MA/CSSE 474

Theory of Computation

NDFSM→DFSM Proof
Minimize DFSM States

Exam 1: Session 12 (Dec 16)

- Resources allowed:
  - one double-sided 8.5 x 11 sheet of paper.
  - No books or electronic devices, especially devices with headphones/earbuds.
- Textbook coverage:
  - Chapters 1-4
  - Sections 5.1-5.7
  - Appendices A and C
- Covers HW 1-4 also

Recap: Nondeterministic and Deterministic FSMs

Clearly: {Languages accepted by some DFSM}

$\subseteq$

{Languages accepted by some NDFSM}

More interestingly:

***Theorem*:**

For each NDFSM, there is an equivalent DFSM.

"equivalent" means "accepts the same language"

Recap: NDFSM → DFSM Construction

***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

$K' = \mathcal{P}(K)$ (a.k.a. 2^K)
$s' = eps(s)$
$A' = \{Q \subseteq K : Q \cap A \neq \emptyset\}$

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

Recap: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 $\delta'$:
        3.1 active-states = {$s'$}.
        3.2 $\delta' = \varnothing$.
        3.3 While there exists some element $Q$ of active-states for
              which $\delta'$ has not yet been computed do:
                For each character $c$ in $\Sigma_M$ 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 $\delta'$.
                  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.

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.

Q2

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

M is the NDFSM, M' is the constructed DFSM

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:

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

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:

M',
- when started in the state q' that corresponds to the set of states that the original machine M could get to from q without consuming any input,
- reads the string w and ands in a state P (which is some 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 "state-set" that contains only states that NDFSM M could get to from q after reading w and then 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 we know what happens one step at a time, we will prove the lemma by induction on $|w|$.

Base Case

- *if* part: Prove:

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

which is the same as:

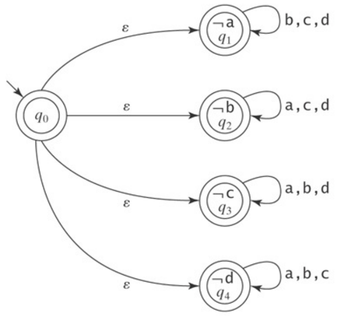
$[\ (eps(q), w) \vdash_{M'}^{*} (P, \varepsilon)$ and $p \in P\ ] \rightarrow [\ (q, w) \vdash_{M}^{*} (p, \varepsilon)\ ]$

- *only if* part: Prove

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

The Number of States May Grow Exponentially

|Σ| = *n*



No. of states after 0 chars: = 1

No. of new states after 1 char: $\binom{n}{n-1}$ = *n*

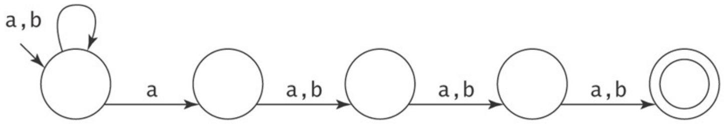
No. of new states after 2 chars: $\binom{n}{n-2}$ = $n(n-1)/2$

No. of new states after 3 chars: $\binom{n}{n-3}$ = $n(n-1)(n-2)/6$

Total number of states after *n* chars: 2^n

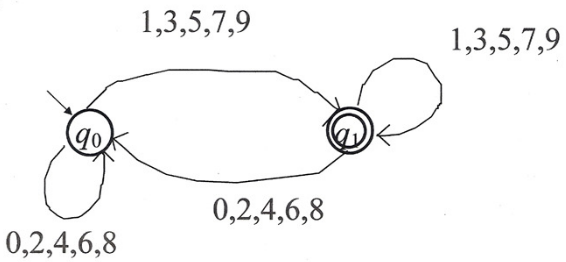
Another Example

$L = \{w \in$ {`a`, `b`}* :
the fourth to the last character is `a`}



If the Original FSM is Deterministic

M =



1. Compute the *eps*(*q*)s:
2. s' = *eps*(*q*0) =
3. Compute δ'

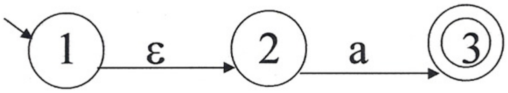
({*q*0}, odd, {*q*1})	({*q*0}, even, {*q*0})
({*q*1}, odd, {*q*1})	({*q*1}, even, {*q*0})

4. *K'* = {{*q*0}, {*q*1}}
5. A' = { {*q*1} }

M' = *M*

The Real Meaning of “Determinism”

Let *M* =

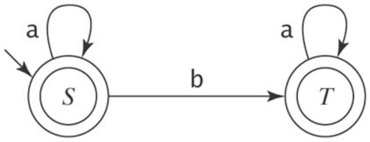


Is *M* deterministic?

An FSM is ***deterministic***, in the most general definition of determinism, if, for each input and state, there is at most one possible transition.

- DFSMs are always deterministic. Why?
- NDFSMs can be deterministic (even with ε-transitions and implicit dead states), but the formalism allows nondeterminism, in general.
- Determinism implies uniquely defined machine behavior.

Deterministic FSMs as Algorithms



```
until accept or reject do:
   S:   s = get-next-symbol
        if s = end-of-input then accept
        else if s = a then go to S
        else if s = b then go to T
   T:   s = get-next-symbol
        if s = end-of-file then accept
        else if s = a then go to T
        else if s = b then reject
        end
```

Deterministic FSMs as Algorithms

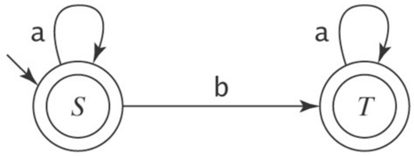
```
until accept or reject do:
   S:   s = get-next-symbol
        if s = end-of-file then accept
        else if s = a then go to S
        else if s = b then go to T
   T:   s = get-next-symbol
        if s = end-of-file then accept
        else if s = a then go to T
        else if s = b then reject
        end
```

Length of Program: $|K| \times (|\Sigma| + 2)$
Time required to analyze string w: $\mathcal{O}(|w| \times |\Sigma|)$

We have to write new code for every new FSM.

A Deterministic FSM Interpreter

```
dfsmsimulate(M: DFSM, w: string) =
    1. $st = s$.
    2. Repeat
        2.1 $c$ = get-next-symbol($w$).
        2.2 If $c \neq$ end-of-file then
            2.2.1 $st = \delta(st, c)$.
      until $c$ = end-of-file.
    3. If $st \in A$ then accept else reject.
```



Input: `aabaa`

Nondeterministic FSMs as Algorithms

Real computers are deterministic, so we have some choices in how to to execute a NDFSM:

1. Convert the NDFSM to a deterministic one:
    - Conversion can take time and space $2^{|K|}$.
    - Time to analyze string *w*: $\mathcal{O}(|w|)$
2. Simulate the behavior of the nondeterministic one by constructing sets of states "on the fly" during execution
    - No conversion cost
    - Time to analyze string *w*: $\mathcal{O}(|w| \times |K|^2)$
3. Do a depth-first search of all paths through the nondeterministic machine.

A NDFSM Interpreter

```
ndfsmsimulate($M = (K, \Sigma, \Delta, s, A)$: NDFSM, $w$: string) =
   1. Declare the set $st$.
   2. Declare the set $st1$.
   3. $st = eps(s)$.
   4. Repeat
          4.1 $c$ = get-next-symbol($w$).
          4.2 If $c \neq$ end-of-file then do
           4.2.1 $st1 = \varnothing$.
           4.2.2 For all $q \in st$ do
                  4.2.2.1 For all $r \in \Delta(q, c)$ do
                         4.2.2.1.1 $st1 = st1 \cup eps(r)$.
           4.2.3 $st = st1$.
           4.2.4 If $st = \varnothing$ then exit.
      until $c$ = end-of-file.
   6. If $st \cap A \neq \varnothing$ then accept else reject.
```

Continue from Day 9
Finite State Machines

State Minimization

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

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.

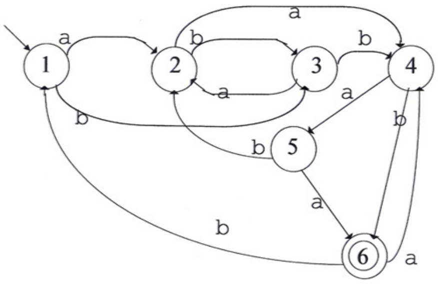
The number of equivalence classes of $\approx_L$ is finite → L regular: If the cardinality of $\approx_L$ is finite, then the construction that was described in the proof of the previous theorem will build an FSM that accepts *L*. So *L* must be regular.

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.

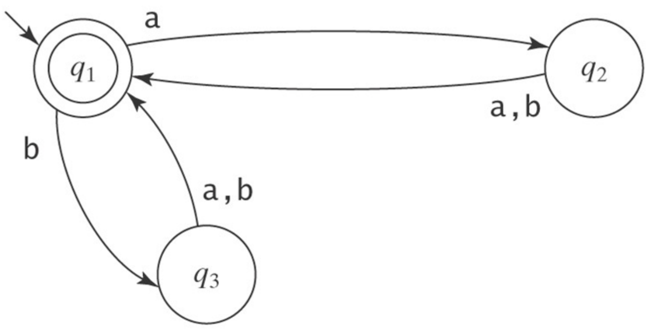
The Overclustering Approach

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

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 $\delta_{M^*}$ is constructed as follows:
            if $\delta_M(q, c) = p$, then $\delta_{M^*}([q], c) = [p]$
```

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:

$$\mathit{buildFSMcanonicalform}(M_1) = \mathit{buildFSMcanonicalform}(M_2)$$

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