MA/CSSE 474

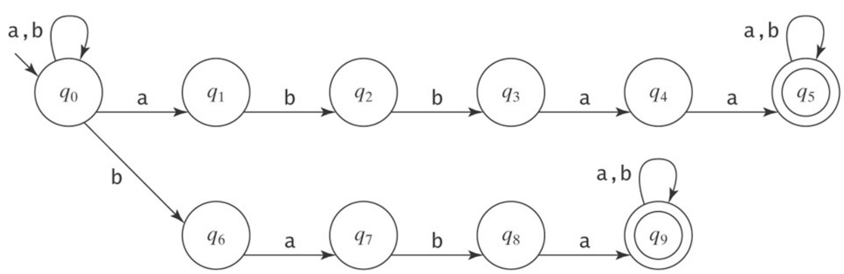
Theory of Computation

NDFSM→DFSM

Pattern Matching: Multiple Keywords

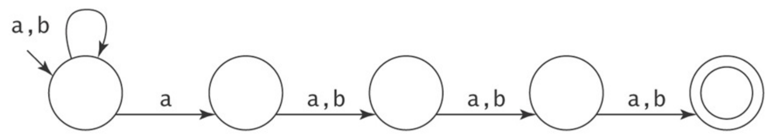
$L = \{w \in \{\texttt{a}, \texttt{b}\}^* : \exists x, y \in \{\texttt{a}, \texttt{b}\}^*$

$((w = x\ \texttt{abbaa}\ y) \vee (w = x\ \texttt{baba}\ y))\}.$



Checking from the End

$L = \{w \in \{\texttt{a}, \texttt{b}\}^* :$
the fourth character from the end is $\texttt{a}\}$



Another Pattern Matching Example

$L = \{w \in \{\texttt{0}, \texttt{1}\}^*$: w is the binary encoding of a positive integer that is divisible by 16 or is odd}

Q1

Another NDFSM

$L_1 = \{w \in \{\texttt{a}, \texttt{b}\}^*: \texttt{aa}$ occurs in $w\}$
$L_2 = \{x \in \{\texttt{a}, \texttt{b}\}^*: \texttt{bb}$ occurs in $x\}$
$L_3 = \{y \in L_1 \cup L_2\}$

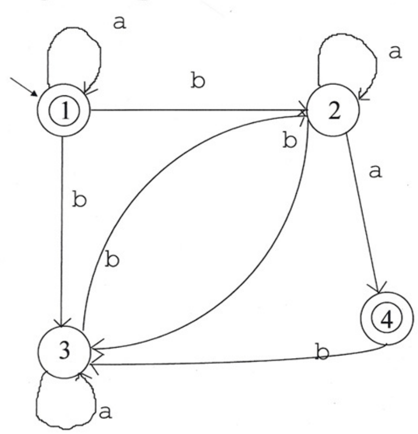
$M_1 =$

$M_2 =$

$M_3 =$

This is a good example for practice later

Analyzing Nondeterministic FSMs



Does this FSM accept:

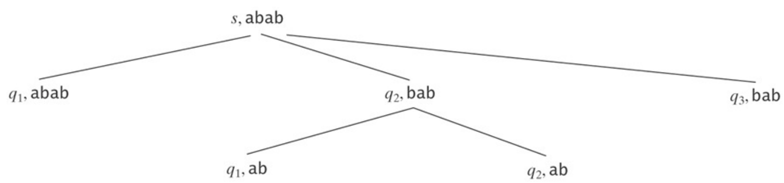
`baaba`

Remember: we just have to find one accepting path.

Simulating Nondeterministic FSMs

Two approaches:

• Explore a search tree:



• Follow all paths in parallel

Dealing with ε Transitions

The epsilon closure of a state:

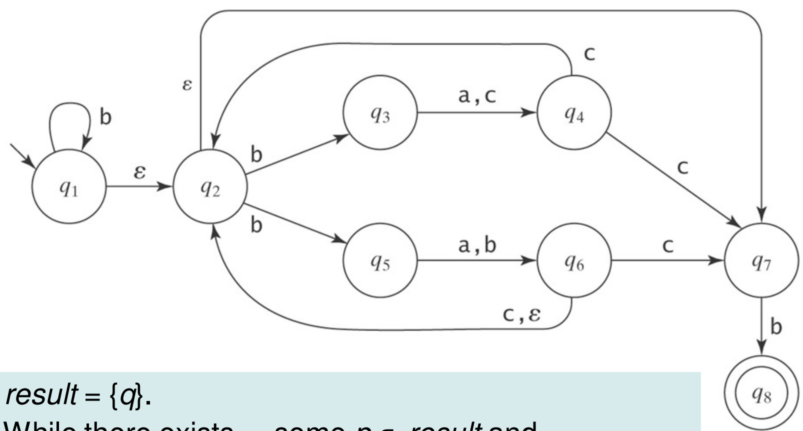
$$eps(q) = \{p \in K : (q, w) \vdash\text{-}^*_M (p, w)\}.$$

$eps(q)$ is the closure of $\{q\}$ under the relation $\{(p, r) :$ there is a transition $(p, \varepsilon, r) \in \Delta\}$.

Algorithm for computing eps(q):

result = $\{q\}$.
While there exists some $p \in$ *result* and
some $r \notin$ *result* and
some transition $(p, \varepsilon, r) \in \Delta$ do:
Insert r into *result.*
Return *result.*

Calculate eps(q) for each state q



result = {*q*}.
While there exists some $p \in$ *result* and
some $r \notin$ *result* and
some transition $(p, \varepsilon, r) \in \Delta$ do:
result = *result* $\cup$ { *r* }
Return *result.*

Q2

Simulating a NDFSM

ndfsmsimulate(*M*: NDFSM, *w*: string) =

1. *current-state* = *eps*(*s*).
2. While any input symbols in *w* remain to be read do:
    1. *c* = get-next-symbol(*w*).
    2. *next-state* = $\varnothing$.
    3. For each state *q* in *current-state* do:
       For each state *p* such that $(q, c, p) \in \Delta$ do:
       next-state = *next-state* $\cup$ *eps*(*p*).
    4. *current-state* = *next-state*.
3. If *current-state* contains any states in *A*, accept.
   Else reject.

Q3

Nondeterministic and Deterministic FSMs

Clearly: {Languages accepted by some DFSM} $\subseteq$ {Languages accepted by some NDFSM}

More interesting:

***Theorem**:*

For each NDFSM, there is an equivalent DFSM.

"equivalent" means "accepts the same language"

Nondeterministic and Deterministic FSMs

***Theorem**:* For each NDFSM, there is an equivalent DFSM.

***Proof**:* By construction:

Given a NDFSM $M = (K, \Sigma, \Delta, s, A)$,
we construct $M' = (K', \Sigma, \delta', s', A')$, where

$K' = \mathcal{P}(K)$
$s' = eps(s)$
$A' = \{Q \subseteq K : Q \cap A \neq \varnothing\}$

$\delta'(Q, a) = \bigcup\{eps(p): p \in K$ and $(q, a, p) \in \Delta$ for some $q \in Q\}$

An Algorithm for Constructing the Deterministic FSM

1. Compute the $eps(q)$'s.

2. Compute $s' = eps(s)$.

3. Compute δ'.

4. Compute K' = a subset of $\mathcal{P}(K)$.

5. Compute $A' = \{Q \in K' : Q \cap A \neq \varnothing\}$.

The Algorithm *ndfsmtodfsm*

ndfsmtodfsm(M: NDFSM) =
1. For each state q in K_M do:
   1.1 Compute $eps(q)$.
2. $s' = eps(s)$
3. Compute δ':
   3.1 *active-states* = $\{s'\}$.
   3.2 $\delta' = \varnothing$.
   3.3 While there exists some element Q of *active-states* for which δ' has not yet been computed do:
      For each character c in Σ_M do:
         new-state = $\varnothing$.
         For each state q in Q do:
            For each state p such that $(q, c, p) \in \Delta$ do:
               new-state = *new-state* $\cup$ $eps(p)$.
         Add the transition (q, c, *new-state*) to δ'.
         If *new-state* $\notin$ *active-states* then insert it.
4. K' = *active-states*.
5. $A' = \{Q \in K' : Q \cap A \neq \varnothing\}$.

Draw part of the transition diagram for the DFSM constructed from the NDFSM that appeared a few slides earlier.

Next week we will prove that it works.

Q4

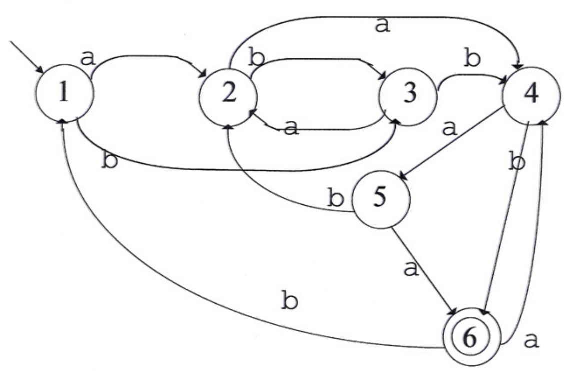
Finite State Machines

Intro to State Minimization

Among all DSFMs that are equivalent to a given DFSM, find one whose number of states is minimal

State Minimization

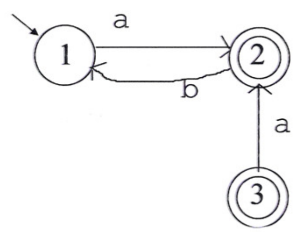
Consider:



Is this a minimal machine?

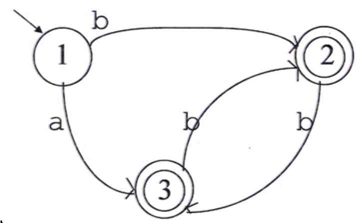
State Minimization

Step (1): Get rid of unreachable states.



State 3 is unreachable.

Step (2): Get rid of redundant states.

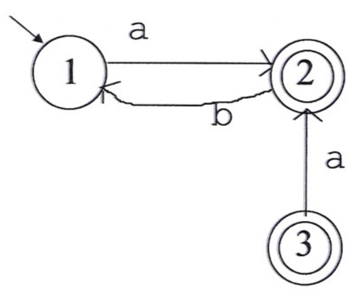


States 2 and 3 are redundant.

Getting Rid of Unreachable States

We can't easily find the unreachable states directly. But we can find the reachable ones and determine the unreachable ones from there.

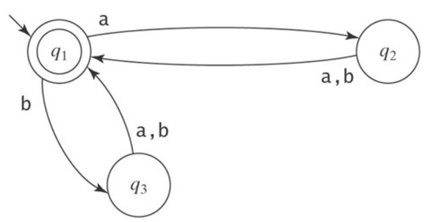
An algorithm for finding the reachable states:



Getting Rid of Redundant States

Intuitively, two states are equivalent to each other (and thus one is redundant) if all string in Σ^* have the same fate, regardless of which of the two states the machine is in. But how can we tell this?

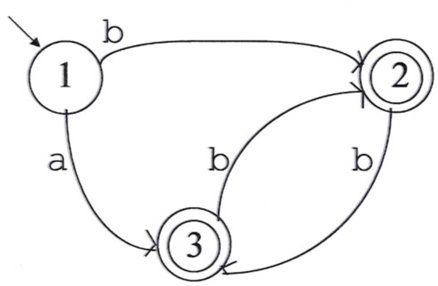
The simple case:



Two states have identical sets of transitions out.

Getting Rid of Redundant States

The harder case:



The outcomes in states 2 and 3 are the same, even though the states aren't.

Finding an Algorithm for Minimization

Capture the notion of equivalence classes of strings with respect to a language.

Prove that we can always find a (unique up to state naming) a deterministic FSM with a number of states equal to the number of equivalence classes of strings.

Describe an algorithm for finding that deterministic FSM.

Defining Equivalence for Strings

We want to capture the notion that two strings are equivalent or ***indistinguishable*** with respect to a language *L* if, no matter what string w tacked on to them on the right, either both concatenated strings will be in *L* or neither will. **Why is this the right notion?** Because it corresponds naturally to what the states of a recognizing FSM have to remember.

Example:

(1) a b a b a b

(2) b a a b a b

Suppose $L = \{w \in \{\texttt{a}, \texttt{b}\}^* : |w| \text{ is even}\}$. Are (1) and (2) equivalent?

Suppose $L = \{w \in \{\texttt{a}, \texttt{b}\}^*$: every a is immediately followed by b$\}$. Are (1) and (2) equivalent?

Q5a

Defining Equivalence for Strings

If two strings are indistinguishable with respect to L, we write:

$$x \approx_L y$$

Formally, $x \approx_L y$ iff $\forall z \in \Sigma^* \ (xz \in L \text{ iff } yz \in L)$.

Q5a

$\approx_L$ is an Equivalence Relation

$\approx_L$ is an equivalence relation because it is:

- Reflexive: $\forall x \in \Sigma^* \ (x \approx_L x)$, because:
  $\forall x, z \in \Sigma^* \ (xz \in L \leftrightarrow xz \in L)$.
- Symmetric: $\forall x, y \in \Sigma^* \ (x \approx_L y \rightarrow y \approx_L x)$, because:
  $\forall x, y, z \in \Sigma^* \ ((xz \in L \leftrightarrow yz \in L) \leftrightarrow$
  $(yz \in L \leftrightarrow xz \in L))$.
- Transitive: $\forall x, y, z \in \Sigma^* \ (((x \approx_L y) \wedge (y \approx_L w)) \rightarrow (x \approx_L w))$, because:
  $\forall x, y, z \in \Sigma^*$
  $(((xz \in L \leftrightarrow yz \in L) \wedge (yz \in L \leftrightarrow wz \in L)) \rightarrow$
  $(xz \in L \leftrightarrow wz \in L))$.

$\approx_L$ is an Equivalence Relation

An equivalence relation on a set partitions the set.
Thus:

- No equivalence class of $\approx_L$ is empty.
- Each string in Σ^* is in exactly one equivalence class of $\approx_L$.

An Example

$\Sigma = \{\texttt{a}, \texttt{b}\}$
$L = \{w \in \Sigma^*$: every `a` is immediately followed by `b`$\}$

The equivalence classes of $\approx_L$: Try:

ε	aa	bbb
a	bb	baa
b	aba	
	aab	

An Example

$\Sigma = \{$a, b$\}$
$L = \{w \in \Sigma^*$: every a is immediately followed by b$\}$

The equivalence classes of $\approx_L$:

[1] [ε, b, abb, …] [all strings in L].

[2] [a, abbba, …] [all strings that end in a and have no prior a that is not followed by a b].

[3] [aa, abaa, …] [all strings that contain at least one instance of aa].

Another Example of $\approx_L$

$\Sigma = \{$a, b$\}$
$L = \{w \in \Sigma^* : |w|$ is even$\}$

ε	bb	aabb
a	aba	bbaa
b	aab	aabaa
aa	bbb	
	baa	

The equivalence classes of $\approx_L$:

Yet Another Example of $\approx_L$

$\Sigma = \{a, b\}$
$L = aab^*a$

ε	bb	aabaa
a	aba	aabbba
b	aab	aabbaa
aa	baa	
	aabb	

The equivalence classes of $\approx_L$:

When More Than One Class Contains Strings in *L*

$\Sigma = \{a, b\}$
$L = \{w \in \Sigma^* :$ no two adjacent characters are the same$\}$

ε	aa	aabb
a	bb	aabaa
b	aba	aabbba
	aab	aabbaa
	baa	

The equivalence classes of $\approx_L$:

When More Than One Class Contains Strings in *L*

$\Sigma = \{a, b\}$
$L = \{w \in \Sigma^* :$ no two adjacent characters are the same$\}$

The equivalence classes of $\approx_L$:

[1] [ε]
[2] [a, aba, ababa, …]
[3] [b, ab, bab, abab, …]
[4] [aa, abaa, ababb…]

Does $\approx_L$ Always Have a Finite Number of Equivalence Classes?

$\Sigma = \{a, b\}$
$L = \{a^n b^n, n \geq 0\}$

ε	aa	aaaa
a	aba	aaaaa
b	aaa	

The equivalence classes of $\approx_L$:

The Best We Can Do

Theorem: Let L be a regular language and let M be a DFSM that accepts L. The number of states in M is greater than or equal to the number of equivalence classes of $\approx_L$.

Proof: Suppose that the number of states in M were less than the number of equivalence classes of $\approx_L$. Then, by the pigeonhole principle, there must be at least one state q that contains strings from at least two equivalence classes of $\approx_L$. But then M's future behavior on those strings will be identical, which is not consistent with the fact that they are in different equivalence classes of $\approx_L$.

The Best Is Unique

Theorem: Let L be a regular language over some alphabet Σ. Then there is a DFSM M that accepts L and that has precisely n states where n is the number of equivalence classes of $\approx_L$. Any other FSM that accepts L must either have more states than M or it must be equivalent to M except for state names.

Proof: (by construction)

$M = (K, \Sigma, \delta, s, A)$, where:

- K contains n states, one for each equivalence class of $\approx_L$.
- $s = [\varepsilon]$, the equivalence class of ε under $\approx_L$.
- $A = \{[x] : x \in L\}$.
- $\delta([x], a) = [xa]$. In other words, if M is in the state that contains some string x, then, after reading the next symbol, a, it will be in the state that contains xa.

Proof, Continued

We must show that:

- K is finite. Since L is regular, it is accepted by some DFSM M'. M' has some finite number of states m. By Theorem 5.4, $n \leq m$. So K is finite.

- δ is a function. In other words, it is defined for all (state, input) pairs and it produces, for each of them, a unique value. The construction defines a value of δ for all (state, input) pairs. The fact that the construction guarantees a unique such value follows from the definition of $\approx_L$.

Proof, Continued

- $L = L(M)$. To prove this, we must first show that $\forall s, t$ $(([\varepsilon], st) \mid\!-_M^* ([s], t))$. We do this by induction on $|s|$.

  If $|s| = 0$ then we have $([\varepsilon], \varepsilon) \mid\!-_M^* ([\varepsilon], t)$, which is true since M simply makes zero moves.

Proof, Continued

Assume that the claim is true if $|s| = k$. Then we consider what happens when $|s| = k+1$. $|s| \geq 1$, so we can let $s = yc$ where $y \in \Sigma^*$ and $c \in \Sigma$. We have:

/* M reads the first k characters:
$([\varepsilon], yct) \mid\!-_M^* ([y], ct)$ (induction hypothesis, since $|y| = k$).

/* M reads one more character:
$([y], ct) \mid\!-_M^* ([yc], t)$ (definition of δ_M).

/* Combining those two, after M has read k+1 characters:
$([\varepsilon], yct) \mid\!-_M^* ([yc], t)$ (transitivity of $\mid\!-_M^*$).
$([\varepsilon], st) \mid\!-_M^* ([s], t)$ (definition of s as yc).

Proof, Continued

So we have :

[*] $\forall s, t\ (([\varepsilon], st) \mid\!-_M^* ([s], t))$.

Let t be ε. Let s be any string in Σ^*. By [*]:

$$([\varepsilon], s) \mid\!-_M^* ([s], \varepsilon).$$

So M will accept s iff $[s] \in A$, which, by the way in which A was constructed, it will be if the strings in $[s]$ are in L. So M accepts precisely those strings that are in M.

Proof, Continued

- There exists no smaller machine $M\#$ that also accepts L. This follows directly from Theorem 5.4, which says that the number of equivalence classes of $\approx_L$ imposes a lower bound on the number of states in any DFSM that accepts L.
- There is no different machine $M\#$ that also has n states and that accepts L.

Constructing the Minimal DFA from $\approx_L$

$\Sigma = \{a, b\}$
$L = \{w \in \Sigma^* :$ no two adjacent characters are the same$\}$

The equivalence classes of $\approx_L$:

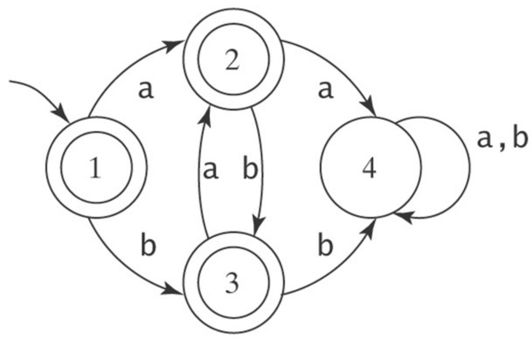
1: [ε]	ε
2: [a, ba, aba, baba, ababa, ...]	(b∪ε)(ab)*a
3: [b, ab, bab, abab, ...]	(a∪ε)(ba)*b
4: [bb, aa, bba, bbb, ...]	the rest

- Equivalence classes become states
- Start state is [ε]
- Accepting states are all equivalence classes in L
- $\delta([x], a) = [xa]$

Constructing the Minimal DFA from $\approx_L$

$\Sigma = \{\mathtt{a}, \mathtt{b}\}$

$L = \{w \in \Sigma^* :$ no two adjacent characters are the same$\}$



The Myhill-Nerode Theorem

Theorem: A language is regular iff the number of equivalence classes of $\approx_L$ is finite.

Proof: *S*how the two directions of the implication:

L regular → the number of equivalence classes of $\approx_L$ ***is finite:*** If *L* is regular, then there exists some FSM *M* that accepts *L*. *M* has some finite number of states *m*. The cardinality of $\approx_L \leq m$. So the cardinality of $\approx_L$ is finite.

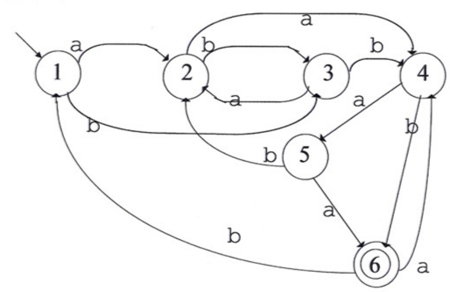
The number of equivalence classes of $\approx_L$ ***is finite → L regular:*** If the cardinality of $\approx_L$ is finite, then the construction that was described in the proof of the previous theorem will build an FSM that accepts *L*. So *L* must be regular.

So Where Do We Stand?

1. We know that for any regular language L there exists a minimal accepting machine M_L.
2. We know that $|K|$ of M_L equals the number of equivalence classes of $\approx_L$.
3. We know how to construct M_L from $\approx_L$.
4. We know that M_L is unique up to the naming of its states.

But is this good enough?

Consider:



Minimizing an Existing DFSM (Without Knowing $\approx L$)

Two approaches:

- Begin with M and collapse redundant states, getting rid of one at a time until the resulting machine is minimal.
- Begin by overclustering the states of L into just two groups, accepting and nonaccepting. Then iteratively split those groups apart until all the distinctions that L requires have been made.

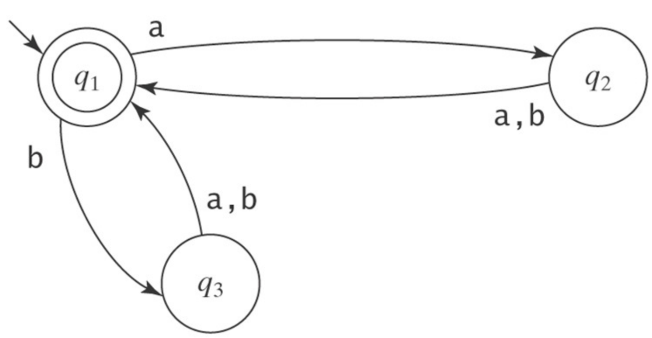
The Overclustering Approach

We need a definition for "equivalent", i.e., mergeable states.

Define $q \equiv p$ iff for all strings $w \in \Sigma^*$, either w drives M to an accepting state from both q and p or it drives M to a rejecting state from both q and p.

An Example

$\Sigma = \{\texttt{a}, \texttt{b}\}$ $L = \{w \in \Sigma^* : |w| \text{ is even}\}$



$q_2 \equiv q_3$

Constructing ≡ as the Limit of a Sequence of Approximating Equivalence Relations $\equiv^n$

(Where *n* is the length of the input strings that have been considered so far)

Consider input strings, starting with ε, and increasing in length by 1 at each iteration. Start by way overgrouping states. Then split them apart as it becomes apparent (with longer and longer strings) that their behavior is not identical.

Constructing $\equiv_n$

- $p \equiv^0 q$ iff they behave equivalently when they read ε. In other words, if they are both accepting or both rejecting states.
- $p \equiv^1 q$ iff they behave equivalently when they read any string of length 1, i.e., if any single character sends both of them to an accepting state or both of them to a rejecting state. Note that this is equivalent to saying that any single character sends them to states that are $\equiv^0$ to each other.
- $p \equiv^2 q$ iff they behave equivalently when they read any string of length 2, which they will do if, when they read the first character they land in states that are $\equiv^1$ to each other. By the definition of $\equiv^1$, they will then yield the same outcome when they read the single remaining character.
- And so forth.

Constructing ≡, Continued

More precisely, $\forall p,q \in K$ and any $n \geq 1$, $q \equiv^n p$ iff:

1. $q \equiv^{n\text{-}1} p$, and
2. $\forall a \in \Sigma\ (\delta(p, a) \equiv^{n\text{-}1} \delta(q, a))$

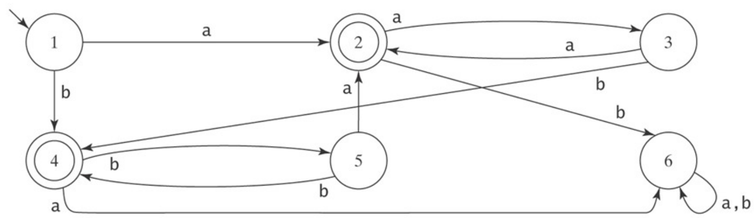
MinDFSM

MinDFSM(*M*: DFSM) =

1. *classes* := {*A*, *K*-*A*};
2. Repeat until no changes are made
   - 2.1. *newclasses* := ∅;
   - 2.2. For each equivalence class *e* in *classes*, if *e* contains more than one state do
     - For each state *q* in *e* do
       - For each character *c* in Σ do
         - Determine which element of *classes q* goes to if *c* is read
     - If there are any two states *p* and *q* that need to be split, split them. Create as many new equivalence classes as are necessary. Insert those classes into *newclasses*.
     - If there are no states whose behavior differs, no splitting is necessary. Insert *e* into *newclasses*.
   - 2.3. *classes* := *newclasses*;
3. Return $M^* = (classes, \Sigma, \delta, [s_M], \{[q\text{: the elements of } q \text{ are in } A_M]\})$, where δ_{M^*} is constructed as follows:
   - if $\delta_M(q, c) = p$, then $\delta_{M^*}([q], c) = [p]$

An Example

Σ = {a, b}

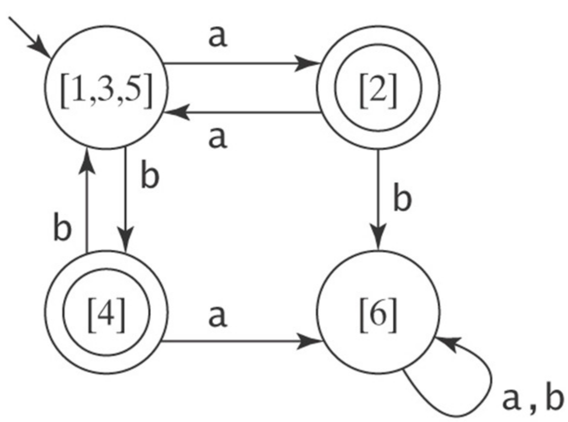


$\equiv^0$ =

$\equiv^1$ =

$\equiv^2$ =

The Result



Summary

- Given any regular language *L*, there exists a minimal DFSM *M* that accepts *L*.
- *M* is unique up to the naming of its states.
- Given any DFSM *M*, there exists an algorithm *minDFSM* that constructs a minimal DFSM that also accepts *L*(*M*).

Canonical Forms

A ***canonical form*** for some set of objects *C* assigns exactly one representation to each class of "equivalent" objects in *C*.

Further, each such representation is distinct, so two objects in *C* share the same representation iff they are "equivalent" in the sense for which we define the form.

A Canonical Form for FSMs

buildFSMcanonicalform(M: FSM) =

1. M' = *ndfsmtodfsm*(M).
2. M^* = *minDFSM*(M').
3. Create a unique assignment of names to the states of M^*.
4. Return M^*.

Given two FSMs M_1 and M_2:

buildFSMcanonicalform(M_1)
=
buildFSMcanonicalform(M_2)

iff $L(M_1) = L(M_2)$.

Correctness Proof of *ndfsmtodfsm*

To prove:

From any NDFSM $M = (K, \Sigma, \Delta, s, A)$, *ndfsmtodfsm* constructs a DFSM $M' = (K', \Sigma, \delta', s', A')$, which is equivalent to M.

$K' \subseteq \mathcal{P}(K)$ (a.k.a. 2^K)

$s' = eps(s)$

$A' = \{Q \subseteq K : Q \cap A \neq \varnothing\}$

$\delta'(Q, a) = \cup \{eps(p): p \in K$ and
$(q, a, p) \in \Delta$ for some $q \in Q\}$

Correctness Proof of *ndfsmtodfsm*

From any NDFSM M, *ndfsmtodfsm* constructs a DFSM M', which is:

(1) **Deterministic:** By the definition in step 3 of δ', we are guaranteed that δ' is defined for all reachable elements of K' and all possible input characters. Further, step 3 inserts a single value into δ' for each state-input pair, so M' is deterministic.

(2) **Equivalent to *M*:** We constructed δ' so that M' mimics an "all paths" simulation of M. We must now prove that that simulation returns the same result that M would.

A Useful Lemma

Lemma: Let w be any string in Σ^*, let p and q be any states in K, and let P be any state in K'. Then:

$$(q, w) \vdash_M^* (p, \varepsilon) \textbf{ iff } ((eps(q), w) \vdash_{M'}^* (P, \varepsilon) \text{ and } p \in P) .$$

INFORMAL RESTATEMENT OF LEMMA: In other words, if the original NDFSM M starts in state q and, after reading the string w, can land in state p (along at least one of its paths), then the new DFSM M' must behave as follows:

When started in the state that corresponds to the set of states the original machine M could get to from q without consuming any input, M' reads the string w and lands in a state P (which is a set of M's states) that contains p.

Furthermore, because of the only- if part of the lemma, M' (starting from q and reading w) must end up in a "set state" that contains only states that M could get to from q after reading w and following any available epsilon-transitions.

A Useful Lemma

Lemma: Let w be any string in Σ^*, let p and q be any states in K, and let P be any state in K'. Then:

$(q, w) \vdash_M^* (p, \varepsilon)$ **iff** $((eps(q), w) \vdash_{M'}^* (P, \varepsilon)$ and $p \in P)$

Recall: NDFSM $M = (K, \Sigma, \Delta, s, A)$, DFSM $M' = (K', \Sigma, \delta', s', A')$,

It turns out that we will only need this lemma for the case where q = s, but the more general form is easier to prove by induction. This is common in induction proofs.

Proof: We must show that δ' has been defined so that the individual steps of M', when taken together, do the right thing for an input string w of any length. Since the definitions describe one step at a time, we will prove the lemma by induction on $|w|$.

Base Case: |w| = 0, so w = ε

- *if* part: Prove:

$$(eps(q), w) \vdash_{M'}^* (P, \varepsilon) \wedge p \in P \longrightarrow (q, w) \vdash_M^* (p, \varepsilon)$$

Since **w** = ε and M' (being deterministic) contains no ε-transitions, M' makes no moves. So M' must end in the same state it started in, namely $eps(q)$. So $P = eps(q)$.

Now, since P contains p, then $p \in eps(q)$. But, given the definition of eps, this means that, in the original NDFSM M, p is reachable from q just by following ε-transitions. So $(q, w) \vdash_M^* (p, \varepsilon)$.

Base Case

- *only if* part: We need to show:

  $[(q, w) \vdash_M{}^* (p, \varepsilon)] \rightarrow [(eps(q), w) \vdash_{M'}{}^* (P, \varepsilon) \text{ and } p \in P]$

  If $|w| = 0$ and the original machine *M* goes from *q* to *p* with only *w* as input, it must go from *q* to *p* following just ε-transitions. So $p \in eps(q)$.

  M' starts in *eps*(*q*). Since *M'* contains no ε-transitions, it will make no moves at all if its input is ε. So it will halt in exactly the same state it started in, namely *eps*(*q*). So *P* = *eps*(*q*) and thus contains *p*.

  So *M'* halts in a state that includes *p*.

Induction Step

Let w have length $k + 1$. Then w = zx where $z \in \Sigma^*$ has length k, and $x \in \Sigma$.

Induction assumption. The lemma is true for Z.

So we show that, assuming that *M* and *M'* behave identically for the first *k* characters, they behave identically for the last character also and thus for the entire string of length $k + 1$.

The Definition of δ′

$\delta'(Q, c) = \bigcup\{eps(p) : \exists q \in Q \, ((q, c, p) \in \Delta)\}$

What We Need to Prove

The relationship between:

- The computation of the NDFSM M:

$$(q, w) \mid\!-_{M}^{*} (p, \varepsilon)$$

and

- The computation of the DFSM M':

$$(eps(q), w) \mid\!-_{M'}^{*} (P, \varepsilon) \text{ and } p \in P$$

What We Need to Prove

Rewriting w as zx:

- The computation of the NDFSM M:

$$(q, zx) \mid\!-_{M}^{*} (p, \varepsilon)$$

and

- The computation of the DFSM M':

$$(eps(q), zx) \mid\!-_{M'}^{*} (P, \varepsilon) \text{ and } p \in P$$

What We Need to Prove

Breaking w into two pieces:

- The computation of the NDFSM *M*:

$$(q, zx) \vdash_M^* (s_i, x) \vdash_M (p, \varepsilon)$$

and

- The computation of the DFSM *M'*:

$$(eps(q), zx) \vdash_{M'}^* (Q, x) \vdash_{M'} (P, \varepsilon) \text{ and } p \in P$$

In other words, after processing z, M will be in some set of states S, whose elements we write as s_i. M' will be in some "set" state that we call Q. Again, well split the proof into two parts:

If Part

We must prove:

$$[\ (eps(q), zx) \vdash_{M'}^* (Q, x) \vdash_{M'} (P, \varepsilon) \text{ and } p \in P\ ] \rightarrow$$
$$[\ (q, zx) \vdash_M^* (s_i, x) \vdash_M (p, \varepsilon)\ ].$$

If, after reading *z*, *M'* is in state *Q*, we know, from the induction hypothesis, that the original machine *M*, after reading *z*, must be in some set of states *S* and that *Q* is precisely that set.

If we have that *M'*, starting in *Q* and reading *x* lands in *P*, then, from the definition of δ', *P* contains precisely the states that *M* could land in after starting in any state in *S* and reading *x*. Thus if $p \in P$, *p* must be a state that *M* could land in if started in s_i on reading *x*.

Only If Part

We must prove:

$$[\ (q, zx) \mid\!-_M^* \ (s_i, x) \mid\!-_M (p, \varepsilon)\ ] \rightarrow$$
$$[\ (eps(q), zx) \mid\!-_{M'}^* (Q, x) \mid\!-_{M'} (P, \varepsilon) \text{ and } p \in P\ ].$$

By the induction hypothesis, if *M*, after processing *z*, can reach some set of states *S*, then *Q* (the state M' is in after processing z) must contain precisely all the states in *S*. So, from *Q*, reading *x*, *M'* must be in some set state *P* that contains precisely the states that *M* can reach starting in any of the states in *S*, reading *x*, and then following all ε transitions. So, after consuming *zx*, *M'*, when started in *eps*(*q*), must end up in a state *P* that contains all and only the states *p* that *M*, when started in *q*, could end up in.

Back to the Theorem

If $w \in L(M)$ then:

- The original machine *M*, when started in its start state, can consume *w* and end up in an accepting state.

- $(eps(s), w) \mid\!-_{M'}^* (Q, \varepsilon)$ for some *Q* containing some $a \in A$. In the statement of the lemma, let *q* equal *s* and $p = a$ for some $a \in A$. Then *M'*, when started in its start state, *eps*(*s*), will consume *w* and end in a state that contains *a*. But if *M'* does that, then it has ended up in one of its accepting states (by the definition of *A'* in step 5 of the algorithm). So *M'* accepts *w* (by the definition of what it means for a machine to accept a string).

Back to the Theorem

If $w \notin L(M)$ (i.e. the original NDFSM does not accept w):

- The original machine *M*, when started in its start state, will not be able to end up in an accepting state after reading *w*.

- If $(eps(s), w) \vdash_{M'}^{*} (Q, \varepsilon)$, then Q contains no state $a \in A$. This follows directly from the lemma.

The two cases, taken together, show that M' accepts exactly the same strings that M accepts.