CS 622
Computing with NFAs
Wednesday, February 21, 2024
UMass Boston CS
Announcements

• HW 2 in
  • Due Wed 2/21 12pm EST (noon)

• HW 3 out
  • Due Mon 3/4 12pm EST (noon)
HW 1 Observations

- Problems must be assigned to the correct pages
- Proof format must be a **Statements** and **Justifications** table
- Machine formal descriptions must have a tuple
How to ask for HW help
(there’s no such thing as a stupid question, but ...)

... there is such thing as a less useful question (gets less useful answers)

- “Is this correct?”
- “I don’t get it”
- “Give me a hint?”
- “Do I need to do the thing DFA thing?”

Useful question examples (gets useful answers):
- “I think string xyz and zyx is in language A but I’m not sure? Can you clarify?”
- “I’m don’t understand this notation $A \otimes B \gg C$ ... and I couldn’t find it in the book”
- “I couldn’t this word’s definition ...”
- “I know I want to change the machine to add an accept state that ... but I can’t figure out how to write it formally. Hint?”
Concatenation of Languages

Let the alphabet $\Sigma$ be the standard 26 letters $\{a, b, \ldots, z\}$.

If $A = \{\text{fort, south}\}$ and $B = \{\text{point, boston}\}$

$$A \circ B = \{\text{fortpoint, fortboston, southpoint, southboston}\}$$
Is Concatenation Closed?

**Theorem**

The class of regular languages is closed under the concatenation operation.

In other words, if $A_1$ and $A_2$ are regular languages then so is $A_1 \circ A_2$.

- **Cannot** combine $A_1$ and $A_2$’s machine to make a DFA because:
  - Unclear when to switch? (can only read input once)
- Need a **different kind of machine**!
Nondeterministic Finite Automata (NFA)

**Definition**

A *nondeterministic finite automaton* is a 5-tuple \((Q, \Sigma, \delta, q_0, F)\), where:

1. \(Q\) is a finite set of states,
2. \(\Sigma\) is a finite alphabet,
3. \(\delta: Q \times \Sigma \rightarrow \mathcal{P}(Q)\) is the transition function, mapping one state and label to a set of states,
4. \(q_0 \in Q\) is the start state, and
5. \(F \subseteq Q\) is the set of accept states.

\[\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}\]

---

- Transition function maps one state and label to a set of states.
- Transition label can be “empty”, \(\varepsilon\), reused as a transition label (i.e., an argument to \(\delta\)).
- It’s not the empty string.
- And, it’s (still) not a character in alphabet \(\Sigma\)!
Deterministic vs Nondeterministic

Deterministic computation

- start
- states
- ...
- accept or reject

DFAs
Deterministic vs Nondeterministic

Deterministic computation
- start
- states
- reject
- accept or reject

Nondeterministic computation
- states
- reject
- NFA

Previously

Nondeterministic computation can be in multiple states at the same time
NFA Computation (JFLAP demo): 010110
NFA Computation Sequence (of set of states)

Symbol read

0

1

0

1

0

NFA accepts input if: at least one path ends in accept state

Each step can branch into multiple states at the same time!

So this is an accepting computation
DFA Computation Rules

**Informally**

Given
- A DFA (~ a “Program”)
- and Input = string of chars, e.g. “1101”

A DFA computation (~ “Program run”):
- **Start** in start state

  - Repeat:
    - Read 1 char from Input, and
    - Change state according to transition rules

Result of computation:
- **Accept** if last state is Accept state
- **Reject** otherwise

**Formally (i.e., mathematically)**

- \( M = (Q, \Sigma, \delta, q_0, F) \)
- \( w = w_1w_2 \cdots w_n \)

A DFA computation is a sequence of states:

- specified by \( \hat{\delta}(q_0, w) \) where:

  - \( M \) accepts \( w \) if \( \hat{\delta}(q_0, w) \in F \)
  - \( M \) rejects otherwise
DFA Computation Rules

**Informally**

Given

- A DFA (~ a “Program”)
- and Input = string of chars, e.g. “1101”

A DFA computation (~ “Program run”):

- **Start** in start state

**Repeat:**

- Read 1 char from Input, and
- Change state according to transition rules

**Result** of computation:

- Accept if last state is Accept state
- Reject otherwise

**Formally (i.e., mathematically)**

- \( M = (Q, \Sigma, \delta, q_0, F) \)
- \( w = w_1w_2 \cdots w_n \)

A DFA computation is a sequence of states:

- specified by \( \hat{\delta}(q_0, w) \) where:

  - \( M \text{ accepts } w \) if \( \hat{\delta}(q_0, w) \in F \)
  - \( M \text{ rejects } w \) otherwise
**Informally**

Given

- An NFA (~ a “Program”)
- and Input = string of chars, e.g. “1101”

An NFA computation (~ “Program run”):

- **Start** in start state

**Repeat:***

- Read 1 char from Input, and
- For each “current” state, go to next states according to transition rules
- ... then combine all “next states”

**Result** of computation:

- Accept if last set of states has accept state
- Reject otherwise

**Formally (i.e., mathematically)**

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

An NFA computation is a ... specified by $\hat{\delta}(q_0, w)$ where:

- $M$ accepts $w$ if ...
- $M$ rejects ...
Informally

Given

- An **NFA** (~ a “Program”)
- and Input = string of chars, e.g. “1101”

A **DFA computation** (~ “Program run”):

- **Start** in **start state**

**Repeat:**

- Read 1 char from Input, and according to **transition rules**

For each “current” state, go to **next states**

... then combine all “next states”

**Result of computation:**

- **Accept** if last set of states has accept state
- **Reject** otherwise

Formally (i.e., mathematically)

- \( M = (Q, \Sigma, \delta, q_0, F) \)
- \( w = w_1 w_2 \cdots w_n \)

An **NFA computation** is a **sequence of:**

- sets of states

specified by \( \hat{\delta}(q_0, w) \) where:

***

- \( M \) accepts \( w \) if ...
- \( M \) rejects ...

Ignoring \( \varepsilon \) transitions, for now!
DFA Extended Transition Function

\[ \hat{\delta} : Q \times \Sigma^* \rightarrow Q \]

- **Domain** (inputs):
  - state \( q \in Q \) (doesn’t have to be start state)
  - string \( w = w_1 w_2 \cdots w_n \) where \( w_i \in \Sigma \)

- **Range** (output):
  - state \( q \in Q \) (doesn’t have to be an accept state)

**Base case**

\[ \hat{\delta}(q, \varepsilon) = \]
DFA Extended Transition Function

\[ \hat{\delta} : Q \times \Sigma^* \rightarrow Q \]

- **Domain** (inputs):
  - state \( q \in Q \) (doesn’t have to be start state)
  - string \( w = w_1 w_2 \cdots w_n \) where \( w_i \in \Sigma \)
- **Range** (output):
  - state \( q \in Q \) (doesn’t have to be an accept state)

(Defined recursively)

Base case \( \hat{\delta}(q, \varepsilon) = q \)

Recursive Case \( \hat{\delta}(q, w'w_n) = \delta(\hat{\delta}(q, w'), w_n) \)

where \( w' = w_1 \cdots w_{n-1} \)

A String is either:
- the empty string \( \varepsilon \), or
- \( xa \) (non-empty string) where
  - \( x \) is a string
  - \( a \) is a “char” in \( \Sigma \)
DFA Extended Transition Function

\[ \hat{\delta} : Q \times \Sigma^* \rightarrow Q \]

- **Domain** (inputs):
  - state \( q \in Q \) (doesn’t have to be start state)
  - string \( w = w_1w_2 \cdots w_n \) where \( w_i \in \Sigma \)
- **Range** (output):
  - state \( q \in Q \) (doesn’t have to be an accept state)

(Defined recursively)

**Base case**

\[ \hat{\delta}(q, \varepsilon) = q \]

**Recursive Case**

\[ \hat{\delta}(q, w'w_n) = \hat{\delta}(\hat{\delta}(q, w'), w_n) \]

where \( w' = w_1 \cdots w_{n-1} \)

\( \delta: Q \times \Sigma \rightarrow Q \) is the transition function

---

**Recursive Input Data needs Recursive Function**

**A String is either:**
- the **empty string** \((\varepsilon)\), or
- \( xa \) (non-empty string) where
  - \( x \) is a string
  - \( a \) is a “char” in \( \Sigma \)

**Single step from “second to last” state and last char gets to last state**
Extended Transition Function

\( \hat{\delta} : Q \times \Sigma^* \rightarrow \mathcal{P}(Q) \)

- **Domain** (inputs):
  - state \( q \in Q \) (doesn’t have to be start state)
  - string \( w = w_1 w_2 \cdots w_n \) where \( w_i \in \Sigma \)

- **Range** (output):
  - states \( q_s \subseteq Q \)

\( \delta : Q \times \Sigma_\varepsilon \rightarrow \mathcal{P}(Q) \) is the transition function
Extended Transition Function

\[ \hat{\delta}: Q \times \Sigma^* \rightarrow \mathcal{P}(Q) \]

- **Domain** (inputs):
  - state \( q \in Q \) (doesn’t have to be start state)
  - string \( w = w_1 w_2 \cdots w_n \) where \( w_i \in \Sigma \)

- **Range** (output):
  - states \( Q_s \subseteq Q \)

(Defined recursively)

**Base case**

\[ \hat{\delta}(q, \varepsilon) = \{q\} \]

\( \delta: Q \times \Sigma \rightarrow \mathcal{P}(Q) \) is the transition function

Recursively Defined Input needs Recursive Function

A String is either:
- the **empty string** \((\varepsilon)\), or
- \( xa \) (non-empty string) where
  - \( x \) is a **string**
  - \( a \) is a “char” in \( \Sigma \)
Extended Transition Function

\[ \hat{\delta} : Q \times \Sigma^* \rightarrow \mathcal{P}(Q) \]

- **Domain** (inputs):
  - state \( q \in Q \) (doesn’t have to be start state)
  - string \( w = w_1 w_2 \cdots w_n \) where \( w_i \in \Sigma \)
- **Range** (output):
  - states \( q_s \subseteq Q \)

(Defined recursively)

**Base case**

\[ \hat{\delta}(q, \varepsilon) = \{q\} \]

**Recursive Case**

\[
\hat{\delta}(q, w'w_n) = \hat{\delta}(q', w_n)
\]

where \( w' = w_1 \cdots w_{n-1} \)

\[ \hat{\delta}(q, w') = \{q_1, \ldots, q_k\} \]
NFA

Extended Transition Function

\[ \hat{\delta} : Q \times \Sigma^* \rightarrow \mathcal{P}(Q) \]

- **Domain** (inputs):
  - state \( q \in Q \) (doesn’t have to be start state)
  - string \( w = w_1w_2 \cdots w_n \) where \( w_i \in \Sigma \)
- **Range** (output):
  - states \( qs \subseteq Q \)

(Defined recursively)

Base case
\[ \hat{\delta}(q, \varepsilon) = \{q\} \]

Recursive Case
\[ \hat{\delta}(q, w_1 \cdots w_n) = \bigcup_{i=1}^{k} \hat{\delta}(q_i, w_n) \]

where \( w' = w_1 \cdots w_{n-1} \)

\[ \hat{\delta}(q, w') = \{q_1, \ldots, q_k\} \]

\( \delta : Q \times \Sigma \varepsilon \rightarrow \mathcal{P}(Q) \) is the transition function

Recursively Defined Input needs Recursive Function

A String is either:
- the **empty string** (\( \varepsilon \)), or
- **\( xa \)** (non-empty string) where
  - \( x \) is a string
  - \( a \) is a “char” in \( \Sigma \)

For each “second to last” state, take single step on last char

Last char
Extended Transition Function

\[ \hat{\delta}: Q \times \Sigma^* \rightarrow \mathcal{P}(Q) \]

**Given**
- An NFA (~ a “Program”)
- and Input = string of chars, e.g. “1101”

**A DFA computation** (~ “Program run”):
- **Start** in start state
- **Repeat:**
  - Read 1 char from Input, and according to transition rules
  - For each “current” state, go to next states...
  - ... then combine all sets of “next states”

**Recursive Case**

\[ \hat{\delta}(q, w' w_n) = \bigcup_{i=1}^{k} \hat{\delta}(q_i, w_n) \]

where \( w' = w_1 \cdots w_{n-1} \)

\[ \hat{\delta}(q, w') = \{q_1, \ldots, q_k\} \]

\[ \delta: Q \times \Sigma \rightarrow \mathcal{P}(Q) \text{ is the transition function} \]
NFA Extended $\delta$ Example

$\hat{\delta}(q_0, \epsilon) = \{q_0\}$

$\hat{\delta}(q_0, 0) = \delta(q_0, 0) = \{q_0, q_1\}$

$\hat{\delta}(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$

$\hat{\delta}(q_0, 001) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$

Base case:
$\hat{\delta}(q, \epsilon) = \{q\}$

Recursive case:
$\hat{\delta}(q, w) = \bigcup_{i=1}^{k} \delta(q, w_n)$

where:
$\hat{\delta}(q, w_1 \cdots w_{n-1}) = \{q_1, \ldots, q_k\}$

We haven’t considered empty transitions!

Combine result of recursive call with “last step”
Adding Empty Transitions

- Define the set $\varepsilon$-REACHABLE($q$)
  - ... to be all states reachable from $q$ via zero or more empty transitions

(Defined recursively)

- **Base case:** $q \in \varepsilon$-REACHABLE($q$)

- **Inductive case:**
  
  $\varepsilon$-REACHABLE($q$) = \{ $r$ | $p \in \varepsilon$-REACHABLE($q$) and $r \in \delta(p, \varepsilon)$ \}
\( \varepsilon\text{-REACHABLE} \) Example

\[ \varepsilon\text{-REACHABLE}(1) = \{1, 2, 3, 4, 6\} \]
Extended Transition Function

\[ \hat{\delta} : Q \times \Sigma^* \rightarrow \mathcal{P}(Q) \]

- **Domain** (inputs):
  - state \( q \in Q \) (doesn’t have to be start state)
  - string \( w = w_1 w_2 \cdots w_n \) where \( w_i \in \Sigma \)

- **Range** (output):
  - states \( q S \subseteq Q \)

(Defined recursively)

**Base case**

\[ \hat{\delta}(q, \varepsilon) = \varepsilon\text{-REACHABLE}(q) \]

**Recursive Case**

\[ \hat{\delta}(q, w' w_n) = \bigcup_{i=1}^{k} \hat{\delta}(q_i, w_n) \]

where \( w' = w_1 \cdots w_{n-1} \)

\[ \hat{\delta}(q, w') = \{q_1, \ldots, q_k\} \]
Extended Transition Function

\[ \hat{\delta} : Q \times \Sigma^* \rightarrow \mathcal{P}(Q) \]

- **Domain** (inputs):
  - state \( q \in Q \) (doesn’t have to be start state)
  - string \( w = w_1w_2\cdots w_n \) where \( w_i \in \Sigma \)
- **Range** (output):
  - states \( q_s \subseteq Q \)

(Defined recursively)

**Base case**

\[ \hat{\delta}(q, \varepsilon) = \varepsilon\text{-REACHABLE}(q) \]

**Recursive Case**

\[ \hat{\delta}(q, w'w_n) = \varepsilon\text{-REACHABLE}\left( \bigcup_{i=1}^{k} \delta(q_i, w_n) \right) \]

where \( w' = w_1 \cdots w_{n-1} \)

\[ \hat{\delta}(q, w') = \{q_1, \ldots, q_k\} \]

“Take single step, then follow all empty transitions”
Summary: NFA vs DFA Computation

**DFAs**
- Can only be in **one** state
- Transition:
  - Must read 1 char
- Acceptance:
  - If final state is accept state

**NFAs**
- Can be in **multiple** states
- Transition
  - Has empty transitions
- Acceptance:
  - If **one** of final states is accept state
Is Concatenation Closed?

**THEOREM**

The class of regular languages is closed under the concatenation operation.

In other words, if $A_1$ and $A_2$ are regular languages then so is $A_1 \circ A_2$.

**Proof requires:** Constructing new machine

- How does it know when to switch machines?
- Can only read input once
Let $M_1$ recognize $A_1$, and $M_2$ recognize $A_2$.

Want: Construction of $N$ to recognize $A_1 \circ A_2$.

$N$ is an NFA! It can:
- Keep checking 1st part with $M_1$ and
- Move to $M_2$ to check 2nd part

$\varepsilon = \text{"empty transition"} = \text{reads no input}$

Allows $N$ to be in both machines at the same time!
Concatenation is Closed for Regular Langs

**Proof** (part of)

Let $\mathcal{M}_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize $A_1$

$\mathcal{M}_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize $A_2$

Construct $N = (Q, \Sigma, \delta, q_1, F)$ to recognize $A_1 \circ A_2$

1. $Q = Q_1 \cup Q_2$
2. The state $q_1$ is the same as the start state of $M_1$
3. The accept states $F_2$ are the same as the accept states of $M_2$
4. Define $\delta$ so that for any $q \in Q$ and any $a \in \Sigma$, $\delta(q, a) = \delta_1(q, a)$ if $q \in Q_1$ and $\delta_2(q, a)$ if $q \in Q_2$. Additionally, $\delta(q_1, \epsilon) = F_2$. 

![Diagram showing the construction of M1, M2, and N]
Concatenation is Closed for Regular Langs

**Proof** (part of)

Let \( M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1) \) recognize \( A_1 \)

DFA \( M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2) \) recognize \( A_2 \)

Construct \( N = (Q, \Sigma, \delta, q_1, F) \) to recognize \( A_1 \circ A_2 \)

1. \( Q = Q_1 \cup Q_2 \)
2. The state \( q_1 \) is the same as the start state of \( M_1 \)
3. The accept states \( F_2 \) are the same as the accept states of \( M_2 \)
4. Define \( \delta \) so that for any \( q \in Q \) and any \( a \in \Sigma \),

\[
\delta(q, a) = \begin{cases} 
\{ \delta_1(q, a) \} & q \in Q_1 \text{ and } q \notin F_1 \\
\{ \delta_2(q, a) \} & q \in Q_2 \\
\{ q \} & q \in F_1 \text{ and } a \neq \varepsilon \\
\varnothing & q \in F_1 \text{ and } a = \varepsilon
\end{cases}
\]

And: \( \delta(q, \varepsilon) = \varnothing \), for \( q \in Q, q \notin F_1 \)
## Is Union Closed For Regular Langs?

**Proof**

### Statements

1. $A_1$ and $A_2$ are regular languages
2. A DFA $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognizes $A_1$
3. A DFA $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognizes $A_2$
4. Construct DFA $M = (Q, \Sigma, \delta, q_0, F)$
5. $M$ recognizes $A_1 \cup A_2$
6. $A_1 \cup A_2$ is a regular language
7. The class of regular languages is closed under the union operation.

### Justifications

1. Assumption
2. Def of Reg Lang (Coro)
3. Def of Reg Lang (Coro)
4. Def of DFA
5. See examples
6. Def of Regular Language
7. From stmt #1 and #6

In other words, if $A_1$ and $A_2$ are regular languages, so is $A_1 \cup A_2$. Q.E.D.
Is Concat Closed For Regular Langs?

Proof?

**Statements**
1. $A_1$ and $A_2$ are regular languages
2. A DFA $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognizes $A_1$
3. A DFA $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognizes $A_2$
4. Construct an NFA $N = (Q, \Sigma, \delta, q_0, F)$
5. $N$ recognizes $A_1 \cup A_2$, $A_1 \circ A_2$
6. $A_1 \cup A_2$, $A_1 \circ A_2$ is a regular language
7. The class of regular languages is closed under concatenation operation.

**Justifications**
1. Assumption
2. Def of Reg Lang (Coro)
3. Def of Reg Lang (Coro)
4. Def of NFA
5. See examples
6. ??? Does NFA recognize reg langs?
7. From stmt #1 and #6

Q.E.D.?
A DFA’s Language

- For DFA $M = (Q, \Sigma, \delta, q_0, F')$
- $M$ accepts $w$ if $\hat{\delta}(q_0, w) \in F$
- $M$ recognizes language $\{w \mid M \text{ accepts } w\}$

Definition: A DFA’s language is a regular language
An NFA’s Language?

• For NFA $N = (Q, \Sigma, \delta, q_0, F)$

  • $N$ accepts $w$ if $\hat{\delta}(q_0, w) \cap F \neq \emptyset$

  • i.e., accept if final states contain at least one accept state

• Language of $N = L(N) = \left\{ w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset \right\}$

Q: What kind of languages do NFAs recognize?
Concatenation Closed for Reg Langs?

• Combining DFAs to recognize concatenation of languages ...
  ... produces an **NFA**

• So to prove concatenation is closed ...
  ... we must prove that **NFAs also recognize regular languages.**

Specifically, we must prove:
**NFAs ↔ regular languages**
“If and only if” Statements

\[ X \leftrightarrow Y = \text{"X if and only if Y"} = X \text{ iff } Y = X \iff Y \]

Represents two statements:

1. \( \Rightarrow \) if \( X \), then \( Y \)
   - "forward" direction

2. \( \Leftarrow \) if \( Y \), then \( X \)
   - "reverse" direction
How to Prove an “iff” Statement

\[ X \leftrightarrow Y = \text{“}X \text{ if and only if } Y\text{”} = X \text{ iff } Y = X \iff Y \]

Proof has two (If-Then proof) parts:

1. \( \implies \) if \( X \), then \( Y \)
   - “forward” direction
   - assume \( X \), then use it to prove \( Y \)

2. \( \iff \) if \( Y \), then \( X \)
   - “reverse” direction
   - assume \( Y \), then use it to prove \( X \)