Caveat lector! This is the first edition of this lecture note. A few topics are missing, and there are almost certainly a few serious errors. Please send bug reports and suggestions to jeffe@illinois.edu.

> *O Marvelous! what new configuration will come next?*
> *I am bewildered with multiplicity.*
>
> — William Carlos Williams

> *Life only avails, not the having lived. Power ceases in the instant of repose; it resides in the moment of transition from a past to a new state, in the shooting of the gulf, in the darting to an aim.*
>
> — Ralph Waldo Emerson, "Self Reliance", *Essays, First Series* (1841)

3 Finite-State Machines

3.1 Intuition

Suppose we want to determine whether a given string $w[1..n]$ of bits represents a multiple of 5 in binary. After a bit of thought, you might realize that you can read the bits in w one at a time, from left to right, keeping track of the value modulo 5 of the prefix you have read so far.

```
MultipleOf5($w[1..n]$):
  $rem \leftarrow 0$
  for $i \leftarrow 1$ to $n$
      $rem \leftarrow (2 \cdot rem + w[i]) \bmod 5$
  if $rem = 0$
      return True
  else
      return False
```

Aside from the loop index i, which we need just to read the entire input string, this algorithm has a single local variable *rem*, which has only four different values (0, 1, 2, 3, or 4).

This algorithm already runs in $O(n)$ time, which is the best we can hope for—after all, we have to read every bit in the input—but we can speed up the algorithm *in practice*. Let's define a ***change*** or ***transition*** function $\delta : \{0,1,2,3,4\} \times \{\mathtt{0},\mathtt{1}\} \to \{0,1,2,3,4\}$ as follows:

$$\delta(q,a) = (2q + a) \bmod 5.$$

(Here I'm implicitly converting the symbols $\mathtt{0}$ and $\mathtt{1}$ to the corresponding integers 0 and 1.) Since we already know all values of the transition function, we can store them in a precomputed table, and then replace the computation in the main loop of MultipleOf5 with a simple array lookup.

We can also modify the return condition to check for different values modulo 5. To be completely general, we replace the final if-then-else lines with another array lookup, using an array $A[0..4]$ of booleans describing which final mod-5 values are "acceptable".

After both of these modifications, our algorithm can be rewritten as follows, either iteratively or recursively (with $q = 0$ in the initial call):

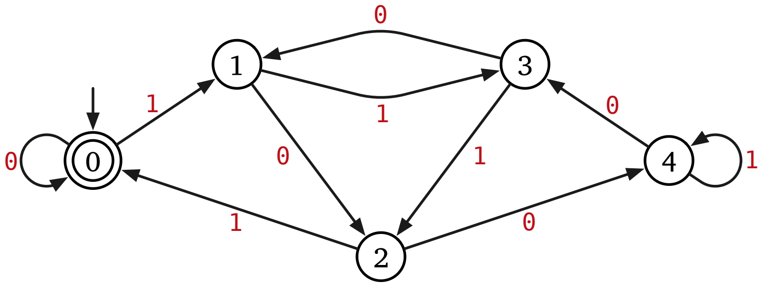
```
DoSomethingCool($w[1..n]$):
  $q \leftarrow 0$
  for $i \leftarrow 1$ to $n$
      $q \leftarrow \delta[q, w[i]]$
  return $A[q]$
```

```
DoSomethingCool($q, w$):
  if $w = \varepsilon$
      return $A[q]$
  else
      decompose $w = a \cdot x$
      return DoSomethingCool($\delta(q, a), x$)
```

If we want to use our new DoSomethingCool algorithm to implement MultipleOf5, we simply give the arrays δ and A the following hard-coded values:

q	$\delta[q, 0]$	$\delta[q, 1]$	$A[q]$
0	0	1	True
1	2	3	False
2	4	0	False
3	1	2	False
4	3	4	False

We can also visualize the behavior of DoSomethingCool by drawing a directed graph, whose vertices represent possible values of the variable q—the possible ***states*** of the algorithm—and whose edges are labeled with input symbols to represent transitions between states. Specifically, the graph includes the labeled directed edge $p \xrightarrow{a} q$ if and only if $\delta(p, a) = q$. To indicate the proper return value, we draw the "acceptable" final states using doubled circles. Here is the resulting graph for MultipleOf5:



State-transition graph for MultipleOf5

If we run the MultipleOf5 algorithm on the string 00101110110 (representing the number 374 in binary), the algorithm performs the following sequence of transitions:

$$0 \xrightarrow{0} 0 \xrightarrow{0} 0 \xrightarrow{1} 1 \xrightarrow{0} 2 \xrightarrow{1} 0 \xrightarrow{1} 1 \xrightarrow{1} 3 \xrightarrow{0} 1 \xrightarrow{1} 3 \xrightarrow{1} 2 \xrightarrow{0} 4$$

Because the final state is not the "acceptable" state 0, the algorithm correctly returns False. We can also think of this sequence of transitions as a walk in the graph, which is completely determined by the start state 0 and the sequence of edge labels; the algorithm returns True if and only if this walk ends at an "acceptable" state.

3.2 Formal Definitions

The object we have just described is an example of a ***finite-state machine***. A finite-state machine is a formal model of any system/machine/algorithm that can exist in a finite number of ***states*** and that transitions among those states based on sequence of ***input*** symbols.

Finite-state machines are also commonly called ***deterministic finite-state automata***, abbreviated ***DFAs***. The word "deterministic" means that the behavior of the machine is completely *determined* by the input string; we'll discuss nondeterministic automata in the next lecture. The word "automaton" (plural "automata") comes from ancient Greek αὐτόματος meaning "self-acting", from the roots αὐτό- ("self") and -ματος ("thinking, willing", the root of Latin *mentus*).

Formally, every finite-state machine consists of five components:

- An arbitrary finite set $\boldsymbol{\Sigma}$, called the ***input alphabet***.
- Another arbitrary finite set $\boldsymbol{Q}$, whose elements are called ***states***.
- An arbitrary ***transition*** function $\boldsymbol{\delta}\colon Q \times \Sigma \to Q$.
- A ***start state*** $\boldsymbol{s} \in Q$.
- A subset $\boldsymbol{A} \subseteq Q$ of ***accepting states***.

The behavior of a finite-state machine is governed by an ***input string*** w, which is a finite sequence of symbols from the input alphabet Σ. The machine ***reads*** the symbols in w one at a time in order (from left to right). At all times, the machine has a *current state* q; initially q is the machine's start state s. Each time the machine reads a symbol a from the input string, its current state ***transitions*** from q to $\delta(q,a)$. After all the characters have been read, the machine ***accepts*** w if the current state is in A and ***rejects*** w otherwise. In other words, every finite state machine runs the algorithm DoSomethingCool! The ***language*** of a finite state machine M, denoted $\boldsymbol{L(M)}$ is the set of all strings that M accepts.

More formally, we extend the transition function $\delta\colon Q \times \Sigma^* \to Q$ of any finite-state machine to a function $\delta^*\colon Q \times \Sigma^* \to Q$ that transitions on *strings* as follows:

$$\delta^*(q,w) := \begin{cases} q & \text{if } w = \varepsilon, \\ \delta^*(\delta(q,a),x) & \text{if } w = ax. \end{cases}$$

Finally, a finite-state machine ***accepts*** a string w if and only if $\delta^*(s,w) \in A$, and ***rejects*** w otherwise. (Compare this definition with the recursive formulation of DoSomethingCool!)

For example, our final MultipleOf5 algorithm is a DFA with the following components:

- input alphabet: $\Sigma = \{\texttt{0},\texttt{1}\}$
- state set: $Q = \{0,1,2,3,4\}$
- transition function: $\delta(q,a) = (2q + a) \bmod 5$
- start state: $s = 0$
- accepting states: $A = \{0\}$

This machine rejects the string `00101110110`, because

$$\begin{aligned}
\delta^*(0,\texttt{00101110110}) &= \delta^*(\delta(0,\texttt{0}),\texttt{0101110110}) \\
= \delta^*(0,\texttt{0101110110}) &= \delta^*(\delta(0,\texttt{0}),\texttt{101110110}) \\
= \delta^*(0,\texttt{101110110}) &= \delta^*(\delta(0,\texttt{1}),\texttt{01110110}) \\
= \delta^*(1,\texttt{01110110}) &= \delta^*(\delta(1,\texttt{0}),\texttt{1110110}) = \cdots \\
&\vdots \\
\cdots = \delta^*(1,\texttt{110}) &= \delta^*(\delta(1,\texttt{1}),\texttt{10}) \\
= \delta^*(3,\texttt{10}) &= \delta^*(\delta(3,\texttt{1}),\texttt{0}) \\
= \delta^*(2,\texttt{0}) &= \delta^*(\delta(3,\texttt{0}),\varepsilon) \\
= \delta^*(4,\varepsilon) &= 4 \notin A.
\end{aligned}$$

We have already seen a more graphical representation of this entire sequence of transitions:

$$0 \xrightarrow{\texttt{0}} 0 \xrightarrow{\texttt{0}} 0 \xrightarrow{\texttt{1}} 1 \xrightarrow{\texttt{0}} 2 \xrightarrow{\texttt{1}} 0 \xrightarrow{\texttt{1}} 1 \xrightarrow{\texttt{1}} 3 \xrightarrow{\texttt{0}} 1 \xrightarrow{\texttt{1}} 3 \xrightarrow{\texttt{1}} 2 \xrightarrow{\texttt{0}} 4$$

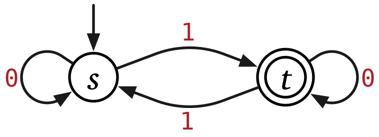
The arrow notation is easier to read and write for specific examples, but surprisingly, most people actually find the more formal functional notation easier to use in formal proofs. Try them both!

We can equivalently define a DFA as a directed graph whose vertices are the states Q, whose edges are labeled with symbols from Σ, such that every vertex has exactly one outgoing edge with each label. In our drawings of finite state machines, the start state s is always indicated by an incoming arrow, and the accepting states A are always indicted by doubled circles. By induction, for any string $w \in \Sigma^*$, this graph contains a unique walk that starts at s and whose edges are labeled with the symbols in w in order. The machine accepts w if this walk ends at an accepting state. This graphical formulation of DFAs is incredibly useful for developing intuition and even designing DFAs. For proofs, it's largely a matter of taste whether to write in terms of extended transition functions or labeled graphs, but (as much as I wish otherwise) I actually find it easier to write ***correct*** proofs using the functional formulation.

3.3 Another Example

The following drawing shows a finite-state machine with input alphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$, state set $Q = \{s, t\}$, start state s, a single accepting state t, and the transition function

$$\delta(s, \texttt{0}) = s, \quad \delta(s, \texttt{1}) = t, \quad \delta(t, \texttt{0}) = t, \quad \delta(t, \texttt{1}) = s.$$



A simple finite-state machine.

For example, the two-state machine M at the top of this page accepts the string `00101110100` after the following sequence of transitions:

$$s \xrightarrow{\texttt{0}} s \xrightarrow{\texttt{0}} s \xrightarrow{\texttt{1}} t \xrightarrow{\texttt{0}} t \xrightarrow{\texttt{1}} s \xrightarrow{\texttt{1}} t \xrightarrow{\texttt{1}} s \xrightarrow{\texttt{0}} s \xrightarrow{\texttt{1}} t \xrightarrow{\texttt{0}} t \xrightarrow{\texttt{0}} t.$$

The same machine M rejects the string `11100101` after the following sequence of transitions:

$$s \xrightarrow{\texttt{1}} t \xrightarrow{\texttt{1}} s \xrightarrow{\texttt{1}} t \xrightarrow{\texttt{0}} t \xrightarrow{\texttt{0}} s \xrightarrow{\texttt{1}} t \xrightarrow{\texttt{0}} t \xrightarrow{\texttt{1}} s.$$

Finally, M rejects the empty string, because the start state s is not an accepting state.

From these examples and others, it is easy to conjecture that the language of M is the set of all strings of `0`s and `1`s with an odd number of `1`s. So let's prove it!

Proof (tedious case analysis): Let $\#(a, w)$ denote the number of times symbol a appears in string w. We will prove the following stronger claims, for any string w.

$$\delta^*(s, w) = \begin{cases} s & \text{if } \#(\texttt{1}, w) \text{ is even} \\ t & \text{if } \#(\texttt{1}, w) \text{ is odd} \end{cases} \qquad \text{and} \qquad \delta^*(t, w) = \begin{cases} t & \text{if } \#(\texttt{1}, w) \text{ is even} \\ s & \text{if } \#(\texttt{1}, w) \text{ is odd} \end{cases}$$

Let w be an arbitrary string. Assume that for any string x that is shorter than w, we have $\delta^*(s, x) = s$ and $\delta^*(t, x) = t$ if x has an even number of `1`s, and $\delta^*(s, x) = t$ and $\delta^*(t, x) = s$ if x has an odd number of `1`s. There are five cases to consider.

- If $w = \varepsilon$, then w contains an even number of $\mathtt{1}$s and $\delta^*(s, w) = s$ and $\delta^*(t, w) = t$ by definition.
- Suppose $w = \mathtt{1}x$ and $\#(\mathtt{1}, w)$ is even. Then $\#(\mathtt{1}, x)$ is odd, which implies

$$\begin{aligned} \delta^*(s,w) &= \delta^*(\delta(s,\mathtt{1}),x) && \text{by definition of } \delta^* \\ &= \delta^*(t,x) && \text{by definition of } \delta \\ &= s && \text{by the inductive hypothesis} \\[1ex] \delta^*(t,w) &= \delta^*(\delta(t,\mathtt{1}),x) && \text{by definition of } \delta^* \\ &= \delta^*(s,x) && \text{by definition of } \delta \\ &= T && \text{by the inductive hypothesis} \end{aligned}$$

  Since the remaining cases are similar, I'll omit the line-by-line justification.

- If $w = \mathtt{1}x$ and $\#(\mathtt{1}, w)$ is odd, then $\#(\mathtt{1}, x)$ is even, so the inductive hypothesis implies

$$\begin{aligned} \delta^*(s,w) = \delta^*(\delta(s,\mathtt{1}),x) = \delta^*(t,x) = t \\ \delta^*(t,w) = \delta^*(\delta(t,\mathtt{1}),x) = \delta^*(s,x) = s \end{aligned}$$

- If $w = \mathtt{0}x$ and $\#(\mathtt{1}, w)$ is even, then $\#(\mathtt{1}, x)$ is even, so the inductive hypothesis implies

$$\begin{aligned} \delta^*(s,w) = \delta^*(\delta(s,\mathtt{0}),x) = \delta^*(s,x) = s \\ \delta^*(t,w) = \delta^*(\delta(t,\mathtt{0}),x) = \delta^*(t,x) = t \end{aligned}$$

- Finally, if $w = \mathtt{0}x$ and $\#(\mathtt{1}, w)$ is odd, then $\#(\mathtt{1}, x)$ is odd, so the inductive hypothesis implies

$$\begin{aligned} \delta^*(s,w) = \delta^*(\delta(s,\mathtt{0}),x) = \delta^*(s,x) = t \\ \delta^*(t,w) = \delta^*(\delta(t,\mathtt{0}),x) = \delta^*(t,x) = s \end{aligned} \qquad \square$$

Notice that this proof contains $|Q|^2 \cdot |\Sigma| + |Q|$ separate inductive arguments. For every pair of states p and q, we must argue about the language so strings w such that $\delta^*(p, w) = q$, and we must consider each first symbol in w. We must also argue about $\delta(p, \varepsilon)$ for every state p. Each of those arguments is typically straightforward, but it's easy to get lost in the deluge of cases.

For this particular proof, however, we can reduce the number of cases by switching from tail recursion to *head* recursion. The following identity holds for all strings $x \in \Sigma^*$ and symbols $a \in \Sigma$:

$$\boxed{\delta^*(q, xa) = \delta(\delta^*(q, x), a)}$$

We leave the inductive proof of this identity as a straightforward exercise (hint, hint).

Proof (clever renaming, head induction): Let's rename the states 0 and 1 instead of s and t. Then the transition function can be described concisely as $\boldsymbol{\delta(q, a) = (q + a) \bmod 2}$.

Now we claim that for every string w, we have $\delta^*(0, w) = \#(\mathtt{1}, w) \bmod 2$. So let w be an arbitrary string, and assume that for any string x that is shorter than w that $\delta^*(0, x) = \#(\mathtt{1}, x) \bmod 2$. There are only two cases to consider: either w is empty or it isn't.

- If $w = \varepsilon$, then $\delta^*(0, w) = 0 = \#(\mathtt{1}, w) \bmod 2$ by definition.

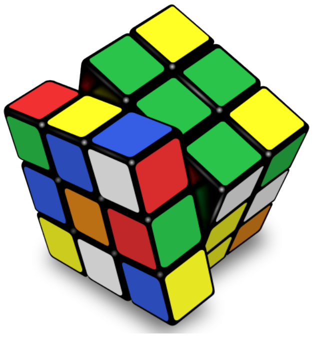
- Otherwise, $w = xa$ for some string x and some symbol a, and we have

$$\begin{aligned}
\delta^*(0, w) &= \delta(\delta^*(0, x), a) \\
&= \delta(\#(\mathtt{1}, x) \bmod 2, a) && \text{by the inductive hypothesis} \\
&= (\#(\mathtt{1}, x) \bmod 2 + a) \bmod 2 && \text{by definition of } \delta \\
&= (\#(\mathtt{1}, x) + a) \bmod 2 && \text{by definition of } \bmod 2 \\
&= (\#(\mathtt{1}, x) + \#(\mathtt{1}, a)) \bmod 2 && \text{because } \#(\mathtt{1}, \mathtt{0}) = 0 \text{ and } \#(\mathtt{1}, \mathtt{1}) = 1 \\
&= (\#(\mathtt{1}, xa)) \bmod 2 && \text{by definition of } \# \\
&= (\#(\mathtt{1}, w)) \bmod 2 && \text{because } w = xa \qquad \square
\end{aligned}$$

Hmmm. This "clever" proof is certainly shorter than the earlier brute-force proof, but is it really "better"? "Simpler"? More intuitive? Easier to understand? I'm skeptical. Sometimes brute force really is more effective.

3.4 Yet Another Example

As a more complex example, consider the ***Rubik's cube***, a well-known mechanical puzzle invented independently by Ernő Rubik in Hungary and Terutoshi Ishigi in Japan in the mid-1970s. This puzzle has precisely 519,024,039,293,878,272,000 distinct configurations. In the unique *solved* configuration, each of the six faces of the cube shows exactly one color. We can change the configuration of the cube by rotating one of the six faces of the cube by 90 degrees, either clockwise or counterclockwise. The cube has six faces (front, back, left, right, up, and down), so there are exactly twelve possible turns, typically represented by the symbols $\mathtt{R}, \mathtt{L}, \mathtt{F}, \mathtt{B}, \mathtt{U}, \mathtt{D}, \bar{\mathtt{R}}, \bar{\mathtt{L}}, \bar{\mathtt{F}}, \bar{\mathtt{B}}, \bar{\mathtt{U}}, \bar{\mathtt{D}}$, where the letter indicates which face to turn and the presence or absence of a bar over the letter indicates turning counterclockwise or clockwise, respectively. Thus, we can represent a Rubik's cube as a finite-state machine with 519,024,039,293,878,272,000 states and an input alphabet with 12 symbols; or equivalently, as a directed graph with 519,024,039,293,878,272,000 vertices, each with 12 outgoing edges. In practice, the number of states is *far* too large for us to actually draw the machine or explicitly specify its transition function; nevertheless, the number of states is still finite. If we let the start state s and the sole accepting state be the solved state, then the language of this finite state machine is the set of all move sequences that leave the cube unchanged.



A complicated finite-state machine.

3.5 Building DFAs

This section describes a few examples of building DFAs that accept particular languages, thereby proving that those languages are automatic. As usual in algorithm design, there is no purely mechanical recipe—no *automatic* method—no *algorithm*—for building DFAs in general. However, the following examples show several useful design strategies.

3.5.1 Superstrings

Perhaps the simplest rule of thumb is to try to construct an algorithm that looks like MultipleOf5: A simple for-loop through the symbols, using a *constant* number of variables, where each variable (except the loop index) has only a *constant* number of possible values. Here, "constant" means an actual number that is not a function of the input size n. You should be able to compute the number of possible values for each variable *at compile time*.

For example, the following algorithm determines whether a given string in $\Sigma = \{\texttt{0}, \texttt{1}\}$ contains the substring `11`.

```
Contains11(w[1..n]):
  found ← False
  for i ← 1 to n
      if i = 1
          last2 ← w[1]
      else
          last2 ← w[1] · w[2]
      if last = 11
          found ← True
  return found
```

Aside from the loop index, this algorithm has exactly two variables.

- A boolean flag *found* indicating whether we have seen the substring `11`. This variable has exactly two possible values: True and False.
- A string *last2* containing the last (up to) three symbols we have read so far. This variable has exactly 7 possible values: ε, `0`, `1`, `00`, `01`, `10`, and `11`.

Thus, altogether, the algorithm can be in at most $2 \times 7 = 14$ possible states, one for each possible pair (*found*, *last2*). Thus, we can encode the behavior of Contains11 as a DFA with fourteen states, where the start state is (False, ε) and the accepting states are all seven states of the form (True, $*$). The transition function is described in the following table (split into two parts to save space):

q	$\delta[q, \texttt{0}]$	$\delta[q, \texttt{1}]$
(False, ε)	(False, `0`)	(False, `1`)
(False, `0`)	(False, `00`)	(False, `01`)
(False, `1`)	(False, `10`)	(**True**, `11`)
(False, `00`)	(False, `00`)	(False, `01`)
(False, `01`)	(False, `10`)	(**True**, `11`)
(False, `10`)	(False, `00`)	(False, `01`)
(False, `11`)	(False, `10`)	(**True**, `11`)
(True, ε)	(True, `0`)	(True, `1`)
(True, `0`)	(True, `00`)	(True, `01`)
(True, `1`)	(True, `10`)	(True, `11`)
(True, `00`)	(True, `00`)	(True, `01`)
(True, `01`)	(True, `10`)	(True, `11`)
(True, `10`)	(True, `00`)	(True, `01`)
(True, `11`)	(True, `10`)	(True, `11`)

For example, given the input string $\texttt{10010}\underline{\texttt{11}}\texttt{100}$, this DFA performs the following sequence of transitions and then accepts.

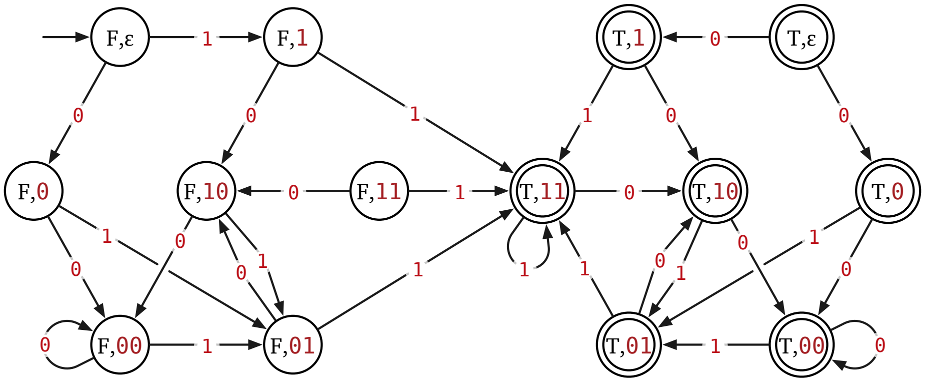
$$(\textsc{False}, \varepsilon) \xrightarrow{\texttt{1}} (\textsc{False}, \texttt{1}) \xrightarrow{\texttt{0}} (\textsc{False}, \texttt{10}) \xrightarrow{\texttt{0}} (\textsc{False}, \texttt{00}) \xrightarrow{\texttt{1}}$$

$$(\textsc{False}, \texttt{01}) \xrightarrow{\texttt{0}} (\textsc{False}, \texttt{10}) \xrightarrow{\texttt{1}} (\textsc{False}, \texttt{01}) \xrightarrow{\texttt{1}}$$

$$(\textsc{True}, \texttt{11}) \xrightarrow{\texttt{1}} (\textsc{True}, \texttt{11}) \xrightarrow{\texttt{0}} (\textsc{True}, \texttt{10}) \xrightarrow{\texttt{0}} (\textsc{True}, \texttt{00})$$

3.5.2 Reducing states

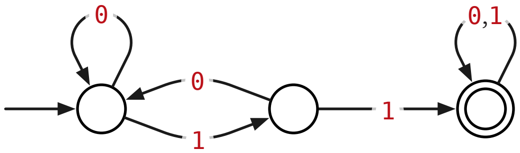
You can probably guess that the brute-force DFA we just constructed has considerably more states than necessary, especially after seeing its transition graph:



Our brute-force DFA for strings containing the substring 11

For example, we don't need actually to remember both of the last two symbols, but only the penultimate symbol, because the last symbol is the one we're currently reading. This observation allows us to reduce the number of states from fourteen to only six. Once the flag part of the state is set to True, we know the machine will eventually accept, so we might as well merge the two accepting states together. Finally, and more subtly, because all transitions out of (False, ε) and (False, 0) are identical, we can merge those two states together as well. In the end, we obtain the following DFA with just three states:

- The start state, which indicates that the machine has not read the substring 11 an did not just read the symbol 1.
- An intermediate state, which indicates that the machine has not read the substring 11 but just read the symbol 1.
- A unique accept state, which indicates that the machine has read the substring 11.



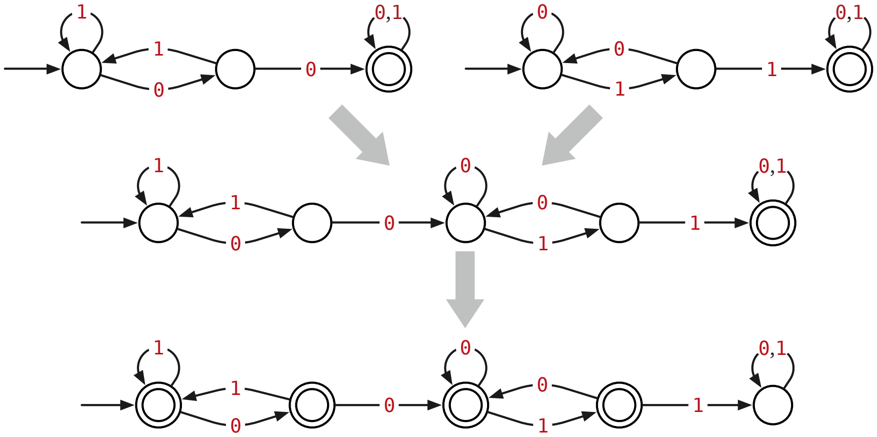
A minimal DFA for superstrings of 11

At the end of this note, I'll describe an efficient algorithm to transform any given DFA into an equivalent DFA with the fewest possible states. Given that this minimization algorithm exists, there is very little incentive to optimize DFAs *by hand*. Clarity is infinitely more important than brevity, especially in this class.

3.5.3 Every this after that

Suppose we want to accept the set of strings in which every occurrence of the substring 00 occurs after every occurrence of the substring 11. Equivalently, we want to *reject* every string in which some 00 occurs before 11. Often the easiest way to design a DFA to check whether a string is *not* in some set is first to build a DFA that *is* in that set and then invert which states in that machine are accepting.

From the previous example, we know that there is a three-state DFA M_{11} that accepts the set of strings with the substring 11 and a nearly identical DFA M_{00} that accepts the set of strings containing the substring 00. By identifying the accept state of M_{00} with the start state of M_{11}, we obtain a five-state DFA that accepts the set of strings with 00 before 11. Finally, by inverting which states are accepting, we obtain the DFA we want.



Building a DFA for the language of strings in which every 00 is after every 11.

3.5.4 Both This and That: The Product Construction

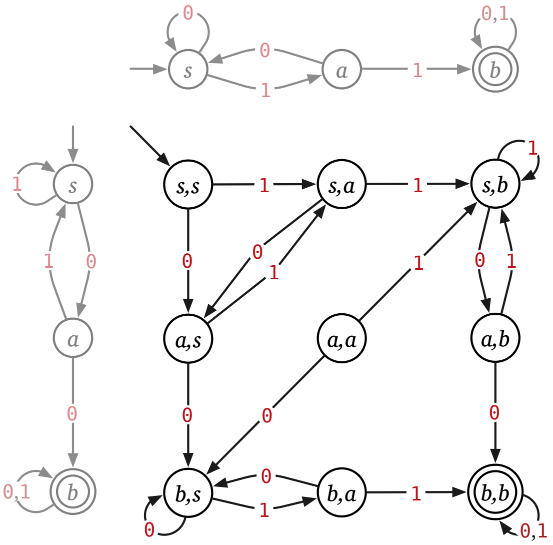
Now suppose we want to accept all strings that contain both 00 and 11 as substrings, in either order. Intuitively, we'd like to run two of our earlier DFAs in parallel—the DFA M_{00} to detect superstrings of 00 and the DFA M_{11} to detect superstrings of 11—and then accept the input string if and only if both of these DFAs accept. In fact, we can encode precisely this "parallel computation" into a single DFA, whose states are all ordered pairs (p,q), where p is a state in M_{00} and q is a state in M_{11}. The new "parallel" DFA includes the transition $(p,q) \xrightarrow{a} (p',q')$ if and only if M_{00} contains the transition $p \xrightarrow{a} p'$ and M_{11} contains the transition $q \xrightarrow{a} q'$. Finally, the state (p,q) is accepting if and only if p and q are accepting states in their respective machines. The resulting nine-state DFA is shown on the next page.

More generally, let $M_1 = (\Sigma, Q_1, \delta_1, s_1, A_1)$ be an arbitrary DFA that accepts some language L_1, and let $M_2 = (\Sigma, Q_2, \delta_2, s_2, A_2)$ be an arbitrary DFA that accepts some language L_2 (over the same alphabet Σ). We can construct a third DFA $M = (\Sigma, Q, \delta, s, A)$ that accepts the intersection language $L_1 \cap L_2$ as follows.

$$\begin{aligned} Q &:= Q_1 \times Q_2 = \big\{(p,q) \bigm| p \in Q_1 \text{ and } q \in Q_2\big\} \\ s &:= (s_1, s_2) \\ A &:= A_1 \times A_2 = \big\{(p,q) \bigm| p \in A_1 \text{ and } q \in A_2\big\} \\ \delta((p,q),a) &:= \big(\delta_1(p,a),\ \delta_2(q,a)\big) \end{aligned}$$

To convince yourself that this product construction is actually correct, consider the extended transition function $\delta^*\colon (Q \times Q') \times \Sigma^* \to (Q \times Q')$, which acts on strings instead of individual symbols. Recall that this function is defined recursively as follows:

$$\delta^*\big((p,q),w\big) := \begin{cases} q & \text{if } w = \varepsilon, \\ \delta^*\big(\delta((p,q),a),\ x\big) & \text{if } w = ax. \end{cases}$$



Building a DFA for the language of strings in which every `00` is after every `11`.

Inductive definition-chasing gives us the identity $\delta^*((p,q),w) = \big(\delta_1^*(p,w),\ \delta_2^*(q,w)\big)$ for any string w:

$$\begin{aligned}
\delta^*\big((p,q),\varepsilon\big) &= (p,q) && \text{by the definition of } \delta^* \\
&= \big(\delta_1^*(p,\varepsilon),\ \delta_2^*(q,\varepsilon)\big) && \text{by the definitions of } \delta_1^* \text{ and } \delta_2^*; \\
\delta^*\big((p,q),ax\big) &= \delta^*\big(\delta((p,q),a),\ x\big) && \text{by the definition of } \delta^* \\
&= \delta^*\big((\delta_1(p,a),\ \delta_2(q,a)),\ x\big) && \text{by the definition of } \delta \\
&= \big(\delta_1^*((\delta_1(p,a),\ x),\ \delta_2^*(\delta_2(q,a),x)\big) && \text{by the induction hypothesis} \\
&= \big(\delta_1^*(p,ax),\ \delta_2^*(q,ax)\big) && \text{by the definitions of } \delta_1^* \text{ and } \delta_2^*.
\end{aligned}$$

It now follows from this seemingly impenetrable wall of notation that for any string w, we have $\delta^*(s,w) \in A$ if and only if both $\delta_1^*(s_1,w) \in A_1$ and $\delta_2^*(s_2,w) \in A_2$. In other words, M accepts w if and only if both M_1 and M_2 accept w, as required.

As usual, this construction technique does not necessarily yield *minimal* DFAs. For example, in our first example of a product DFA, illustrated above, the central state (a,a) cannot be reached by any other state and is therefore redundant. Whatever.

Similar product constructions can be used to build DFAs that accept any other boolean combination of languages; in fact, the only part of the construction that needs to be changed is the choice of accepting states. For example:

- To accept the union $L_1 \cup L_2$, define $A = \big\{(p,q) \bigm| p \in A_1 \textbf{ or } q \in A_2\big\}$.
- To accept the difference $L_1 \setminus L_2$, define $A = \big\{(p,q) \bigm| p \in A_1 \textbf{ but not } q \notin A_2\big\}$.
- To accept the symmetric difference $L_1 \oplus L_2$, define $A = \big\{(p,q) \bigm| p \in A_1 \textbf{ xor } q \in A_2\big\}$.

Moreover, by cascading this product construction, we can construct DFAs that accept arbitrary boolean combinations of arbitrary finite collections of regular languages.

3.6 Decision Algorithms

★★★

It's unclear how much we can say here, since we haven't yet talked about graph algorithms, or even really about graphs. Perhaps this discussion should simply be moved to the graph-traversal notes.

- ***Is $w \in L(M)$?*** Follow the unique path from q_0 with label w. By definition, $w \in L(M)$ if and only if this path leads to an accepting state.
- ***Is $L(M)$ empty?*** The language $L(M)$ is empty if and only if no accepting state is reachable from q_0. This condition can be checked in $O(n)$ time via whatever-first search, where n is the number of states. Alternatively, but less usefully, $L(M) = \varnothing$ if and only if $L(M)$ contains no string w such that $|w| < n$.
- ***Is $L(M)$ finite?*** Remove all states unreachable from q_0 (via whatever first search). Then $L(M)$ is finite if and only if the reduced DFA is a dag; this condition can be checked by depth-first search. Alternatively, but less usefully, $L(M)$ is finite if and only if $L(M)$ contains no string w such that $n \le |w| < 2n$.
- ***Is $L(M) = \Sigma^*$?*** Remove all states unreachable from q_0 (via whatever first search). Then $L(M) = \Sigma^*$ if and only if every state in M is an accepting state.
- ***Is $L(M) = L(M')$?*** Build a DFA N such that $L(N) = L(M) \setminus L(M')$ using a standard product construction, and then check whether $L(N) = \varnothing$.

3.7 Closure Properties

★★★

We haven't yet proved that automatic languages are regular yet, so formally, for now, some of these are closure properties of ***automatic*** languages.

- Complement (easy for DFAs, hard for regular expressions.)
- Concatenation (trivial for regular expressions, hard for DFAs)
- Union (trivial for regular expressions, easy for DFAs via product)
- Intersection (hard for regular expressions, easy for DFAs via product)
- Difference (hard for regular expressions, easy for DFAs via product)
- Kleene star: wait for NFAs (trivial for regular expression, hard for DFAs)
- Homomorphism: only mention in passing
- Inverse homomorphism: only mention in passing

3.8 Fooling Sets

Fix an arbitrary language L over an arbitrary alphabet Σ. For any strings $x, y, z \in \Sigma^*$, we say that ***z distinguishes x from y*** if exactly one of the strings xz and yz is in L. If no string distinguishes x and y, we say that x and y are ***L-equivalent*** and write $\boldsymbol{x \equiv_L y}$. Thus,

$$x \equiv_L y \iff \text{For every string } z \in \Sigma^*\text{, we have } xz \in L \text{ if and only if } yz \in L.$$

For example, let L_{eo} denote the language of strings over $\{\texttt{0}, \texttt{1}\}$ with an even number of `0`s and an odd number of `1`s. Then the strings $x = \texttt{01}$ and $y = \texttt{0011}$ are distinguished by the string $z = \texttt{100}$, because

$$xz = \texttt{01} \bullet \texttt{100} = \texttt{01100} \in L_{eo}$$
$$yz = \texttt{0011} \bullet \texttt{100} = \texttt{0011100} \notin L_{eo}.$$

On the other hand, it is quite easy to prove (hint, hint) that the strings `0001` and `1011` are L_{eo}-equivalent.

Let M be an arbitrary DFA for an arbitrary language L, and let x be y be arbitrary strings. If x and y lead to the same state in M—that is, if $\delta^*(s,x) = \delta^*(s,y)$—then we have

$$\delta^*(s,xz) = \delta^*(\delta^*(s,x),z) = \delta^*(\delta^*(s,y),z) = \delta^*(s,yz)$$

for any string z. In particular, either M accepts both x and y, or M rejects both x and y, and therefore $x \equiv_L y$. It follows that if x and y are not L-equivalent, then ***any*** DFA that accepts L has at least two distinct states $\delta^*(s,x) \neq \delta^*(s,y)$.

Finally, a ***fooling set*** for L is a set F of strings such that *every* pair of strings in F has a distinguishing suffix. For example, $F = \{\mathtt{01}, \mathtt{101}, \mathtt{010}, \mathtt{1010}\}$ is a fooling set for the language L_{eo} of strings with an even number of `0`s and an odd number of `1`s, because each pair of strings in F has a distinguishing suffix:

- `0` distinguishes `01` and `101`;
- `0` distinguishes `01` and `010`;
- `0` distinguishes `01` and `1010`;
- `10` distinguishes `101` and `010`;
- `1` distinguishes `101` and `1010`;
- `1` distinguishes `010` and `1010`.

The pigeonhole principle now implies that for any integer k, if language L is accepted by a DFA with k states, then *every* fooling set for L contains at most k strings. This simple observation has two immediate corollaries.

First, for any integer k, if L has a fooling set of size k, then *every* DFA that accepts L has at least k states. For example, the fooling set $\{\mathtt{01}, \mathtt{101}, \mathtt{010}, \mathtt{1010}\}$ proves that any DFA for L_{eo} has at least four states. Thus, we can use fooling sets to prove that certain DFAs are as small as possible.

Second, and more interestingly, if a language L is accepted by *any* DFA, then *every* fooling set for L must be finite. Equivalently: ***If L has an infinite fooling set, then L is not accepted by any DFA.*** This is arguably both the simplest and most powerful method for proving that a language is non-regular. Here are a few canonical examples of the fooling-set technique in action.

Lemma 3.1. *The language $L = \{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$ is not regular.*

Proof: Consider the set $F = \{\mathtt{0}^n \mid n \geq 0\}$, or more simply $F = \mathtt{0}^*$. Let x and y be arbitrary distinct strings in F. Then we must have $x = \mathtt{0}^i$ and $y = \mathtt{0}^j$ for some integers $i \neq j$. The suffix $z = \mathtt{1}^i$ distinguishes x and y, because $xz = \mathtt{0}^i\mathtt{1}^i \in L$, but $yz = \mathtt{0}^i\mathtt{1}^j \notin L$. We conclude that F is a fooling set for L. Because F is infinite, L cannot be regular. □

Lemma 3.2. *The language $L = \{ww^R \mid w \in \Sigma^*\}$ of even-length palindromes is not regular.*

Proof: Let x and y be arbitrary distinct strings in $\mathtt{0}^*\mathtt{1}$. Then we must have $x = \mathtt{0}^i\mathtt{1}$ and $y = \mathtt{0}^j\mathtt{1}$ for some integers $i \neq j$. The suffix $z = \mathtt{10}^i$ distinguishes x and y, because $xz = \mathtt{0}^i\mathtt{110}^i \in L$, but $yz = \mathtt{0}^i\mathtt{110}^j \notin L$. We conclude that $\mathtt{0}^*\mathtt{1}$ is a fooling set for L. Because $\mathtt{0}^*\mathtt{1}$ is infinite, L cannot be regular. □

Lemma 3.3. *The language $L = \{\mathtt{0}^{2^n} \mid n \geq 0\}$ is not regular.*

Proof: Let x and y be arbitrary distinct strings in L. Then we must have $x = \mathtt{0}^{2^i}$ and $y = \mathtt{0}^{2^j}$ for some integers $i \neq j$. The suffix $z = \mathtt{0}^{2^i}$ distinguishes x and y, because $xz = \mathtt{0}^{2^i+2^i} = \mathtt{0}^{2^{i+1}} \in L$, but $yz = \mathtt{0}^{2^i+2^j} \notin L$. We conclude that L itself is a fooling set for L. Because L is infinite, L cannot be regular. □

Lemma 3.4. *The language $L = \{0^p \mid p \text{ is prime}\}$ is not regular.*

Proof: Again, we use 0^* as our fooling set, but but the actual argument is somewhat more complicated than in our earlier examples.

Let x and y be arbitrary distinct strings in 0^*. Then we must have $x = 0^i$ and $y = 0^j$ for some integers $i \neq j$. Without loss of generality, assume that $i < j$. Let p be any prime number larger than i. Because $p + 0(j-i)$ is prime and $p + p(j-i) > p$ is not, there must be a positive integer $k \leq p$ such that $p + (k-1)(j-i)$ is prime but $p + k(j-i)$ is not. Then the suffix $0^{p+(k-1)j-ki}$ distinguishes x and y:

$$xz = 0^i\,0^{p+(k-1)j-ki} = 0^{p+(k-1)(j-i)} \in L \qquad \text{because } p+(k-1)(j-i) \text{ is prime;}$$
$$yz = 0^j\,0^{p+(k-1)j-ki} = 0^{p+k(j-i)} \notin L \qquad \text{because } p+k(j-i) \text{ is not prime.}$$

(Because $i < j$ and $i < p$, the suffix $0^{p+(k-1)j-ki} = 0^{(p-i)+(k-1)(j-i)}$ has positive length and therefore *actually exists!*) We conclude that 0^* is indeed a fooling set for L, which implies that L is not regular. □

One natural question that many students ask is "How did you come up with that fooling set?" Perhaps the simplest rule of thumb is that for most languages L—in particular, for almost all languages that students are asked to prove non-regular on homeworks or exams—either some simple regular language like 0^* or 10^*1 is a fooling set, or the language L itself is a fooling set. (Of course, there are well-engineered counterexamples.)

⋆3.9 The Myhill-Nerode Theorem

The fooling set technique implies a *necessary* condition for a language to be accepted by a DFA—the language must have no infinite fooling sets. In fact, this condition is also *sufficient*. The following powerful theorem was first proved by Anil Nerode in 1958, strengthening a 1957 result of John Myhill.[1]

The Myhill-Nerode Theorem. *For any language L, the following are equal:*
(a) the minimum number of states in a DFA that accepts L,
(b) the maximum size of a fooling set for L, and
(c) the number of equivalence classes of $\equiv_L$.
In particular, L is accepted by a DFA if and only if every fooling set for L is finite.

Proof: Let L be an arbitrary language.

We have already proved that the size of any fooling set for L is at most the number of states in any DFA that accepts L, so (a)≤(b). It also follows directly from the definitions that $F \subseteq \Sigma^*$ is a fooling set for L if and only if F contains at most one string in each equivalence class of $\equiv_L$; thus, (b)=(c). We complete the proof by showing that (a)≥(c).

We have already proved that if $\equiv_L$ has an infinite number of equivalence classes, there is no DFA that accepts L, so assume that the number of equivalence classes is finite. For any string w, let $[w]$ denote its equivalence class. We define a DFA $M_\equiv = (\Sigma, Q, s, A, \delta)$ as follows:

$$\begin{aligned} Q &:= \big\{[w] \bigm| w \in \Sigma^*\big\} \\ s &:= [\varepsilon] \\ A &:= \big\{[w] \bigm| w \in L\big\} \\ \delta([w], a) &:= [w \bullet a] \end{aligned}$$

[1] Myhill considered the finer equivalence relation $x \sim_L y$, meaning $wxz \in L$ if and only if $wyz \in L$ for all strings w and z, and proved that L is regular if and only if $\sim_L$ defines a finite number of equivalence classes. Like most of Myhill's early automata research, this result appears in an unpublished Air Force technical report. The modern Myhill-Nerode theorem appears (in an even more general form) as a minor lemma in Nerode's 1958 paper, which (not surprisingly) does not cite Myhill.

We claim that this DFA accepts the language L; this claim completes the proof of the theorem.

But before we can prove anything about this DFA, we first need to verify that it is actually well-defined. Let x and y be two strings such that $[x] = [y]$. By definition of L-equivalence, for any string z, we have $xz \in L$ if and only if $yz \in L$. It immediately follows that for any symbol $a \in \Sigma$ and any string z', we have $xaz' \in L$ if and only if $yaz' \in L$. Thus, by definition of L-equivalence, we have $[xa] = [ya]$ for every symbol $a \in \Sigma$. We conclude that the function δ is indeed well-defined.

An easy inductive proof implies that $\delta^*([\varepsilon], x) = [x]$ for every string x. Thus, M accepts string x if and only if $[x] = [w]$ for some string $w \in L$. But if $[x] = [w]$, then by definition (setting $z = \varepsilon$), we have $x \in L$ if and only if $w \in L$. So M accepts x if and only if $x \in L$. In other words, M accepts L, as claimed, so the proof is complete. □

*3.10 Minimal Automata

Given a DFA $M = (\Sigma, Q, s, A, \delta)$, suppose we want to find another DFA $M' = (\Sigma, Q', s', A', \delta')$ with the fewest possible states that accepts the same language. In this final section, we describe an efficient algorithm to minimize DFAs, first described (in slightly different form) by Edward Moore in 1956. We analyze the running time of Moore's in terms of two parameters: $n = |Q|$ and $\sigma = |\Sigma|$.

In the preprocessing phase, we find and remove any states that cannot be reached from the start state s; this filtering can be performed in $O(n\sigma)$ time using any graph traversal algorithm. So from now on we assume that all states are reachable from s.

Now define two states p and q in the trimmed DFA to be ***distingusiable***, written $\boldsymbol{p \not\sim q}$, if at least one of the following conditions holds:

- $p \in A$ and $q \notin A$,
- $p \notin A$ and $q \in A$, or
- $\delta(p,a) \not\sim \delta(q,a)$ for some $a \in \Sigma$.

Equivalently, $p \not\sim q$ if and only if there is a string z such that exactly one of the states $\delta^*(p,z)$ and $\delta^*(q,z)$ is accepting. (Sound familiar?) Intuitively, the main algorithm assumes that all states are equivalent until proven otherwise, and then repeatedly looks for state pairs that can be proved distinguishable.

The main algorithm maintains a two-dimensional table, indexed by the states, where $\mathit{Dist}[p,q] = \textsc{True}$ indicates that we have proved states p and q are distinguished. Initially, for all states p and q, we set $\mathit{Dist}[p,q] \leftarrow \textsc{True}$ if $p \in A$ and $q \notin A$ or vice versa, and $\mathit{Dist}[p,q] = \textsc{False}$ otherwise. Then we repeatedly consider each pair of states and each symbol to find more distinguished pairs, until we make a complete pass through the table without modifying it. The table-filling algorithm can be summarized as follows:[2]

★★★ Don't just whine; actually explain Moore's algorithm as a dynamic programming. Need to prove that if two states can be distinguished at all, they can be distinguished by a string of length at most n.

[2] More experienced readers should become violently ill at the mere suggestion that any algorithm is merely *filling in a table* instead of *evaluating a recurrence*; this algorithm is no exception. Consider the boolean function $\mathit{Dist}(p,q,k)$, which equals $\textsc{True}$ if and only if p and q can be distinguished by some string of length at most k. This function obeys the following recurrence:

$$\mathit{Dist}(p,q,k) = \begin{cases} (p \in A) \oplus (q \in A) & \text{if } k = 0, \\ \mathit{Dist}(p,q,k-1) \vee \bigvee_{a \in \Sigma} \mathit{Dist}(\delta(p,a), \delta(q,a), k-1) & \text{otherwise.} \end{cases}$$

The "table-filling" algorithm presented here is just a space-efficient dynamic programming algorithm to evaluate this recurrence.

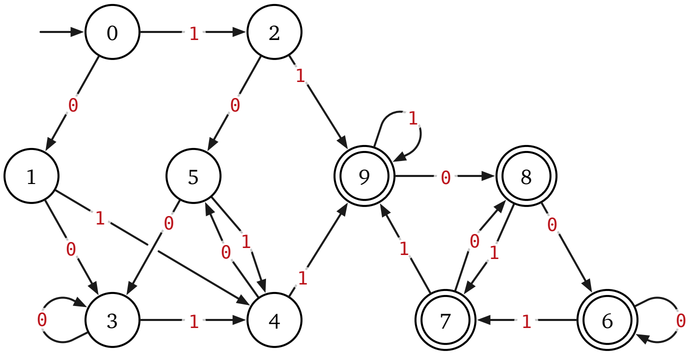
```
MinDFATable($\Sigma, Q, s, A, \delta$):
  for all $p \in Q$
      for all $q \in Q$
          if ($p \in A$ and $q \notin A$) or ($p \notin A$ and $q \in A$)
              $Dist[p,q] \leftarrow$ True
          else
              $Dist[p,q] \leftarrow$ False
  $notdone \leftarrow$ True
  while $notdone$
      $notdone \leftarrow$ False
      for all $p \in Q$
          for all $q \in Q$
              if $Dist[p,q] =$ False
              for all $a \in \Sigma$
                  if $Dist[\delta(p,a), \delta(q,a)]$
                      $Dist[p,q] \leftarrow$ True
                      $notdone \leftarrow$ True
  return $Dist$
```

The algorithm must eventually halt, because there are only a finite number of entries in the table that can be marked. In fact, the main loop is guaranteed to terminate after at most n iterations, which implies that the entire algorithm runs in $\boldsymbol{O(\sigma n^3)}$ time. Once the table is filled, any two states p and q such that $Dist(p,q) =$ False are equivalent and can be merged into a single state. The remaining details of constructing the minimized DFA are straightforward.

With more care, Moore's minimization algorithm can be modified to run in $O(\sigma n^2)$ time. A faster DFA minimization algorithm, due to John Hopcroft, runs in $O(\sigma n \log n)$ time.

Example

To get a better idea how this algorithm works, let's visualize the algorithm running on our earlier brute-force DFA for strings containing the substring 11. This DFA has four unreachable states: (False, 11), (True, ε), (True, 0), and (True, 1). We remove these states, and relabel the remaining states for easier reference. (In an actual implementation, the states would almost certainly be represented by indices into an array anyway, not by mnemonic labels.)



Our brute-force DFA for strings containing the substring 11, after removing all four unreachable states

The main algorithm initializes (the bottom half of) a 10×10 table as follows. (In the implementation, cells marked $\not\sim$ have value True and blank cells have value False.)

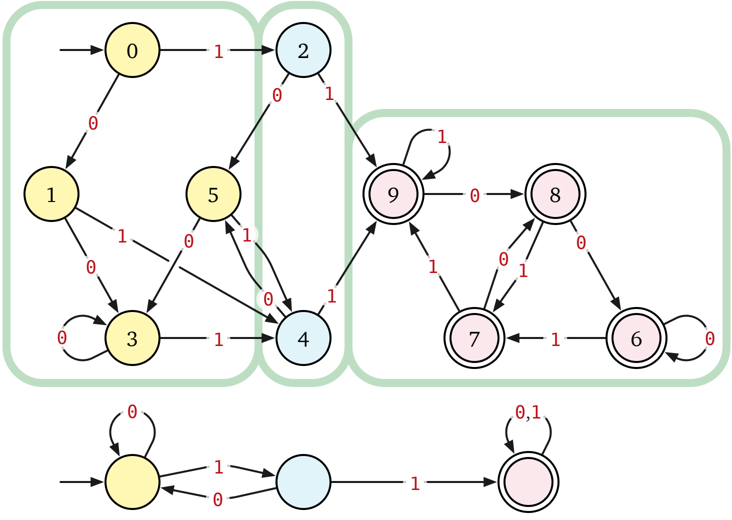
	0	1	2	3	4	5	6	7	8
1									
2									
3									
4									
5									
6	≁	≁	≁	≁	≁	≁			
7	≁	≁	≁	≁	≁	≁			
8	≁	≁	≁	≁	≁	≁			
9	≁	≁	≁	≁	≁	≁			

In the first iteration of the main loop, the algorithm discovers several distinguishable pairs of states. For example, the algorithm sets $\mathit{Dist}[0,2] \leftarrow \textsc{True}$ because $\mathit{Dist}[\delta(0,1),\delta(2,1)] = \mathit{Dist}[2,9] = \textsc{True}$. After the iteration ends, the table looks like this:

	0	1	2	3	4	5	6	7	8
1									
2	≁	≁							
3			≁						
4	≁	≁		≁					
5			≁		≁				
6	≁	≁	≁	≁	≁	≁			
7	≁	≁	≁	≁	≁	≁			
8	≁	≁	≁	≁	≁	≁			
9	≁	≁	≁	≁	≁	≁			

The second iteration of the while loop makes no further changes to the table—We got lucky!—so the algorithm terminates.

The final table implies that the states of our trimmed DFA fall into exactly three equivalence classes: $\{0,1,3,5\}$, $\{2,4\}$, and $\{6,7,8,9\}$. Replacing each equivalence class with a single state gives us the three-state DFA that we already discovered.



Equivalence classes of states in the trimmed DFA, and the resulting minimal equivalent DFA.

Exercises

1. For each of the following languages in $\{0,1\}^*$, describe a deterministic finite-state machine that accepts that language. There are infinitely many correct answers for each language. "Describe" does not necessarily mean "draw".

   (a) Only the string 0110.

   (b) Every string except 0110.

   (c) Strings that contain the substring 0110.

   (d) Strings that do not contain the substring 0110.

   *(e) Strings that contain an even number of occurrences of the substring 0110. (For example, this language contains the strings 0110110 and 01011.)

   (f) Strings that contain the *subsequence* 0110.

   (g) Strings that do not contain the *subsequence* 0110.

   (h) Strings that contain an even number of 1s and an odd number of 0s.

   (i) Strings that represent a number divisible by 7 in binary.

   (j) Strings whose reversals represent a number divisible by 7 in binary.

   (k) Strings in which the substrings 01 and 10 appear the same number of times.

   (l) Strings such that in every prefix, the number of 0s and the number of 1s differ by at most 1.

   (m) Strings such that in every prefix, the number of 0s and the number of 1s differ by at most 4.

   (n) Strings that end with $0^{10} = 0000000000$.

   (o) Strings in which the number of 1s is even, the number of 0s is divisible by 3, the overall length is divisible by 5, the binary value is divisible by 7, and the binary value of the reversal is divisible by 11. *[Hint: This is more tedious than difficult.]*

2. (a) Let $L \subseteq 0^*$ be an arbitrary *unary* language. Prove that L^* is regular.

   (b) Prove that there is a binary language $L \subseteq (0+1)^*$ such that L^* is not regular.

3. Describe and analyze algorithms for the following problems. In each case, the input is a DFA M over the alphabet $\Sigma = \{0,1\}$.

★★★ Move these to the graph traversal notes?

   (a) Does M accept any string whose length is a multiple of 5?

   (b) Does M accept every string that represents a number divisible by 7 in binary?

   (c) Does M accept an infinite number of strings containing an odd number of 0s?

   (d) Does M accept a finite number of strings that contain the substring 0110110 and whose length is divisible by five?

   (e) Does M accept *only* strings whose lengths are perfect squares?

   (f) Does M accept any string whose length is *composite*?

   *(g) Does M accept any string whose length is *prime*?

4. Prove that each of the following languages cannot be accepted by a DFA.

   (a) $\left\{\mathtt{0}^{n^2} \mid n \ge 0\right\}$

   (b) $\left\{\mathtt{0}^{n^3} \mid n \ge 0\right\}$

   (c) $\left\{\mathtt{0}^{f(n)} \mid n \ge 0\right\}$, where $f(n)$ is *any* fixed polynomial in n with degree at least 2.

   (d) $\left\{\mathtt{0}^{n} \mid n \text{ is composite}\right\}$

   (e) $\left\{\mathtt{0}^{n}\mathtt{1}\mathtt{0}^{n} \mid n \ge 0\right\}$

   (f) $\left\{\mathtt{0}^{i}\mathtt{1}^{j} \mid i \ne j\right\}$

   (g) $\left\{\mathtt{0}^{i}\mathtt{1}^{j} \mid i < 3j\right\}$

   (h) $\left\{\mathtt{0}^{i}\mathtt{1}^{j} \mid i \text{ and } j \text{ are relatively prime}\right\}$

   (i) $\left\{\mathtt{0}^{i}\mathtt{1}^{j} \mid j - i \text{ is a perfect square}\right\}$

   (j) $\{w\mathtt{\#}w \mid w \in (\mathtt{0}+\mathtt{1})^*\}$

   (k) $\{ww \mid w \in (\mathtt{0}+\mathtt{1})^*\}$

   (l) $\left\{w\mathtt{\#}\mathtt{0}^{|w|} \mid w \in (\mathtt{0}+\mathtt{1})^*\right\}$

   (m) $\left\{w\mathtt{0}^{|w|} \mid w \in (\mathtt{0}+\mathtt{1})^*\right\}$

   (n) $\{xy \mid w, x \in (\mathtt{0}+\mathtt{1})^* \text{ and } |x| = |y| \text{ but } x \ne y\}$

   (o) $\left\{\mathtt{0}^{m}\mathtt{1}^{n}\mathtt{0}^{m+n} \mid m, n \ge 0\right\}$

   (p) $\{\mathtt{0}^{m}\mathtt{1}^{n}\mathtt{0}^{mn} \mid m, n \ge 0\}$

   (q) Strings in which the substrings $\mathtt{00}$ and $\mathtt{11}$ appear the same number of times.

   (r) The set of all palindromes in $(\mathtt{0}+\mathtt{1})^*$ whose length is divisible by 7.

   (s) $\{w \in (\mathtt{0}+\mathtt{1})^* \mid w \text{ is the binary representation of a perfect square}\}$

   ★(t) $\{w \in (\mathtt{0}+\mathtt{1})^* \mid w \text{ is the binary representation of a prime number}\}$

5. For each of the following languages over the alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, either describe a DFA that accepts the language or prove that no such DFA exists. Recall that Σ^+ denotes the set of all *nonempty* strings over Σ. *[Hint: Believe it or not, most of these languages **can** be accepted by DFAs.]*

   (a) $\left\{wxw \mid w, x \in \Sigma^*\right\}$

   (b) $\left\{wxw \mid w, x \in \Sigma^+\right\}$

   (c) $\left\{wxw^R \mid w, x \in \Sigma^+\right\}$

   (d) $\left\{wwx \mid w, x \in \Sigma^+\right\}$

   (e) $\left\{ww^Rx \mid w, x \in \Sigma^+\right\}$

   (f) $\left\{wxwy \mid w, x, y \in \Sigma^+\right\}$

   (g) $\left\{wxw^Ry \mid w, x, y \in \Sigma^+\right\}$

   (h) $\left\{xwwy \mid w, x, y \in \Sigma^+\right\}$

   (i) $\left\{xww^Ry \mid w, x, y \in \Sigma^+\right\}$

   (j) $\left\{wxxw \mid w, x \in \Sigma^+\right\}$

   ★(k) $\left\{wxw^Rx \mid w, x \in \Sigma^+\right\}$