

We conclude this section with two important observations. First, as the sequence is extended, the initial state uncertainty becomes more and more refined. I.e., by applying more input symbols the blocks of the partition may be split but not merged:

Lemma 1.9. *For any sequences $x, y \in I^*$, the following holds.*

$$\forall\, B_i \in \pi(xy)\, \exists\, B_j \in \pi(x) : B_i \subseteq B_j.$$

Proof. All states in a block of $\pi(xy)$ give the same output on xy. In particular they give the same output on x, so they all belong to the same block of $\pi(x)$. □

Second, as a sequence x is extended, the sum $\sum_{B \in \sigma(x)} |B|$ of sizes of all blocks in the current state uncertainty can never increase. This is because if $B \in \sigma(xy)$ then $B = \delta(Q, y)$, where $Q \subseteq B'$ is a subset of some block $B' \in \sigma(x)$: here we must have $|B'| \geq |Q| \geq |B|$. (When is each inequality strict?) Moreover, the number of blocks can only decrease if two blocks are mapped to the same block, in which case the sum of sizes of all blocks also decreases. This (very informally) explains the following lemma, whose proof we delegate to Exercise 1.2.

Lemma 1.10. *If $x, y \in I^*$ are any two sequences, the following holds.*

$$\left(\sum_{B \in \sigma(x)} |B| \right) - |\sigma(x)| \geq \left(\sum_{B \in \sigma(xy)} |B| \right) - |\sigma(xy)|. \qquad \square$$

As we will see in the next section, algorithms for computing homing sequences work by concatenating sequences so that in each step the inequality in Lemma 1.10 is *strict*: note that x is homing when $\sum_{B \in \sigma(x)} |B| - |\sigma(x)|$ reaches zero.

Initial and current state uncertainty has been used since the introduction of homing sequences [Moo56, Gil61] although we use a slightly different definition of current state uncertainty [LY96].

Exercise 1.2. Prove Lemma 1.10.

Exercise 1.3. Recall from the discussion before Lemma 1.10 that if $B \in \sigma(xy)$ then $B = \delta(Q, y)$ where $Q \subseteq B'$ for some $B' \in \sigma(x)$. When is $|B'| > |Q|$, and when is $|Q| > |B|$?

1.3 Algorithms for Computing Homing and Synchronizing Sequences

1.3.1 Computing Homing Sequences for Minimized Machines

This section presents an algorithm to compute homing sequences, assuming the machine is *minimized* (for definitions and algorithms for minimization, refer to Chapter 21). This is an important special case that occurs in many practical applications, cf. Section 4.5 and the article by Rivest and Schapire [RS93]. The algorithm for minimized machines is a simpler special case of the general Algorithm 3 in Section 1.3.3: they can be implemented to act identically on minimized machines. Both algorithms run in time $O(n^3 + n^2 \cdot |I|)$, but the one for minimized machines requires less space ($O(n)$ instead of $O(n^2 + n \cdot |I|)$) and produces shorter sequences (bounded by $(n^2 - n)/2$ instead of $(n^3 - n)/6$). The general algorithm, on the other hand, gives additional insight into the relation between homing and synchronizing sequences, and is of course applicable to more problem instances.

The algorithm of this section builds a homing sequence by concatenating many *separating sequences*. A separating sequence for two states gives different output for the states:

Definition 1.11. A **separating sequence** for two states $s, t \in S$ is a sequence $x \in I^*$ such that $\lambda(s, x) \neq \lambda(t, x)$. $\square$

The Algorithm. Algorithm 1 computes a homing sequence for a minimized machine as follows. It first finds a separating sequence for some two states of the machine. By the definition of separating sequence, the two states give different outputs, hence they now belong to different blocks of the resulting current state uncertainty. Next iteration finds two new states that belong to the same block of the current state uncertainty and applies a separating sequence for them. Again, the two states end up in different blocks of the new current state uncertainty. This process is repeated until the current state uncertainty contains only singleton blocks, at which point we have a homing sequence.

Algorithm 1 Computing a homing sequence for a minimized machine.

```
1  function HOMING-FOR-MINIMIZED(Minimized Mealy machine M)
2      x ← ε
3      while there is a block B ∈ σ(x) with |B| > 1
4          find two different states s, t ∈ B
5          let y be a separating sequence for s and t
6          x ← xy
7      return x
```

Step 5 of the algorithm can always be performed because the machine is minimized. Since y is separating, the block B splits into at least two new blocks (one containing s and one containing t). Thus, Lemma 1.10 holds with strict inequality between any two iterations, i.e., the quantity $\sum_{B\in\sigma(x)}|B| - |\sigma(x)|$ strictly decreases in each iteration. When the algorithm starts, it is $n-1$ and when it terminates it is 0. Hence the algorithm terminates after concatenating at most $n-1$ separating sequences.

The algorithm is due to Ginsburg [Gin58] who relies on the adaptive version of Moore [Moo56] which we will see more of in Section 1.3.4.

Length of Separating Sequences. We now show that any two states in a minimized machine have a separating sequence of length at most $n-1$. Since we only need to concatenate $n-1$ separating sequences, this shows that the computed homing sequence has length at most $(n-1)^2$. The argument will also help understanding how to compute separating sequences.

Define a sequence $\rho_0, \rho_1, \ldots$ of partitions, so that two states are in the same class of ρ_i if and only if they do not have any separating sequence of length i. Thus, ρ_i is the partition induced by the relation $s \equiv_i t \stackrel{\text{def}}{\Leftrightarrow}$ "$\lambda(s,x) = \lambda(t,x)$ for all $x \in I^*$ of length at most i". In particular, $\rho_0 = \{S\}$, and ρ_{i+1} is a refinement of ρ_i. These partitions are also used in algorithms for machine minimization; cf. Section 21. The following lemma is important.

Lemma 1.12 ([Moo56]). *If $\rho_{i+1} = \rho_i$ for some i, then the rest of the sequence of partitions is constant, i.e., $\rho_j = \rho_i$ for all $j > i$.*

Proof. We prove the equivalent, contrapositive form: $\rho_{i+1} \neq \rho_i \Rightarrow \rho_i \neq \rho_{i-1}$ for all $i \geq 1$. If $\rho_{i+1} \neq \rho_i$ then there are two states $s, t \in S$ with a shortest separating sequence of length $i+1$, say $ax \in I^{i+1}$ (i.e., a is the first letter and x the tail of the sequence). Since ax is separating for s and t but a is not, x must be separating for $\delta(s, a)$ and $\delta(t, a)$. It is also a shortest separating sequence, because if $y \in I^*$ was shorter than x, then ay would be a separating sequence for s and t, and shorter than ax. This proves that there are two states $\delta(s, a), \delta(t, a)$ with a shortest separating sequence of length i, so $\rho_i \neq \rho_{i-1}$. □

Since a partition of n elements can only be refined n times, the sequence $\rho_0, \rho_1, \ldots$ of partitions becomes constant after at most n steps. And since the machine is minimized, after this point the partitions contain only singletons. So any two states have a separating sequence of length at most $n-1$, and the homing sequence has length at most $(n-1)^2$.

Hibbard [Hib61] improved this bound, showing that the homing sequence computed by Algorithm 1 has length at most $n(n-1)/2$, provided we choose two states with the shortest possible separating sequence in each iteration. Moreover, for every n there is an n-state machine whose shortest homing sequence has length $n(n-1)/2$, so the algorithm has the optimal worst case behavior in terms of output length.

Computing Separating Sequences. We are now ready to fill in the last detail of the algorithm. To compute separating sequences, we first construct the partitions $\rho_1, \rho_2, \ldots, \rho_r$ described above, where r is the smallest index such that ρ_r contains only singletons. Two states $s, t \in S$ belong to different blocks of ρ_1 if and only if there is an input $a \in I$ so that $\lambda(s, a) \neq \lambda(t, a)$, and thus ρ_1 can be computed. Two states $s, t \in S$ belong to different blocks of ρ_i for $i > 1$ if and only if there is an input a such that $\delta(s, a)$ and $\delta(t, a)$ belong to different blocks of ρ_{i-1}, and thus all ρ_i with $i > 1$ can be computed.

To find a separating sequence for two states $s, t \in S$, find the smallest index i such that s and t belong to different blocks of ρ_i. As argued in the proof of Lemma 1.12, the separating sequence has the form ax, where x is a shortest separating sequence for $\delta(s, a)$ and $\delta(t, a)$. Thus, we find the a that takes s and t to different blocks of ρ_{i-1} and repeat the process until we reach ρ_0. The concatenation of all such a is our separating sequence.

This algorithm can be modified to use only $O(n)$ memory, not counting the space required by the output. Typically, the size of the output does not contribute to the memory requirements, since the sequence is applied to some machine on the fly rather than stored explicitly.

Exercise 1.4. Give an example of a Mealy machine that is not minimized but has a homing sequence. Is there a Mealy machine that is not minimized and has a homing but no synchronizing sequence?

1.3.2 Computing Synchronizing Sequences

Synchronizing sequences are computed in a way similar to homing sequences. The algorithm also concatenates many short sequences into one long, but this time the sequences take two states to the same final state. Analogously to separating sequences, we define merging sequences:

Definition 1.13. A **merging sequence** for two states $s, t \in S$ is a sequence $x \in I^*$ such that $\delta(s, x) = \delta(t, x)$. □

The Algorithm. Algorithm 2 first finds a merging sequence y for two states. This ensures that $|\delta(S, y)| < |S|$, because each state in S gives rise to at most one state in $\delta(S, y)$, but the two states for which the sequence is merging give rise to the same state. This process is repeated, in each step appending a new merging sequence for two states in $\delta(S, x)$ to the result x, thus decreasing $|\delta(S, x)|$.

If at some point the algorithm finds two states that do not have a merging sequence, then there is no synchronizing sequence: if there was, it would merge them. And if the algorithm terminates by finishing the loop, the sequence x is clearly synchronizing. This shows correctness. Since $|\delta(S, x)|$ is n initially and 1 on successful termination, the algorithm needs at most $n - 1$ iterations.

Algorithm 2 Computing a synchronizing sequence.

```
1   function SYNCHRONIZING(Mealy machine M)
2       x ← ε
3       while |δ(S, x)| > 1
4           find two different states s0, t0 ∈ δ(S, x)
5           let y be a merging sequence for s0 and t0
              (if none exists, return FAILURE)
6           x ← xy
7       return x
```

As a consequence of this algorithm, we have (see, e.g., Starke [Sta72]):

Theorem 1.14. *A machine has a synchronizing sequence if and only if every pair of states has a merging sequence.*

Proof. If the machine has a synchronizing sequence, then it is merging for every pair of states. If every pair of states has a merging sequence, then Algorithm 2 computes a synchronizing sequence. □

To convince yourself, it may be instructive to go back and see how this algorithm works on Exercise 1.1.

Computing Merging Sequences. It remains to show how to compute merging sequences. This can be done by constructing a product machine $\mathcal{M}' = \langle I', S', \delta' \rangle$, with the same input alphabet $I' = I$ and no outputs (so we omit O' and λ'). Every state in $\mathcal{M}'$ is a set of one or two states in $\mathcal{M}$, i.e., $S' = \{\{s\}, \{s,t\} : s,t \in S\}$. Intuitively, these sets correspond to possible final states in $\mathcal{M}$ after applying a sequence to the s_0, t_0 chosen on line 4 of Algorithm 2. Thus we define δ' by setting $\{s,t\} \xrightarrow{a} \{s',t'\}$ in $\mathcal{M}'$ if and only if $s \xrightarrow{a} s'$ and $t \xrightarrow{a} t'$ in $\mathcal{M}$, where we may have $s = t$ or $s' = t'$. In other words, δ' is the restriction of δ to sets of size one or two. Clearly, $\delta'(\{s,t\}, x) = \{s',t'\}$ if and only if $\delta(\{s,t\}, x) = \{s',t'\}$ (where in the first case $\{s,t\}$ and $\{s',t'\}$ are interpreted as states of $\mathcal{M}'$ and in the second case as sets of states in $\mathcal{M}$). See Figure 1.5 for an example of the product machine. Thus, to find a merging sequence for s and t we only need to check if it is possible to reach a singleton set from $\{s,t\}$ in $\mathcal{M}'$. This can be done, e.g., using breadth-first search [CLRS01].

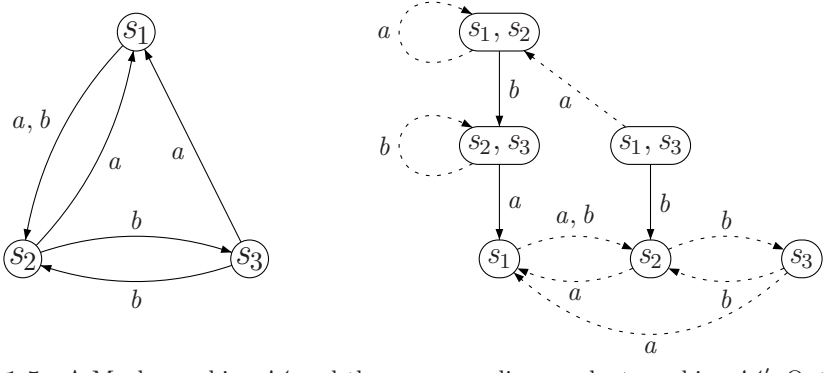

Fig. 1.5. A Mealy machine $\mathcal{M}$ and the corresponding product machine $\mathcal{M}'$. Outputs are omitted here. In the product machine, edges in the shortest path forest are solid and other edges are dashed.

Efficient Implementation. The resulting algorithm is easily seen to run in time $O(n^4 + n^3 \cdot |I|)$. In each of the $O(n)$ iterations, we compute $\delta(S, xy)$ by applying y to every element of $\delta(S, x)$. Since $|y| = O(n^2)$ and $|\delta(S,x)| = O(n)$, this needs $O(n^3)$ time per iteration. The breadth first search needs linear time in the size of $\mathcal{M}'$, i.e., $O(n^2 \cdot |I|)$. We now show how to save a factor n, using several clever tricks due to Eppstein [Epp90].

Theorem 1.15 ([Epp90])**.** *Algorithm 2 can be implemented to consume time in $O(n^3 + n^2 \cdot |I|)$ and working space $O(n^2 + n \cdot |I|)$ (not counting the space for the output).*

The extra condition "not counting the space for the output" is necessary because the only known upper bound on the length of synchronizing sequences is

$O(n^3)$ (cf. Theorem 1.16). The output may not contribute to the space requirement, in case the sequence is not stored explicitly but applied to some machine one letter at a time as it is being constructed.

Proof.
Overview. The proof is in several steps. First, we show how to implement the algorithm to use only the required time, not bothering about how much space is used. The real bottleneck is computing $\delta(S, x)$. For this to be fast, we precompute one lookup table per node of the shortest path forest of $\mathcal{M}'$. Each table has size $O(n)$, and since there are $O(n^2)$ nodes, the total space is $O(n^3)$, which is too big. To overcome this, we then show how to leave out all but every n'th table, without destroying the time requirement.
The Shortest Path Forest. We first show how to satisfy the time requirements. Run breadth-first search in advance, starting simultaneously from all singletons in the product machine $\mathcal{M}'$ and taking transitions backward. Let it produce a shortest path forest, i.e., a set of trees where the roots are the singletons and the path from any node to the root is of shortest length. This needs $O(n^2 \cdot |I|)$ time. For any $\{s, t\} \in S'$, denote by $\tau_{s,t}$ the path from $\{s, t\}$ to the root in this forest. The algorithm will always select $y = \tau_{s_0,t_0}$ on line 5.
Tables. Recall that we obtain $\delta(S, xy)$ by iterating through all elements u of the already known set $\delta(S, x)$ and computing $\delta(u, y)$. In the worst case, y is quadratically long and thus the total work for all $O(n)$ choices of u in all $O(n)$ iterations becomes $O(n^4)$. We now improve this bound to $O(n^2)$. Since $y = \tau_{s_0,t_0}$, we precompute $\delta(u, \tau_{s,t})$ for every $s, t, u \in S$. Thus, computing $\delta(u, y)$ is done in $O(1)$ time by a table lookup and we obtain $\delta(S, xy)$ from $\delta(S, x)$ in $O(n)$ time. The tables need $O(n^3)$ space, but we will improve that later.
Computing Tables. We now show how to compute the tables. For every $\{s, t\} \in S'$ we compute an n element table, with the entries $\delta(u, \tau_{s,t})$ for each $u \in S$, using totally $O(n^3)$ time and space, as follows. Traverse the shortest path forest in pre-order, again following transitions backward. When visiting node $\{s, t\} \in S'$, let $\{s', t'\}$ be its parent in the forest. Thus, there is some a such that $\tau_{s,t} = a\tau_{s',t'}$. Note that $\delta(u', \tau_{s',t'})$ has already been computed for all $u' \in S$, since we traverse in pre-order. To compute $\delta(u, \tau_{s,t})$ we only need to compute $\delta(u, a)$ and plug it into the table in the parent node: $\delta(u, \tau_{s,t}) = \delta(\delta(u, a), \tau_{s',t'})$. This takes constant time, so doing it for every $u \in S$ and every node in the tree requires only $O(n^3)$ time.

We thus achieved the time bound, but the algorithm now needs $O(n^2 \cdot |I|)$ space to store $\mathcal{M}'$ and $O(n^3)$ to store the tables of all $\delta(u, \tau_{s,t})$. We will reduce the first to $O(n \cdot |I| + n^2)$ and the second to $O(n^2)$.
Compact Representation of $\mathcal{M}'$. The graph $\mathcal{M}'$ has $O(n^2)$ nodes and $O(n^2 \cdot |I|)$ edges. The breadth-first search needs one flag per node to indicate if it has been visited, so we cannot get below $O(n^2)$ space. But we do not have to store edges explicitly. The forward transitions δ' can be computed on the fly using δ. The breadth-first search takes transitions backward. To avoid representing backwards transitions explicitly, we precompute for every $s' \in S$ and $a \in I$, the set $\delta^{-1}(s', a) \stackrel{\text{def}}{=} \{s \in S : \delta(s, a) = s'\}$ (requiring totally $O(n \cdot |I|)$ space).

By definition, the backward transitions on input a from some state $\{s', t'\} \in S'$ are all $\{s, t\}$ so that $s \in \delta^{-1}(s', a)$ and $t \in \delta^{-1}(t', a)$. For a single state and a single input, these can be found in time $O(r)$, where r is the number of resulting backward transitions. Consequently, the breadth-first search can still be done in $O(n^2 \cdot |I|)$ time even if edges are not represented explicitly.

Leaving out Tables. To reduce the space needed by tables, we will leave out the tables for all but at most every n'th node of the forest, so the distribution of tables in the forest becomes "sparse". At the same time we will guarantee that following the shortest path from any node toward a root, a node with a table will be found after at most n steps. Thus, when the main algorithm computes $\delta(\delta(S, x), y)$ it has to follow the shortest path in the forest for at most n steps per state in $\delta(S, x)$ before it can look up the answer. As a result, the total time over all iterations to update $\delta(S, xy)$ grows to $O(n^3)$, but that is within the time limit.

Which Tables to Leave out. To determine which nodes in the shortest path forest that should have a table, we first take a copy of the forest. Take a leaf of maximal depth, follow the path from this leaf toward the root for n steps and let $\{s, t\}$ be the node we arrive at. Mark $\{s, t\}$ as a node for which the table should be computed, and remove the entire subtree rooted at $\{s, t\}$. Repeat this process as long as possible, i.e., until the resulting forest has depth less than n. Since every removed subtree has depth n, the path from any node to a marked node has length at most n, thus guaranteeing that updating $\delta(u, xy)$ needs at most n steps. Moreover, every removed subtree has at least n nodes, so tables will be stored in at most every n'th node.

Computing Tables When They Are Few. Finally, computing the tables when they are more sparsely occurring is done almost as before, but instead of using the table value from a parent, we find the nearest ancestor that has a table, requiring $O(n)$ time for every element of every table, summing up to $O(n^3)$ because there are $O(n)$ tables with n entries each.

We conclude the proof with a summary of the requirements of the algorithm.

- The graph $\mathcal{M}'$ needs $O(n^2)$ space for nodes and $O(n \cdot |I|)$ for edges.
- There are $O(n)$ tables, each one taking $O(n)$ space, so totally $O(n^2)$.
- The breadth-first search needs $O(n^2 \cdot |I|)$ time.
- Computing the tables needs $O(n^3)$ time.
- In each of the $O(n)$ iterations, computing $\delta(S, xy)$ needs $O(n^2)$ time.
- In each iteration, writing the merging sequence to the output is linear in its length, which is bounded by $O(n^2)$. □

Length of Synchronizing Sequences. The merging sequences computed are of minimal length, because breadth-first search computes shortest paths. Unfortunately, this does not guarantee that Algorithm 2 finds a shortest possible synchronizing sequence, since the order in which states to merge are picked may not be optimal. It is possible to pick the states that provide for the shortest merging sequence without increasing the asymptotic running time, but there are machines where this strategy is not the best. In fact, we will see in Section 1.4.1

that finding shortest possible sequences is NP-hard, meaning that it is extremely unlikely that a polynomial time algorithm exists.

Note that each merging sequence has length at most $n(n-1)/2$ because it is a simple path in $\mathcal{M}'$; thus the length of the synchronizing sequence is at most $n(n-1)^2/2$. We now derive a slightly sharper bound.

Theorem 1.16. *If a machine has a synchronizing sequence, then it has one of length at most* $n^3/3$.

Proof. At any iteration of Algorithm 2, among all states in $Q \stackrel{\text{def}}{=} \delta(S,x)$, find two that provide for a shortest merging sequence. We first show that when $|Q| = k$, there is a pair of states in Q with a merging sequence of length at most $n(n-1)/2 - k(k-1)/2 + 1$. Every shortest sequence passes through each node in the shortest path forest at most once: otherwise we could cut away the sub-sequence between the repeated nodes to get a shorter sequence. Also, it cannot visit any node in the forest that has both states in Q, because those two states would then have a shorter merging sequence. There are $n(n-1)/2$ nodes in the shortest path forest (not counting singletons)[4], and $k(k-1)/2$ of them correspond to pairs with both nodes in $\delta(S,x)$. Thus, there is a merging sequence of length at most $n(n-1)/2 - k(k-1)/2 + 1$.

The number $|\delta(S,x)|$ of possible states is n initially, and in the worst case it decreases by only one in each iteration until it reaches 2 just before the last iteration. Thus, summing the length of all merging sequences, starting from the end, we get

$$\sum_{i=2}^{n} \left(\frac{n(n-1)}{2} - \frac{i(i-1)}{2} + 1 \right),$$

which evaluates to $n^3/3 - n^2 + \frac{5}{3}n - 1 < n^3/3$. □

This is not the best known bound: Klyachko, Rystsov, and Spivak [KRS87] improved it to $(n^3-n)/6$. Similarly to the proof above, they bound the length of each merging sequence, but with a much more sophisticated analysis they achieve the bound $(n-k+2)\cdot(n-k+1)/2$ instead of $n(n-1)/2 - k(k-1)/2 + 1$. The best known lower bound is $(n-1)^2$, and it is an open problem to close the gap between the lower quadratic and upper cubic bounds. Černý [Čer64] conjectured that the upper bound is also $(n-1)^2$.

Exercise 1.5. Consider the problem of finding a synchronizing sequence that ends in a specified final state. When does such a sequence exist? Extend Algorithm 2 to compute such a sequence.

Exercise 1.6. Show how to modify the algorithm of this section, so that it tests whether a machine has a synchronizing sequence without computing it, in time $O(n^2 \cdot |I|)$.

A similar algorithm for the problem in Exercise 1.6 was suggested by Imreh and Steinby [IS95].

[4] Recall that $|S' \setminus \{\text{singletons}\}|$ = the number of two-element subsets of $S = \binom{n}{2} = n(n-1)/2$.

1.3.3 Computing Homing Sequences for General Machines

In this section we remove the restriction from Section 1.3.1 that the machine has to be minimized. Note that for general machines, an algorithm to compute homing sequences can be used also to compute synchronizing sequences: just remove all outputs from the machine and ask for a homing sequence. Since there are no outputs, homing and synchronizing sequences are the same thing. It is therefore natural that the algorithm unifies Algorithm 1 of Section 1.3.1 and Algorithm 2 of Section 1.3.2 by computing separating *or* merging sequences in each step.

Recall Lemma 1.10, saying that the quantity $\sum_{B\in\sigma(x)}|B| - |\sigma(x)|$ does not increase as the sequence x is extended. Algorithm 3 repeatedly applies a sequence that strictly decreases this quantity: it takes two states from the same block of the current state uncertainty and applies either a merging or a separating sequence for them. If the sequence is merging, then the sum of sizes of all blocks diminishes. If it is separating, then the block containing the two states is split. Since the quantity is $n-1$ initially and 0 when the algorithm finishes, it finishes in at most $n-1$ steps.

If the algorithm does not find either a merging or a separating sequence on line 5, then the machine has no homing sequence. Indeed, any homing sequence that does not take s and t to the same state must give different outputs for them, so it is either merging or separating. This shows correctness of the algorithm.

Algorithm 3 Computing a homing sequence for a general machine.

```
1   function HOMING(Mealy machine M)
2       x ← ε
3       while there is a block B ∈ σ(x) with |B| > 1
4           find two different states s, t ∈ B
5           let y be a separating or merging sequence for s and t
              (if none exists, return FAILURE)
6           x ← xy
7       return x
```

Similar to Theorem 1.14, we have the following for homing sequences.

Theorem 1.17 ([Rys83])**.** *A Mealy machine has a homing sequence if and only if every pair of states either has a merging sequence or a separating sequence.*

Note that, as we saw already in Section 1.3.1, every minimized machine has a homing sequence.

Proof. Assume there is a homing sequence and let $s, t \in S$ be any pair of states. If the homing sequence takes s and t to the same final state, then it is a merging sequence. Otherwise, by the definition of homing sequence, it must be possible

to tell the two final states apart by looking at the output. Thus the homing sequence is a separating sequence.

Conversely, if every pair of states has either a merging or a separating sequence, then Algorithm 3 computes a homing sequence. □

We cannot hope for this algorithm to be any faster than the one to compute synchronizing sequences, because they have to do the same job if there is no separating sequence. But it is easy to see that it can be implemented not to be worse either. By definition, two states have a separating sequence if and only if they are not equivalent (two states are equivalent if they give the same output for all input sequences: see Section 21). Hence, we first minimize the machine to find out which states have separating sequences. As long as possible, the algorithm chooses non-equivalent states on line 4 and only looks for a separating sequence. Thus, the first half of the homing sequence is actually a homing sequence for the minimized machine, and can be computed by applying Algorithm 1 to the minimized machine. The second half of the sequence is computed as described in Section 1.3.2, but only selecting states from the same block of the current state uncertainty.

1.3.4 Computing Adaptive Homing Sequences

Recall that an **adaptive homing sequence** is applied to a machine as it is being computed, and that each input symbol depends on the previous outputs. An adaptive homing sequence is formally defined as a decision tree, where each node is labeled with an input symbol and each edge is labeled with an output symbol. The experiment consists in first applying the input symbol in the root, then following the edge corresponding to the observed output, applying the input symbol in the reached node and so on. When a leaf is reached, the final state can be uniquely determined. The **length** of an adaptive homing sequence is defined as the depth of this tree.

Using adaptive sequences can be an advantage because they are often shorter than preset sequences. However, it has been shown that machines possessing the longest possible preset homing sequences (of length $n(n-1)/2$) require equally long adaptive homing sequences [Hib61].

It is easy to see that a machine has an adaptive homing sequence if and only if it has a preset one. One direction is immediate: any preset homing sequence corresponds to an adaptive one. For the other direction, note that by Theorem 1.17 it is sufficient to show that if a machine has an adaptive homing sequence, then every pair of states has a merging or a separating sequence. Assume toward a contradiction that a machine has an adaptive homing sequence but there are two states $s, t \in S$ that have neither a merging nor a separating sequence. Consider the leaf of the adaptive homing sequence tree that results when the initial state is s. Since s and t have no separating sequence, the same leaf would be reached also if t was the initial state. But since s and t have no merging sequence, there are at least two possible final states, contradicting that there must be only one possible final state in a leaf.

Algorithms 1 and 3 for computing preset homing sequences can both be modified so that they compute adaptive homing sequences. To make the sequence adaptive (and possibly shorter), note that it can always be determined from the output which block of the current state uncertainty that the current state belongs to. Only separating or merging sequences for states in this block need to be considered. Algorithm 4 is similar to Algorithm 3, except we only consider the relevant block of the current state uncertainty (called B in the algorithm). For simplicity, we stretch the notation a bit and describe the algorithm in terms of the intuitive definition of adaptive homing sequence, i.e., it applies the sequence as it is constructed, rather than computes an explicit decision tree.

Algorithm 4 Computing an adaptive homing sequence.

```
1   function ADAPTIVE-HOMING(Mealy machine M)
2       B ← S
3       while |B| > 1
4           find two different states s, t ∈ B
5           let y be a separating or merging sequence for s and t
              (if none exists, return FAILURE)
6           apply y to M and let z be the observed output sequence
7           B ← {u ∈ δ(B, y) : λ(B, y) = z}
```

The same arguments as before show correctness and cubic running time. Algorithm 1 for minimized machines can similarly be made adaptive, resulting in the same algorithm except with the words "or merging" on line 5 left out. Although the computed sequences are never longer than the non-adaptive ones computed by Algorithms 1 and 3, we stress once more that they do not have to be the shortest possible.

Algorithm 4 occurred already in the paper by Moore [Moo56], even before the algorithm of Section 1.3.1 for preset sequences.

Adaptive *synchronizing* sequences were suggested by Pomeranz and Reddy [PR94]. They can be computed, e.g., by first applying a homing sequence (possibly adaptive), and then from the final known state find a sequence that takes the machine to one particular final state.

1.3.5 Computing Minimal Homing and Synchronizing Sequences

The algorithms we saw so far do not necessarily compute the shortest possible sequences. It is of practical interest to minimize the length of sequences: the Mealy machine may be a model of some system where each transition is very expensive, such as a remote machine or the object pushing in Example 1.3 where making a transition means moving a physical object, which can take several seconds. An extreme example is the subway map in Exercise 1.1, where, for each transition, a human has to buy a ticket and travel several kilometers. Moreover, the sequence

may be computed once and then applied a large number of times. We will see in Section 1.4.1 that finding a minimal length sequence is an NP-complete problem, hence unlikely to have a polynomial time algorithm. The algorithms in this section compute minimal length sequences but need exponential time and space in the worst case.

To compute a shortest synchronizing sequence, we define the *synchronizing tree*. This is a tree describing the behavior of the machine for each possible input string, but pruning off branches that are redundant when computing synchronizing sequences:

Definition 1.18. The **synchronizing tree** (for a Mealy machine) is a rooted tree where edges are labeled with input symbols and nodes with sets of states, satisfying the following conditions.

(1) Each non-leaf has exactly $|I|$ children, and the edges leading to them are labeled with different input symbols.
(2) Each node is labeled with $\delta(S, x)$, where x is the sequence of input symbols occurring as edge labels on the path from the root to the node.
(3) A node is a leaf iff:
   (a) either its label is a singleton set,
   (b) or it has the same label as a node of smaller depth in the tree. □

By (2), the root node is labeled with S. To find the shortest synchronizing sequence, compute the synchronizing tree top-down. When the first leaf satisfying condition (3a) is found, the labels on the path from the root to the leaf form a synchronizing sequence. Since no such leaf was found on a previous level, this is the shortest sequence and the algorithm stops and outputs it. To prove correctness, it is enough to see that without condition (3b) the algorithm would compute every possible string of length 1, then of length 2, and so on until it finds one that is synchronizing. No subtree pruned away by (3b) contains any *shortest* synchronizing sequence, because it is identical to the subtree rooted in the node with the same label, except every node has a bigger depth.

The term "smaller depth" in (3b) is deliberately a bit ambiguous: it is not incorrect to interpret it as "strictly smaller depth". However, an algorithm that generates the tree in a breadth-first manner would clearly benefit from making a node terminal also if it occurs on the same level and has already been generated.

Example 1.19. Figure 1.6 depicts a Mealy machine and its synchronizing tree. The root note is labeled with the set of all states. It has two children, one per input symbol. The leftmost child is labeled with $\delta(\{s_1, s_2, s_3\}, a) = \{s_1, s_2\}$ and the rightmost with $\delta(\{s_1, s_2, s_3\}, b) = \{s_1, s_2, s_3\}$. Thus, it is pointless for the sequence to start with a b, and the right child is made a leaf by rule (3b) of the definition. The next node to expand is the one labeled by $\{s_1, s_2\}$. Applying a gives again $\delta(\{s_1, s_2\}, a) = \{s_2, s_2\}$, so we make the left child a leaf. Applying b gives the child $\delta(\{s_1, s_2\}, b) = \{s_1, s_3\}$. Finally, we expand the node labeled $\{s_1, s_3\}$, and arrive at the singleton $\delta(\{s_1, s_3\}, a) = s_3$. It is not necessary to expand any further, as the labels from the root to the singleton leaf form a shortest synchronizing sequence, *aba*.

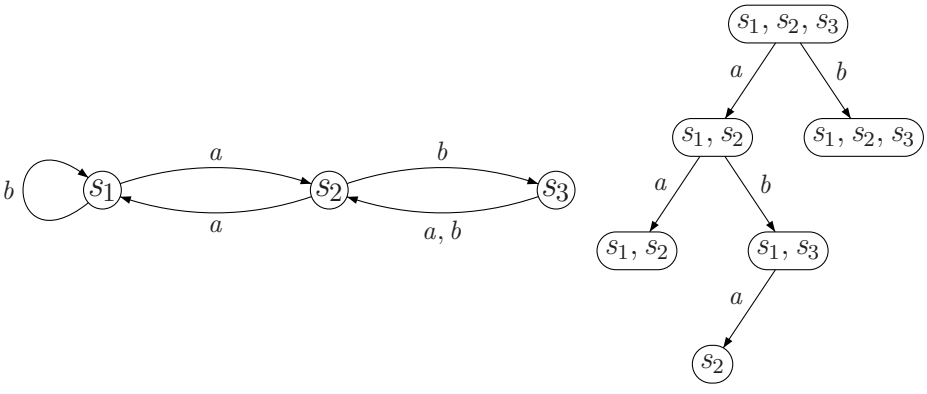

Fig. 1.6. A machine and its corresponding synchronizing tree. Note that the rightmost and leftmost nodes of the tree have been made leaves due to rule (3b) of Definition 1.18. Since the lowest leaf is labeled with a singleton s_2, the path leading to it from the root indicates a shortest synchronizing sequence, *aba*.

It is possible to replace condition (3b) of Definition 1.18 with a stronger one, allowing to prune the tree more: stipulate that a node becomes a leaf also if its label is a *superset* of some node of smaller depth. This clearly works, because if $P \subseteq Q \subseteq S$ are the node labels, then $\delta(P, x) \subseteq \delta(Q, x)$ for all sequences x. The drawback is that it can be more costly to test this condition.

The homing tree is used analogously to compute shortest *homing* sequences.

Definition 1.20. The **homing tree** (for a Mealy machine) is a rooted tree where edges are labeled with input symbols and nodes with current state uncertainties (i.e., sets of sets of states), satisfying the following conditions.

(1) Each non-leaf has exactly $|I|$ outgoing edges, labeled with different input symbols.
(2) Each node is labeled with $\sigma(x)$, where x is the sequence of input symbols occurring as edge labels on the path from the root to the node.
(3) A node is a leaf iff:
    (a) either each block of its label is a singleton set,
    (b) or it has the same label as a node of smaller depth. □

Condition (3b) can be strengthened in a similar way for the homing tree as for the synchronizing tree. Here we turn a node into a leaf also if each block of its label is a superset of some block in the label of another node at a smaller depth.

The homing tree method was introduced by Gill [Gil61] and the synchronizing tree method has been described by Hennie [Hen64]. Synchronizing sequences are sometimes used in test generation for circuits without a reset (a reset, here, would be an input symbol that takes every state to one and the same state,

i.e., a trivial synchronizing sequence). In this application, the state space is $\{0,1\}^k$ and typically very big. Rho, Somenzi and Pixley [RSP93] suggested a more practical algorithm for this special case based on binary decision diagrams (BDDs).

1.4 Complexity

This section shows two hardness results for related problems. First, Section 1.4.1 shows that it is NP-hard to find a shortest homing or synchronizing sequence. Second, Section 1.4.2 shows that it is PSPACE-complete to determine if there is a homing or synchronizing sequence when the initial state is known to be in a specified subset $Q \subseteq S$ of the states.

1.4.1 Computing Shortest Homing and Synchronizing Sequences Is NP-hard

If a homing or synchronizing sequence is going to be used in practice, it is natural to ask for it to be as short as possible. We saw in Section 1.3.1 that we can always find a homing sequence of length at most $n(n-1)/2$ if one exists and the machine is minimized, and Sections 1.3.2 and 1.3.3 explain how to find synchronizing or homing sequences of length $O(n^3)$, for general machines. The algorithms for computing these sequences run in polynomial time. But the algorithms of Section 1.3.5 that compute minimal-length homing and synchronizing sequences are exponential. In this section, we explain this exponential running time by proving that the problems of finding homing and synchronizing sequences *of minimal length* are significantly harder than those of finding just any sequence: the problems are NP-hard, meaning they are unlikely to have polynomial-time algorithms.

The Reduction Since only decision problems can be NP-complete, formally it does not make sense to talk about NP-completeness of *computing* homing or sequences. Instead, we look at the decision version of the problems: is there a sequence of length at most k?

Theorem 1.21 ([Epp90])**.** *The following problems, taking as input a Mealy machine $\mathcal{M}$ and a positive integer k, are NP-complete:*

(1) *Does $\mathcal{M}$ have a homing sequence of length $\leq k$?*
(2) *Does $\mathcal{M}$ have a synchronizing sequence of length $\leq k$?*

Proof. To show that the problems belong to NP, note that a nondeterministic algorithm easily guesses a sequence of length $\leq k$ (where k is polynomial in the size of the machine) and verifies that it is homing or synchronizing in polynomial time.

To show that the problems are NP-hard, we reduce from the NP-complete problem 3SAT [GJ79]. Recall that in a boolean formula, a **literal** is either a

variable or a negated variable, a **clause** is the "or" of several literals, and a formula is in **conjunctive normal form** (**CNF**) if it is the "and" of several clauses. In **3SAT** we are given a boolean formula φ over n variables $v_1, \ldots, v_n$ in CNF with exactly three literals per clause, so it is on the form $\varphi = \bigwedge_{i=1}^{m}(l_1^i \vee l_2^i \vee l_3^i)$, where each l_j^i is a literal. The question is whether φ is satisfiable, i.e., whether there is an assignment that sets each variable to either T or F and makes the formula true.

Given any such formula φ with n variables and m clauses, we create a machine with $m(n+1)+1$ states. The machine gives no output (or one and the same output on all transitions, to formally fit the definition of Mealy machines), so synchronizing and homing sequences are the same thing and we can restrict the discussion to synchronizing sequences. There will always be a synchronizing sequence of length $n+1$, but there will be one of length n if and only if the formula is satisfiable. The input alphabet is $\{\mathrm{T}, \mathrm{F}\}$, and the i'th symbol in the sequence roughly corresponds to assigning T or F to variable i.

The machine has one special state s, and for each clause $(l_1^i \vee l_2^i \vee l_3^i)$ a sequence of $n+1$ states $s_1^i, \ldots, s_{n+1}^i$. Intuitively, s_j^i leads to s_{j+1}^i, except if variable v_j is in the clause and satisfied by the input letter, in which case a shortcut to s is taken. The last state s_{n+1}^i leads to s and s has a self-loop. See Figure 1.7 for an example.

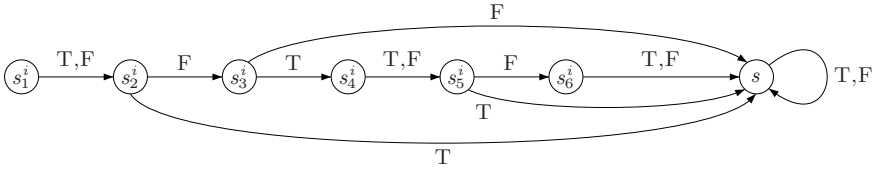

Fig. 1.7. Example of the construction for the clause $(v_2 \vee \neg v_3 \vee v_5)$, where the formula has five variables $v_1, \ldots, v_5$. States s_1^i and s_4^i only have transitions to the next state, because they do not occur in the clause. States s_2^i, and s_5^i have shortcuts to s on input T because v_2 and v_5 occur without negation, and s_3^i has a shortcut to s on input F because it occurs negated. Note that such a chain is constructed for each clause, and they are all different except for the last state s.

Formally, we have the following transitions, for all $1 \le i \le m$:

- The last state goes to s, $s_{n+1}^i \xrightarrow{\mathrm{T,F}} s$, and s has a self-loop, $s \xrightarrow{\mathrm{T,F}} s$.
- If v_j does not occur in the clause, then $s_j^i \xrightarrow{\mathrm{T,F}} s_{j+1}^i$.
- If v_j occurs positively in the clause, i.e., one of l_1^i, l_2^i, or l_3^i is v_j, then $s_j^i \xrightarrow{\mathrm{T}} s$ and $s_j^i \xrightarrow{\mathrm{F}} s_{j+1}^i$.
- If v_j occurs negatively in the clause, i.e., one of l_1^i, l_2^i, or l_3^i is $\neg v_j$, then $s_j^i \xrightarrow{\mathrm{F}} s$ and $s_j^i \xrightarrow{\mathrm{T}} s_{j+1}^i$.

To finish the proof, we have to show that the machine thus constructed has a synchronizing sequence of length n if and only if φ is satisfiable. First, assume

φ is satisfiable and let ν be the satisfying assignment, so $\nu(v_i) \in \{\mathrm{T}, \mathrm{F}\}$. Then the corresponding sequence $\nu(v_1)\nu(v_2)\ldots\nu(v_n) \in I^*$ is synchronizing: starting from any state s_j^i with $j \geq 2$ or from s, we reach s in $\leq n$ steps. Consider state s_1^i and recall that at least one of the literals in the i'th clause is satisfied. Thus, if this literal contains variable v_j, the shortcut from s_j^i to s will be taken, so also from s_1^i will s be reached in $\leq n$ steps.

Conversely, assume there is a synchronizing sequence $b = b_1 b_2 \ldots b_k$ of length $k \leq n$. Hence $\delta(t, b) = s$ for every state t. In particular, starting from s_1^i one of the shortcuts must be taken, say from s_j^i to s. Thus v_j occurs in the i:th clause and setting it to b_j makes the clause true. It follows that the assignment that sets v_j to b_j, for $1 \leq j \leq n$, makes all clauses true. This completes the proof. □

Rivest and Schapire [RS93] mention without proof that it is also possible to reduce from the problem *exact 3-set cover*.

Exercise 1.7. Show that the problem of computing the shortest homing sequence is NP-complete, even if the machine is minimized and the output alphabet has size at most two.

1.4.2 PSPACE-Completeness of a More General Problem

So far we assumed the biggest possible amount of ignorance – the machine can initially be in any state. However, it is sometimes known that the initial state belongs to a particular subset Q of S. If a sequence takes every state in Q to the same final state, call it an *Q-synchronizing sequence*. Similarly, say that an *Q-homing sequence* is one for which the output reveals the final state if the initial state is in Q. In particular, homing and synchronizing are the same as S-homing and S-synchronizing. Even if no homing or synchronizing sequence exists, a machine can have Q-homing or Q-synchronizing sequences (try to construct such a machine, using Theorem 1.14). However, it turns out that even determining if such sequences exist is far more difficult: as we will show soon, this problem is PSPACE-complete. PSPACE-completeness is an ever stronger hardness result than NP-completeness, meaning that the problem is "hardest" among all problems that can be solved using polynomial space. It is widely believed that such problems do not have polynomial time algorithms, not even if nondeterminism is allowed. It is interesting to note that Q-homing and Q-synchronizing sequences are not polynomially bounded: as we will see later in this section, there are machines that have synchronizing sequences but only of exponential length. The following theorem was proved by Rystsov [Rys83]. It is similar to Theorem 3.2 in Section 3.

Theorem 1.22 ([Rys83])**.** *The following problems, taking as input a Mealy machine $\mathcal{M}$ and a subset $Q \subseteq S$ of its states, are PSPACE-complete:*

(1) *Does $\mathcal{M}$ have an Q-homing sequence?*
(2) *Does $\mathcal{M}$ have an Q-synchronizing sequence?*

Proof. We first prove that the problems belong to NPSPACE, by giving polynomial space nondeterministic algorithms for both problems. It then follows from the general result PSPACE = NPSPACE (in turn a consequence of Savitch's theorem [Sav70, Pap94]) that they belong to PSPACE. The algorithm for synchronizing sequences works as follows. Let $Q_0 = Q$, nondeterministically select one input symbol a_0, apply it to the machine and compute $Q_1 = \delta(Q_0, a_0)$. Iterate this process, in turn guessing a_1 to compute $Q_2 = \delta(Q_1, a_1)$, a_2 to compute $Q_3 = \delta(Q_2, a_2)$, and so on until $|Q_i| = 1$, at which point we verified that $a_0 a_1 \dots a_{i-1}$ is synchronizing (because $Q_i = \delta(Q, a_0 a_1 \dots a_i)$). This needs at most polynomial space, because the previously guessed symbols are forgotten, so only the current Q_i needs to be stored. The algorithm for homing sequences is similar, but instead of keeping track of the current $\delta(Q, a_1 a_2 \dots a_i)$, the algorithm keeps track of the current state uncertainty $\sigma(a_0 a_1 \dots a_i)$ and terminates when it contains only singletons.

To show PSPACE-hardness, we reduce from the PSPACE-complete problem **Finite Automata Intersection** [Koz77, GJ79]. In this problem, we are given k finite, total, deterministic automata $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_k$ (all with the same input alphabet) and asked whether there is a string accepted by all $\mathcal{A}_i$, i.e., whether the intersection of their languages is nonempty. Recall that finite automata are like Mealy machines, except they do not produce outputs, they have one distinguished initial state and a set of distinguished final states. We construct a Mealy machine $\mathcal{M}$ with the same input alphabet as the automata, and specify a subset Q of its states, such that a sequence is Q-synchronizing for $\mathcal{M}$ if and only if it is accepted by all $\mathcal{A}_i$. As in Theorem 1.21, $\mathcal{M}$ does not give output, so a sequence is homing if and only if it is synchronizing, and the rest of the discussion will be restricted to synchronizing sequences. To construct $\mathcal{M}$, first take a copy of all $\mathcal{A}_i$. Add one new input symbol z, and two new states, Good and Bad. Make z-transitions from each accepting state of the automata to Good and from each non-accepting state to Bad, and finally make self-loops on Good and Bad: $\textsc{Good} \xrightarrow{I \cup \{z\}} \textsc{Good}$ and $\textsc{Bad} \xrightarrow{I \cup \{z\}} \textsc{Bad}$. See Figure 1.8 for an example.

Let Q be the set of all initial states of the automata, together with Good. We will show that all the automata accept a common word x if and only if $\mathcal{M}$ has an Q-synchronizing sequence (and that sequence will be xz). First assume all automata accept x. Then xz is an Q-synchronizing sequence of $\mathcal{M}$: starting from the initial state of any automaton, the sequence x will take us to a final state. If in addition we apply the letter z, we arrive at Good. Also, any sequence applied to Good arrives at Good. Thus $\delta(Q, xz) = \textsc{Good}$ so xz is an Q-synchronizing sequence.

Conversely, assume $\mathcal{M}$ has an Q-synchronizing sequence. Since we can only reach Good from Good, the final state must be Good. In order to reach Good from any state in $Q \setminus \{\textsc{Good}\}$, the sequence must contain z. In the situation just before the first z was applied, there was one possible current state in each of the automata. If any of these states was non-accepting, applying z would take us to Bad, and afterward we would be trapped in Bad and never reach Good. Thus

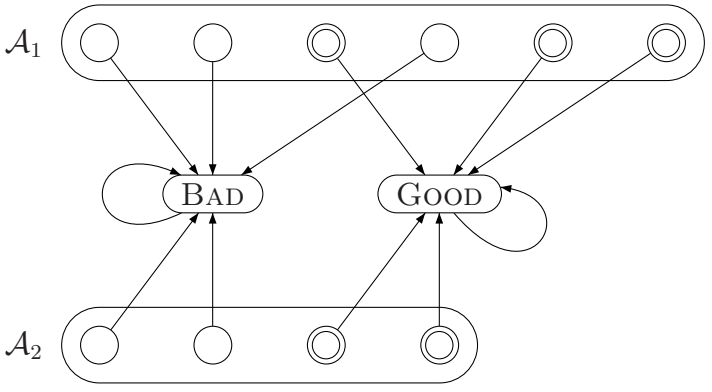

Fig. 1.8. Example of the reduction in Theorem 1.22. We are given two (in general many) automata, $\mathcal{A}_1$ and $\mathcal{A}_2$, and asked if there is a string accepted by all of them. Add a new input symbol z and make z-transitions from accepting and nonaccepting states to the new states GOOD and BAD, respectively, as in the picture. The new states only have self-loops. Let Q be the set of all initial states, together with GOOD; thus a sequence is Q-synchronizing iff it takes every initial state to a final state of the same automaton and then applies z. Thus it corresponds to a word accepted by all automata.

all the automata were in a final state, so the word applied so far is accepted by all automata. This finishes the proof. □

This result also implies that Q-homing and Q-synchronizing sequences may be exponentially long, another indication that they are fundamentally different from the cubically bounded homing and synchronizing sequences. First, a polynomial upper bound would imply that the sequence can be guessed and checked by an NP-algorithm. So the length is superpolynomial unless PSPACE equals NP. Second, Lee and Yannakakis [LY94] gave a stronger result, providing an explicit family of sets of automata, such that the shortest sequence accepted by all automata in one set is exponentially long. Since, in the reduction above, every Q-synchronizing or Q-homing sequence corresponds to a sequence in the intersection language, it follows that these are also of exponential length.

Theorem 1.23 ([LY94])**.** *The shortest sequence accepted simultaneously by n automata is exponentially long in the total size of all automata, in the worst case (even with a unary input alphabet).*

Proof. Denote by p_i the i'th prime. We will construct n automata, the i'th of which accepts sequences of positive length divisible by p_i. Thus, the shortest word accepted by all automata must be positive, and divisible by $p_1 p_2 \cdots p_n > 2^n$. The input alphabet has only one symbol. The i'th automaton consists of a loop of length p_i, one state of which is accepting. To assure that the empty word is not accepted, the initial state is an extra state outside the loop, that points to the successor of the accepting state: see Figure 1.9. By Gauss' prime number

theorem, each automaton has size $O(n \log n)$, so the total size is polynomial in n but the shortest sequence accepted by all automata is exponential. □

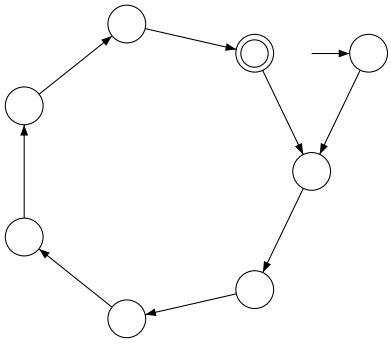

Fig. 1.9. An automaton that accepts exactly the words of positive length divisible by 7. A similar automaton is created for all primes $p_1, \ldots, p_n$.

Consider another generalization of synchronizing sequences, where we are again given a subset $Q \subseteq S$ but now the sequence has to *end* in Q, that is, we want to find a sequence x such that $\delta(S, x) \subseteq Q$. It is not more difficult to show that this problem also is PSPACE-complete; however, it can be solved in time $n^{O(|Q|)}$, so it is polynomial if the size of Q is bounded by a constant [Rys83]. Rystsov shows in the same article that several related problems are PSPACE-complete, and concludes the following result in another paper [Rys92].

Exercise 1.8. A *nondeterministic* Mealy machine is like a Mealy machine except $\delta(s, a)$ is a *set* of states. The transition function δ is extended similarly, so $\delta(Q, a) = \bigcup\{\delta(s, a) : s \in Q\}$ and $\delta(Q, a_1 \ldots a_n) = \delta(\delta(Q, a_1, \ldots, a_{n-1}), a_n)$. Show that the synchronizing sequence problem for nondeterministic Mealy machines is PSPACE-complete. Here, a sequence x is synchronizing for a nondeterministic machine if $|\delta(S, x)| = 1$.

Hint: Use Theorem 1.22.

1.5 Related Topics and Bibliography

The experimental approach to automata theory was initiated by the classical article by Moore [Moo56], who introduces several testing problems, including homing sequences and the adaptive version of Algorithm 1. He also shows the upper bound of $n(n-1)/2$ for the length of adaptive homing sequences. The worst-case length of homing sequences for minimized automata was studied by Ginsburg [Gin58] and finally resolved by Hibbard [Hib61]. The book by Kohavi [Koh78] and the article by Gill [Gil61] contain good overviews of the problem.

Length of Synchronizing Sequences and Černý's Conjecture. Synchronizing sequences were introduced a bit later by Černý [Čer64] and studied mostly independently from homing sequences, with some exceptions [Koh78, Rys83, LY96]. The focus has largely been on the worst-case length of sequences, except an article by Rystsov that classifies the complexity of several related problems [Rys83], the article by Eppstein, which introduces the algorithm in Section 1.3.2 [Epp90], and the survey by Lee and Yannakakis [LY96]. Černý [Čer64] showed an upper bound of $2^n - n - 1$ for the length of synchronizing sequences and conjectured that it can be improved to $(n-1)^2$, a conjecture that inspired much of the research in the area. The first polynomial bound was $\frac{1}{2}n^3 - \frac{3}{2}n^2 + n + 1$ due to Starke [Sta66], and as mentioned in Section 1.3.2 the best known bound is $\frac{1}{6}(n^3 - n)$ due to Klyachko, Rystsov and Spivak [KRS87]. Already Černý [Čer64] proved that there are automata that require synchronizing sequences of length at least $(n-1)^2$, so if the conjecture is true then it is optimal.

Proving or disproving Černý's conjecture is still an open problem, but it has been settled for several special cases: Eppstein [Epp90] proved it for *monotonic* automata, which arise in the orientation of parts that we saw in Example 1.3; Kari [Kar03] showed it for *Eulerian* machines (i.e., where each state has the same in- and out-degrees); Pin [Pin78b] showed it when n is prime and the machine is *cyclic* (meaning that there is an input letter $a \in I$ such that the a-transitions form a cycle through all states); Černý, Pirická and Rosenauerová [ČPR71] showed it when there are at most 5 states. Other classes of machines were studied by Pin [Pin78a], Imreh and Steinby [IS95], Rystsov [Rys97], Bogdanović et al. [BIĆP99], Trakhtman [Tra02], Göhring [Göh98] and others. See also Trakhtman's [Tra02] and Göhring's [Göh98] articles for more references.

Parallel Algorithms. In a series of articles, Ravikumar and Xiong study the problem of computing homing sequences on parallel computers. Ravikumar gives a deterministic $O(\sqrt{n}\log^2 n)$ time algorithm [Rav96], but it is reported not to be practical due to large communication costs. There is also a randomized algorithm requiring only $O(\log^2 n)$ time but $O(n^7)$ processors [RX96]. Although not practical, this is important as it implies that the problem belongs to the complexity class RNC. The same authors also introduced and implemented a practical randomized parallel algorithm requiring time essentially $O(n^3/k)$, where the number k of processors can be specified [RX97]. It is an open problem whether there are parallel algorithms for the synchronizing sequence problem, but in the special case of monotonic automata, Eppstein [Epp90] gives a randomized parallel algorithm. See also Ravikumar's survey of parallel algorithms for automata problems [Rav98].

Nondeterministic and Probabilistic Automata. The homing and synchronizing sequence problems become much harder for some generalizations of Mealy machines. As shown in Exercise 1.8, they are PSPACE-complete for nondeterministic automata, where $\delta(s, a)$ is a subset of S. This was noted by Rystsov [Rys92] as a consequence of the PSPACE-completeness theorem in another of

his papers [Rys83] (our Theorem 1.22). The generalization to nondeterministic automata can be made in several ways; Imreh and Steinby [IS99] study algebraic properties of three different formulations. For probabilistic automata, where $\delta(s, a)$ is a random distribution over S, Kfoury [Kfo70] showed that the problems are algorithmically unsolvable, by a reduction from the problem in a related article by Paterson [Pat70].

Related Problems. As a generalization of synchronizing sequences, many authors study the *rank* of a sequence [Rys92, Kar03, Pin78b]. The rank of a synchronizing sequence is 1, and for a general sequence x it is $|\delta(S, x)|$. Thus, Algorithm 2 decreases the rank by one every time it appends a merging sequence.

A problem related to synchronizing sequences is the *road coloring problem*. Here we are given a machine where the edges have not yet been labeled with input symbols, and asked whether there is a way of labeling so that the machine has a synchronizing sequence. This problem was introduced by Adler [AGW77] and studied in relation to synchronizing sequences, e.g., by Culik, Karhumäki and Kari [CKK02], and by Mateescu and Salomaa [MS99].

The parts orienting problem of Example 1.3 was studied in relation to automata by Natarajan [Nat86] and Eppstein [Epp90]. They have a slightly different setup, but the setup of our example was considered by Rao and Goldberg [RG95]. The field has been extensively studied for a long time, and many other approaches have been investigated.

Synchronizing sequences have been used to generate test cases for sequential circuits [RSP93, PJH92, CJSP93]. Here, the states of the machine is the set of all length k bitstrings. This set is too big to be explicitly enumerated, so both the state space and the transition function are specified implicitly. The algorithm of Section 1.3.2 becomes impractical in this setting as it uses too much memory. Instead, several authors considered algorithms that work directly with this symbolic representation of the state space and transition functions [RSP93, PJH92]. Pomeranz and Reddy [PR94] compute both preset and adaptive *homing* sequences for the same purpose, the advantage being that homing sequences exist more often than synchronizing do, and can be shorter since there is more information available to determine the final state.

1.6 Summary

We considered two fundamental and closely related testing problems for Mealy machines. In both cases, we look for a sequence of input symbols to apply to the machine so that the *final* state becomes known. A *synchronizing sequence* takes the machine to one and the same state no matter what the initial state was. A *homing sequence* produces output, so that one can learn the final state by looking at this output. These problems can be completely solved in polynomial time.

Homing sequences always exist if the machine is minimized. They have at most quadratic length and can be computed in cubic time, using the algorithm

in Section 1.3.1, which works by concatenating many separating sequences. Synchronizing sequences do not always exist, but the cubic time algorithm of Section 1.3.2 computes one if it exists, or reports that none exists, by concatenating many merging sequences. Synchronizing sequences have at most cubic length, but it is an open problem to determine if this can be improved to quadratic. Combining the methods of these two algorithms, we get the algorithm of Section 1.3.3 for computing homing sequences for general (non-minimized) machines.

It is practically important to compute as short sequences as possible. Unfortunately, the problems of finding the shortest possible homing or synchronizing sequences are NP-complete, so it is unlikely that no polynomial algorithm exists. This was proved in Section 1.4.1, and Section 1.3.5 gave exponential algorithms for both problems. Section 1.4.2 shows that only a small relaxation of the problem statement gives a PSPACE-complete problem.