CS420
(Deterministic) Finite Automata

Thursday, September 8, 2022
UMass Boston Computer Science
Announcements

• Quizzes
  • Quiz 1 returned
  • Use gradescope issue ticket for questions / complaints

• HW
  • Weekly; in/out Sunday midnight
  • HW 0 due Sunday 9/11 11:59pm EST
  • ~4-5 questions, Paper-and-pencil proofs (no programming)
  • Discussing with classmates ok; Final answers written up / submitted individually

• Lectures
  • Not recorded but closely follow the listed textbook chapters

• Office Hours
  • Thurs 12:30-2pm (in person)
  • Fri 4-5:30pm (zoom, access link from blackboard)
  • Let me know if advance if possible, but drop-ins also fine
Last Time: The Theory of Computation ...

Formally defines **mathematical models of computation**

In order to:

1. **Make predictions** (about computer programs)
   - If possible
     ```
     function( x, y, z, n ) {
       if n > 2 && x^n + y^n == z^n {
         printf("hello, world!\n");
       }
     }
     ```

2. **Compare** the models to each other
   - Java vs Python? The same?

3. **Explore the limits** of computation
   - What programs cannot be written?

Fermat's Last Theorem (unknown for ~350 years, solved in 1990s)
Last Time: Computation = Programs!

Intuition for this course:

- A model of computation defines a class of machines (each box)
- Think of: a class of machines = a “Programming Language”!
- Think of: a single machine instance = a “Program”!
Last Time: Computation = Programs!

Very important Note: I use this “programs” and “programming language” analogy to help you understand CS420 topics, by comparing them to ideas you’ve seen before.

But don’t get confused: “programs” and “programming languages” are not formal terms defined in this course.

In fact, the term language will formally mean something else (later).

Intuition for this course:
- A model of computation
- Think of: a class of machines = a “Programming Language”!
- Think of: a single machine instance = a “Program”!
Last Time: Models of Computation Hierarchy

- Turing Machines
- Linear bounded Automata
- Push-down Automata
- Finite State Automata

We’ll start here ...
Finite Automata: “Simple” Computation / “Programs”
Finite Automata

• A finite automata or finite state machine (FSM) ...

• ... computes with a finite number of states
A Microwave Finite Automata

Inputs change states
(possibly)

press stop
press start

press start
press stop

idle
cook

States
Finite Automata: Not Just for Microwaves

Finite Automata: a common programming pattern

State pattern
From Wikipedia, the free encyclopedia

The state pattern is a behavioral software design pattern that allows an object to alter its behavior when its internal state changes. This pattern is close to the concept of finite-state machines. The state pattern can be interpreted as a strategy pattern, which is able to switch a strategy through invocations of methods defined in the pattern's interface.

Computation Simulating Other Computation
(a common theme this semester)
Video Games Love Finite Automata

The basic idea is that a character is engaged in some particular kind of action at any given time. The actions available will depend on the type of gameplay but typical actions include things like idling, walking, running, jumping, etc. These actions are referred to as states, in the sense that the character is in a “state” where it is walking, idling or whatever. In general, the character will have restrictions on the next state it can go to rather than being able to switch immediately from any state to any other. For example, a running jump can only be taken when the character is already running and not when it is at a standstill, so it should never switch straight from the idle state to the running jump state. The options for the next state that a character can enter from its current state are referred to as state transitions. Taken together, the set of states, the set of transitions and the variable to remember the current state form a state machine.

The states and transitions of a state machine can be represented using a graph diagram, where the nodes represent the states and the arcs (arrows between nodes) represent the transitions. You can think of the current state as being a marker or highlight that is placed on one of the nodes and can then only jump to another node along one of the arrows.
Finite Automata in Video Games

```cpp
1 // simple_state_machine.h
2 // Simple finite state machine encapsulation
3 // Author: Michael S. Booth (mike@turtlerockstudios.com), November 2003
4
5 ifndef SIMPLE_STATE_MACHINE_H_
6 #define SIMPLE_STATE_MACHINE_H_
7
8 /**
9 * Encapsulation of a finite-state-machine state
10 */
11
template < typename T >
12 class SimpleState
13 {
14```
Model-view-controller (MVC) is an FSM

- **States**
- **Inputs change states**
- **The View draws states**
A Finite Automata = a “Program”

- A very limited “program” that uses finite memory
  - Actually, only 1 “cell” of memory!
  - States = the possible things that can be written to memory

- Finite Automata has different representations:
  - Code (wont use in this class)
  - State diagrams
Finite Automata state diagram

Start State
States
Accept State
Inputs cause state transitions
A Finite Automata = a “Program”

• A very limited program with finite memory
  • Actually, only 1 “cell” of memory!
  • States = the possible things that can be written to memory

• Finite Automata has different representations:
  • Code
  • State diagrams
  ➢ Formal mathematical description
Finite Automata: The 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**.
Sets and Sequences

- Both are: mathematical objects that group other objects
- **Members** of the group are called **elements**
- Can be: empty, finite, or infinite
- Can contain: other sets or sequences

<table>
<thead>
<tr>
<th>Sets</th>
<th>Sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Unordered</td>
<td>• Ordered</td>
</tr>
<tr>
<td>• Duplicates <strong>not</strong> allowed</td>
<td>• Duplicates ok</td>
</tr>
<tr>
<td>• Common notation: { }</td>
<td>• Common notation: ( ), or just commas</td>
</tr>
<tr>
<td>• “Empty set” denoted: $\emptyset$ or { }</td>
<td>• “Empty sequence”: ( )</td>
</tr>
<tr>
<td>• A <strong>language</strong> is a (possibly infinite) set of strings</td>
<td>• A <strong>tuple</strong> is a finite sequence</td>
</tr>
<tr>
<td></td>
<td>• A <strong>string</strong> is a finite sequence of characters</td>
</tr>
</tbody>
</table>
A \textbf{function} is ... (1^{st} \text{ of each pair from domain, } 2^{nd} \text{ from range})

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

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

\begin{itemize}
  \item A \textbf{pair} is ... (1 element)
  \item A \textbf{sequence} of 2 elements
\end{itemize}

... can write it in many ways: as a \textbf{mapping, a table, ...}

---

**Set or Sequence?**
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**.
**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**.

Example: as **state diagram**
**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**.

**Example:** as **formal description**

\[ M_1 = (Q, \Sigma, \delta, q_1, F) \], where

1. \(Q = \{q_1, q_2, q_3\}\),  
2. \(\Sigma = \{0, 1\}\),  
3. \(\delta\) is described as

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q_1 )</td>
<td>(q_1)</td>
<td>(q_2)</td>
</tr>
<tr>
<td>(q_2 )</td>
<td>(q_3)</td>
<td>(q_2)</td>
</tr>
<tr>
<td>(q_3 )</td>
<td>(q_2)</td>
<td>(q_2)</td>
</tr>
</tbody>
</table>

4. \(q_1\) is the start state, and  
5. \(F = \{q_2\}\).

**Example:** as **state diagram**
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**.

**Example: as formal description**

\[
M_1 = (Q, \Sigma, \delta, q_1, F),
\]

1. \(Q = \{q_1, q_2, q_3\}\),  
2. \(\Sigma = \{0, 1\}\),  
3. \(\delta\) is described as  

\[
\begin{array}{c|cc}
    & 0 & 1 \\
\hline
q_1 & q_1 & q_2 \\
q_2 & q_3 & q_2 \\
q_3 & q_2 & q_2 \\
\end{array}
\]

4. \(q_1\) is the start state, and  
5. \(F = \{q_2\}\).
**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**.

**Example: as formal description**

\[ M_1 = (Q, \Sigma, \delta, q_1, F), \text{ where} \]

1. \(Q = \{q_1, q_2, q_3\}\),
2. \(\Sigma = \{0, 1\}\),
3. \(\delta\) is described as
   
   \[
   \begin{array}{c|cc}
   & 0 & 1 \\
   \hline
   q_1 & q_1 & q_2 \\
   q_2 & q_3 & q_2 \\
   q_3 & q_2 & q_2 \\
   \end{array}
   \]
4. \(q_1\) is the start state, and
5. \(F = \{q_2\}\).

**Example: as state diagram**
**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**.

**Example:** as **formal description**

\[M_1 = (Q, \Sigma, \delta, q_1, F)\], where

1. \(Q = \{q_1, q_2, q_3\}\),  
2. \(\Sigma = \{0, 1\}\),  
3. \(\delta\) is described as

\[
\begin{array}{c|cc}
\delta & 0 & 1 \\
\hline
q_1 & q_1 & q_2 \\
q_2 & q_3 & q_2 \\
q_3 & q_2 & q_2 \\
\end{array}
\]

4. \(q_1\) is the start state, and  
5. \(F = \{q_2\}\).

**Example:** as **state diagram**
**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**.

**Example: as formal description**

\[ M_1 = (Q, \Sigma, \delta, q_1, F), \]

1. \(Q = \{q_1, q_2, q_3\}\),
2. \(\Sigma = \{0, 1\}\),
3. \(\delta\) is described as

\[
\begin{array}{c|cc}
0 & 1 \\
\hline
q_1 & q_1 & q_2 \\
q_2 & q_3 & q_2 \\
q_3 & q_2 & q_2 \\
\end{array}
\]

4. \(q_1\) is the start state, and 
5. \(F = \{q_2\}\).

**Example: as state diagram**
**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**.

**Example:** as a formal description

\[
M_1 = (Q, \Sigma, \delta, q_1, F), \text{ where }
\]

1. \(Q = \{q_1, q_2, q_3\}\), 
2. \(\Sigma = \{0, 1\}\), 
3. \(\delta\) is described as 
   
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q_1)</td>
<td>(q_1)</td>
<td>(q_2)</td>
</tr>
<tr>
<td>(q_2)</td>
<td>(q_3)</td>
<td>(q_2)</td>
</tr>
<tr>
<td>(q_3)</td>
<td>(q_2)</td>
<td>(q_2)</td>
</tr>
</tbody>
</table>

4. \(q_1\) is the start state, and 
5. \(F = \{q_2\}\).
In-class Exercise

Come up with a formal description of the following machine:

---

**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

- $Q = \{q_1, q_2, q_3\}$
- $\Sigma = \{a, b\}$
- $\delta$
  - $\delta( q_1, a ) = q_2$
  - $\delta( q_1, b ) = q_1$
  - $\delta( q_2, a ) = q_3$
  - $\delta( q_2, b ) = q_3$
  - $\delta( q_3, a ) = q_2$
  - $\delta( q_3, b ) = q_1$
- $q_0 = q_1$
- $F = \{q_2\}$

$M = (Q, \Sigma, \delta, q_0, F)$
A Computation Model is ... (from lecture 1)

• Some base definitions and axioms ...

**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**.

• And rules that use the definitions ...
Computation with FSMs (JFLAP demo)

- **FSM:**

- **Input:** “1101”
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)

- \( M = (Q, \Sigma, \delta, q_0, F) \)
- \( w = w_1 w_2 \cdots w_n \)
- \( r_0 = q_0 \)
- \( \delta(r_i, w_{i+1}) = r_{i+1}, \text{ for } i = 0, \ldots, n - 1 \)

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 \( r_n \in F \)

Still a little verbose
Check-in Quiz 9/8

On gradescope