What have we seen so far?

- Representing many types of data (text, numbers, images, sound) using binary representations.
- Solving problems using logical variables, logical expressions, truth tables, and logic gates.

Any limitations to this?

- So far we have assumed that the state of each logical variable stays constant over time.
  
  E.g. Reset the computer between each round of Rock-Paper-Scissors.

Today: Representing many configurations with memory.
The eight-puzzle

- Can define a sequence of moves to go from start configuration to goal configuration.
- Need to be able to represent any configuration in memory.
- BUT! Need a way to describe how the configuration (a.k.a. the state) changes over time. **What state-to-state transition is allowed in 1 move?**

---

Other examples

- Retractable ballpoint pen
- Automatic door
- Traffic-light control
- Combination lock
- Elevator control system
- Language generation

What do these things have in common?
- The state changes over time.
- Need memory to capture the state of the system at any point in time.
Ballpoint pen

- Open
  - Click
    - No-click
  - No-click
- Closed
  - Click
    - No-click

Eight-puzzle

Some state space of the Eight Puzzle:

- Initial state: 1 2 3 8 5 7 4 6
- Final state: 1 2 3 8 5 7 4 6

- Transition states:
  - 1 2 3 8 5 7 4 6
    - Click:
      - 1 2 3 8 5 7 4 6
    - No-click:
      - 1 2 3 8 5 7 4 6

- Other states:
  - 1 3 2 8 5 7 4 6
  - 1 2 3 8 4 7 6 5
Finite State Machine

• The Finite State Machine combines a look-up table (constructed with binary logic) with a memory device (to store the state).

• Components of the finite state machine:
  – Set of states: $S = \{s_1, s_2, \ldots, s_n\}$
  – Set of observations: $O = \{o_1, o_2, \ldots, o_n\}$
  – A transition function: describing how the state changes in response to the observation seen.

Transition function

• For each state and observation, need to know what is the next state.

• Denote this as: $T(s, o) = s'$
  
  where $s$ is the current state of the device
  $o$ is the most recent observation
  $s'$ is the next state of the device
Storing the state of a finite-state machine

- Observations are set through inputs.
- Register is used to store bits. It has an additional timing input that tells it when to change state (think of a clock that ‘ticks’ every second).
- The logic block implements the transition function.

To implement a finite-state machine (FSM)

1. Select the set of states and the set of observations.
2. Choose a different pattern of bits for each state
   - In traffic light example: need 3 bits for state, 1 bit for observation.
3. Generate transition table or transition graph.
4. Implement transition table using logic gates.
   - Input = bits representing the observation and the last state.
   - Output = bits representing the next state.
Traffic-light controller

- **State** = 2-way car traffic light
  - \( S = (\text{red, yellow, green})_{\text{direction 1}} \times (\text{red, yellow, green})_{\text{direction 2}} \)
  - \( S = \{\text{red-red, red-yellow, yellow-red, red-green, green-red}\} \)
  - Not all configurations of lights are included (e.g. green-green) at least one light must be red.

- **Observation** = pedestrian light request
  - \( O = \{\text{pressed, not pressed}\} \)

- **Memory** = Current state of the light in both directions.

Traffic-light controller: Transition function

- **State** = 2-way car traffic light
  - \( S = \{\text{red-red, red-yellow, yellow-red, red-green, green-red}\} \)

- **Observation** = pedestrian light request
  - \( O = \{\text{pressed, not pressed}\} \)

- **Transition function**: \( T(\text{current state, observation}) = \text{next state} \)

Here is one way to show the transition function as a graph.
Traffic-light controller: Transition table

<table>
<thead>
<tr>
<th>State</th>
<th>Observation</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green-Red</td>
<td>Pressed</td>
<td>Yellow-Red</td>
</tr>
<tr>
<td>Green-Red</td>
<td>Not-Pressed</td>
<td>Yellow-Red</td>
</tr>
<tr>
<td>Yellow-Red</td>
<td>Pressed</td>
<td>Red-Red</td>
</tr>
<tr>
<td>Yellow-Red</td>
<td>Not-Pressed</td>
<td>Red-Red</td>
</tr>
<tr>
<td>Red-Green</td>
<td>Pressed</td>
<td>Red-Yellow</td>
</tr>
<tr>
<td>Red-Green</td>
<td>Not-Pressed</td>
<td>Red-Yellow</td>
</tr>
<tr>
<td>Red-Yellow</td>
<td>Pressed</td>
<td>Green-Red</td>
</tr>
<tr>
<td>Red-Yellow</td>
<td>Not-Pressed</td>
<td>Red-Red</td>
</tr>
<tr>
<td>Red-Red</td>
<td>Pressed</td>
<td>Red-Green</td>
</tr>
<tr>
<td>Red-Red</td>
<td>Not-Pressed</td>
<td>Red-Green</td>
</tr>
</tbody>
</table>

• This can be seen as just a standard truth table.
• Simply need to pick logical variables for the states and observations.
• Only subtlety: Need to use same set of variables for the “State” and “Next state” variables.
• Also need to store those variables between time steps.

Combination Lock

• **State** = Summary of the sequence of numbers.

• **Observation** = Number entered.

• **Memory** = Not all numbers ever dialed need to be stored, but need to remember enough about recent numbers to know if the sequence opens the lock.
Recognizing sequences with a FSM

- Recall: Combination Lock
  - Assume the lock opens only when it sees sequence 0 - 5 - 2.
- States: Number of digits in the sequence that have been recognized already = {0, 1, 2, 3, done}
- Observations: Digits that can be selected = {0, 1, ..., 9}
- Transition function:

Recognizing sequences with patterns

- Consider a lock which recognizes sequences that start with "1", have any number of "0"s, and end with "3".
- Transition function:
Which sequences can be recognized?

- Can recognize any pre-specified sequences of numbers or letters of a finite length.
  - E.g. Misspelled word within a stream of text.

- Cannot recognize all types of patterns.
  - E.g. Cannot build a finite-state machine that unlocks a lock whenever you enter any palindrome: 3-2-1-1-2-3
  - Why? Palindromes can be of any length, and to recognize the 2nd half, you need to remember every character in the first half. Because there are infinitely many possible first halves, this would require a machine with an infinite number of states.

Other tasks we cannot do with an FSM

- Problems with non-deterministic transitions, e.g. backgammon.
- Problems where we don’t know the set of state/observations in advances.
- Problems where the transitions change over time.
Example: Behavior-based robotics

- Consider building a Light Seeking robot
  - States = {Seek-light, Follow-light, Avoid-obstacle}
  - Observations = {Light, NoLight, Obstacle, NoObstacle}

- Draw the transition graph.

- What other states/observations could we include?

Take-home message

- Finite-state machines let us reason about functions that change over time.
- Understand the components of finite state machines (state, observation, transition) and the implementation steps (transition table, logic encoding through gates, use of registers)
- Understand what tasks can and cannot be done with a FSM.