State Diagrams

Lecture 24
State
State

Consider a (discrete) system which takes a stream of inputs and produces a stream of outputs (a “transducer”)

State

Consider a (discrete) system which takes a stream of inputs and produces a stream of outputs (a "transducer")
State

Consider a (discrete) system which takes a stream of inputs and produces a stream of outputs (a “transducer”)

The system’s output at any moment depends not only on the “current” input but also on what the system “remembers” about the past
Consider a (discrete) system which takes a stream of inputs and produces a stream of outputs (a “transducer”).

The system’s output at any moment depends not only on the “current” input but also on what the system “remembers” about the past.

State of the system: what is in the system’s memory
State

Consider a (discrete) system which takes a stream of inputs and produces a stream of outputs (a “transducer”)

The system's output at any moment depends not only on the “current” input but also on what the system “remembers” about the past

State of the system: what is in the system’s memory

The number of possible states could be finite or infinite (for e.g. if the system remembers the sequence of inputs seen so far, or even just the number of inputs so far)
State Diagram
State Diagram

A graph with nodes as the states and arcs from a state to another if the system can make that transition in one step.
A graph with nodes as the states and arcs from a state to another if the system can make that transition in one step.

e.g. A system in which the inputs are pairs of binary digits (Least Significant Bit first) and the outputs are the digits of their sum.
State Diagram

A graph with nodes as the states and arcs from a state to another if the system can make that transition in one step.

e.g. A system in which the inputs are pairs of binary digits (Least Significant Bit first) and the outputs are the digits of their sum.
A graph with nodes as the states and arcs from a state to another if the system can make that transition in one step.

E.g. A system in which the inputs are pairs of binary digits (Least Significant Bit first) and the outputs are the digits of their sum.
A graph with nodes as the states and arcs from a state to another if the system can make that transition in one step.

e.g. A system in which the inputs are pairs of binary digits (Least Significant Bit first) and the outputs are the digits of their sum.

What should the system remember?
A graph with nodes as the states and arcs from a state to another if the system can make that transition in one step

e.g. A system in which the inputs are pairs of binary digits (Least Significant Bit first) and the outputs are the digits of their sum

What should the system remember?

The "carry": a single bit
A graph with nodes as the states and arcs from a state to another if the system can make that transition in one step.

e.g. A system in which the inputs are pairs of binary digits (Least Significant Bit first) and the outputs are the digits of their sum.

What should the system remember?

The “carry”: a single bit.

State diagram has two nodes.
State Diagram

0 0 1

[0][1][1][1]

carry

[1][0][0][0]
Initially carry is 0
Initially carry is 0
If carry is 0, and input is [0,0], then output is 0
Initially carry is 0

If carry is 0, and input is [0,0], then output is 0

And carry remains 0
Initially carry is 0

If carry is 0, and input is [0,0], then output is 0
  And carry remains 0

If carry is 0, and input is [1,1], then output is 0, but new carry is 1 ...

State Diagram
Initially carry is 0
If carry is 0, and input is [0,0], then output is 0
    And carry remains 0
If carry is 0, and input is [1,1], then output is 0, but new carry is 1 ...

<table>
<thead>
<tr>
<th>carry</th>
<th>input</th>
<th>output</th>
<th>new carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0,0]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[0,1]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[1,0]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[1,1]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[0,0]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>[0,1]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[1,0]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[1,1]</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Initially carry is 0

If carry is 0, and input is [0,0], then output is 0
   And carry remains 0

If carry is 0, and input is [1,1], then output is 0, but new carry is 1 ...

<table>
<thead>
<tr>
<th>carry</th>
<th>input</th>
<th>output</th>
<th>new carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0,0]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[0,1]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[1,0]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[1,1]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[0,0]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>[0,1]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[1,0]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[1,1]</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
State Diagram

<table>
<thead>
<tr>
<th>carry</th>
<th>input</th>
<th>output</th>
<th>new carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0,0]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[0,1]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[1,0]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[1,1]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[0,0]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>[0,1]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[1,0]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[1,1]</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Transition function: maps (state,input) pairs to (state,output) pairs

<table>
<thead>
<tr>
<th>carry</th>
<th>input</th>
<th>output</th>
<th>new carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0,0]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[0,1]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[1,0]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[1,1]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[0,0]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>[0,1]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[1,0]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[1,1]</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
State Diagram

- **Transition function**: maps (state,input) pairs to (state,output) pairs
- \( \delta_{\text{deterministic}}: S \times \Sigma_{\text{in}} \rightarrow S \times \Sigma_{\text{out}} \) (\( S \): state space, \( \Sigma \): “alphabet”)

<table>
<thead>
<tr>
<th>carry</th>
<th>input</th>
<th>output</th>
<th>new carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0,0]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[0,1]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[1,0]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[1,1]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[0,0]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>[0,1]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[1,0]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[1,1]</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**State Diagram**

- **Transition function**: maps (state, input) pairs to (state, output) pairs
- \( \delta_{\text{deterministic}}: S \times \Sigma_{\text{in}} \rightarrow S \times \Sigma_{\text{out}} \) (S: state space, \( \Sigma \): “alphabet”)
- **Deterministic**: given a state and an input, the system’s behavior on next input is completely determined

<table>
<thead>
<tr>
<th>carry</th>
<th>input</th>
<th>output</th>
<th>new carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0,0]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[0,1]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[1,0]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[1,1]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[0,0]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>[0,1]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[1,0]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[1,1]</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**State Diagram**

- **Transition function**: maps \((\text{state}, \text{input})\) pairs to \((\text{state}, \text{output})\) pairs
- \(\delta_{\text{deterministic}}: S \times \Sigma_{\text{in}} \rightarrow S \times \Sigma_{\text{out}}\) (\(S\): state space, \(\Sigma\): “alphabet”)
- **Deterministic**: given a state and an input, the system’s behavior on next input is completely determined

<table>
<thead>
<tr>
<th>carry</th>
<th>input</th>
<th>output</th>
<th>new carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0,0]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[0,1]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[1,0]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>[1,1]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[0,0]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>[0,1]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[1,0]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[1,1]</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Another Example
Another Example

Binary addition for 3 bit numbers
Another Example

- Binary addition for 3 bit numbers

- In the previous example, the answer is complete only if carry is 0 (can enforce by feeding [0,0] as a last input)
Another Example

- Binary addition for 3 bit numbers

- In the previous example, the answer is complete only if carry is 0 (can enforce by feeding [0,0] as a last input)

- Here, accepts only up to 3 bits for each number, and produces a 4 bit output
Another Example

- Binary addition for 3 bit numbers

- In the previous example, the answer is complete only if carry is 0 (can enforce by feeding [0,0] as a last input)

- Here, accepts only up to 3 bits for each number, and produces a 4 bit output

- State space?
Another Example

Binary addition for 3 bit numbers

In the previous example, the answer is complete only if carry is 0 (can enforce by feeding [0,0] as a last input)

Here, accepts only up to 3 bits for each number, and produces a 4 bit output

State space?

Need to remember carry, and number of inputs seen so far
Another Example

- Binary addition for 3 bit numbers

- In the previous example, the answer is complete only if carry is 0 (can enforce by feeding \([0,0]\) as a last input)

- Here, accepts only up to 3 bits for each number, and produces a 4 bit output

- State space?

- Need to remember carry, and number of inputs seen so far
Question
Question

On giving which of the following strings as input does this transducer give a different string as output?

A. $\epsilon$ (empty string)
B. 0011010
C. 0010110
D. 100
E. 1100011
Question

On giving which of the following strings as input does this transducer give a different string as output?

A. \(\varepsilon\) (empty string)
B. 0011010
C. 0010110
D. 100
E. 1100011

\[(0^*11)^* 10 0^* 1 (0|1)^*\]
Acceptors
Acceptors

The machines we saw are deterministic transducers
Acceptors

The machines we saw are deterministic transducers

Converts an input stream to an output stream
Acceptors

- The machines we saw are deterministic transducers
  - Converts an input stream to an output stream
- Acceptors don’t produce an output stream
Acceptors

- The machines we saw are deterministic transducers
  - Converts an input stream to an output stream

- Acceptors don't produce an output stream
  - At the end of input, either “accepts” or “rejects” the input. Indicated by the state it is in at that point.
Acceptors

- The machines we saw are deterministic **transducers**
  - Converts an input stream to an output stream

- **Acceptors** don’t produce an output stream
  - At the end of input, either “accepts” or “rejects” the input. Indicated by the state it is in at that point.

- Accepting states are called **final states**
Acceptors

The machines we saw are deterministic transducers.

- Converts an input stream to an output stream
- Acceptors don't produce an output stream
- At the end of input, either "accepts" or "rejects" the input. Indicated by the state it is in at that point.
- Accepting states are called final states
- Transition function: $\delta_{\text{det-acceptor}} : S \times \Sigma \rightarrow S$
An Example
An Example

Input: a number given as binary digits, MSB first.
Accept iff the number is even (or empty)
An Example

Input: a number given as binary digits, MSB first. Accept iff the number is even (or empty)

Just remember the last digit seen
An Example

- Input: a number given as binary digits, MSB first.
  Accept iff the number is even (or empty)
- Just remember the last digit seen
- What if input is given LSB first?
Example

- Input: a number given as binary digits, MSB first. Accept iff the number is even (or empty).
- Just remember the last digit seen.
- What if input is given LSB first?
- Remember the first digit seen.

Diagram:
An Example

Input: a number given as binary digits, MSB first. Accept iff the number is even (or empty)

Just remember the last digit seen

What if input is given LSB first?

Remember the first digit seen

How about deciding if the number is a multiple of say 5?
Question
Question

How many states must an acceptor for multiples of 5 have, when the inputs are given as binary digits of a non-negative number, with MSB first? (Treat empty input as number 0.)

A. 2  
B. 4  
C. 5  
D. 6  
E. Infinitely many
Question

How many states must an acceptor for multiples of 5 have, when the inputs are given as binary digits of a non-negative number, with MSB first? (Treat empty input as number 0.)

A. 2
B. 4
C. 5
D. 6
E. Infinitely many

Need to only remember $x \pmod{5}$, where $x$ is the number seen so far. Next number $x'$ is $2x$ or $2x+1$ depending on the current input bit. $x' \pmod{5}$ is determined by $x \pmod{5}$. 
Question
Question

Which of the following strings is **not accepted** by this acceptor:

A. $\epsilon$ (empty string)
B. 101
C. 001000110
D. 1011001
E. 10000001
Which of the following strings is not accepted by this acceptor:

A. $\epsilon$ (empty string)
B. 101
C. 001000110
D. 1011001
E. 10000001
Counting Number of States: An Example
Counting Number of States: An Example

Game of Nim:
- 2 piles of matchsticks, with T matchsticks each.
- Each round a player removes one or more matchsticks from one pile.
- Alice makes the first move.
Counting Number of States: An Example

Game of Nim:
- 2 piles of matchsticks, with $T$ matchsticks each.
- Each round a player removes one or more matchsticks from one pile.
- Alice makes the first move.

What are the states?
Counting Number of States: An Example

Game of Nim:
- 2 piles of matchsticks, with $T$ matchsticks each.
- Each round a player removes one or more matchsticks from one pile.
- Alice makes the first move.

What are the states?

$(|\text{pile}_1|, |\text{pile}_2|, \text{next-player})$
Counting Number of States: An Example

Game of Nim:
- 2 piles of matchsticks, with T matchsticks each.
- Each round a player removes one or more matchsticks from one pile.
- Alice makes the first move.

What are the states?

(|pile\textsubscript{1}|, |pile\textsubscript{2}|, next-player)

Number of such states? 2(T+1)^2
Counting Number of States: An Example

Game of Nim:
- 2 piles of matchsticks, with $T$ matchsticks each.
- Each round a player removes one or more matchsticks from one pile.
- Alice makes the first move.

What are the states?

$(|pile_1|, |pile_2|, next-player)$

Number of such states? $2(T+1)^2$

Number of reachable states? $2(T+1)^2 - 4$
Counting Number of States: An Example

Game of Nim:
- 2 piles of matchsticks, with T matchsticks each.
- Each round a player removes one or more matchsticks from one pile.
- Alice makes the first move.

What are the states?

(|pile$_1|, |pile$_2|, next-player)

Number of such states? $2(T+1)^2$

Number of reachable states? $2(T+1)^2 - 4$
Finite-State Machines
Finite-State Machines

Many sets of strings have finite-state acceptors
Finite-State Machines

Many sets of strings have finite-state acceptors

- e.g., numbers divisible by d, LSB first, or MSB first. strings matching a "pattern" like 0*10*10* (strings with exactly two 1s)
Finite-State Machines

Many sets of strings have finite-state acceptors

e.g., numbers divisible by d, LSB first, or MSB first. strings matching a "pattern" like 0*10*10* (strings with exactly two 1s)

Can run for ever without needing more memory
Finite-State Machines

Many sets of strings have finite-state acceptors

e.g., numbers divisible by d, LSB first, or MSB first. strings matching a “pattern” like 0*10*10* (strings with exactly two 1s)

Can run for ever without needing more memory

Many interesting sets of strings do not have finite-state acceptors
Finite-State Machines

Many sets of strings have finite-state acceptors

- e.g., numbers divisible by \(d\), LSB first, or MSB first. strings matching a “pattern” like \(0*10*10*\) (strings with exactly two 1s)
- Can run for ever without needing more memory

Many interesting sets of strings do not have finite-state acceptors

- e.g., strings with equal number of 0s and 1s, palindromes, strings representing prime numbers, ...
Finite-State Machines

Many sets of strings have finite-state acceptors

- e.g., numbers divisible by d, LSB first, or MSB first. strings matching a "pattern" like $0^*10^*10^*$ (strings with exactly two 1s)
- Can run for ever without needing more memory

Many interesting sets of strings do not have finite-state acceptors

- e.g., strings with equal number of 0s and 1s, palindromes, strings representing prime numbers, ...
- How do we know they don't have finite-state acceptors?
Finite-State Machines

Many sets of strings have finite-state acceptors

- e.g., numbers divisible by d, LSB first, or MSB first. strings matching a “pattern” like 0*10*10* (strings with exactly two 1s)
- Can run for ever without needing more memory

Many interesting sets of strings do not have finite-state acceptors

- e.g., strings with equal number of 0s and 1s, palindromes, strings representing prime numbers, ...
- How do we know they don’t have finite-state acceptors?
- If only finite memory, can come up with two input sequences which result in same state, but one to be accepted and one to be rejected
Finite-State Machines

- Many sets of strings have finite-state acceptors
  - e.g., numbers divisible by d, LSB first, or MSB first. strings matching a “pattern” like 0*10*10* (strings with exactly two 1s)
  - Can run for ever without needing more memory
- Many interesting sets of strings do not have finite-state acceptors
  - e.g., strings with equal number of 0s and 1s, palindromes, strings representing prime numbers, ...
- How do we know they don’t have finite-state acceptors?
  - If only finite memory, can come up with two input sequences which result in same state, but one to be accepted and one to be rejected
- Later (in CS173).
Non-determinism
Non-determinism

At a state, on an input, the system could make zero, one or more different transitions
Non-determinism

At a state, on an input, the system could make zero, one or more different transitions

\[ \delta_{\text{nondet-acceptor}} : S \times \Sigma \rightarrow \mathcal{P}(S) \]
Non-determinism

- At a state, on an input, the system could make zero, one or more different transitions

\[ \delta_{\text{nondet-acceptor}} : S \times \Sigma \rightarrow \mathcal{P}(S) \]

- \( \delta(s,a) \): At a state \( s \), on input \( a \), what is the set of all the states to which the system can transition
Non-determinism

- At a state, on an input, the system could make zero, one or more different transitions

\[ \delta_{\text{nondet-acceptor}} : S \times \Sigma \rightarrow \mathcal{P}(S) \]

- \( \delta(s,a) \): At a state \( s \), on input \( a \), what is the set of all the states to which the system can transition

- System’s behavior not necessarily fixed by its state and input
Non-determinism

- At a state, on an input, the system could make zero, one or more different transitions

\[ \delta_{\text{nondet-acceptor}} : S \times \Sigma \rightarrow \mathcal{P}(S) \]

- \( \delta(s,a) \): At a state \( s \), on input \( a \), what is the set of all the states to which the system can transition

- System’s behavior not necessarily fixed by its state and input

- Sometimes probabilistic machine: Non-deterministic machine + probabilities associated with the multiple transitions
Representing a Finite-State Machine
Representing a Finite-State Machine

- If your program uses only a constant amount of memory (irrespective of how large the input (stream) is) then it is a finite state machine.
Representing a Finite-State Machine

- If your program uses only a constant amount of memory (irrespective of how large the input (stream) is) then it is a finite state machine.

- But often useful to explicitly design a finite state machine (drawing out all its states), and then implement it.
If your program uses only a constant amount of memory (irrespective of how large the input (stream) is) then it is a finite state machine.

But often useful to explicitly design a finite state machine (drawing out all its states), and then implement it.

To represent the transition function of a deterministic acceptor, a look-up table mapping (state,input) pair to a state.
Representing a Finite-State Machine

If your program uses only a constant amount of memory (irrespective of how large the input (stream) is) then it is a finite state machine.

But often useful to explicitly design a finite state machine (drawing out all its states), and then implement it.

To represent the transition function of a deterministic acceptor, a look-up table mapping (state,input) pair to a state.

But if sparse - i.e., for many states, many inputs lead to a "crash state" (and hence can be omitted) - it is more space-efficient to simply list valid (state, input, next state) tuples.
Representing a Finite-State Machine

- If your program uses only a constant amount of memory (irrespective of how large the input (stream) is) then it is a finite state machine.

- But often useful to explicitly design a finite state machine (drawing out all its states), and then implement it.

- To represent the transition function of a deterministic acceptor, a look-up table mapping (state,input) pair to a state.

- But if sparse - i.e., for many states, many inputs lead to a "crash state" (and hence can be omitted) - it is more space-efficient to simply list valid (state, input, next state) tuples.

Or, in the case of non-deterministic machines, ∅
Representing a Finite-State Machine

- If your program uses only a constant amount of memory (irrespective of how large the input (stream) is) then it is a finite state machine.

- But often useful to explicitly design a finite state machine (drawing out all its states), and then implement it.

- To represent the transition function of a deterministic acceptor, a look-up table mapping (state,input) pair to a state.

- But if sparse — i.e., for many states, many inputs lead to a “crash state” (and hence can be omitted) — it is more space-efficient to simply list valid (state, input, next state) tuples.

- This would slow down look-up.

Or, in the case of non-deterministic machines, ☐
Representing a Finite-State Machine

- If your program uses only a constant amount of memory (irrespective of how large the input (stream) is) then it is a finite state machine.

- But often useful to explicitly design a finite state machine (drawing out all its states), and then implement it.

- To represent the transition function of a deterministic acceptor, a look-up table mapping (state,input) pair to a state.

- But if sparse - i.e., for many states, many inputs lead to a "crash state" (and hence can be omitted) - it is more space-efficient to simply list valid (state, input, next state) tuples.

- This would slow down look-up.

- An appropriate data structure (sometimes a "hash table") can give (almost) the best of both worlds.
Infinite-State Systems
Infinite-State Systems

If we consider an infinite set of possible inputs (all possible strings), many systems are best modeled as infinite-state systems.
Infinite-State Systems

If we consider an infinite set of possible inputs (all possible strings), many systems are best modeled as infinite-state systems.

E.g., a counter that keeps track of the number of inputs so far.
Infinite-State Systems

If we consider an infinite set of possible inputs (all possible strings), many systems are best modeled as infinite-state systems. e.g., a counter that keeps track of the number of inputs so far. In practice, your machine has only a finite memory, but it is not very useful to model it as a finite-state machine if the number of states is huge.
Infinite-State Systems

If we consider an infinite set of possible inputs (all possible strings), many systems are best modeled as infinite-state systems.

- e.g., a counter that keeps track of the number of inputs so far.

In practice, your machine has only a finite memory, but it is not very useful to model it as a finite-state machine if the number of states is huge.

- e.g., if a program stores 100 bits of input in memory, already the number of possible states it can have is more than the age of the universe in pico seconds.
Infinite-State Systems

If we consider an infinite set of possible inputs (all possible strings), many systems are best modeled as infinite-state systems.

- e.g., a counter that keeps track of the number of inputs so far.

In practice, your machine has only a finite memory, but it is not very useful to model it as a finite-state machine if the number of states is huge.

- e.g., if a program stores 100 bits of input in memory, already the number of possible states it can have is more than the age of the universe in pico seconds.

In general infeasible to explicitly describe the state diagram.
Infinite-State Systems

If we consider an infinite set of possible inputs (all possible strings), many systems are best modeled as infinite-state systems.

- e.g., a counter that keeps track of the number of inputs so far.

In practice, your machine has only a finite memory, but it is not very useful to model it as a finite-state machine if the number of states is huge.

- e.g., if a program stores 100 bits of input in memory, already the number of possible states it can have is more than the age of the universe in pico seconds.

In general infeasible to explicitly describe the state diagram.

An infinite-state system can still be a “finite-control” system.
Infinite-State Systems

If we consider an infinite set of possible inputs (all possible strings), many systems are best modeled as infinite-state systems.

- e.g., a counter that keeps track of the number of inputs so far.
- In practice, your machine has only a finite memory, but it is not very useful to model it as a finite-state machine if the number of states is huge.
  - e.g., if a program stores 100 bits of input in memory, already the number of possible states it can have is more than the age of the universe in pico seconds.
- In general infeasible to explicitly describe the state diagram.

An infinite-state system can still be a “finite-control” system.

- i.e., system’s behavior defined by a small “program”
Infinite-State Systems

If we consider an infinite set of possible inputs (all possible strings), many systems are best modeled as infinite-state systems. e.g., a counter that keeps track of the number of inputs so far.

In practice, your machine has only a finite memory, but it is not very useful to model it as a finite-state machine if the number of states is huge. e.g., if a program stores 100 bits of input in memory, already the number of possible states it can have is more than the age of the universe in pico seconds.

In general infeasible to explicitly describe the state diagram.

An infinite-state system can still be a “finite-control” system, i.e., system’s behavior defined by a small “program”

This is what we consider computation.
Infinite-State Systems
Infinite-State Systems

Even a few simple rules can lead to complex behavioral patterns (or non-patterns)
Infinite-State Systems

Even a few simple rules can lead to complex behavioral patterns (or non-patterns)

Popular examples
Infinite-State Systems

Even a few simple rules can lead to complex behavioral patterns (or non-patterns)

Popular examples

Game of Life
Infinite-State Systems

- Even a few simple rules can lead to complex behavioral patterns (or non-patterns)
- Popular examples
  - Game of Life
  - Cellular automata
Infinite-State Systems

Even a few simple rules can lead to complex behavioral patterns (or non-patterns)

Popular examples

- Game of Life
- Cellular automata
- Aperiodic tilings/Quasicrystals
Infinite-State Systems

Even a few simple rules can lead to complex behavioral patterns (or non-patterns)

- Popular examples
  - Game of Life
  - Cellular automata
  - Aperiodic tilings/Quasicrystals

- A simple model for computation
Infinite-State Systems

Even a few simple rules can lead to complex behavioral patterns (or non-patterns)

- Popular examples
  - Game of Life
  - Cellular automata
  - Aperiodic tilings/Quasicrystals

- A simple model for computation
- Turing Machines
Infinite-State Systems

Even a few simple rules can lead to complex behavioral patterns (or non-patterns)

Popular examples

- Game of Life
- Cellular automata
- Aperiodic tilings/Quasicrystals

A simple model for computation

- Turing Machines

Later...