CS420
Computing With Finite Automata

Tuesday, September 13, 2022
UMass Boston Computer Science
Announcements

• HW 0 in extended, due Wed 11:59pm

• HW 1 released soon

• Please ask all HW questions on Piazza!
  • So all course staff can see,
  • and entire class can benefit
  • Please do not directly email course staff with HW questions

• TA: Sean Rasku-Casas
  • Office Hours Mondays 12:30-2pm, in the TA room (McCormack 3rd floor)
Last Time: Finite Automata Formal Definition

**Definition**

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

1. \(Q\) is a finite set called the **states**, 
2. \(\Sigma\) is a finite set called the **alphabet**, 
3. \(\delta: Q \times \Sigma \rightarrow Q\) is the **transition function**, 
4. \(q_0 \in Q\) is the **start state**, and 
5. \(F \subseteq Q\) is the **set of accept states**.

**Also called a Deterministic Finite Automata (DFA)**

(Will be important later)
Informally

- "Program" = a finite automata
- Input = string of chars, e.g. "1101"

To run a "program":
- Start in “start state”
- Repeat:
  - Read 1 char;
  - Change state according to the transition table
- Result =
  - Accept if last state is “Accept” state
  - Reject otherwise

Formally (i.e., mathematically)

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

Define variables \( r_i \), \( i = 0 \ldots n \), representing sequence of states in the computation

- \( r_0 = q_0 \)
  
  e.g., \( i = 1 \), \( r_1 = \delta(r_0, w_1) \) \( r_2 = \delta(r_1, w_2) \) ...

- \( r_i = \delta(r_{i-1}, w_i) \), for \( i = 1, \ldots, n \)

Let’s come up with nicer notation to represent this part

- \( M \) accepts \( w \) if
  sequence of states \( r_0, r_1, \ldots, r_n \) in \( Q \) exists ...
  with \( r_n \in F \)

This is still a little verbose / informal
An Extended Transition Function

Define **extended transition function**:

- **Domain**:
  - Beginning state $q \in Q$ (not necessarily the start state)
  - Input string $w = w_1w_2 \cdots w_n$ where $w_i \in \Sigma$
- **Range**:
  - Ending state (not necessarily an accept state)

(Defined recursively)

- **Base case**: ...
Recursive Definitions

```javascript
function factorial( n )
{
    if ( n == 0 )
        return 1;
    else
        return n * factorial( n - 1 );
}
```

- **Why is this allowed?**
  - It’s a feature (i.e., an axiom) of the programming language

- **Why does this work?**
  - Because the recursive call always has a “smaller” argument ...
  - ... and so eventually reaches the base case and stops
Recursive Definitions

A Natural Number is either:

- Zero, or
- the Successor of a Natural Number

Examples

- Zero
- Successor of Zero ( = “one” )
- Successor of Successor of Zero ( = “two” )
- Successor of Successor of Successor of Successor of Zero ( = “three” ) ...
An Extended Transition Function

Define **extended transition function**: |

- **Domain:**
  - Beginning state $q \in Q$ (not necessarily the start state)
  - Input string $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- **Range:**
  - Ending state (not necessarily an accept state)

(Defined recursively)

- **Base case:** $\hat{\delta}(q, \varepsilon) = q$
- **Recursive case:** $\hat{\delta}(q, w) = \hat{\delta}(\hat{\delta}(q, w_1), w_2 \cdots w_n)$
# FSM Computation Model

<table>
<thead>
<tr>
<th>Informally</th>
<th>Formally (i.e., mathematically)</th>
</tr>
</thead>
</table>
| - “Program” = a finite automata  
  - Input = string of chars, e.g. “1101” | - $M = (Q, \Sigma, \delta, q_0, F)$  
  - $w = w_1 w_2 \cdots w_n$ |
| To run a “program”:  
  - Start in “start state” | - $r_0 = q_0$  
  - $r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$ |
| **Repeat:**  
  - Read 1 char;  
  - Change state according to the transition table | Let’s come up with nicer notation to represent this part  
  - $M$ accepts $w$ if  
    sequence of states $r_0, r_1, \ldots, r_n$ in $Q$ exists . . .  
    with $r_n \in F$ |
| Result =  
  - “Accept” if last state is “Accept” state  
  - “Reject” otherwise |
FSM Computation Model

**Informally**

- “Program” = a finite automata
- **Input** = string of chars, e.g. “1101”

To run a “program”:
- **Start** in “start state”

- **Repeat:**
  - Read 1 char;
  - Change state according to the **transition** table

- **Result** =
  - “Accept” if last state is “Accept” state
  - “Reject” otherwise

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

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

- $r_0 = q_0$

- $r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$

- $M$ **accepts** $w$ if $\hat{\delta}(q_0, w) \in F$
  - sequence of states $r_0, r_1, \ldots, r_n$ in $Q$ exists …
    - with $r_n \in F$
Definition of Accepting Computations

An **accepting computation**, for FSM \( M = (Q, \Sigma, \delta, q_0, F) \) and string \( w \):

1. starts in the start state \( q_0 \)
2. goes through a valid sequence of states according to \( \delta \)
   - this implies that all \( w_i \in \Sigma \)
3. ends in an accept state

---

All 3 must be true for a computation to be an **accepting computation**!

\( M \) accepts \( w \) if
\[
\hat{\delta}(q_0, w) \in F
\]
Accepting Computation or Not?

FSM:

\[ \delta(q_1, 1101) \]
\[ \text{yes} \]

\[ \delta(q_1, 110) \]
\[ \text{No (doesn’t end in accept state)} \]

\[ \delta(q_2, 101) \]
\[ \text{No (doesn’t start in start state)} \]

\[ \delta(q_1, 123) \]
\[ \text{No (doesn’t follow delta transition function)} \]
Languages and Strings

• A **language** is a **set** of strings

• A **string** is a **finite sequence** of symbols from an **alphabet**

• An **alphabet** is a **non-empty finite set** of symbols

\[ \Sigma_1 = \{0, 1\} \]

\[ \Sigma_2 = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\} \]
Computation and Languages

• The **language** of a machine is the **set of all strings** that it accepts.

• E.g., An **FSM** $M$ **accepts** $w$ if $\delta(q_0, w) \in F$.

• **Language** of $M = L(M) = \{w \mid M$ accepts $w\}$
Language Terminology

- $M$ accepts $w$ ← string

- $M$ recognizes language $A$ ← Set of strings

if $A = \{ w \mid M$ accepts $w \}$
Computation and Classes of Languages

• The **language** of a machine is the **set of all strings** that it accepts

• A **computation model** is equivalent to the **set of machines** it defines

• E.g., all possible FSMs are a computation model

• Thus: a **computation model** is also equivalent to a **set of languages**
Regular Languages: Definition

If a finite automaton (FSM) recognizes a language, then that language is called a regular language.

A language is a set of strings. If \( A = \{w \mid M \text{ accepts } w\} \), then \( M \) recognizes language \( A \).
A Language, Regular or Not?

• If given: a Finite Automaton $M$
  • We know: $L(M)$, the language recognized by $M$, is a regular language

  If a finite automaton (FSM) recognizes a language, then that language is called a **regular language**.

• If given: a Language $A$
  • Is $A$ a regular language?
    • Not necessarily!
  • How do we determine, i.e., **prove**, that $A$ is a regular language?
Kinds of Mathematical Proof

• Deductive Proof
  • Start with known facts (i.e., premises)
  • Use logical inference rules to reach new conclusions
An Inference Rule: Modus Ponens

Premises
• If $P$ then $Q$
• $P$ is true

Conclusion
• $Q$ must also be true
An Inference Rule: Modus Ponens

Premises
- If $P$ then $Q$
- $P$ is true

Conclusion
- $Q$ must also be true

Example Premises
- If there is an FSM recognizing language $A$, then $A$ is a regular language
- We know an FSM $M$ where $L(M) = A$

Conclusion
- $A$ is a regular language!
A Language, Regular or Not?

• If given: a Finite Automaton $M$
  • We know: $L(M)$, the language recognized by $M$, is a regular language

If a finite automaton (FSM) recognizes a language, then that language is called a regular language.

• If given: a Language $A$
  • Is $A$ is a regular language?
    • Not necessarily!
  • How do we determine, i.e., prove, that $A$ is a regular language?

Create an FSM recognizing $A$!
Designing Finite Automata: Tips

• Input may only be read once, one char at a time

• Must decide accept/reject after that

• States = the machine’s memory!
  • # states must be decided in advance
  • So think about what information must be remembered.

• Every state/symbol pair must have a transition (for DFAs)
Design a DFA: accept strs with odd # 1s

- **States:**
  - 2 states:
    - seen even 1s so far
    - seen odds 1s so far

- **Alphabet:** 0 and 1

- **Transitions:**

- **Start / Accept states:**
In-class exercise

• **Prove:** the following language is a regular language:
  • \( A = \{ w \mid w \text{ has exactly three } 1\text{'s} \} \)
  • i.e., design a finite automata that recognizes it!

• Where \( \Sigma = \{ 0, 1 \} \),

• Remember:

---

**DEFINITION**

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

1. \( Q \) is a finite set called the **states**, 
2. \( \Sigma \) is a finite set called the **alphabet**, 
3. \( \delta: Q \times \Sigma \rightarrow Q \) is the **transition function**, 
4. \( q_0 \in Q \) is the **start state**, and 
5. \( F \subseteq Q \) is the **set of accept states**.
In-class exercise Solution

- Design finite automata recognizing:
  - \{w \mid w \text{ has exactly three 1's}\}

- States:
  - Need one state to represent how many 1's seen so far
  - \(Q = \{q_0, q_1, q_2, q_3, q_{4+}\}\)

- Alphabet: \(\Sigma = \{0, 1\}\)

- Transitions:

  ![Diagram of a finite automaton]

- Start state:
  - \(q_0\)

- Accept states:
  - \(\{q_3\}\)

So finite automata are used to recognize simple string patterns? Yes!

Have you ever used a programming language feature to recognize simple string patterns?
Check-in Quiz 9/13

On gradescope