2. Finite-state (string) automata

First some standard but important stage-setting . . .

(1) For any set Σ, we define Σ^* as the smallest set such that:
- $\varepsilon \in \Sigma^*$, and
- if $x \in \Sigma$ and $u \in \Sigma^*$ then $(x{:}u) \in \Sigma^*$.

We often call Σ an *alphabet*, call the members of Σ *symbols*, and call the members of Σ^* *strings*.

So if our alphabet is $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, then some of the strings in Σ^* are shown in (2). But we usually abbreviate these as in (3).

(2)
- $\mathtt{b}{:}(\mathtt{a}{:}\varepsilon)$
- $\mathtt{c}{:}(\mathtt{c}{:}(\mathtt{c}{:}\varepsilon))$
- $\mathtt{a}{:}\varepsilon$
- ε

(3)
- ba
- ccc
- a
- ε

I'll generally use x, y and z for individual symbols of an alphabet Σ, and use u, v and w for strings in Σ^*.

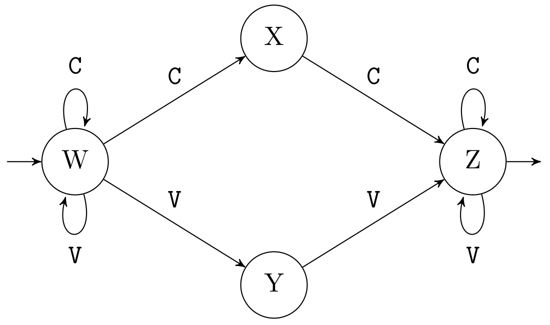
1 Finite-state automata, informally

Below are graphical representations of some finite-state automata (FSAs).

- The circles represent *states*.
- The *initial* states are indicated by an "arrow from nowhere".
- The *final* or *ending* states are indicated by an "arrow to nowhere".

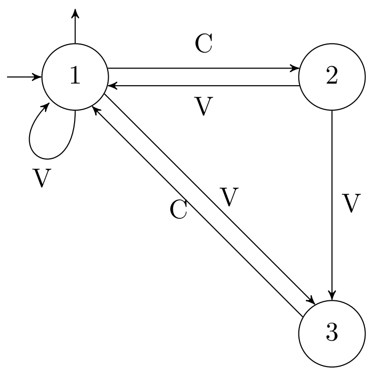
The FSA in (4) generates the subset of $\{\mathtt{C}, \mathtt{V}\}^*$ consisting of all and only strings that contain either two adjacent Cs or two adjacent Vs (or both).

(4)



If we think of state 1 as indicating syllable boundaries, then the FSA in (5) generates sequences of syllables of the form (C)V(C). The string VCV, for example, can be generated via two different paths, 1-1-2-1 and 1-3-1-1, corresponding to different syllabifications.

(5)



2 Formal definition of an FSA

(6) A finite-state automaton (FSA) is a five-tuple $(Q, \Sigma, I, F, \Delta)$ where:

- Q is a finite set of states;
- Σ, the alphabet, is a finite set of symbols;
- $I \subseteq Q$ is the set of initial states;
- $F \subseteq Q$ is the set of ending states; and
- $\Delta \subseteq Q \times \Sigma \times Q$ is the set of transitions.

So strictly speaking, (4) is a picture of the following mathematical object:

(7)
$$\big(\ \{\mathrm{W}, \mathrm{X}, \mathrm{Y}, \mathrm{Z}\},\quad \{\mathtt{C}, \mathtt{V}\},\quad \{\mathrm{W}\},\quad \{\mathrm{Z}\},\quad \{(\mathrm{W}, \mathtt{C}, \mathrm{W}), (\mathrm{W}, \mathtt{C}, \mathrm{X}), (\mathrm{W}, \mathtt{V}, \mathrm{W}), (\mathrm{W}, \mathtt{V}, \mathrm{Y}), (\mathrm{X}, \mathtt{C}, \mathrm{Z}), (\mathrm{Y}, \mathtt{V}, \mathrm{Z}), (\mathrm{Z}, \mathtt{C}, \mathrm{Z}), (\mathrm{Z}, \mathtt{V}, \mathrm{Z})\}\ \big)$$

Now let's try to say more precisely what it means for an automaton $M = (Q, \Sigma, I, F, \Delta)$ to generate/accept a string.

(8) For M to generate a string of three symbols, say $x_1x_2x_3$, there must be four states q_0, q_1, q_2, and q_3 such that

- $q_0 \in I$, and
- $(q_0, x_1, q_1) \in \Delta$, and
- $(q_1, x_2, q_2) \in \Delta$, and
- $(q_2, x_3, q_3) \in \Delta$, and
- $q_3 \in F$.

To take a concrete example:

(9) The automaton in (4)/(7) generates the string `VCCVC` because we can choose q_0, q_1, q_2, q_3, q_4 and q_5 to be the states W, W, X, Z, Z and Z (respectively), and then it's true that:

- $\mathrm{W} \in I$, and
- $(\mathrm{W}, \mathtt{V}, \mathrm{W}) \in \Delta$, and
- $(\mathrm{W}, \mathtt{C}, \mathrm{X}) \in \Delta$, and
- $(\mathrm{X}, \mathtt{C}, \mathrm{Z}) \in \Delta$, and
- $(\mathrm{Z}, \mathtt{V}, \mathrm{Z}) \in \Delta$, and
- $(\mathrm{Z}, \mathtt{C}, \mathrm{Z}) \in \Delta$, and
- $\mathrm{Z} \in F$.

We'll write $\mathcal{L}(M)$ for the set of strings generated by an FSA M. So stated roughly, the important idea is:

(10) $w \in \mathcal{L}(M)$

$$\iff \bigvee_{\text{all possible paths } p} \Big[\text{string } w \text{ can be generated by path } p\Big]$$

$$\iff \bigvee_{\text{all possible paths } p} \Big[\bigwedge_{\text{all steps } s \text{ in } p} \big[\text{step } s \text{ is allowed and generates the appropriate part of } w\big]\Big]$$

It's handy to write $I(q_0)$ in place of $q_0 \in I$, and likewise for F and Δ. Then one way to make (10) precise is:

(11) $x_1 x_2 \dots x_n \in \mathcal{L}(M)$

$$\iff \bigvee_{q_0 \in Q} \bigvee_{q_1 \in Q} \cdots \bigvee_{q_{n-1} \in Q} \bigvee_{q_n \in Q} \Big[I(q_0) \wedge \Delta(q_0, x_1, q_1) \wedge \cdots \wedge \Delta(q_{n-1}, x_n, q_n) \wedge F(q_n)\Big]$$

But it's convenient — both for computational efficiency, and as an aid to understanding — to break this down in a couple of different ways, making use of *recursion on strings*.

3 Recursive calculations: forward and backward values

3.1 Forward values

For any FSA M there's a two-place predicate fwd_M, relating states to strings in an important way:

(12) $\text{fwd}_M(w)(q)$ is true iff there's a path through M from some initial state to the state q, emitting the string w

Given a way to work out $\text{fwd}_M(w)(q)$ for any string and any state, we can easily use this to check for membership in $\mathcal{L}(M)$:

(13) $$w \in \mathcal{L}(M) \iff \bigvee_{q_n \in Q} \Big[\text{fwd}_M(w)(q_n) \wedge F(q_n)\Big]$$

We can represent the predicate fwd_M in a table. Each column shows fwd_M values for the *entire prefix* consisting of the header symbols to its *left*. The first column shows values for the empty string.

Here's the table of forward values for the string `CVCCV` using the FSA in (4):

(14)

State		`C`	`V`	`C`	`C`	`V`
W	1	1	1	1	1	1
X	0	1	0	1	1	0
Y	0	0	1	0	0	1
Z	0	0	0	0	1	1

Important things to note:

- Filling in the values in the leftmost column is easy: this column just says which states are initial states.
- In order to fill in any other column, you only need to know
  - the values in the column immediately to its left, and
  - the symbol immediately to its left.

More generally, this means that:

(15) The fwd_M values for a non-empty string $x_1 \dots x_n$ depend only on
- the fwd_M values for the string $x_1 \dots x_{n-1}$, and
- the symbol x_n.

This means that we can give a recursive definition of fwd_M:

(16)
$$\text{fwd}_M(\varepsilon)(q) = I(q)$$
$$\text{fwd}_M(x_1 \dots x_n)(q) = \bigvee_{q_{n-1} \in Q} \Big[\text{fwd}_M(x_1 \dots x_{n-1})(q_{n-1}) \wedge \Delta(q_{n-1}, x_n, q)\Big]$$

This suggests a natural and efficient algorithm for calculating these values: write out the table, start by filling in the leftmost column, and then fill in other columns from left to right. This is where the name "forward" comes from.

3.2 Backward values

We can do all the same things, flipped around in the other direction.

For any FSA M there's a two-place predicate bwd_M, relating states to strings in an important way:

(17) $\text{bwd}_M(w)(q)$ is true iff there's a path through M from the state q to some ending state, emitting the string w

Given a way to work out $\text{bwd}_M(w)(q)$ for any string and any state, we can easily use this to check for membership in $\mathcal{L}(M)$:

(18)
$$w \in \mathcal{L}(M) \iff \bigvee_{q_0 \in Q} \Big[I(q_0) \wedge \text{bwd}_M(w)(q_0)\Big]$$

We can represent the predicate bwd_M in a table. Each column shows bwd_M values for the *entire suffix* consisting of the header symbols to its *right*. The last column shows values for the empty string.

Here's the table of backward values for the string `CVCCV` using the FSA in (4):

(19)

State		`C`	`V`	`C`	`C`	`V`
W	1	1	1	0	0	0
X	1	0	1	1	0	0
Y	0	1	0	0	1	0
Z	1	1	1	1	1	1

In this case, filling in the last column is easy, and each other column can be filled in simply by looking at the values immediately to its right.

(20) The bwd_M values for a non-empty string $x_1 \dots x_n$ depend only on
- the bwd_M values for the string $x_2 \dots x_n$, and
- the symbol x_1.

So bwd_M can also be defined recursively.

(21)
$$\text{bwd}_M(\varepsilon)(q) = F(q)$$
$$\text{bwd}_M(x_1 \dots x_n)(q) = \bigvee_{q_1 \in Q} \Big[\Delta(q, x_1, q_1) \wedge \text{bwd}_M(x_2 \dots x_n)(q_1)\Big]$$

3.3 Forward values and backward values together

Now we can say something beautiful:

(22) $$uv \in \mathcal{L}(M) \iff \bigvee_{q \in Q} \Big[\text{fwd}_M(u)(q) \wedge \text{bwd}_M(v)(q) \Big]$$

And in fact (13) and (18) are just special cases of (22), with u or v chosen to be the empty string:

(23) $$w \in \mathcal{L}(M) \iff \bigvee_{q \in Q} \Big[\text{fwd}_M(w)(q) \wedge \text{bwd}_M(\varepsilon)(q) \Big] \iff \bigvee_{q \in Q} \Big[\text{fwd}_M(w)(q) \wedge F(q) \Big]$$

(24) $$w \in \mathcal{L}(M) \iff \bigvee_{q \in Q} \Big[\text{fwd}_M(\varepsilon)(q) \wedge \text{bwd}_M(w)(q) \Big] \iff \bigvee_{q \in Q} \Big[I(q) \wedge \text{bwd}_M(w)(q) \Big]$$

4 Interchangeable subexpressions

Now forget about FSAs for a moment, and just consider sets of strings "out of the blue". We'll connect things back to FSAs shortly, in section 5.

(25) Given some stringset $L \subseteq \Sigma^*$, the L-remainders of a string u are all the strings v such that $uv \in L$. I'll write $\text{rem}_L(u)$ for the set of L-remainders of u, so we can write this definition in symbols as: $\text{rem}_L(u) = \{v \mid v \in \Sigma^*, uv \in L\}$.

Roughly, $\text{rem}_L(u)$ gives us a handle on "all the things we're still allowed to do, if we've done u so far".

(26) If $L_1 = \{\texttt{cat}, \texttt{cap}, \texttt{cape}, \texttt{cut}, \texttt{cup}, \texttt{dog}\}$, then:
- a. $\text{rem}_{L_1}(\texttt{ca}) = \{\texttt{t}, \texttt{p}, \texttt{pe}\}$
- b. $\text{rem}_{L_1}(\texttt{c}) = \{\texttt{at}, \texttt{ap}, \texttt{ape}, \texttt{ut}, \texttt{up}\}$
- c. $\text{rem}_{L_1}(\texttt{cap}) = \{\varepsilon, \texttt{e}\}$
- d. $\text{rem}_{L_1}(\texttt{d}) = \{\texttt{og}\}$

(27) If $L_2 = \{\texttt{ad}, \texttt{add}, \texttt{baa}, \texttt{bad}, \texttt{cab}, \texttt{cad}, \texttt{dab}, \texttt{dad}\}$, then:
- a. $\text{rem}_{L_2}(\texttt{a}) = \{\texttt{d}, \texttt{dd}\}$
- b. $\text{rem}_{L_2}(\texttt{ad}) = \{\varepsilon, \texttt{d}\}$
- c. $\text{rem}_{L_2}(\texttt{ca}) = \text{rem}_{L_2}(\texttt{da}) = \{\texttt{b}, \texttt{d}\}$

When we notice that $\text{rem}_{L_2}(\texttt{ca}) = \text{rem}_{L_2}(\texttt{da})$, this tells us something useful about how we can go about designing a grammar to generate the stringset L_2: such a grammar *doesn't need to care about* the distinction between starting with `ca` and starting with `da`, because for any string v that you choose, $\texttt{ca}v$ and $\texttt{da}v$ will either both be in L_2 or both not be in L_2. An initial `ca` and an initial `da` are *interchangeable subexpressions*.

(28) Given a stringset $L \subseteq \Sigma^*$ and two strings $u \in \Sigma^*$ and $v \in \Sigma^*$, we define a relation $\equiv_L$ such that: $u \equiv_L v$ iff $\text{rem}_L(u) = \text{rem}_L(v)$.

Some slightly more linguistics-ish examples:

(29) Suppose that $\Sigma = \{\texttt{C}, \texttt{V}\}$, and L is the subset of Σ^* containing all strings that contain at least one `V`. Then:
- a. $\texttt{C} \equiv_L \texttt{CC}$, because both can only be followed by strings that fulfill the requirement for a `V`.

b. `VC` $\equiv_L$ `CV`, because both can be followed by anything at all.

c. So two strings are L-equivalent iff they either both do or both don't contain a `V`.

(30) Suppose that $\Sigma = \{\texttt{C}, \texttt{V}\}$, and L is the subset of Σ^* containing all strings that have two adjacent `C`s or two adjacent `V`s (or both). Then

a. `C` $\equiv_L$ `CVC` $\equiv_L$ `CVCVC`, because these all require remainders that have two adjacent `C`s *or* two adjacent `V`s *or* an initial `C`.

b. `V` $\equiv_L$ `VCV` $\equiv_L$ `VCVCV`, because these all require remainders that have two adjacent `C`s *or* two adjacent `V`s *or* an initial `V`.

c. `CC` $\equiv_L$ `VCVCVVCVC`

(31) Suppose that $\Sigma = \{\texttt{C}, \texttt{V}\}$, and L is the subset of Σ^* containing all strings that *do not* have two adjacent occurrences of `V`. Then:

a. `CCCC` $\equiv_L$ `VC`, because both can be followed by anything without adjacent `V`s.

b. `CCV` $\equiv_L$ `V`, because both can be followed by anything without adjacent `V`s that does not begin with `V`.

c. `CCV` $\not\equiv_L$ `CCC`, because only the latter can be followed by `VC`.

d. In fact: two strings are L-equivalent iff they end with the same symbol!

(32) Suppose that Σ is the set of English words, and L is the set of all grammatical English word-sequences. Then (probably?):

a. `John` $\equiv_L$ `the brown furry rabbit`

b. `John` $\equiv_L$ `Mary thinks that John`

c. `John` $\not\equiv_L$ `the fact that John`

5 The Myhill-Nerode Theorem

Recall that $\text{fwd}_M(u)(q)$ is a boolean, true or false; so we can think of $\text{fwd}_M(u)$ as a *set of states*, namely all those states reachable from an initial state of M by taking transitions that produce the string u.

Now here's the important connection:

(33) For any FSA $M = (Q, \Sigma, I, F, \Delta)$ and for any two strings $u \in \Sigma^*$ and $v \in \Sigma^*$, if $\text{fwd}_M(u) = \text{fwd}_M(v)$ then $u \equiv_{\mathcal{L}(M)} v$.

Given any particular stringset L, we can think of the relation $\equiv_L$ as sorting out all possible strings into buckets (or "equivalence classes").

- Two strings belong in the same bucket iff they are equivalent prefixes.
- So what (33) says is that for an FSA to generate L it must be arranged so that fwd only maps two strings to the same state-sets if those two strings are equivalent prefixes; the machine can ignore distinctions between bucket-mates, but only between bucket-mates.
- This idea of ignoring at least some distinctions is exactly what makes a grammar different from a list of strings.

And now we can put our finger on the capacities/limitations of finite-state automata.

(34) **The Myhill-Nerode Theorem:** Given a particular stringset L, there is an FSA that generates L iff the relation $\equiv_L$ sorts strings into only finitely-many buckets.

Why is this, exactly?

5.1 Why do FSAs make only finitely-many distinctions?

Well, if we have a particular FSA whose set of states is Q, then there are only finitely many distinct subsets of Q that fwd_M can map strings to; specifically, there are $2^{|Q|}$ of them. So there are only finitely-many "candidate forward sets", meaning that the FSA is necessarily making only those finitely-many distinctions.

Having noticed this, it's very easy to convince ourselves that no FSA can generate the stringset $L = \{\mathtt{a}^n\mathtt{b}^n \mid n > 0\}$, for example.

- Notice that $\mathtt{a} \not\equiv_L \mathtt{aa}$, and $\mathtt{aa} \not\equiv_L \mathtt{aaa}$, and so on.
- In fact any string of `a`s is non-equivalent to each different-length string of `a`s — so this stringset sorts strings into infinitely many buckets, one bucket for each length.
- So there is no way for an FSA M to be set up such that $\text{fwd}_M(\mathtt{a}^j) \neq \text{fwd}_M(\mathtt{a}^k)$ whenever $j \neq k$. Any FSA will incorrectly collapse the distinction between two such strings of '`a`'s.

5.2 Why can any finitely-many distinctions be captured with an FSA?

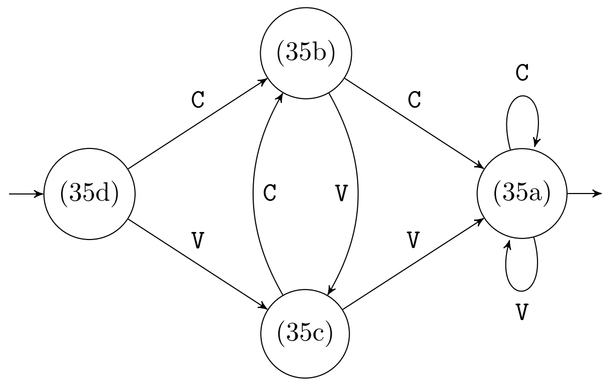
On the other hand, if we have a particular stringset L whose equivalence relation $\equiv_L$ makes only finitely-many distinctions, then there is a straightforward way to construct a minimal FSA whose states track exactly those distinctions.

Consider again the stringset from (30), consisting of all strings with either two adjacent `C`s or two adjacent `V`s (or both). This stringset's equivalence relation sorts strings into four buckets:

(35) a. a bucket containing strings that have either two adjacent `C`s or two adjacent `V`s;
b. a bucket containing strings that don't have two adjacent `C`s or `V`s, but end in `C`;
c. a bucket containing strings that don't have two adjacent `C`s or `V`s, but end in `V`;
d. a bucket containing only the empty string.

We can mechanically construct an appropriate FSA which has one state corresponding to each bucket. The crucial idea here is that if $u \equiv_L v$, then $ux \equiv_L vx$ for any $x \in \Sigma$, i.e. adding a symbol at the end can't "break" an equivalence; and similarly, for any FSA M, if $\text{fwd}_M(u) = \text{fwd}_M(v)$ then $\text{fwd}_M(ux) = \text{fwd}_M(vx)$.

(36)



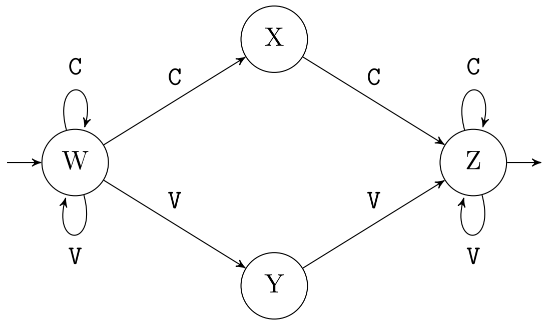
5.3 One loose end: determinization and minimization

The strategy illustrated in (36) produces a specific kind of automaton, a *deterministic automaton*. In a deterministic automaton, there are never two arcs leading out of the same state that are labeled with the same symbol; each string corresponds to at most one path through the states, and so fwd only ever produces singleton sets or the empty set.

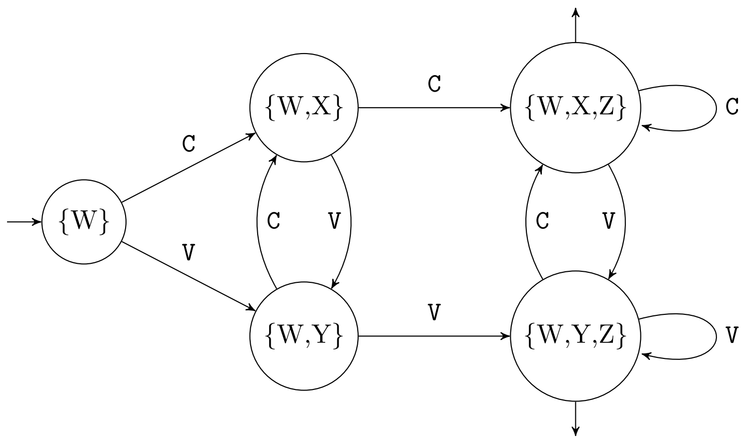
Any FSA M can be converted into an equivalent deterministic one M', by setting up the states of M' to correspond to sets of the states of M. Then, for any string u, the one state in $\text{fwd}_{M'}(u)$ will be the one corresponding to the set $\mathrm{fwd}_M(u)$. Working out the transitions of the new automaton M' for a symbol x amounts to working out how to calculate $\mathrm{fwd}_M(ux)$ from $\mathrm{fwd}_M(u)$, which we saw before.

Applying this procedure to the FSA in (4), which generates the stringset in (30), produces the new FSA in (37).

(4)



(37)



To see the connection, compare (37) with the table of forward values for the string `CVCCV` that we saw earlier, using the original FSA in (4).

(14)

State		`C`	`V`	`C`	`C`	`V`
W	1	1	1	1	1	1
X	0	1	0	1	1	0
Y	0	0	1	0	0	1
Z	0	0	0	0	1	1

It turns out that the two states $\{W, X, Z\}$ and $\{W, Y, Z\}$ in (37) "do the same thing": in either case, it's possible to end, it's possible to transition to state $\{W, X, Z\}$ on a `C`, and it's possible to transition to state $\{W, Y, Z\}$ on a `V` (and that's all). Collapsing these two states will produce the minimal FSA in (36).