Week 3: Introducing finite-state automata

First some standard stage-setting definitions:

(1) For any set Σ, we define Σ^* as the smallest set such that:
- $\epsilon \in \Sigma^*$, 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*.

(2) For any two strings $u \in \Sigma^*$ and $v \in \Sigma^*$, we define $u +\!\!+\, v$ as follows:
- $\epsilon +\!\!+\, v = v$
- $(x{:}w) +\!\!+\, v = x{:}(w +\!\!+\, v)$

Although these definitions provide the "official" notation, I'll sometimes be slightly lazy and abbreviate '$x{:}\epsilon$' as 'x', and abbreviate both '$s{:}t$' and '$s +\!\!+\, t$' as just 'st' in cases where it should be clear what's intended.

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^*. This should help to clarify whether a ':' or a '$+\!\!+$' has been left out.

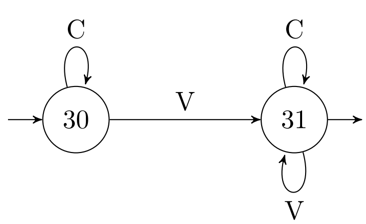
1 Finite-state automata, informally

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 *accepting* states are indicated by an "arrow to nowhere".

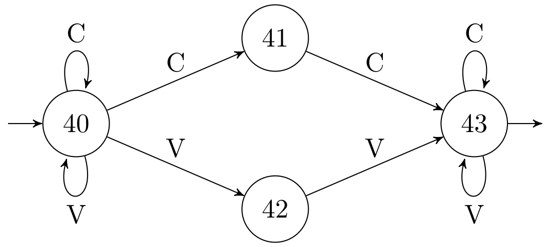
The FSA in (3) generates the subset of $\{\mathrm{C}, \mathrm{V}\}^*$ consisting of all and only strings that have at least one occurrence of 'V'.

(3)

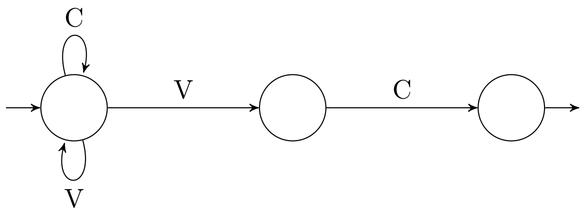


The FSA in (4) generates the subset of $\{\mathrm{C}, \mathrm{V}\}^*$ consisting of all and only strings that contain either two adjacent 'C's or two adjacent 'V's (or both).

(4)

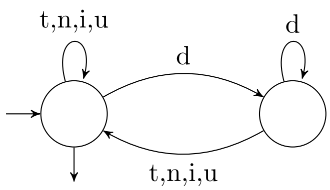


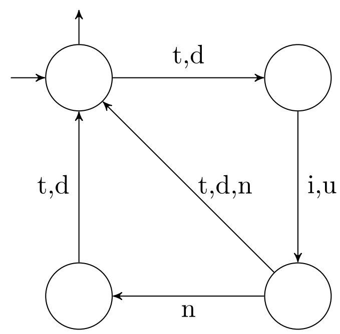
The FSA in (5) generates the subset of $\{C, V\}^*$ consisting of all and only strings which end in 'VC'.

(5)


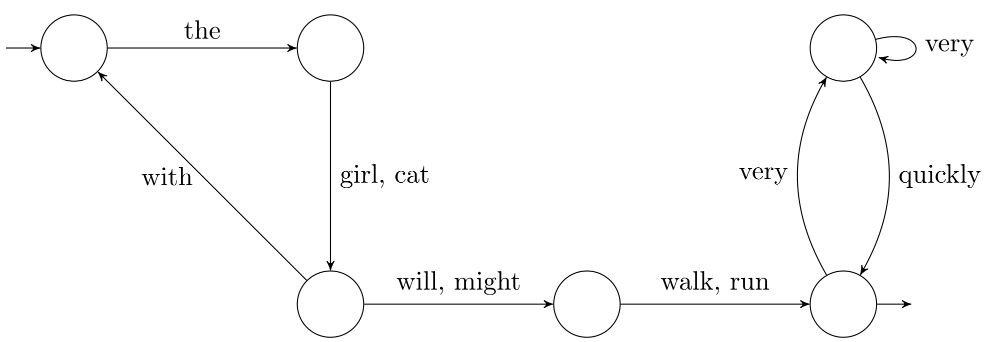
The FSA in (6) generates the subset of $\{t, d, n, i, u\}^*$ consisting of all and only strings satisfying a requirement that voiced obstruents cannot appear at the end of a string.

The FSA in (7) generates a subset of $\{t, d, n, i, u\}^*$ consisting of strings with a certain "sensible" syllable structure.

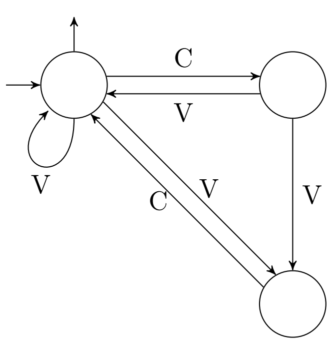
(6)


(7)


The FSA in (8) generates an infinite set of strings of words that look something like a portion of English.

(8)


If we think of the initial state as indicating syllable boundaries, then FSA in (9) generates sequences of syllables of the form '(C)V(C)'. The string 'VCV', for example, can be generated via two different paths, corresponding to different syllabifications.

(9)


2 Formal definition of an FSA

(10) 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:

(11) $\big(\ \{40, 41, 42, 43\},\quad \{\mathrm{C}, \mathrm{V}\},\quad \{40\},\quad \{43\},$
$\{(40, \mathrm{C}, 40), (40, \mathrm{C}, 41), (40, \mathrm{V}, 40), (40, \mathrm{V}, 42), (41, \mathrm{C}, 43), (42, \mathrm{V}, 43), (43, \mathrm{C}, 43), (43, \mathrm{V}, 43)\}\ \big)$

You should convince yourself that (4) and (11) really do contain the same information.

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.

(12) 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
- $q_3 \in F$.

.

(13) More generally, M generates a string of n symbols, say $x_1x_2 \dots x_n$, iff: there are $n+1$ states q_0, q_1, q_2, $\dots q_n$ such that
- $q_0 \in I$, and
- for every $i \in \{1, 2, \dots, n\}$, $(q_{i-1}, x_i, q_i) \in \Delta$, and
- $q_n \in F$.

To take a concrete example:

(14) The automaton in (4)/(11) 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 40, 40, 41, 43, 43 and 43 (respectively), and then it's true that:
- $40 \in I$, and
- $(40, \mathrm{V}, 40) \in \Delta$, and
- $(40, \mathrm{C}, 41) \in \Delta$, and
- $(41, \mathrm{C}, 43) \in \Delta$, and
- $(43, \mathrm{V}, 43) \in \Delta$, and
- $(43, \mathrm{C}, 43) \in \Delta$, and
- $43 \in F$.

Side remark: Note that abstractly, (13) is not all that different from:

(15) A tree-based grammar will generate a string $x_1x_2 \dots x_n$ iff: there is some collection of nonterminal symbols that we can choose such that
- those nonterminal symbols and the symbols x_1, x_2, etc. can all be clicked together into a tree structure in ways that the grammar allows, and
- the nonterminal "at the top" is the start symbol.

(Much more on this in a few weeks!)

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

(16) $w \in \mathcal{L}(M)$
$$\iff \bigvee_{\text{all possible paths } p} \Big[\text{string } w \text{ can be generated by path } p\Big]$$
$$\iff \bigvee_{\text{all possible paths } p} \Big[\bigwedge_{\text{all steps } s \text{ in } p} \big[\text{step } s \text{ is allowed and generates the appropriate part of } w\big]\Big]$$

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 (16) precise is:

(17) $x_1 x_2 \dots x_n \in \mathcal{L}(M)$
$$\iff \bigvee_{q_0 \in Q} \bigvee_{q_1 \in Q} \cdots \bigvee_{q_{n-1} \in Q} \bigvee_{q_n \in Q} \Big[I(q_0) \wedge \Delta(q_0, x_1, q_1) \wedge \cdots \wedge \Delta(q_{n-1}, x_n, q_n) \wedge F(q_n)\Big]$$

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:

(18) $\text{fwd}_M(w)(q)$ 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 $\mathcal{L}(M)$:

(19)
$$w \in \mathcal{L}(M) \iff \bigvee_{q_n \in Q} \Big[\text{fwd}_M(w)(q_n) \wedge F(q_n)\Big]$$

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):

(20)

State		C	V	C	C	V
40	1	1	1	1	1	1
41	0	1	0	1	1	0
42	0	0	1	0	0	1
43	0	0	0	0	1	1

Notice that filling in the values in the leftmost column is easy: this column just says which states are initial states. And with a little bit of thought you should be able to convince yourself that, in order to fill in a column of this table, 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:

(21) 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:

(22)
$$\text{fwd}_M(\epsilon)(q) = I(q)$$
$$\text{fwd}_M(x_1 \dots x_n)(q) = \bigvee_{q_{n-1} \in Q} \Big[\text{fwd}_M(x_1 \dots x_{n-1})(q_{n-1}) \wedge \Delta(q_{n-1}, x_n, q) \Big]$$

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:

(23) $\text{bwd}_M(w)(q)$ 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 $\mathcal{L}(M)$:

(24)
$$w \in \mathcal{L}(M) \iff \bigvee_{q_0 \in Q} \Big[I(q_0) \wedge \text{bwd}_M(w)(q_0) \Big]$$

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):

(25)

State		C	V	C	C	V
40	1	1	1	0	0	0
41	1	0	1	1	0	0
42	0	1	0	0	1	0
43	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.

(26) 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.

(27)
$$\text{bwd}_M(\epsilon)(q) = F(q)$$
$$\text{bwd}_M(x_1 \dots x_n)(q) = \bigvee_{q_1 \in Q} \Big[\Delta(q, x_1, q_1) \wedge \text{bwd}_M(x_2 \dots x_n)(q_1) \Big]$$

3.3 Forward values and backward values together

Now we can say something beautiful:

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

And in fact (19) and (24) are just special cases of (28), with u or v chosen to be the empty string:

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

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