CS 10:
Problem solving via Object Oriented Programming

Winter 2017

Tim Pierson
260 (255) Sudikoff

Day 19 – Pattern Matching

Agenda

1. Regular expressions
2. Finite automata
3. Validating input
4. Modeling a complex system

Sometimes it is useful to be able to detect or require patterns

Email addresses follow a pattern:
mailbox@domain.TLD
example: tjp@cs.dartmouth.edu

We can specify a pattern or rules for email addresses:
<characters> @ <characters>.<com | edu | org | ...>

One or more characters

Followed by @

One or more characters

Followed by .

Followed by one of a set predefined of values

Regular expressions (RegEx) are a common way of looking for patterns in Strings

Regular expressions (regex)

- Most programming languages have support for regex
- Can be really complex and messy, but there are basic patterns

Operation	Meaning	Example
Character	Match a character next	“a” matches “a”

Regular expressions (RegEx) are a common way of looking for patterns in Strings

Regular expressions (regex)

- Most programming languages have support for regex
- Can be really complex and messy, but there are basic patterns

Operation	Meaning	Example
Character	Match a character next	“a” matches “a”
Concatenation: R_1 R_2	One after the other	“cat” matches “c” then “a” then “t”

Regular expressions (RegEx) are a common way of looking for patterns in Strings

Regular expressions (regex)

- Most programming languages have support for regex
- Can be really complex and messy, but there are basic patterns

Operation	Meaning	Example
Character	Match a character next	“a” matches “a”
Concatenation: R_1 R_2	One after the other	“cat” matches “c” then “a” then “t”
Alternative: R_1 | R_2	One or the other	a|e|i|o|u matches any vowel

Regular expressions (RegEx) are a common way of looking for patterns in Strings

Regular expressions (regex)

- Most programming languages have support for regex
- Can be really complex and messy, but there are basic patterns

Operation	Meaning	Example
Character	Match a character next	“a” matches “a”
Concatenation: $R_1 R_2$	One after the other	“cat” matches “c” then “a” then “t”
Alternative: $R_1 \mid R_2$	One or the other	a|e|i|o|u matches any vowel
Grouping: (R)	Establishes order; allows reference/extraction	c(a|o)t matches “cat” or “cot”

Regular expressions (RegEx) are a common way of looking for patterns in Strings

Regular expressions (regex)

- Most programming languages have support for regex
- Can be really complex and messy, but there are basic patterns

Operation	Meaning	Example
Character	Match a character next	"a" matches "a"
Concatenation: $R_1 R_2$	One after the other	"cat" matches "c" then "a" then "t"
Alternative: R_1 | R_2	One or the other	a|e|i|o|u matches any vowel
Grouping: (R)	Establishes order; allows reference/extraction	c(a|o)t matches "cat" or "cot"
Character classes [c_1-c_2] and [^c_1-c_2]	Alternative characters and excluded characters	[a-c] matches "a" or "b" or "c", while [^a-c] matches any but abc

Regular expressions (RegEx) are a common way of looking for patterns in Strings

Regular expressions (regex)

- Most programming languages have support for regex
- Can be really complex and messy, but there are basic patterns

Operation	Meaning	Example
Character	Match a character next	"a" matches "a"
Concatenation: R_1 R_2	One after the other	"cat" matches "c" then "a" then "t"
Alternative: R_1 | R_2	One or the other	a|e|i|o|u matches any vowel
Grouping: (R)	Establishes order; allows reference/extraction	c(a|o)t matches "cat" or "cot"
Character classes [c_1-c_2] and [^c_1-c_2]	Alternative characters and excluded characters	[a-c] matches "a" or "b" or "c", while [^a-c] matches any but abc
Repetition: R*	Matches 0 or more times	"ca*t" matches "ct", "cat", "caat"

Regular expressions (RegEx) are a common way of looking for patterns in Strings

Regular expressions (regex)

- Most programming languages have support for regex
- Can be really complex and messy, but there are basic patterns

Operation	Meaning	Example
Character	Match a character next	“a” matches “a”
Concatenation: R_1 R_2	One after the other	“cat” matches “c” then “a” then “t”
Alternative: R_1 | R_2	One or the other	a|e|i|o|u matches any vowel
Grouping: (R)	Establishes order; allows reference/extraction	c(a|o)t matches “cat” or “cot”
Character classes [c_1-c_2] and [^c_1-c_2]	Alternative characters and excluded characters	[a-c] matches “a” or “b” or “c”, while [^a-c] matches any but abc
Repetition: R*	Matches 0 or more times	“ca*t” matches “ct”, “cat”, “caat”
Non-zero repetition: R+	Matches 1 or more times	“ca+t” matches “cat” or “caat” or “caaat”, but not “ct”

We can use RegEx to see if an email address is valid

Email addresses follow a pattern:
mailbox@domain.TLD
example: tjp@cs.dartmouth.edu

We can specify a pattern or rules for email addresses:
<characters> @ <characters>.<com | edu | org | …>

As a simple RegEx: [a-z]+@[a-z.]* [a-z]+. (com | edu | org …)

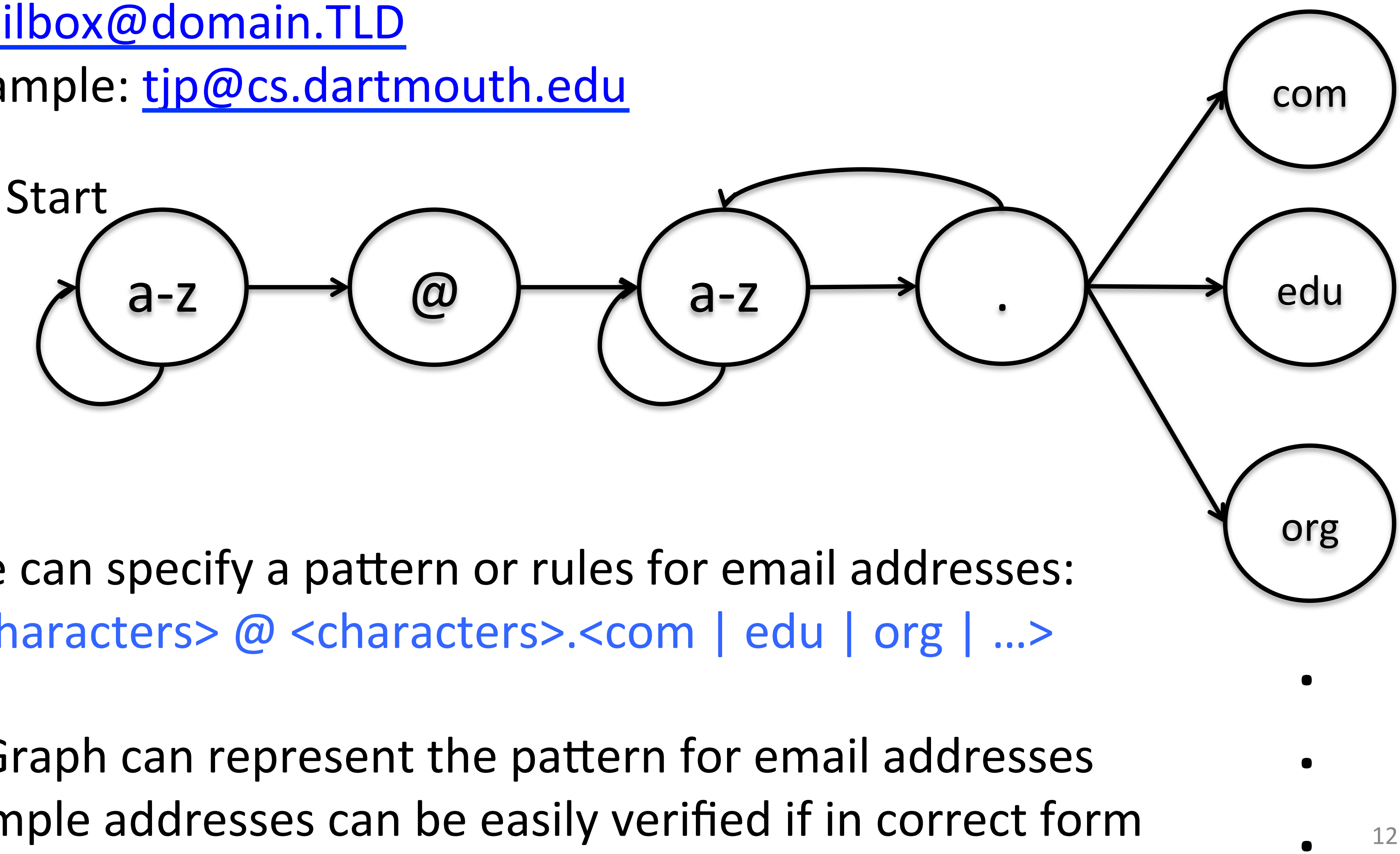
Check:
tjp@cs.dartmouth.edu -- valid
Blob.x -- invalid

A Graph can implement a RegEx

Email addresses follow a pattern:

mailbox@domain.TLD

example: tjp@cs.dartmouth.edu



We can specify a pattern or rules for email addresses:

<characters> @ <characters>.<com | edu | org | …>

A Graph can represent the pattern for email addresses

Sample addresses can be easily verified if in correct form

Key points

1. We can define a set of rules that must be followed

2. We can represent those rules with a graph

Agenda

1. Regular expressions
2. Finite automata
3. Validating input
4. Modeling a complex system

Finite Automata (FA) can be used for many problems, two uses are common

Common Finite Automata use cases

1. Validating input

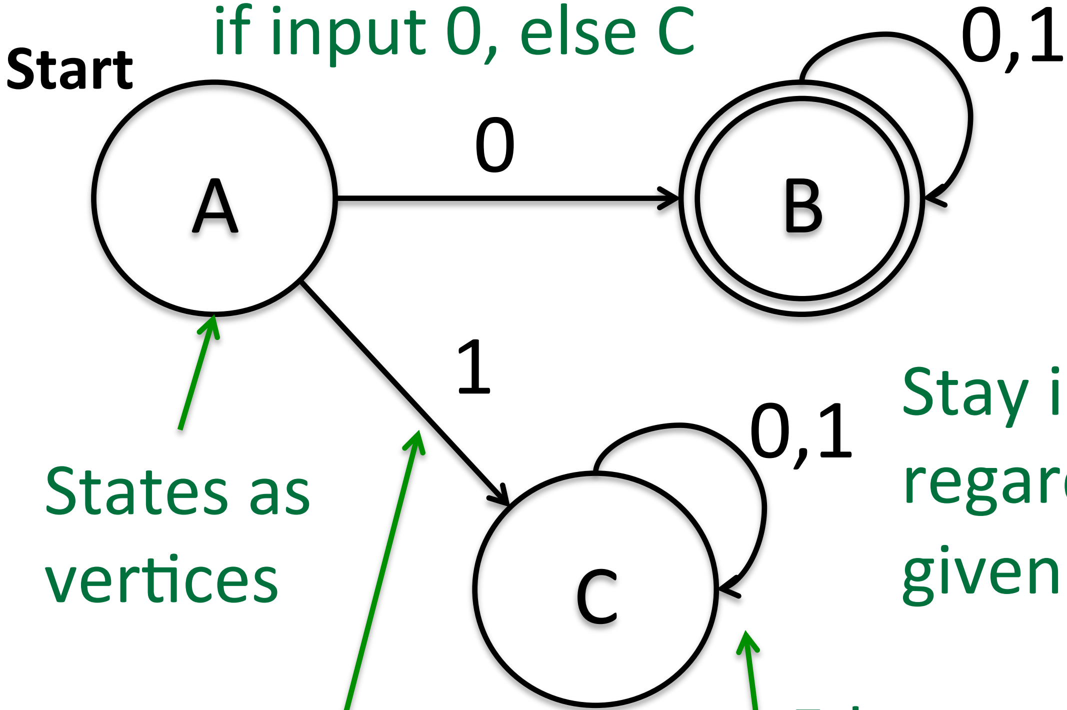
2. Tracking the state of a system, changing state as a response to events

We can model States as vertices and Transitions as edges in a directed graph

Finite Automata validating input

Transition from A to B if input 0, else C

Set of labels called alphabet



Double circle indicates valid end states

Stay in C regardless if given 0 or 1

States as vertices

Edges as transitions

Edges can loop back to same vertex

What does this do?

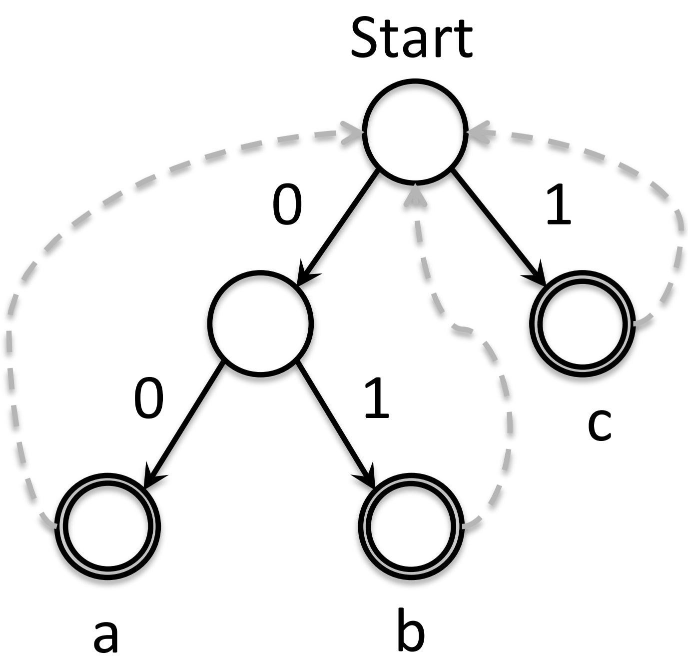
Accepts any input starting with 0

Agenda

1. Regular expressions
2. Finite automata
3. Validating input
4. Modeling a complex system

Finite Automata can validate input

Finite Automata validating input

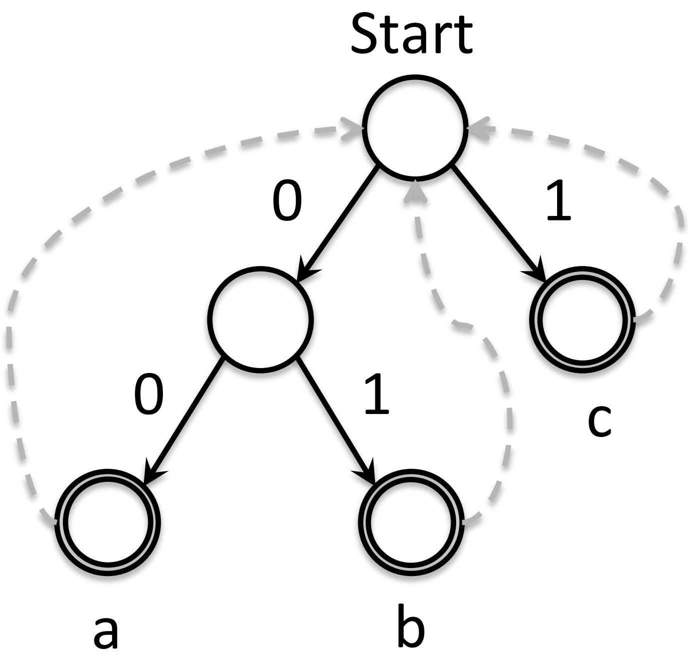


Input	**Result**
00	a

Assume leaves represent valid end states
Can loop back to root from leaf
Invalid if input ends and not at end state
Extension of Huffman, go back to root after finding leaf

Finite Automata can validate input

Finite Automata validating input

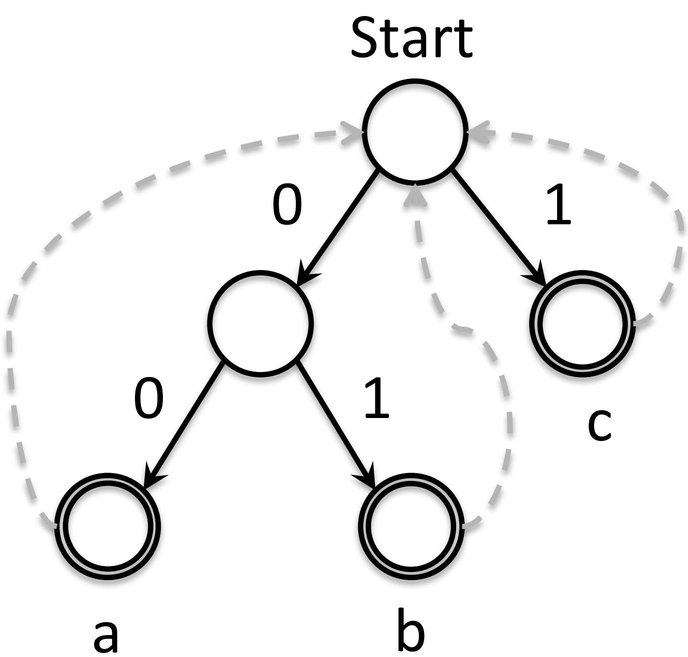


Input	**Result**
00	a
01	b

Assume leaves represent valid end states
Can loop back to root from leaf
Invalid if input ends and not at end state
Extension of Huffman, go back to root after finding leaf

Finite Automata can validate input

Finite Automata validating input

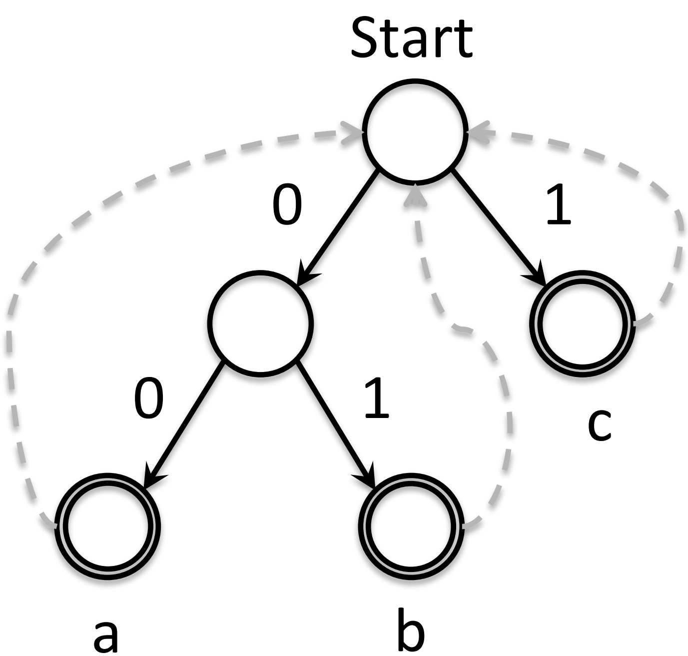


Input	**Result**
00	a
01	b
1	c

Assume leaves represent valid end states
Can loop back to root from leaf
Invalid if input ends and not at end state
Extension of Huffman, go back to root after finding leaf

Finite Automata can validate input

Finite Automata validating input

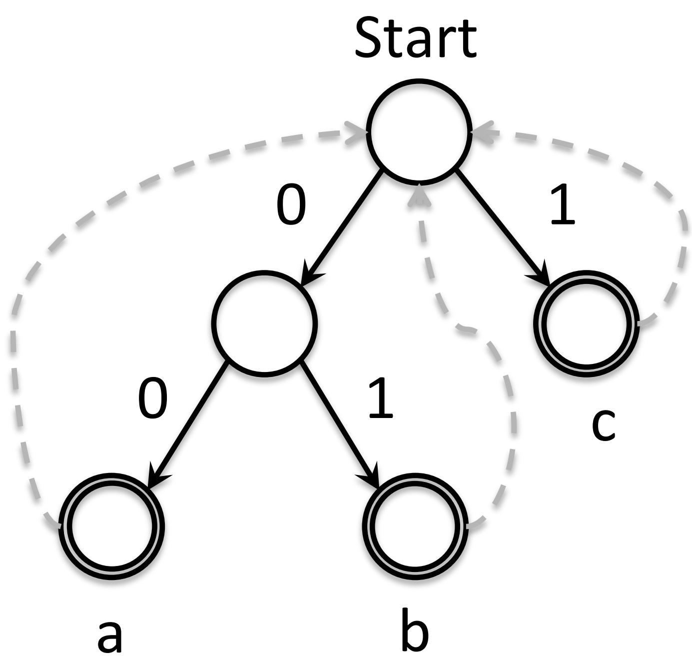


Input	**Result**
00	a
01	b
1	c
0	invalid

Assume leaves represent valid end states
Can loop back to root from leaf
Invalid if input ends and not at end state
Extension of Huffman, go back to root after finding leaf

Finite Automata can validate input

Finite Automata validating input

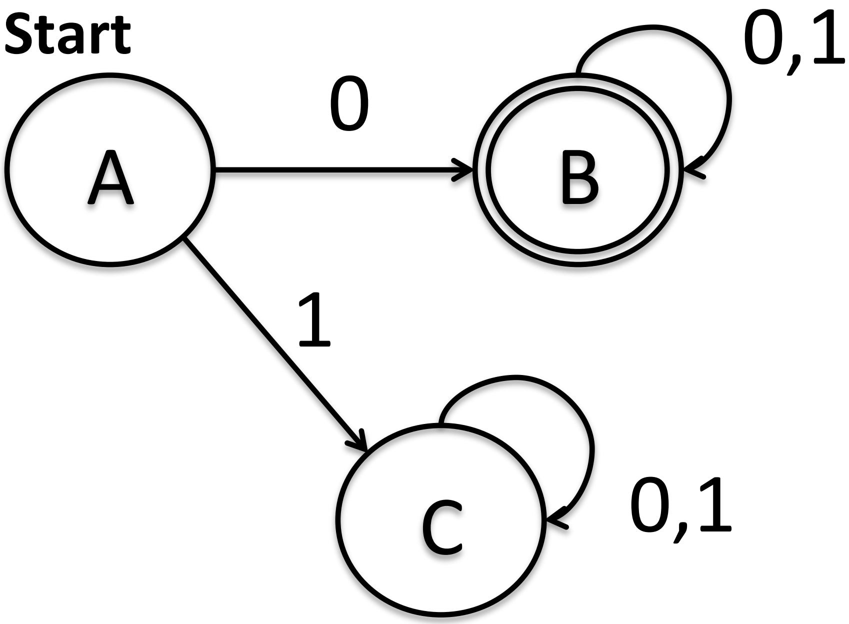


Input	**Result**
00	a
01	b
1	c
0	Invalid
001100	acca

Assume leaves represent valid end states
Can loop back to root from leaf
Invalid if input ends and not at end state
Extension of Huffman, go back to root after finding leaf

Finite Automata come in two flavors, Deterministic and Nondeterministic

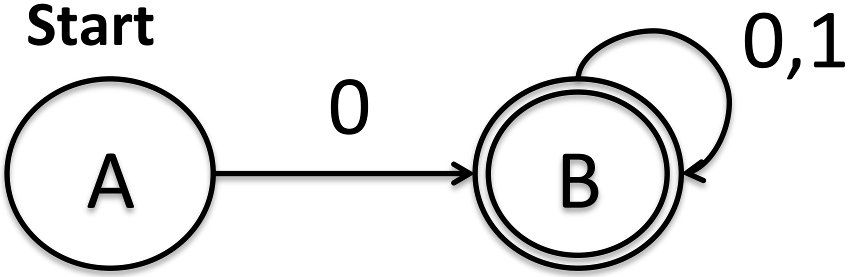
Deterministic Finite Automaton (DFA)



Exactly one choice for each possible input

No ambiguity

Nondeterministic Finite Automaton (NFA)

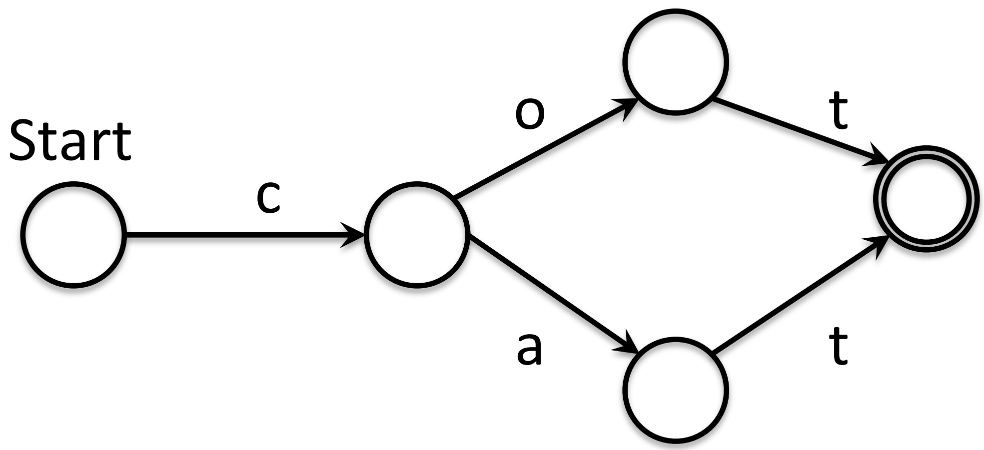


May have 0, 1, or more choices for transition from each state

These both do the same thing

With DFAs we have to specify each State transition, with NFAs we do not

NFA validating input



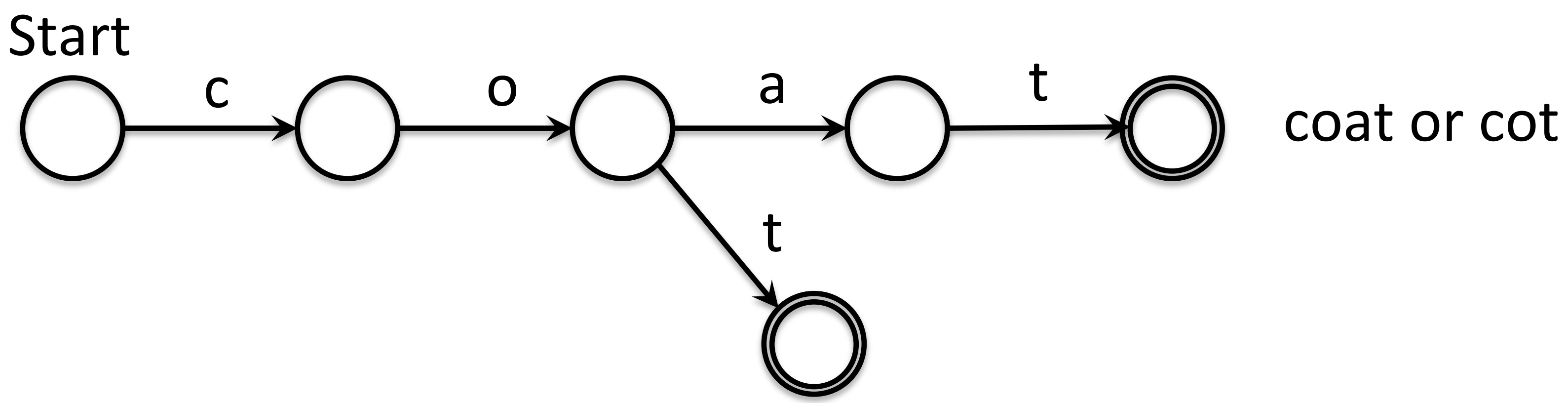
Valid inputs:
cot or cat

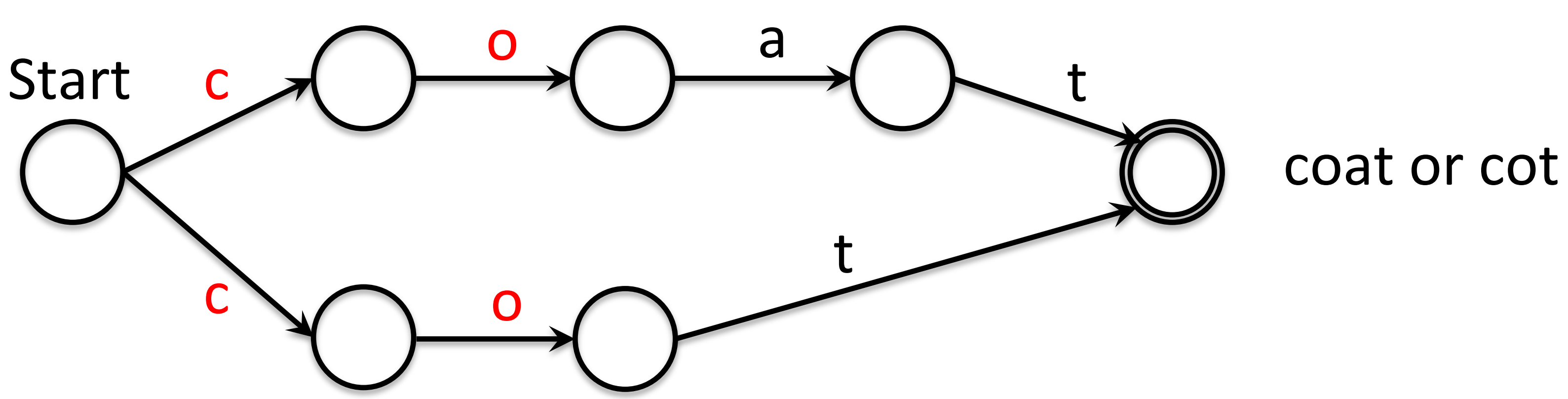
All else invalid

There are 0, 1, or more choices for each letter

With NFAs we can have 0, 1 or more choices for each input

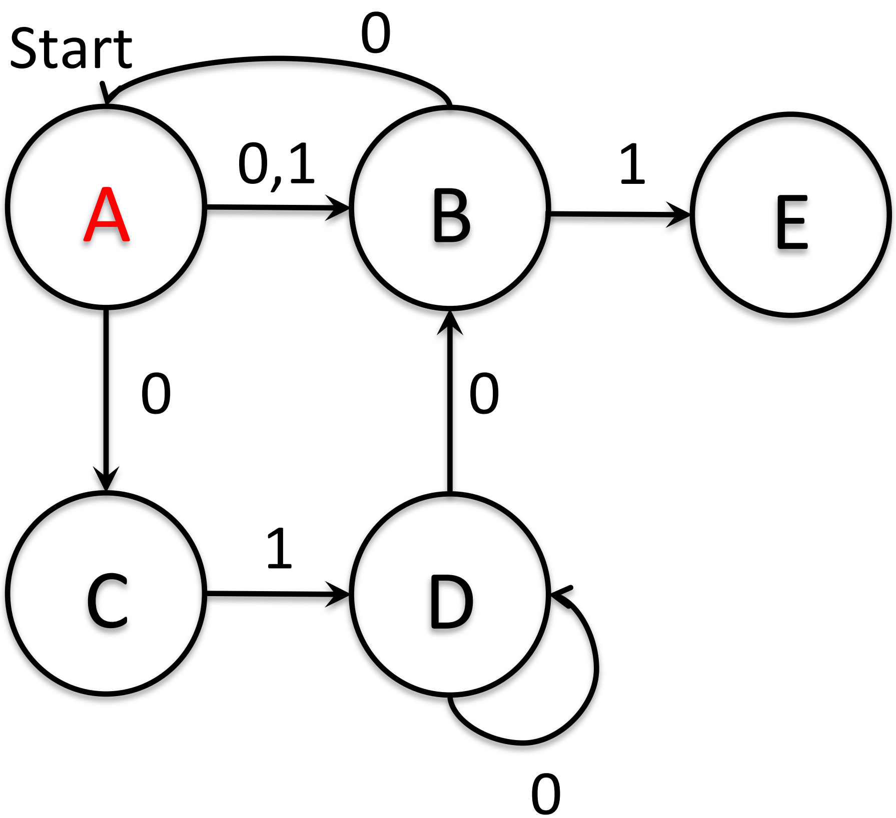
NFA validating input





Sometimes we cannot map from a State a single next State

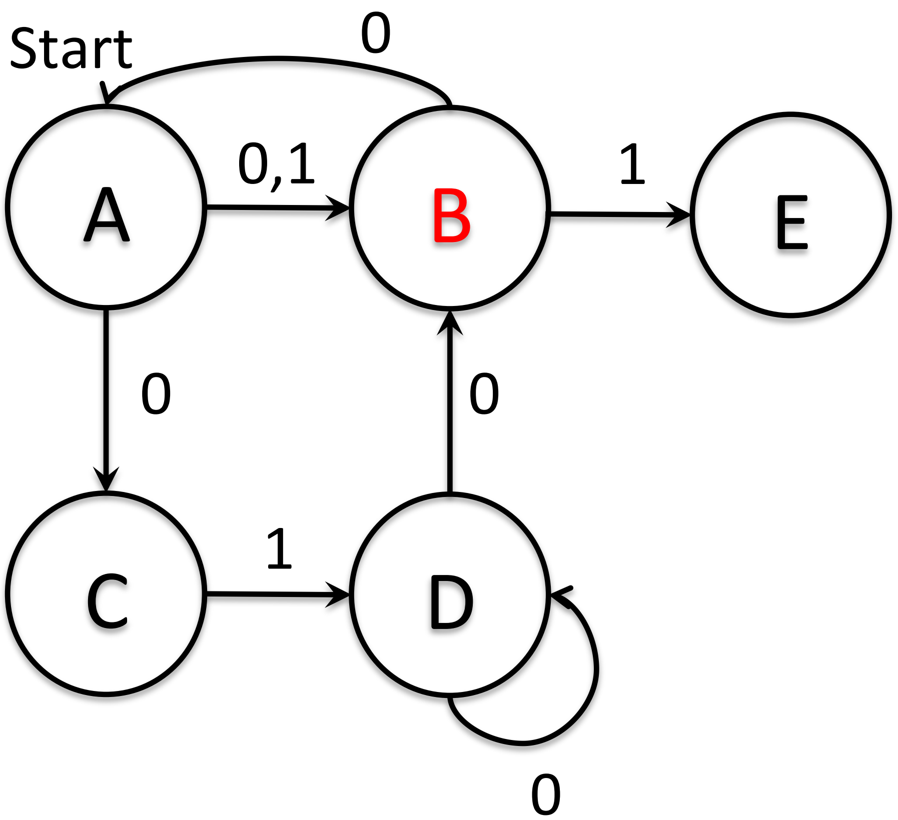
NFAs can have multiple next States



Key	Input	Paths
A	0	{B,C}
A	1	{B}

Sometimes we cannot map from a State a single next State

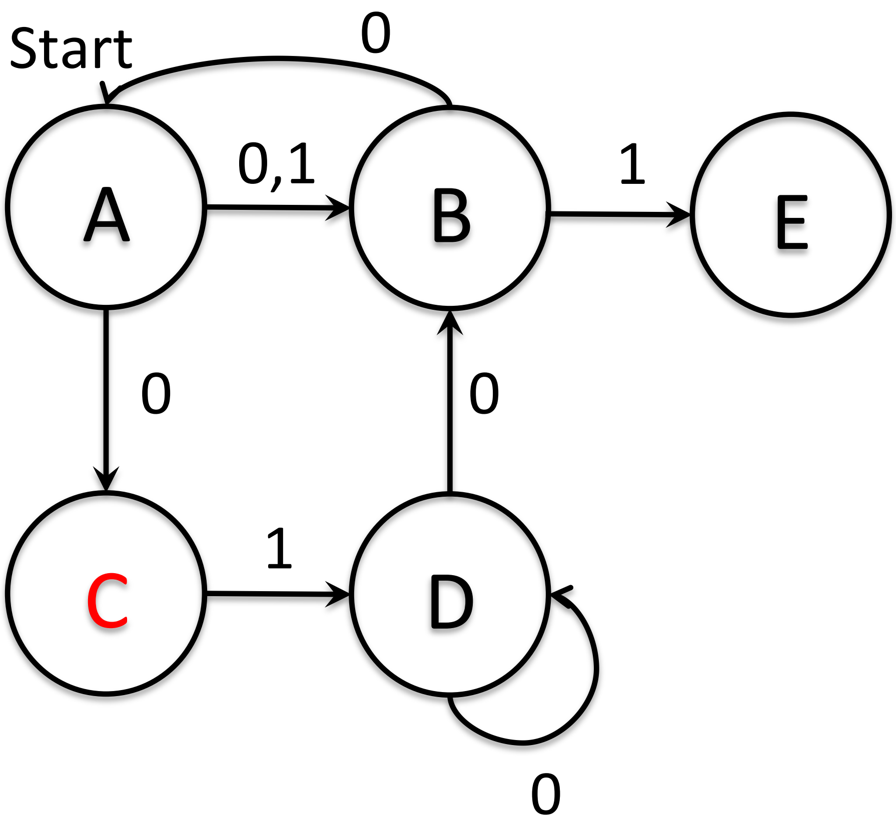
NFAs can have multiple next States



Key	Input	Paths
A	0	{B,C}
A	1	{B}
B	0	{A}
B	1	{E}

Sometimes we cannot map from a State a single next State

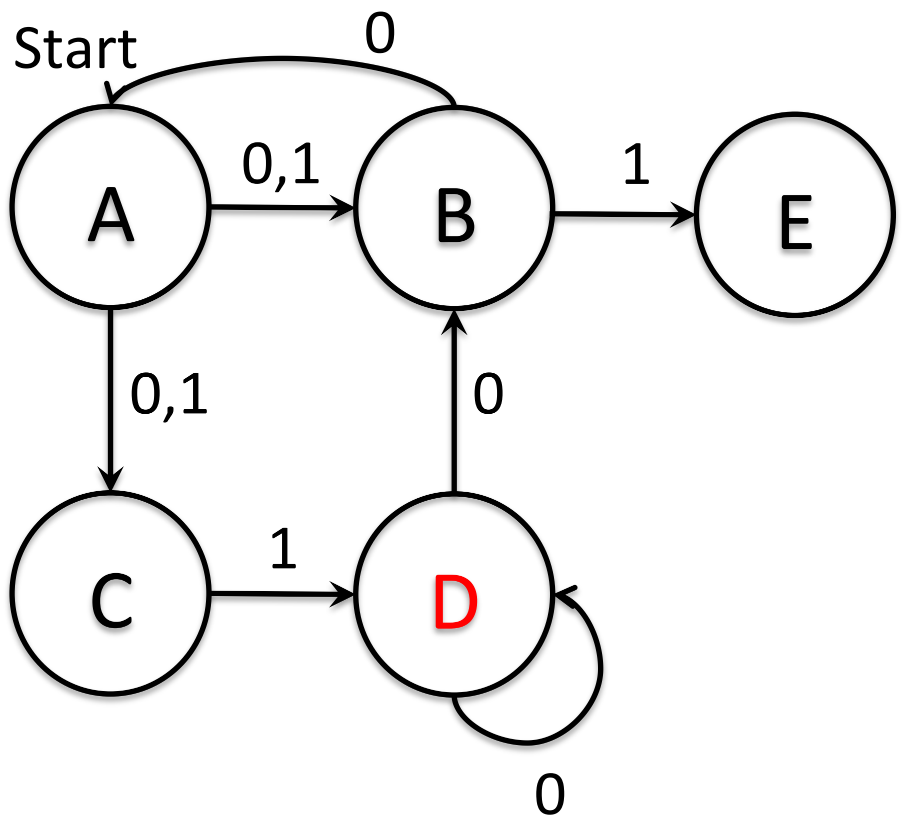
NFAs can have multiple next States



Key	Input	Paths
A	0	{B,C}
A	1	{B}
B	0	{A}
B	1	{E}
C	0	{}
C	1	{D}

Sometimes we cannot map from a State a single next State

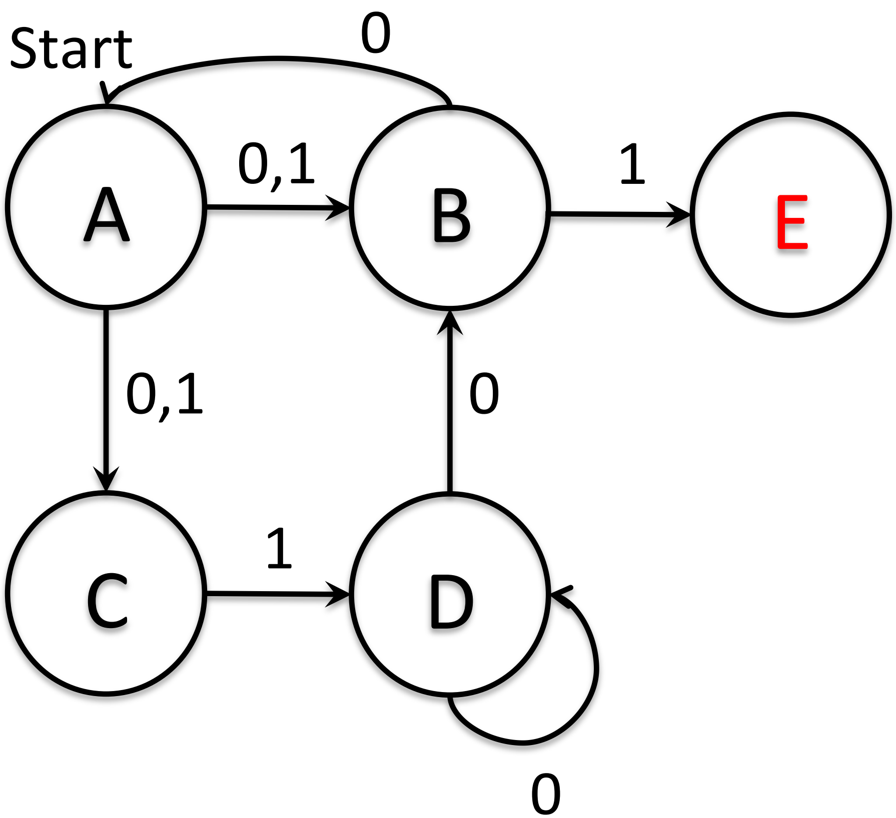
NFAs can have multiple next States



Key	Input	Paths
A	0	{B,C}
A	1	{B}
B	0	{A}
B	1	{E}
C	0	{}
C	1	{D}
D	0	{B,D}
D	1	{}

Sometimes we cannot map from a State a single next State

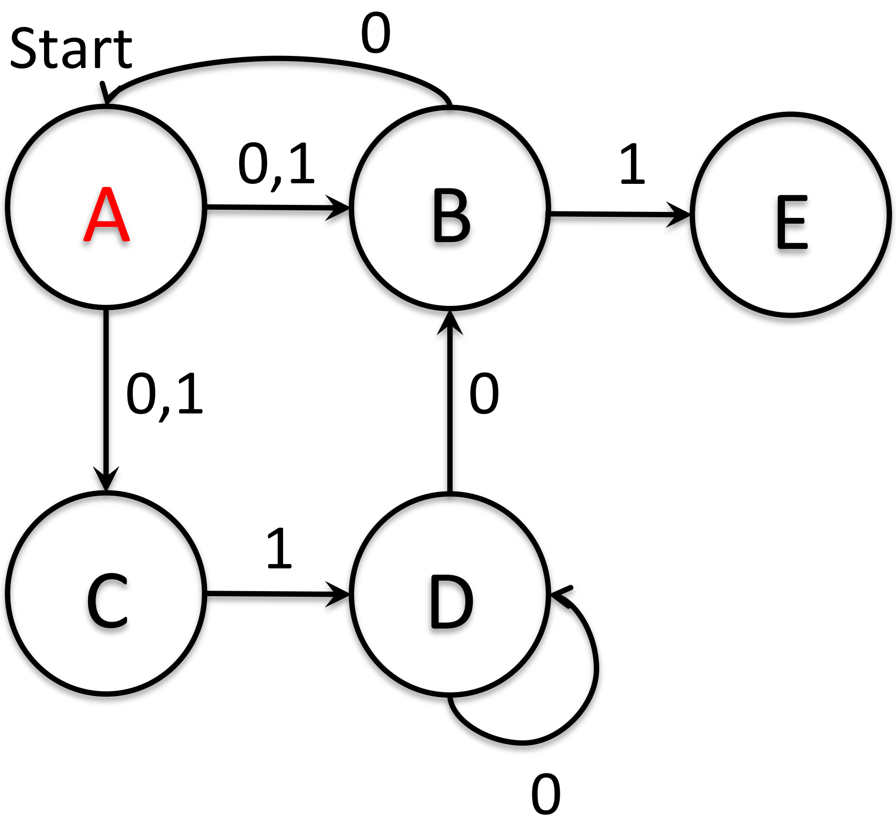
NFAs can have multiple next States



Key	**Input**	**Paths**
A	0	{B,C}
A	1	{B}
B	0	{A}
B	1	{E}
C	0	{}
C	1	{D}
D	0	{B,D}
D	1	{}
E	0	{}
E	1	{}

In that case, must keep track of all possible States

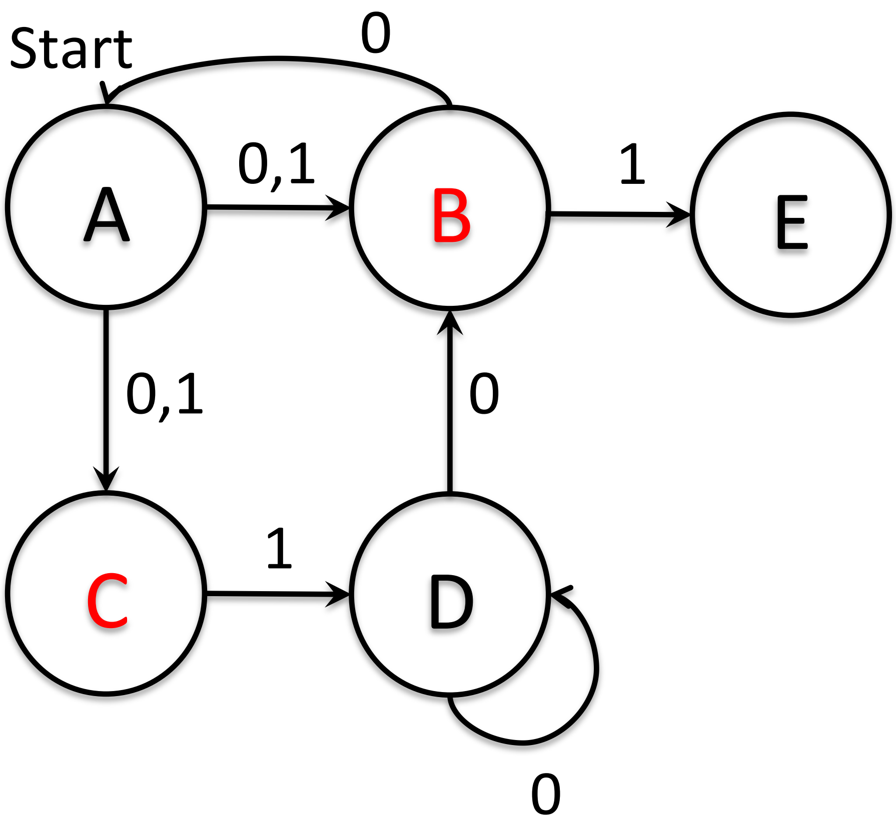
NFAs can have multiple next States



Input	**Possible States**
Start	{A}

In that case, must keep track of all possible States

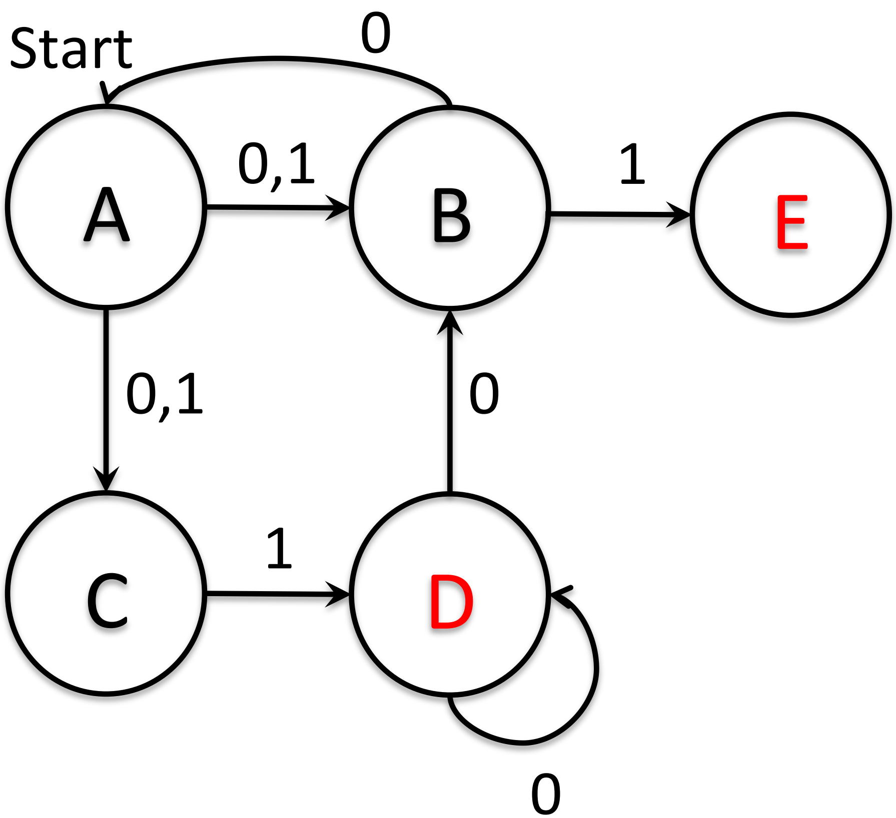
NFAs can have multiple next States



Input	Possible States
Start	{A}
0	{B,C}

In that case, must keep track of all possible States

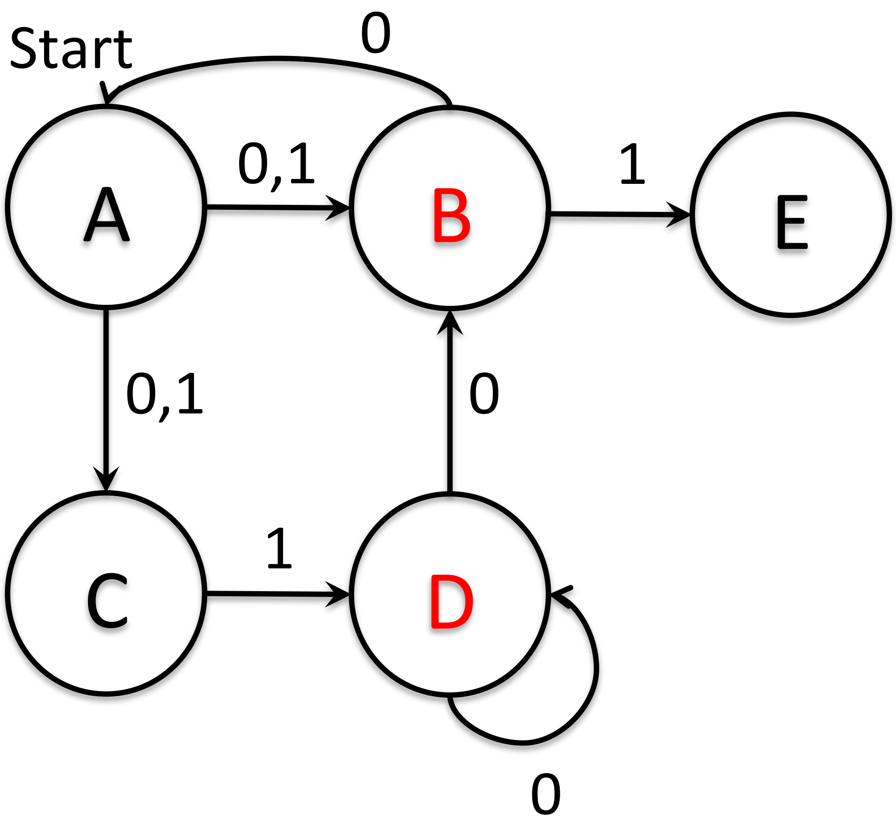
NFAs can have multiple next States



Input	**Possible States**
Start	{A}
0	{B,C}
1	{E,D}

In that case, must keep track of all possible States

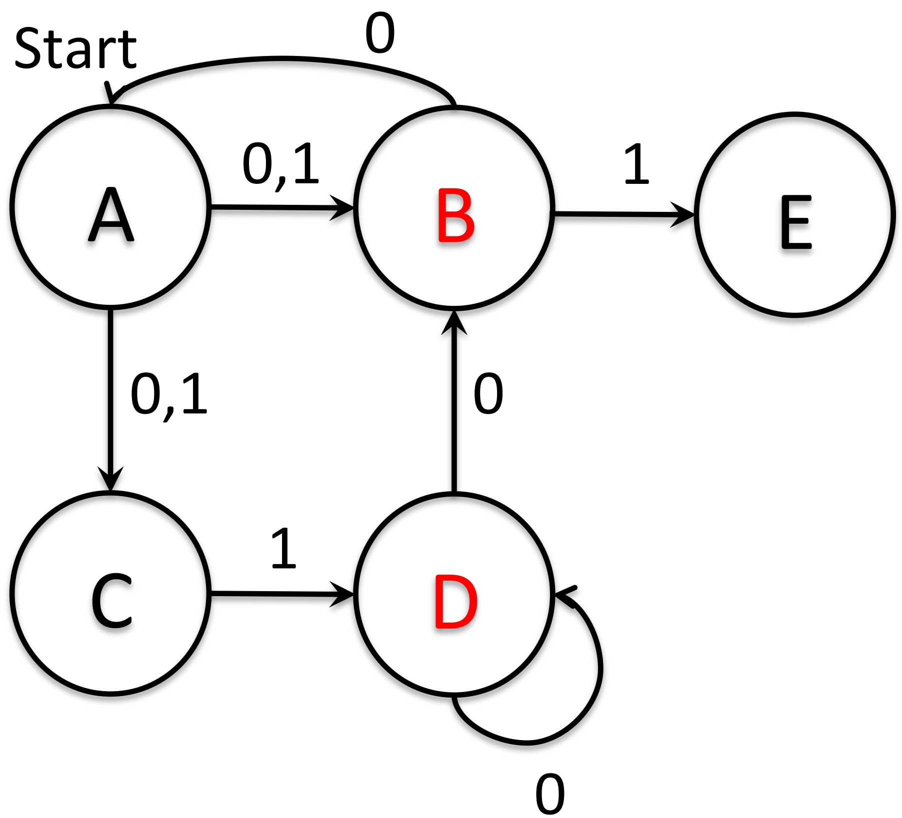
NFAs can have multiple next States



Input	Possible States
Start	{A}
0	{B,C}
1	{E,D}
0	{B,D}

In that case, must keep track of all possible States

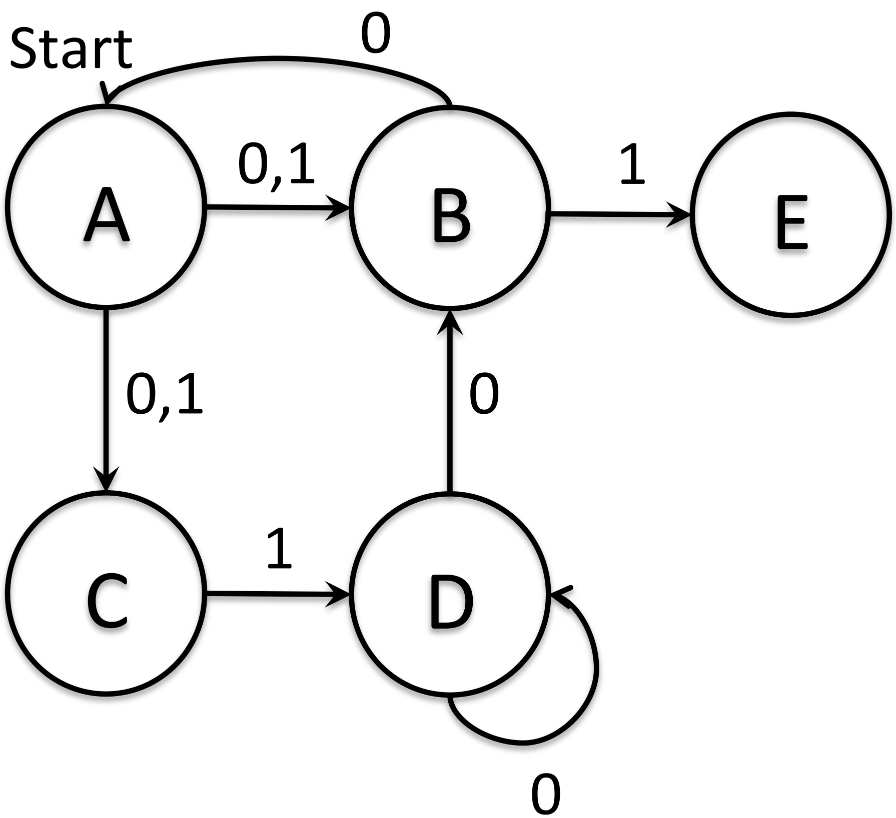
NFAs can have multiple next States



Input	**Possible States**
Start	{A}
0	{B,C}
1	{E,D}
0	{B,D}

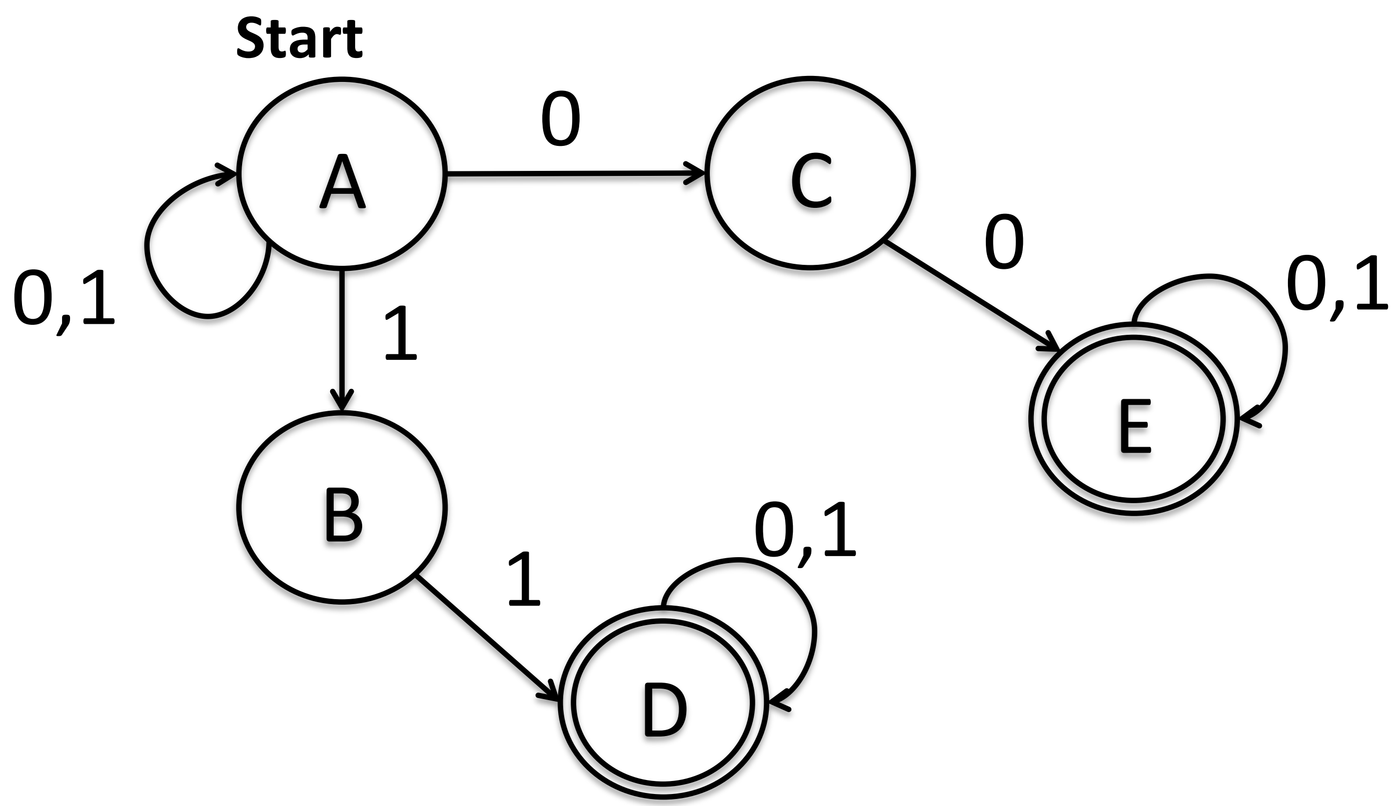
In that case, must keep track of all possible States

NFAs can have multiple next States



Input	**Possible States**
Start	{A}
0	{B,C}
1	{E,D}
0	{B,D}
…	

What does this NFA do?



Agenda

1. Regular expressions
2. Finite automata
3. Validating input
4. Modeling a complex system

Finite Automata are also used to track the State of a system as event occur

Sensors detect arrival and departure of cars in spaces

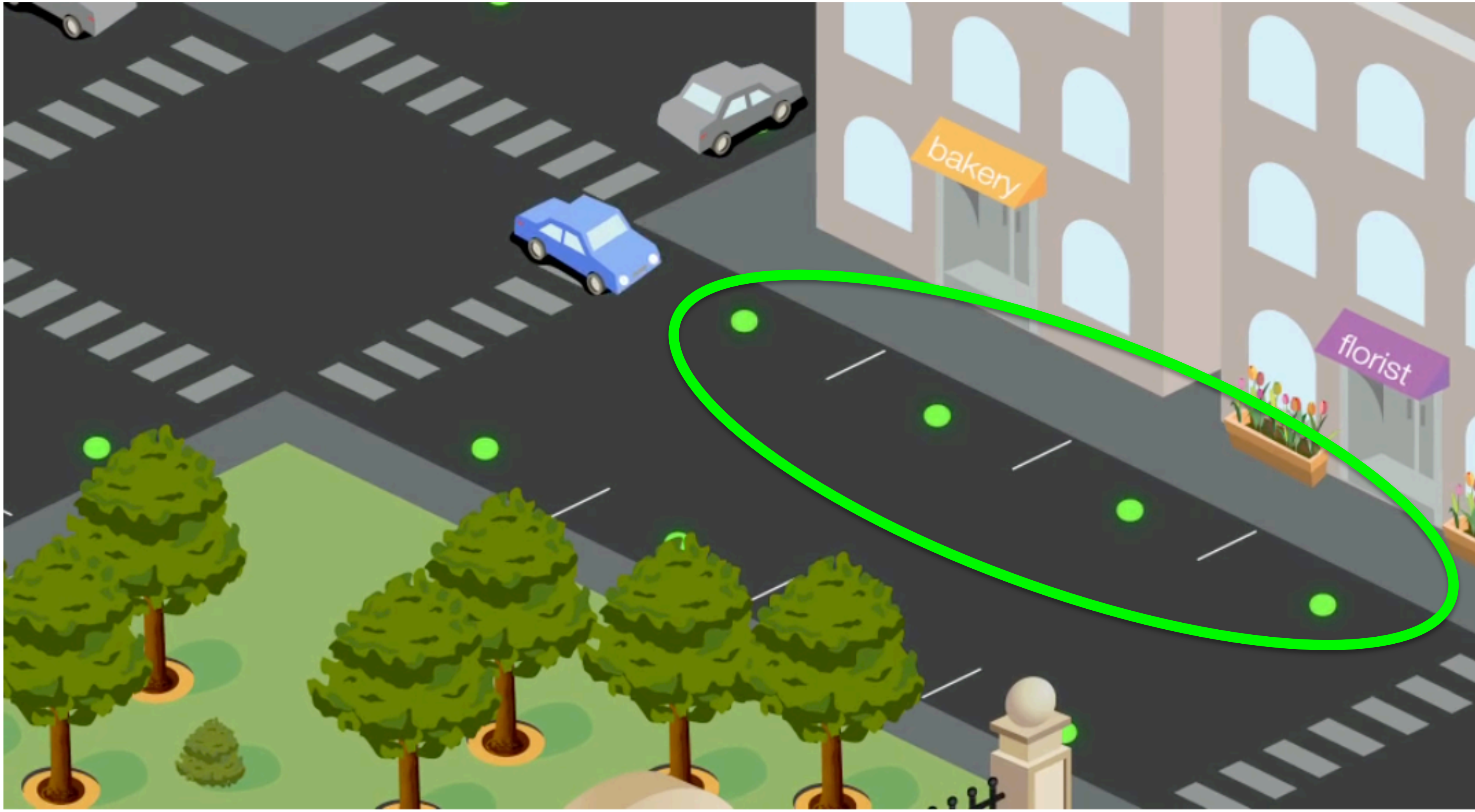


Image: Fybr.com

Finite Automata are also used to track the State of a system as event occur

Parking meters detect payments and payment expirations

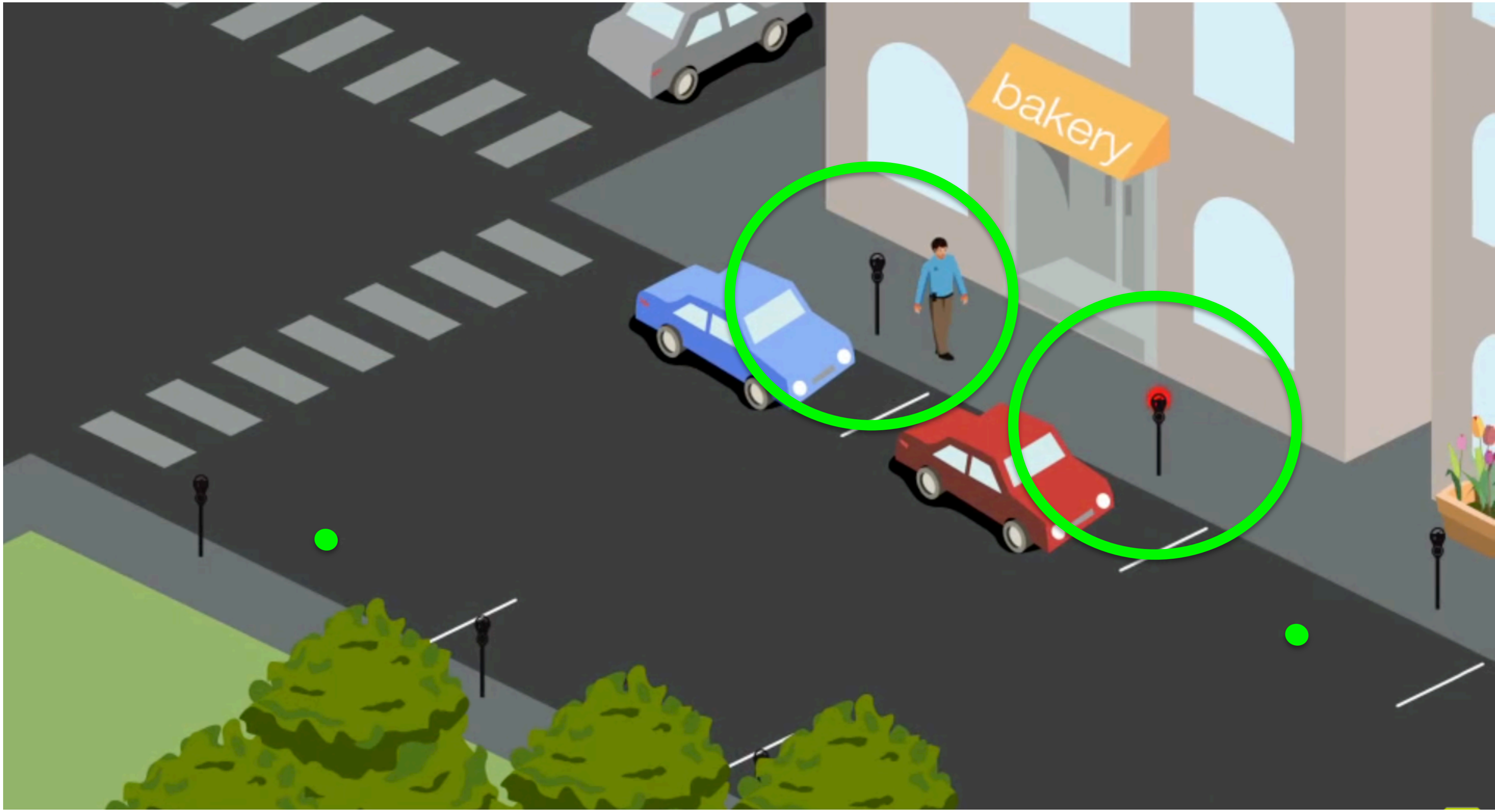


Image: Fybr.com

Combine data for all spaces on a block to show drivers where they can find parking

Fisherman's Wharf in San Francisco, CA

Green < 75% occupied, yellow = 75-90% occupied, red > 90% occupied

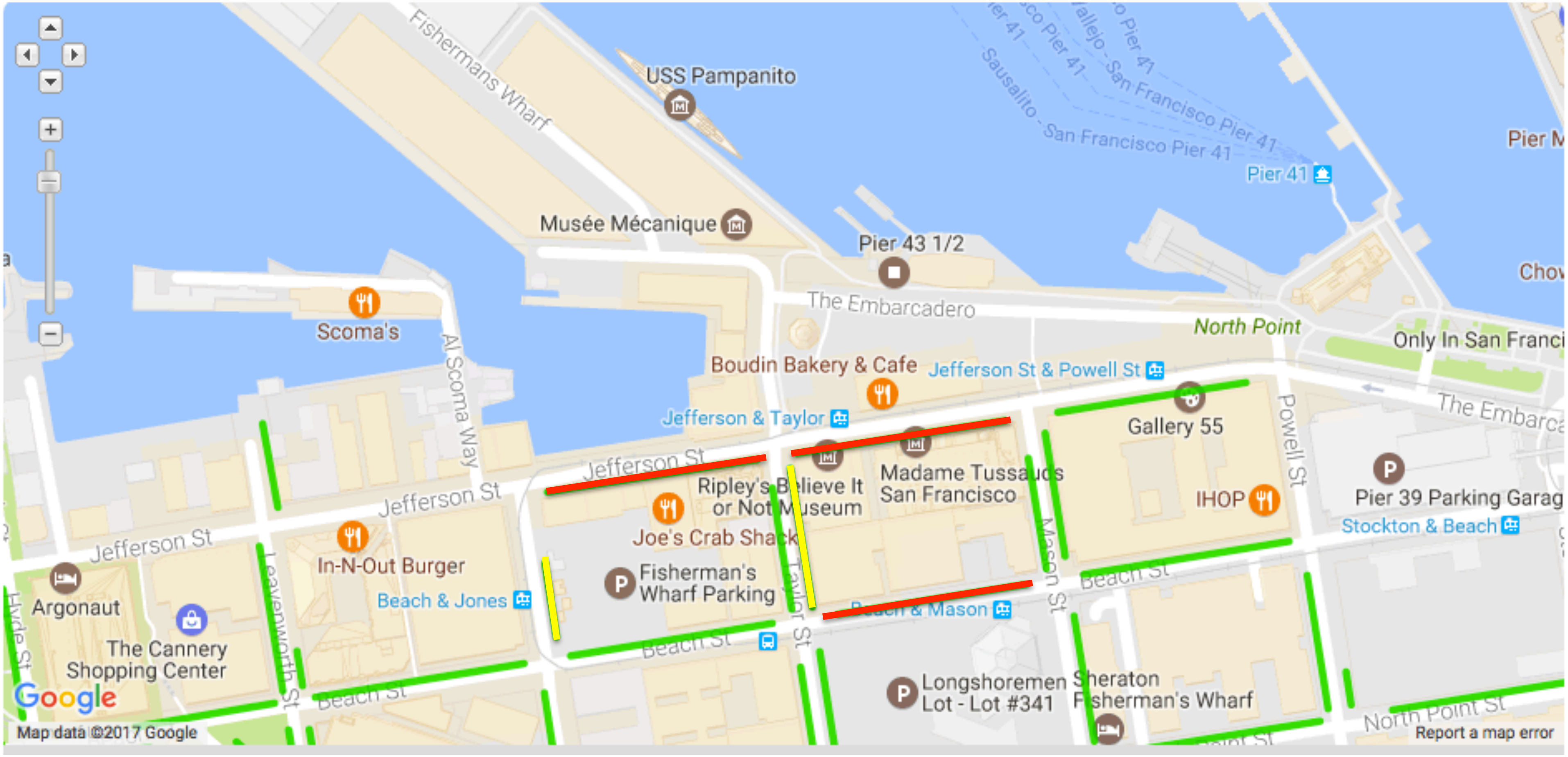


Image: sfpark.org

Combination of occupancy and payments leads to four states for each space

Simplified automobile parking

		Occupancy	
		Vacant	**Occupied**
Payment status	**Not Paid**	Vacant Not paid	Occupied Not paid
	Paid	Vacant Paid	Occupied Paid

Occupancy and payment events can occur and change the state of the space

Simplified automobile parking

		Occupancy	
		Vacant	**Occupied**
Payment status	**Not Paid**	Vacant Not paid	Occupied Not paid
	Paid	Vacant Paid	Occupied Paid

Occupancy event raised by sensor:

- Vehicle arrives
- Vehicle departs

Payment events raised by parking meter:

- Payment made
- Time expires

Events cause the system to transition between states

The parking space could be modeled with a complicated if-then structure

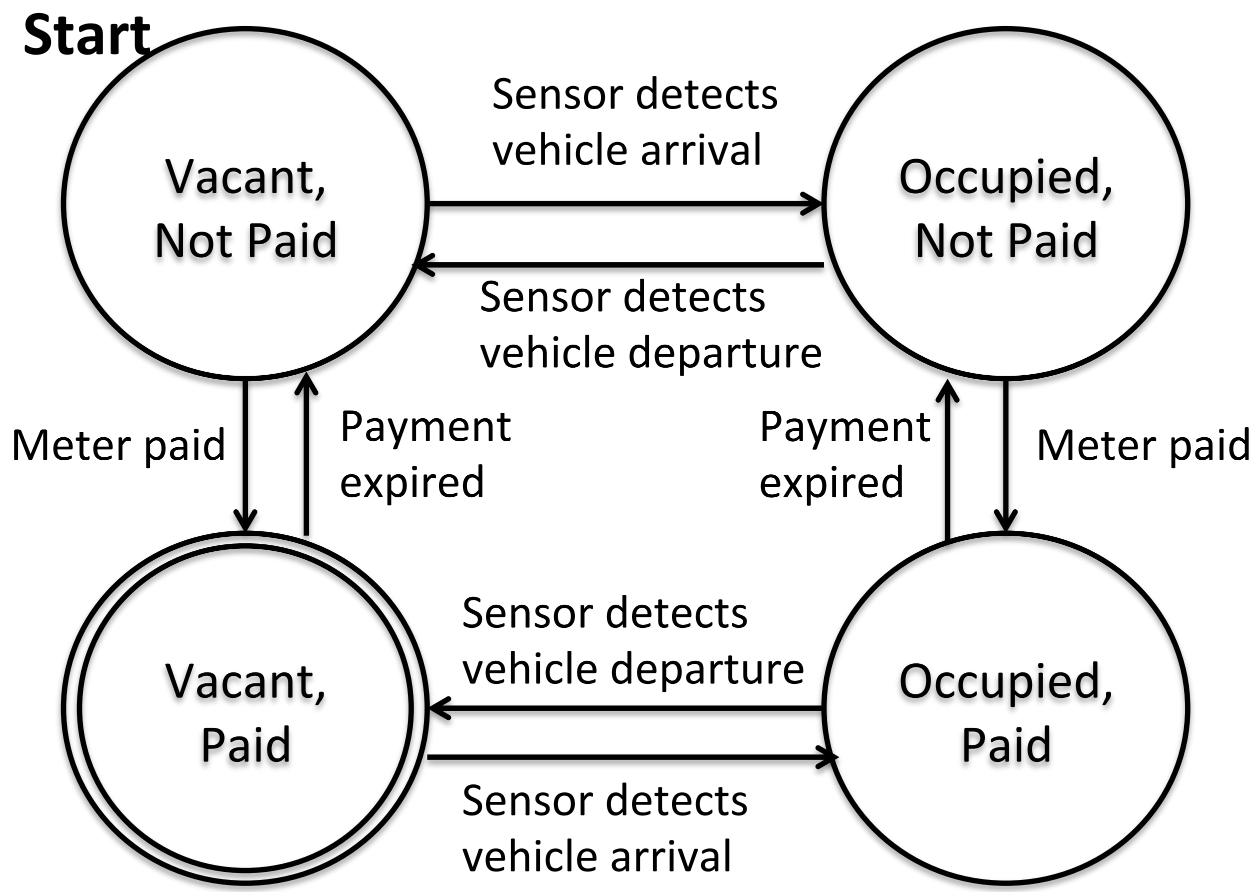
Simplified automobile parking

		Occupancy	
		Vacant	**Occupied**
Payment status	**Not Paid**	Vacant Not paid	Occupied Not paid
	Paid	Vacant Paid	Occupied Paid

```
void handleEvent(Event e) {
    if (event==“Payment”) {
        if (occupancy==“Occupied” && payment==“Not Paid”) {
            //add time on meter
        elseif (occupancy=“Occupied” && payment==“Paid”) {
            //increment time on meter
        …
```

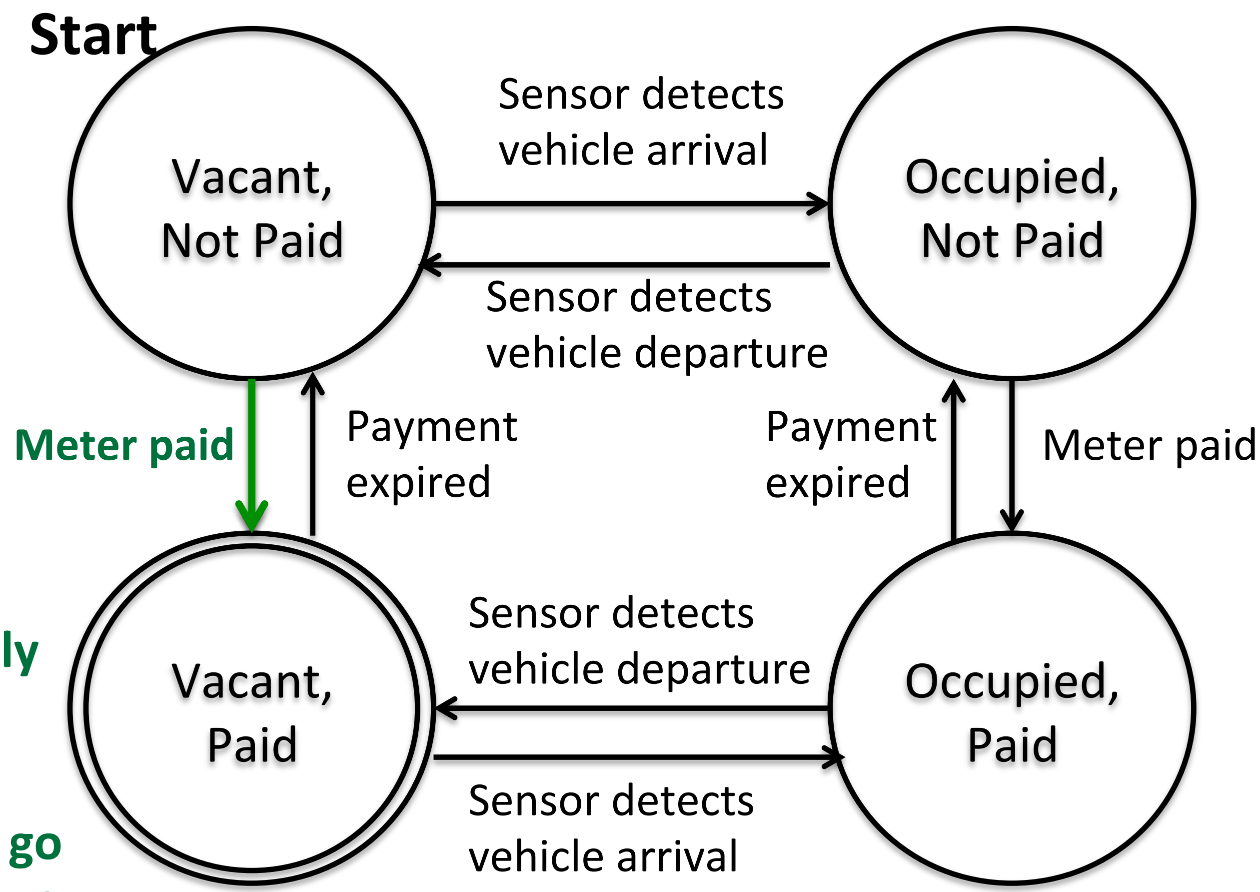
The parking space could be modeled more simply with a Finite Automata

Simplified automobile parking



The parking space could be modeled more simply with a Finite Automata

Simplified automobile parking



Sensor probably missed vehicle arrival, go figure out why!

Code review

DFA.java

- Store start as String
- Store ends as a Set (could be multiple ends)
- Constructor
  - Takes set of States (with Start and Ends labeled) and transitions (A,B,0 means from A go to B if given 0)
  - Track start and end vertices
  - Track transitions as Map (state-> Map(character, next state))
- match
  - Start with current = start
  - For each input
    - Ensure transition to next state is valid
    - Move to next state
  - Return final state

Code review

NFA.java

- Store start as String
- Store ends as a Set (could be multiple ends)
- Transitions now store list of possible states
- Constructor
  - Takes set of States (with Start and Ends labeled) and transitions (A,B,0 means from A go to B if given 0)
  - Track start and end vertices
  - Transactions: Map (state-> Map(character, **<u>List(String)</u>**))
- match
  - Start with currentStates = start (could be multiple valid current states!)
  - For each input
    - Check possible next states from all valid current states