CDM

Memoryless Machines

Klaus Sutner

Carnegie Mellon University

Spring 2024

The Idea

- Suppose you have a system that has finitely many internal states.
- The system reads a string of symbols from some external source.
- Initially, the system is set up in some special state q_0.
- Each input symbol causes a state transition:

  $p \xrightarrow{a} q$ move from state p to state q under input a

- When the last symbol has been read, the last state determines whether the system accepts the input or not.

Transition Systems

Definition

A transition system is a structure

$$\langle Q, \Sigma, \tau \rangle$$

where Q (states) and Σ (alphabet) are finite sets and $\delta \subseteq Q \times \Sigma \times Q$ (transition relation).

It is often best to think about a transition system as a directed graph whose edges are labeled in Σ, the diagram or transition graph of the transition system.

We do allow multiple edges and self-loops in the digram: $p \xrightarrow{a} q$ and $p \xrightarrow{b} q$ or $p \xrightarrow{a} p$ are all fine.

Runs

Suppose $\mathcal{T} = \langle Q, \Sigma, \tau \rangle$ is a transition system. Given a word $u = a_1 a_2 \ldots a_m$ over Σ, a run of $\mathcal{T}$ on u is an alternating sequence of states and letters

$$p_0, a_1, p_1, a_2, p_2, \ldots, p_{m-1}, a_m, p_m$$

such that $p_{i-1} \xrightarrow{a_i} p_i$ is a valid transition for all i. p_0 is the source of the run and p_m its target, and $m \geq 0$ its length. So a run is just a path in a labeled digraph.

Traces

Given a run

$$\pi = p_0, a_1, p_1, a_2, p_2, \ldots, p_{m-1}, a_m, p_m$$

of an automaton, the corresponding sequence of labels

$$a_1 a_2 \ldots a_{m-1} a_m \in \Sigma^\star$$

is referred to as the trace or label of the run.

Every run has exactly one associated trace, but the same trace may have several runs, even if we fix the source and target states (ambiguous automata).

Special Transition Systems

Definition

A transition system is complete if for all $p \in Q$ and $a \in \Sigma$ there is some $q \in Q$ such that

$$p \xrightarrow{a} q$$

is a transition.

In other words, the system cannot get stuck in any state, we always can consume all input symbols.

Definition

A transition system is deterministic if for all $p, q, q' \in Q$ and $a \in \Sigma$

$$p \xrightarrow{a} q, p \xrightarrow{a} q' \quad \text{implies} \quad q = q'$$

Thus, a deterministic system can have at most one run from a given state for any input.

Finite State Machines

Definition

A finite state machine or finite automaton is a structure

$$\mathcal{A} = \langle \mathcal{T}; \mathsf{acc} \rangle$$

where $\mathcal{T} = \langle Q, \Sigma, \delta \rangle$ is a transition system and acc is an acceptance condition.

The vanilla acceptance condition is comprised of a collection of initial states $I \subseteq Q$ and a collection of final states $F \subseteq Q$. The idea is that $\mathcal{A}$ accepts some input $x \in \Sigma^\star$ if there is a run from a state in I to a state in F with label x in $\mathcal{T}$.

The (acceptance) language $\mathcal{L}(\mathcal{A})$ of the automaton $\mathcal{A}$ is the set of all words accepted by the automaton.

Semiautomata

In a sense the most basic type of a FSM is a semiautomaton: we think of all states as being initial and final.

$$\mathcal{A} = \langle Q, \Sigma, \tau; Q, Q \rangle$$

The language of a semiautomaton corresponds to the labels of all possible paths in the diagram.

For example, these languages are always closed under factors:

$$uv \in L \text{ implies } u \in L, v \in L$$

DFAs

Combining the previous acceptance condition with completeness and determinism produces a particularly useful type of automaton.

Definition

A deterministic finite automaton (DFA) is a structure

$$\mathcal{A} = \langle \mathcal{T}; q_0, F \rangle$$

where $\mathcal{T} = \langle Q, \Sigma, \delta \rangle$ is a deterministic and complete transition system and $q_0 \in Q$, $F \subseteq Q$ is the standard acceptance condition.

It is straightforward to see that a DFA[†] has exactly one trace (or run) on any possible input word.

Since we use the standard acceptance condition, a run is accepting if it leads from q_0 to some $q \in F$ (a slight asymmetry).

[†]These machines should be called DCFA or CDFA, but that ship sailed a long time ago.

Partial DFA

Definition

A partial DFA (PDFA) is a structure

$$\mathcal{A} = \langle \mathcal{T}; q_0, F \rangle$$

where $\mathcal{T} = \langle Q, \Sigma, \delta \rangle$ is a deterministic transition system and $q_0 \in Q$, $F \subseteq Q$ is the standard acceptance condition.

Algorithmically, PDFA are in many ways better behaved than DFAa, since they may have fewer useless transitions.

If need be, we can always complete a PDFA to a DFA by adding another state $\perp$ and fill in missing transitions to $p \xrightarrow{a} \perp$, plus $\perp \xrightarrow{a} \perp$.

Useless States

Definition

Let $\mathcal{A} = \langle Q, \Sigma, \tau; I, F \rangle$ be a FSM. A state $p \in Q$ is

- accessible if it is reachable from I in the diagram
- coaccessible if it is coreachable from F in the diagram
- useful if it is both accessible and coaccessible; useless otherwise
- trap if all transitions with source p also have target p
- sink if it is a trap and is not final..

The machine is accessible, coaccessible, trim if all its states are accessible, coaccessible, useful, respectively.

Hence we can compute the trim version $\operatorname{trim} \mathcal{A}$ in linear time using standard graph algorithms.

Note that there is an issue with DFA: the accessible part $\operatorname{acc} \mathcal{A}$ is still a DFA, but $\operatorname{trim} \mathcal{A}$ may be just a partial DFA: all the sinks have to be removed.

Recognizable Languages

Definition

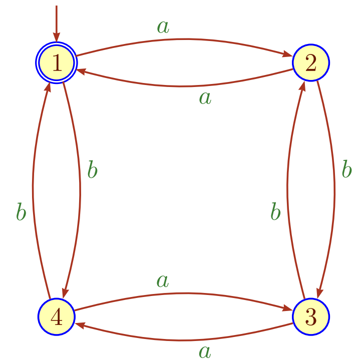
A language $L \subseteq \Sigma^\star$ is recognizable or regular* if there is a DFA M that accepts L: $\mathcal{L}(M) = L$.

Thus a recognizable language has a simple, finite description in terms of a particular type of finite state machine. As we will see, one can manipulate the languages in many ways by manipulating the corresponding machines.

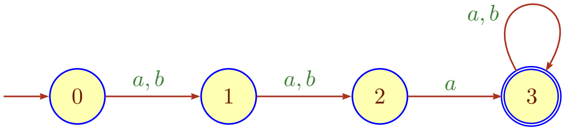
In a sense, finite state machines are the bottom of the barrel when it comes to computation, at least when infinitely many inputs are involved. They are (fast) linear time and constant space.

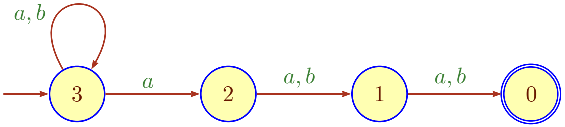
*Regular is more popular in the US, but hopelessly overloaded.

Example: Even/Even



Example: kth-Letter



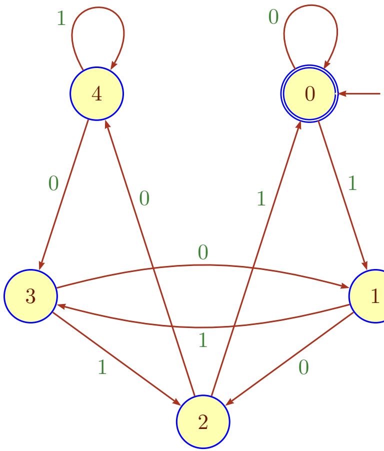


Example: $k = -3$

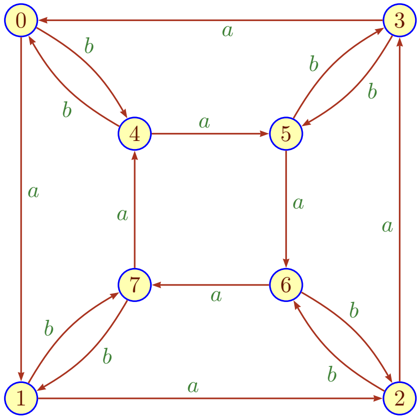
$$\begin{tikzcd}
 & aab \arrow[dr] \arrow[rrr, bend left] & & & abb \arrow[dr] \arrow[dd] & \\
aaa \arrow[loop left] \arrow[ur] & & aba \arrow[r, bend left] \arrow[dl] & bab \arrow[l, bend left] \arrow[ur] & & bbb \arrow[loop right] \arrow[dl] \\
 & baa \arrow[ul] \arrow[uu] & & & bba \arrow[ul] \arrow[lll, bend left] &
\end{tikzcd}$$

Transitions are $xyz \xrightarrow{s} yzs$.

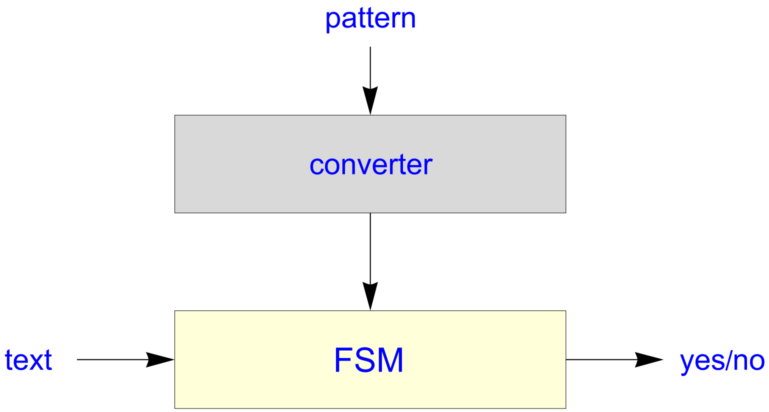
Example: Divisibility



Example: Dihedral Group



A Killer App



Basic Results

Theorem

Recognizable languages are closed under:

1. *intersection, union, complement*
2. *concatenation, Kleene star, reversal*
3. *homomorphisms and inverse homomorphisms*

The key point here is that all these results are perfectly constructive, there are nice algorithms to build the corresponding machines.

Example: Union and Intersection

Suppose we have two FMSs over Σ: $\mathcal{A}_i = \langle Q_i, \Sigma, \tau_i; I_i, F_i \rangle$. We may safely assume that the state sets are disjoint. There are two simple operations the combine the computations of both machines:

- Sum Automaton

$$\mathcal{A}_1 + \mathcal{A}_2 = \langle Q_1 \cup Q_2, \tau_1 \cup \tau_2; I_1 \cup I_2, F_1 \cup F_2 \rangle.$$

- Cartesian Product Automaton

$$\mathcal{A}_1 \times \mathcal{A}_2 = \langle Q_1 \times Q_2, \tau_1 \times \tau_2; I_1 \times I_2, F_1 \times F_2 \rangle$$

In the product construction

$$((p,q), a, (p', q')) \in \tau_1 \times \tau_2 \iff \tau_1(p, a, p') \wedge \tau_2(q, a, q')$$

Computations

Clearly, the computations of $\mathcal{A}_1 + \mathcal{A}_2$ are exactly the union of the computations of $\mathcal{A}_1$ and $\mathcal{A}_2$.

The size of the sum automaton is linear in the size of the components.

The computations of $\mathcal{A}_1 \times \mathcal{A}_2$ are the computations of $\mathcal{A}_1$ combined with the computations of $\mathcal{A}_2$ provided both have the same label: essentially, we are running both machines in parallel.

A real implementation will only construct the accessible part, but still, the size of $\mathcal{A}_1 \times \mathcal{A}_2$ is potentially quadratic in the sizes of $\mathcal{A}_1$ and $\mathcal{A}_2$. This causes problems if a product machine construction is used repeatedly.

The Languages

By our choice of acceptance condition we have

$$\mathcal{L}(\mathcal{A}_1 + \mathcal{A}_2) = L_1 \cup L_2$$
$$\mathcal{L}(\mathcal{A}_1 \times \mathcal{A}_2) = L_1 \cap L_2$$

By changing the final states in the product, we can also get union:

union	$F = F_1 \times Q_2 \cup Q_1 \times F_2$
intersection	$F = F_1 \times F_2$

Determinization

Theorem

Languages recognized by nondeterministic FSM are all recognizable by a DFA.

Suppose we have a nondeterministic machine $\mathcal{A} = \langle Q, \Sigma, \tau; I, F \rangle$. Define a DFA $\mathcal{B} = \langle Q', \Sigma, \delta; q_0, F' \rangle$ by

- $Q' = \mathfrak{P}(Q)$
- $q_0 = I$
- $F' = \{\, P \subseteq Q \mid F \cap P \neq \emptyset \,\}$
- $\delta(P, a) = \{\, q \in Q \mid \exists\, p \in P\; \tau(p, a, q) \,\}$

Theorem (Minimal DFA)

For every recognizable language L, there is exactly one DFA (up to isomorphism) that recognizes L and has the least number of states of all such DFA.

This provides a (bad) way to check whether two DFA accept the same language: construct the respective minimal DFA and check that they are isomorphic.

Determinization is the only way to produce complements, alas, it may require exponential time and space.

State Complexity of Operations

	DFA	NFA
intersection	mn	mn
union	mn	$m+n$
concatenation	$(m-1)2^n-1$	$m+n$
Kleene star	$3 \cdot 2^{n-2}$	$n+1$
reversal	2^n	n
complement	n	2^n

Worst case blow-up starting from machine(s) of size m, n and applying the corresponding operation (accessible part only).

Note that we are only dealing the state complexity, not transition complexity (which is arguably a better measure for NFAs).

Finite State Machines with Output

A transducer is a finite state machine that reads some input string x and returns an output string y. In general, transducers are nondeterministic: a single input is can be associated with multiple potential outputs. We will focus on the functional case.

The maps $\Sigma^\star \to \Sigma^\star$ described by a transducer are called transductions.

In the most restricted case, the transducer generates exactly one output symbol for each input symbol, so these are length-preserving transductions. Moreover, we will insist that our machines are deterministic: there is exactly one output for each input.

Warning: For the general theory of transducers this is much too narrow, we really need to confront nondeterministic and non-length-preserving machines.

Mealy Transducers

Definition

A Mealy transducer or Mealy machine is a deterministic and complete transition system $\mathcal{A} = \langle Q, \Sigma \times \Sigma, \tau \rangle$.

Since we require the transition system to be both deterministic and complete (just like a DFA), we can think of the transitions as being given by a transition function

$$\tau : Q \times \Sigma \longrightarrow \Sigma \times Q$$

We can copy&paste all the definitions from the acceptor case: the only difference is that the language of a transducer is a map rather than a language.

In general, transducers and Mealy machines have initial and final states, but that will not be necessary for our purposes.

Alternative Definition

Instead of using a single transition function $\tau : Q \times \Sigma \to \Sigma \times Q$ it is sometimes more convenient to split things into two functions

$\delta : Q \times \Sigma \to Q$	state transitions
$\rho : Q \times \Sigma \to \Sigma$	output

In particular the curried versions of these functions

$$\delta_a : Q \to Q \qquad a \in \Sigma$$
$$\rho_p : \Sigma \to \Sigma \qquad p \in Q$$

are often convenient.

Output Modules

One can add initial states and final states, but in our definition there are none (Sam Eilenberg referred to this type of machine as output modules).

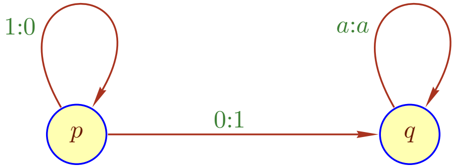
For any state p in a Mealy machine we get a transduction

$$\underline{p} : \Sigma^\star \to \Sigma^\star$$

by starting the computation at state p.

Note that this is really a function since Mealy machines are deterministic.

Successor



We can describe the transductions defined by this machine as follows:

$$\begin{aligned} \underline{p}\,(0x) &= 1\,\underline{q}\,(x) \\ \underline{p}\,(1x) &= 0\,\underline{p}\,(x) \\ \underline{q}\,(x) &= x \end{aligned}$$

On k-bit strings[†], $\underline{p}$ is the successor modulo 2^k.

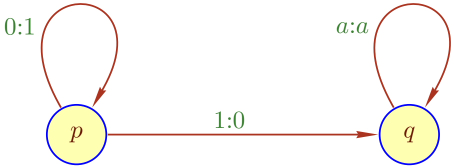
In the literature, this is called the "adding machine."

[†]On infinite strings we get the true successor function, but on 2-adic numbers.

Predecessor

The last machine has a special property: its transduction is a bijection on $\mathbf{2}^\star$.

Moreover, the corresponding transducer is obtained by simply flipping the labels:

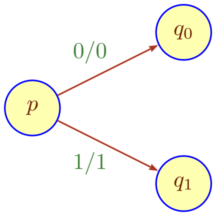


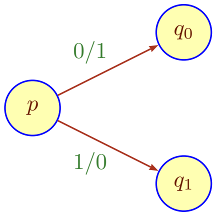
Invertible Mealy Automata

Definition

A Mealy transducer is invertible if, for every state p, the output maps ρ_p are permutations of Σ.

In particular for $\Sigma = \mathbf{2}$ there are only two types of states: copy and toggle.





We will focus on invertible transducers from now on.

Transduction Semigroups and Groups

By moving the initial state in an invertible Mealy machine $\mathcal{A}$ around, we get a collection of transductions $\underline{p}$, $p \in Q$ that are all permutations of $\Sigma^\star$. Define

$$\begin{aligned} \mathsf{Sgrp}(\mathcal{A}) &= \text{ the semigroup generated by the } \underline{p} \\ \mathsf{Grp}(\mathcal{A}) &= \text{ the group generated by the } \underline{p} \end{aligned}$$

For the semigroup we take all possible compositions of the basic transductions, producing an automaton semigroup.

For the group we add the inverse transductions of all the basic ones. producing an automaton group.

There is a very similar concept of automatic semigroup/group, but that is slightly different. And sometimes there is confusion about the two ideas.

Semigroups are a bit of a mess, but for groups we have a huge body of results, it seems plausible that we might be able to analyze these groups.

But Why?

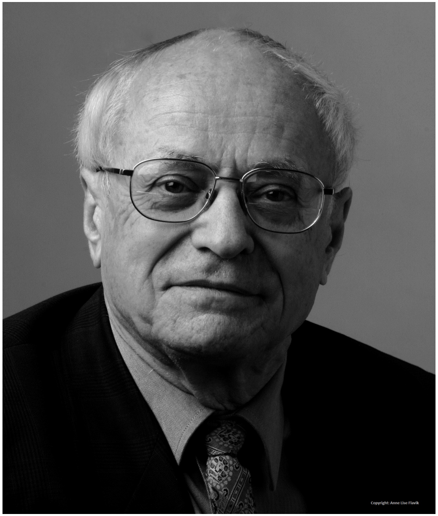
Why should anyone care about these semigroups/groups?

Two reasons:

- Automatic groups have become the goto source for examples and counterexamples in group theory. The key is that some enormously complicated groups have descriptions in terms of automata with just a handful of states.

- Transducers operating on binary and 2-adic numbers are quite useful in understanding digital circuits. Take a look at Vuillemin.

Jean-Pierre Serre



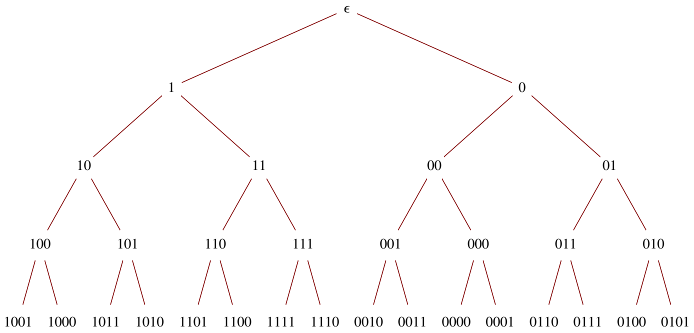
Fields Medal, Abel Prize, Steele Prize, Wolf Prize

Member Bourbaki

Arbres, Amalgames, SL_2 (1977)

Trees?

One can think of the transductions of our Mealy machines as automorphisms of the infinite binary tree:



More later.