



# Automata, Transducers, and Hidden Markov Models

**Natalie Parde, Ph.D.**

Department of Computer  
Science

University of Illinois at  
Chicago

CS 421: Natural Language  
Processing

Fall 2019

Many slides adapted from Jurafsky and Martin  
(<https://web.stanford.edu/~jurafsky/slp3/>),  
Universiteit Utrecht's NLP course  
([http://www.phil.uu.nl/tst/2012/Slides/SLP\\_Lecture2.pdf](http://www.phil.uu.nl/tst/2012/Slides/SLP_Lecture2.pdf)), and Ray Mooney's NLP course  
(<https://www.cs.utexas.edu/~mooney/cs388/>).

# What are finite state automata?

- **Computational models that can generate regular languages** (such as those specified by a regular expression)
- Also used in other NLP applications that function by **transitioning between finite states**
  - Dialogue systems
  - Morphological parsing
- Singular: Finite State Automaton (FSA)
- Plural: Finite State Automata (FSAs)

# Key Components

---

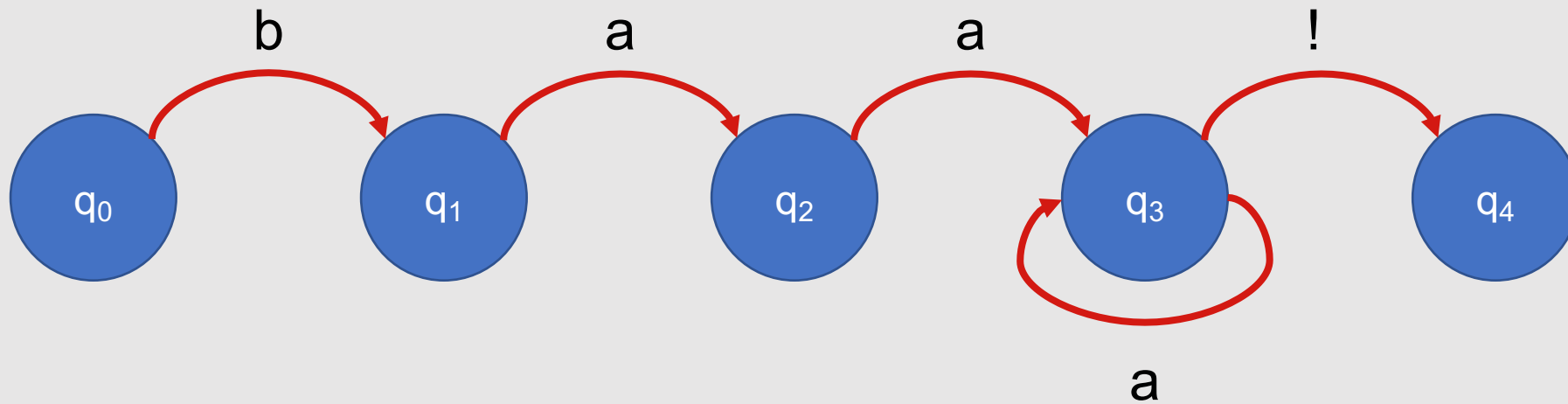
- Finite set of states
  - Start state
  - Final state
- Set of transitions from one state to another

# How do FSAs work?

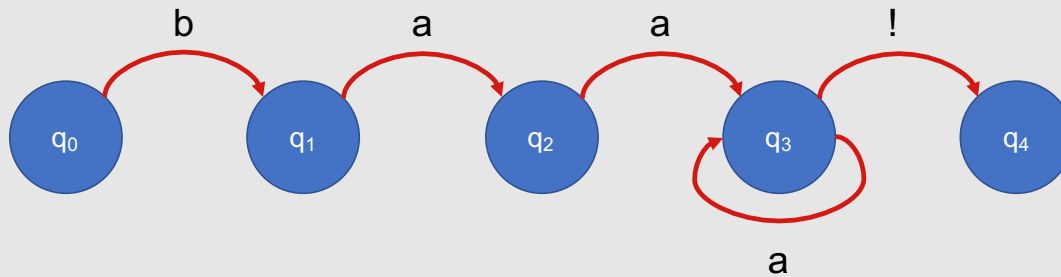
- For a given sequence of items (characters, words, etc.) to match, **begin in the start state**
- **If the next item** in the sequence **matches a state that can be transitioned to** from the current state, **go to that state**
- **Repeat**
  - If no transitions are possible, **stop**
  - If the state you stopped in is a final state, **accept the sequence**

# FSAs are often represented graphically.

- Nodes = states
- Arcs = transitions



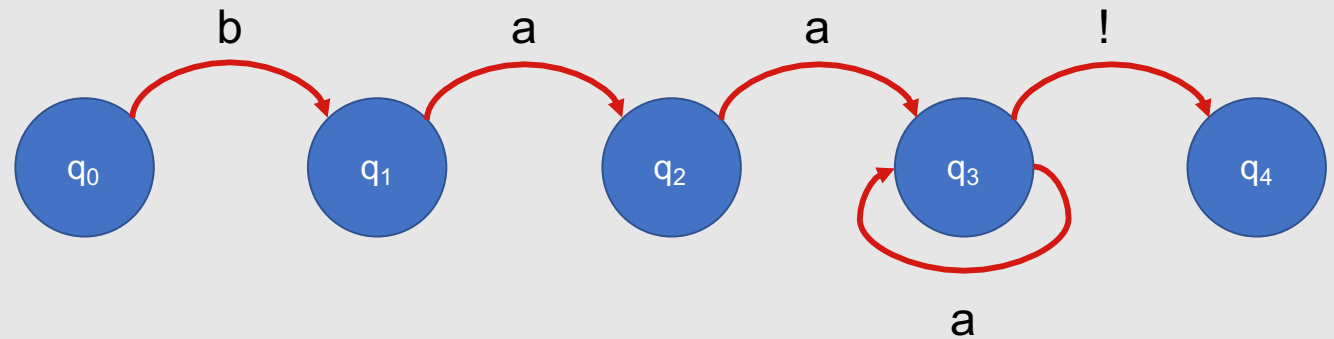
# What do we know about this FSA?



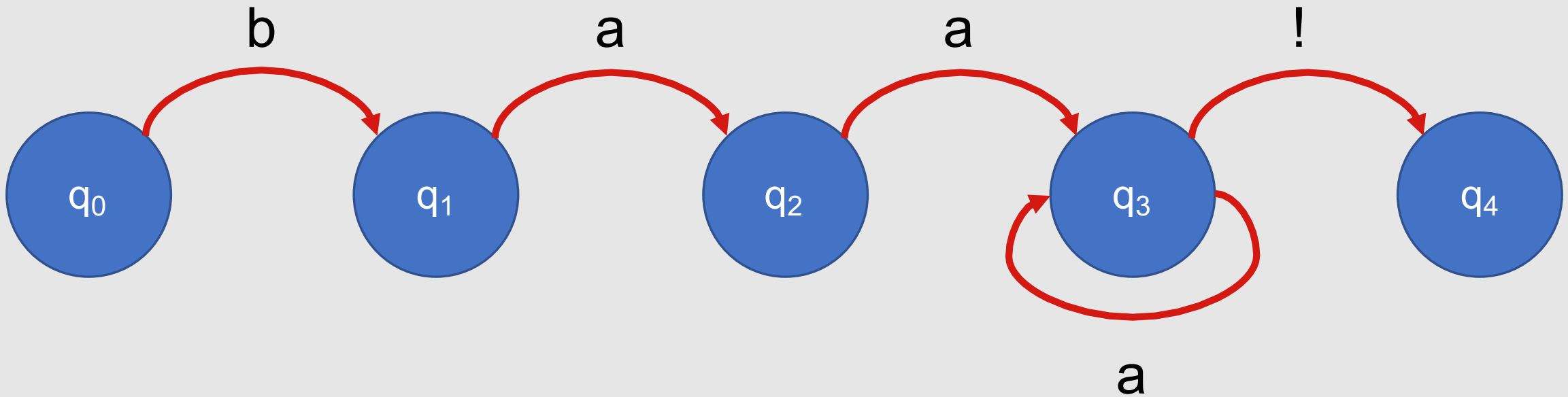
- Five states
  - $q_0$  is the start state
  - $q_4$  is the final (accept) state
- Five transitions
- Alphabet =  $\{a, b, !\}$

Which strings could  
this FSA match?

- baa!
  - baaaa!
  - ba!
  - baaaaaaaaa!
  - baaaa
  - baabaa!
- 
- <https://www.google.com/search?q=timer>

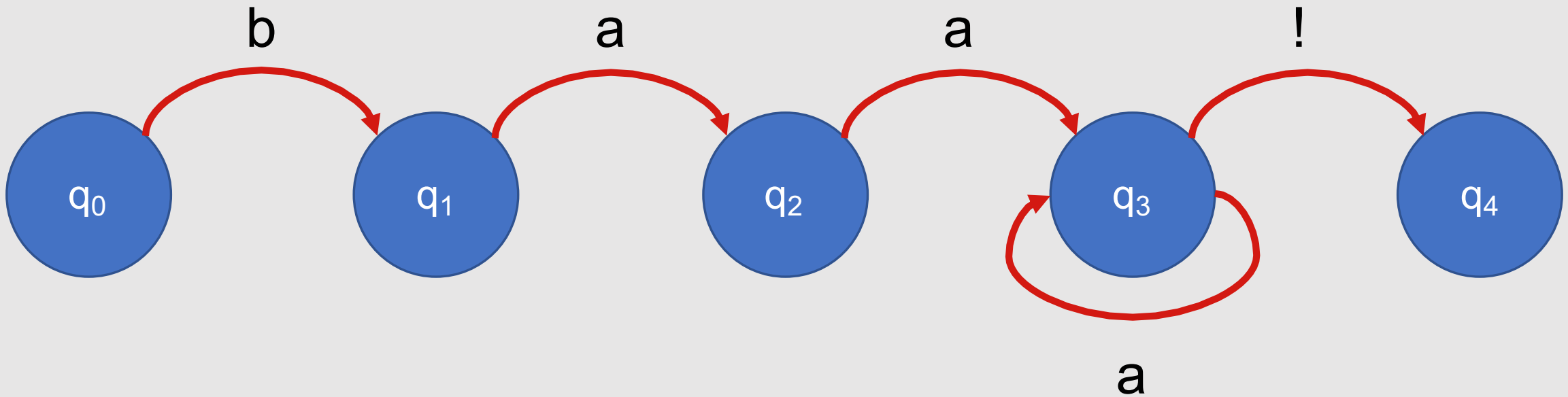


**Regex that this FSA matches: baa+!**



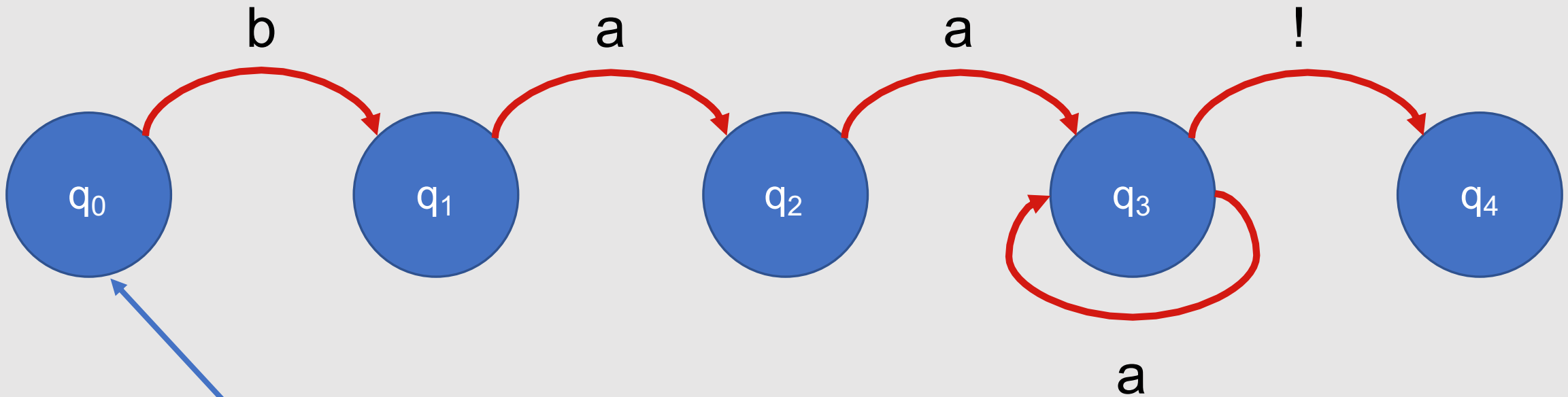


**Regex that this FSA matches: baa+!**



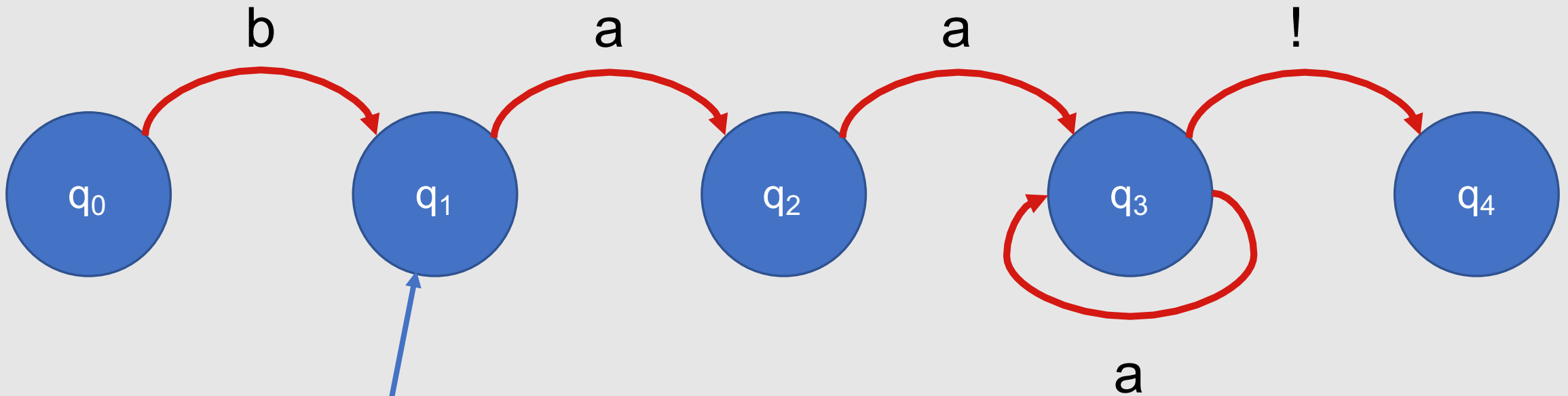
Test String: baa!

**Regex that this FSA matches: baa+!**



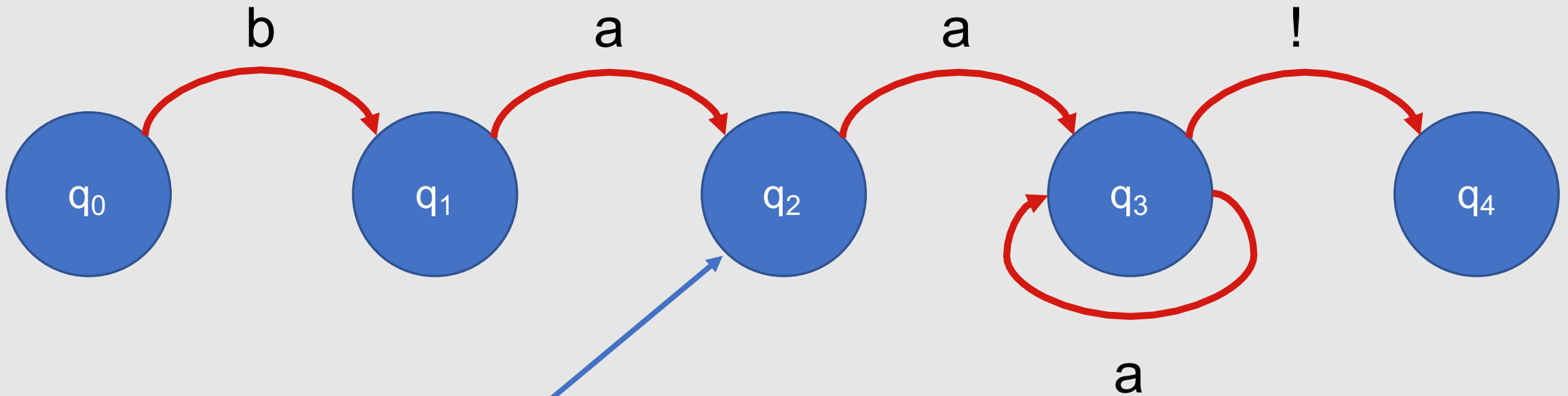
Test String: baa!

# Regex that this FSA matches: **baa+!**



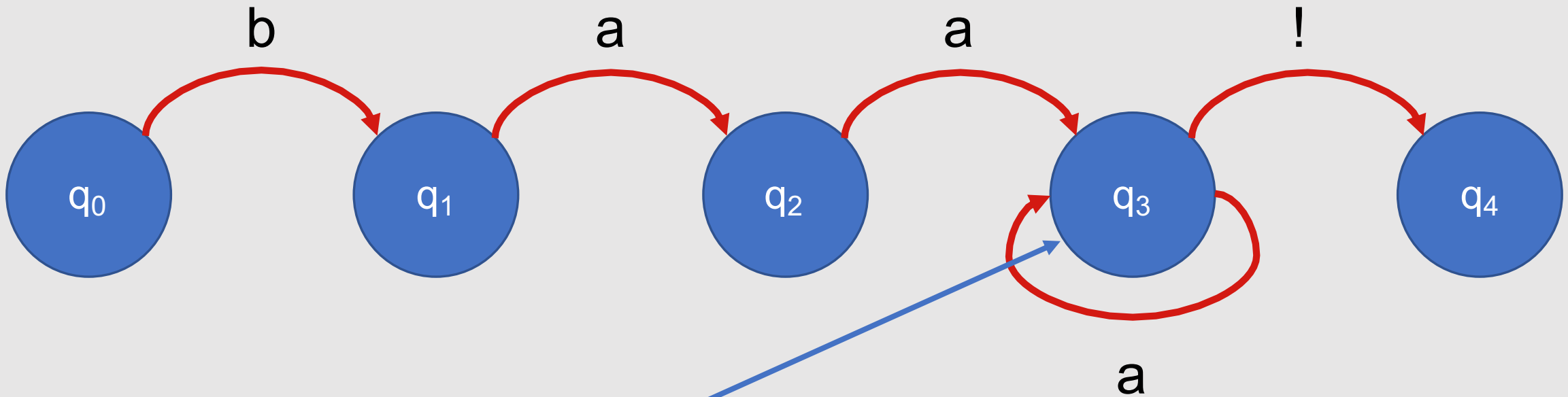
Test String: **b**aa!

# Regex that this FSA matches: baa+!



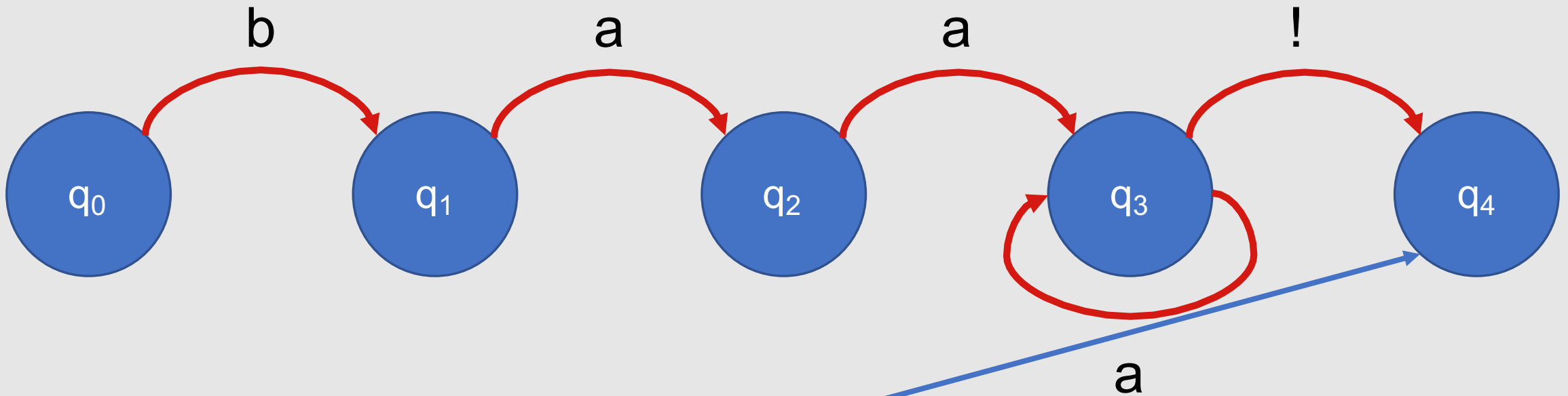
Test String: baa!

# Regex that this FSA matches: **baa+!**



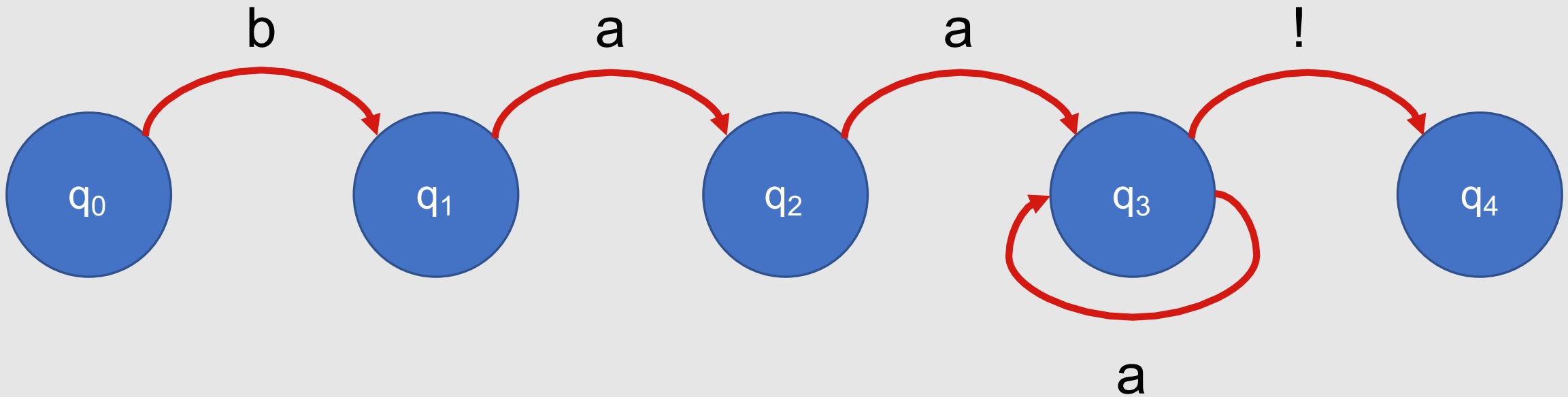
Test String: **baa!**

# Regex that this FSA matches: **baa+!**



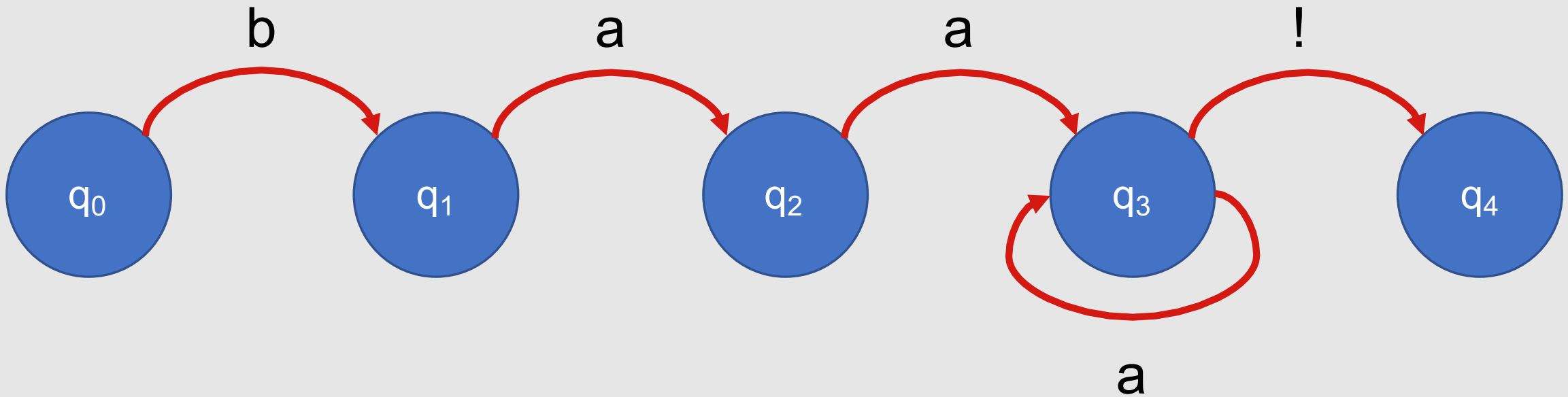
Test String: **baa!**

**Regex that this FSA matches: baa+!**



Test String: baa! 😊

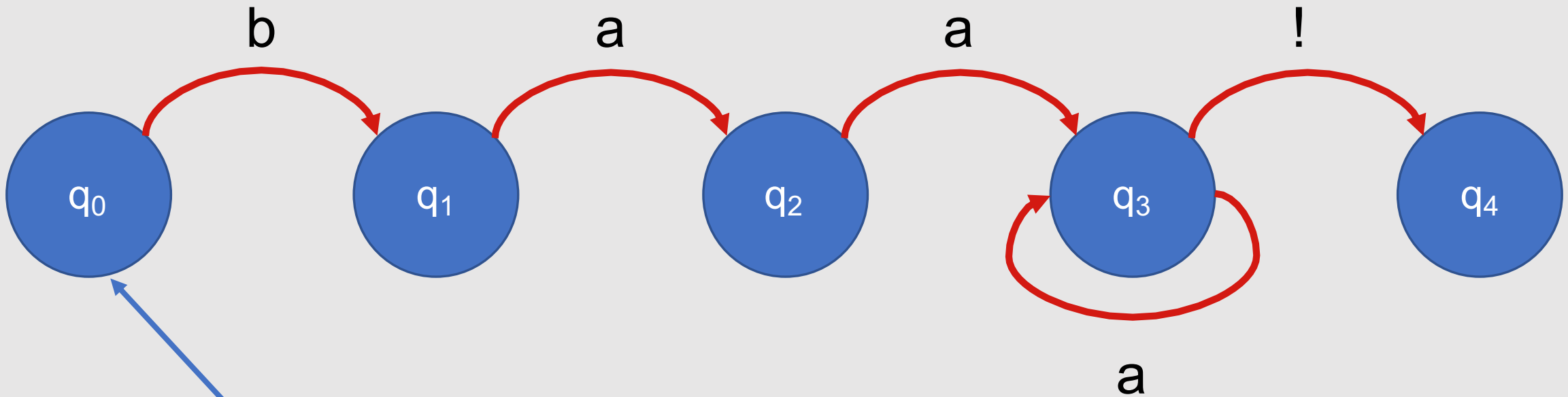
**Regex that this FSA matches: baa+!**



Test String: baabaa!

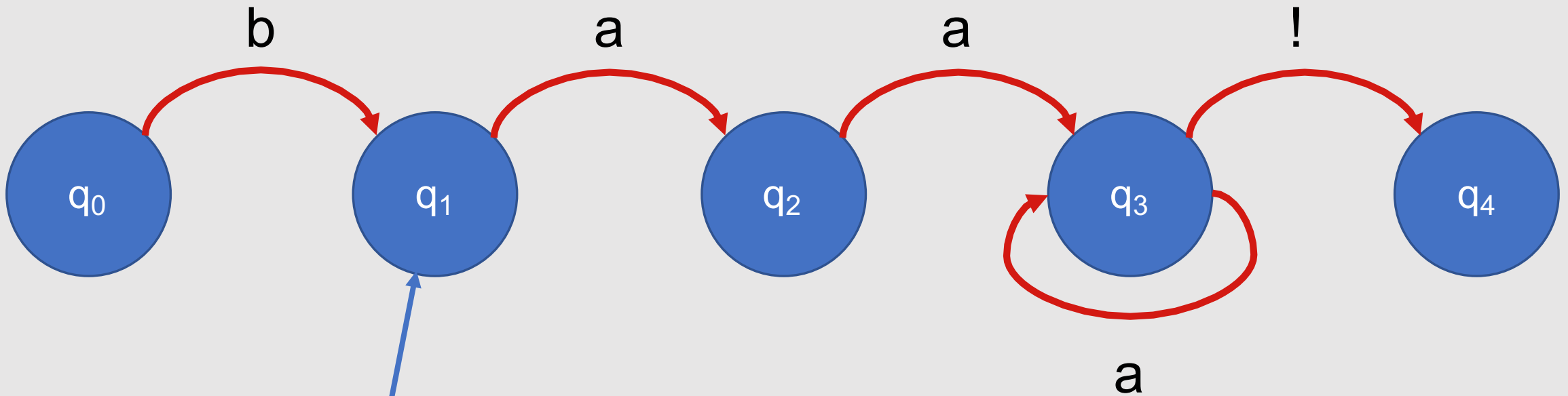


**Regex that this FSA matches: baa+!**



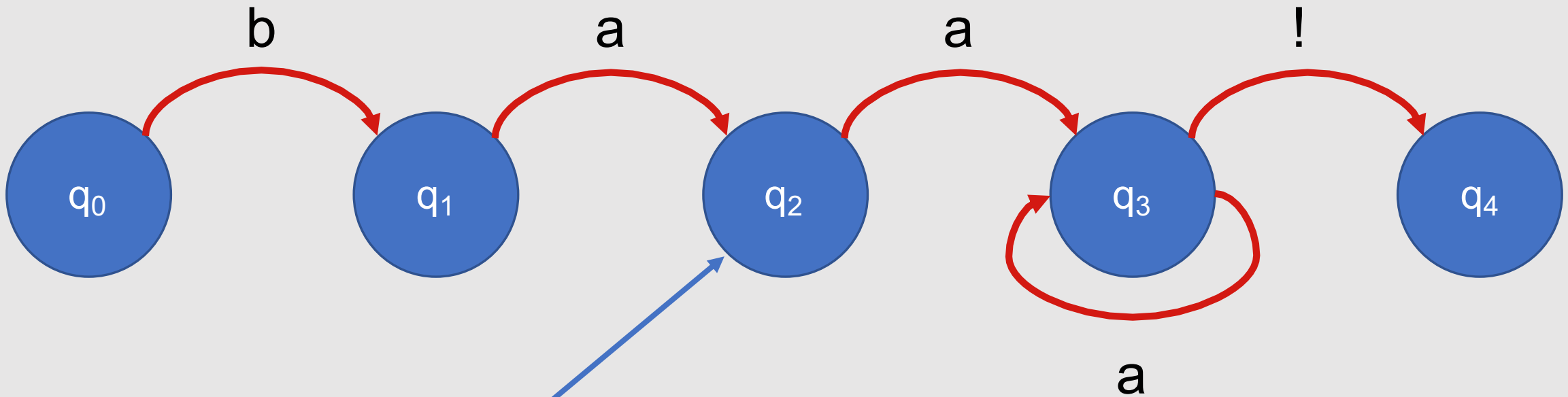
Test String: baabaa!

# Regex that this FSA matches: **baa+!**



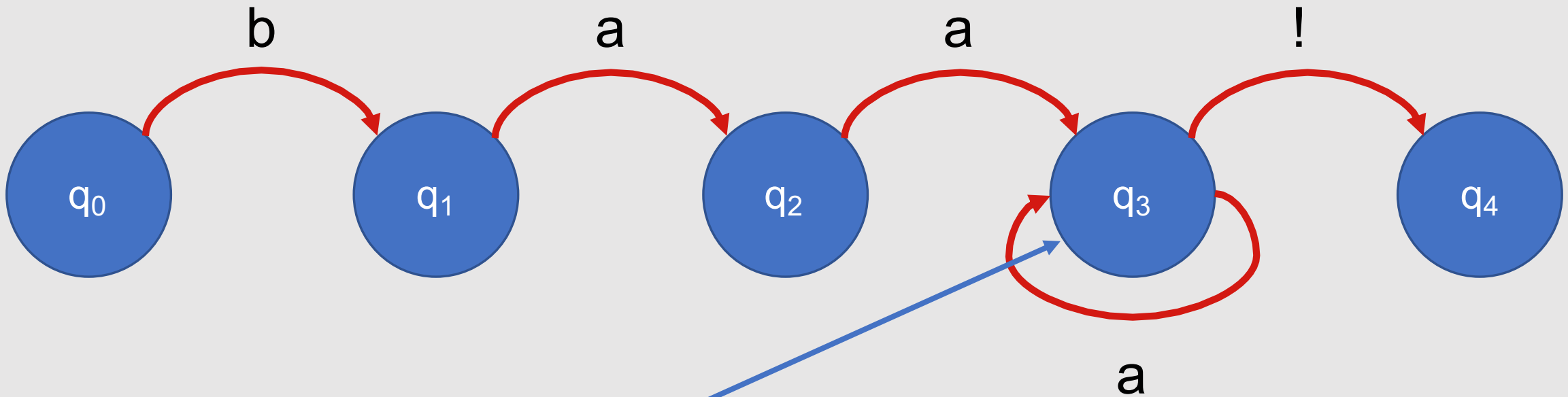
Test String: **b**aaabaa!

**Regex that this FSA matches: baa+!**



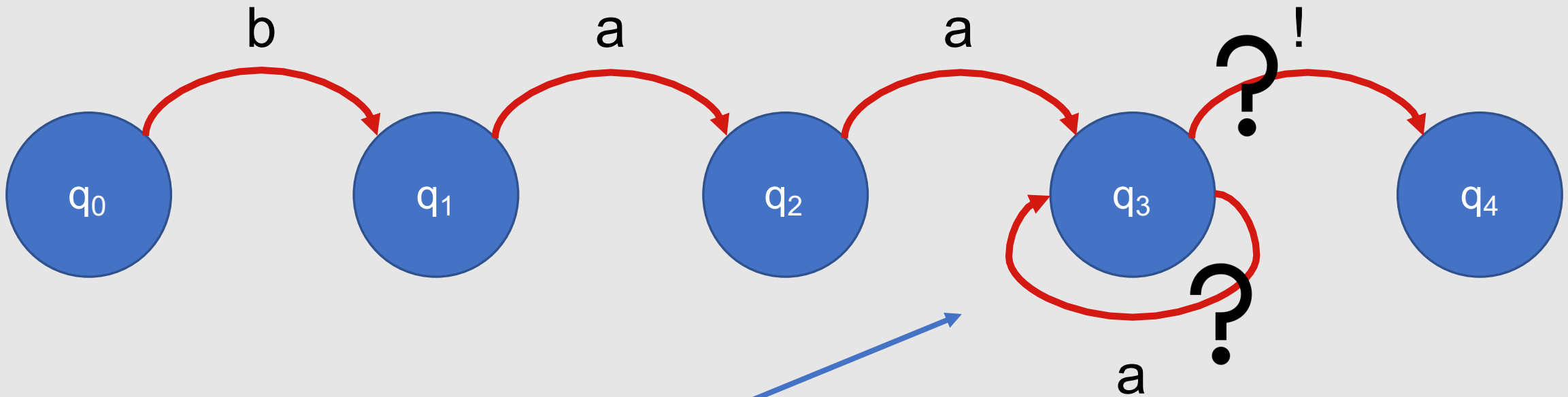
Test String: baabaa!

# Regex that this FSA matches: baa+!



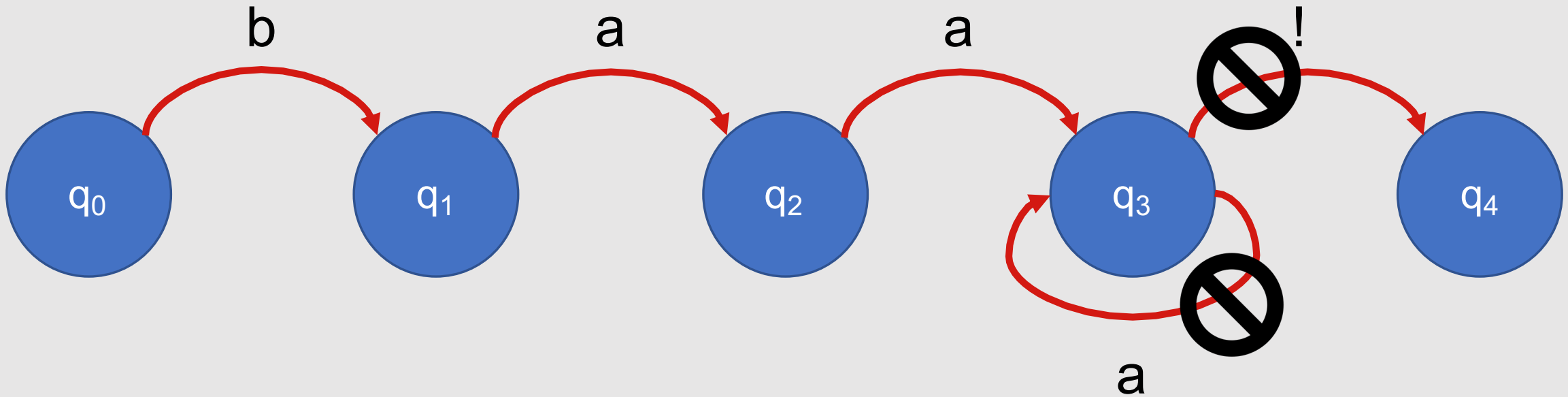
Test String: baabaa!

# Regex that this FSA matches: baa+!



Test String: baabaa! 😞

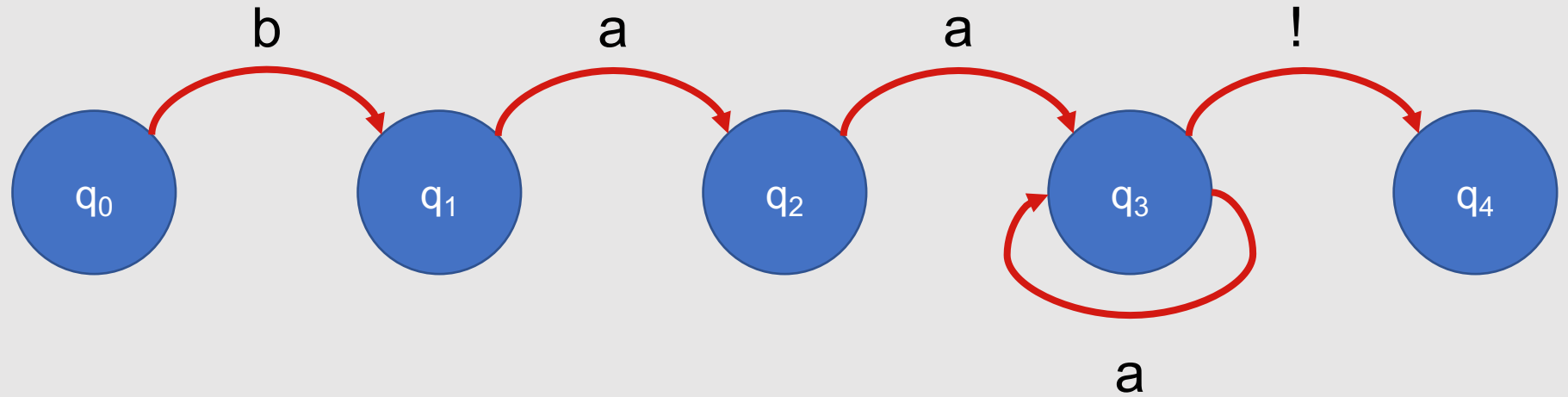
**Regex that this FSA matches: baa+!**



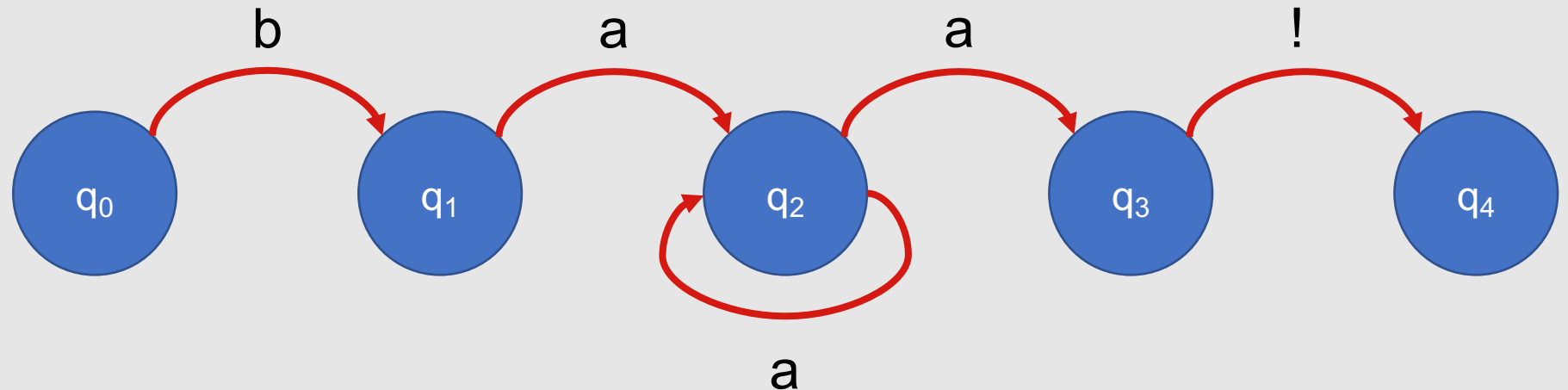
Test String: baabaa! ☹️

# Note: More than one FSA can correspond to the same regular language!

Test String:  
baaa!



Test String:  
baaa!



# Formal Definition

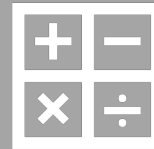
- A finite state automaton can be specified by enumerating the following properties:
  - The set of states,  $Q$
  - A finite alphabet,  $\Sigma$
  - A start state,  $q_0$
  - A set of accept/final states,  $F \subseteq Q$
  - A transition function or transition matrix between states,  $\delta(q, i)$
- $\delta(q, i)$ : Given a state  $q \in Q$  and input  $i \in \Sigma$ ,  $\delta(q, i)$  returns a new state  $q' \in Q$ .



# Alphabets



In the previous definition, alphabet does not necessarily mean  $[a-zA-Z]!$

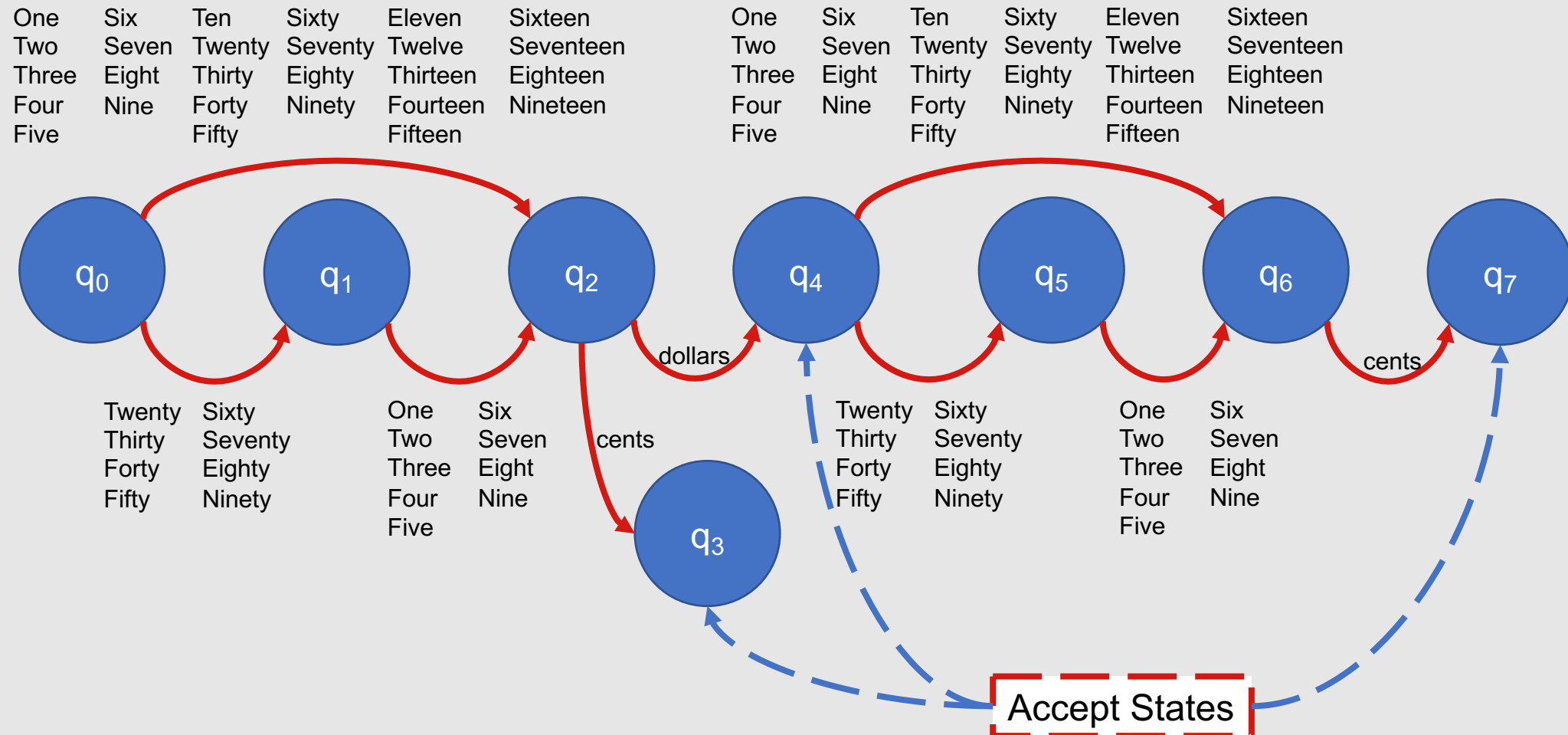


Alphabet = finite set of possible input symbols

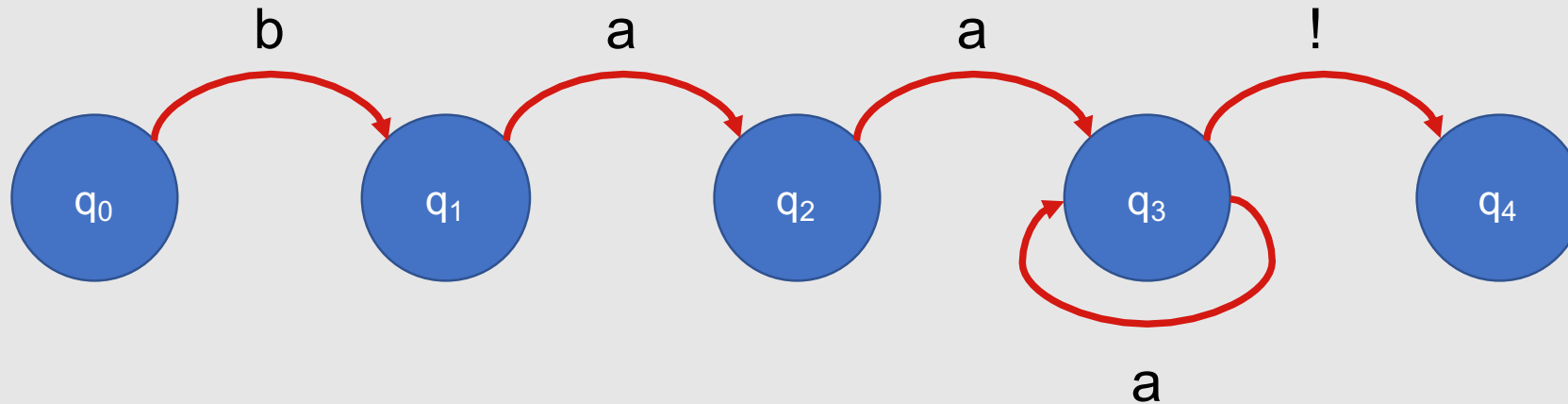


An alphabet can be a subset of letters (e.g.,  $\{a, b\}$ ), a combination of letters and other characters (e.g.,  $\{a, b, !\}$ ), a subset of words (e.g.,  $\{\text{lamb, sheep, baa!}\}$ ), etc.

# Example: FSA for Dollar Amounts



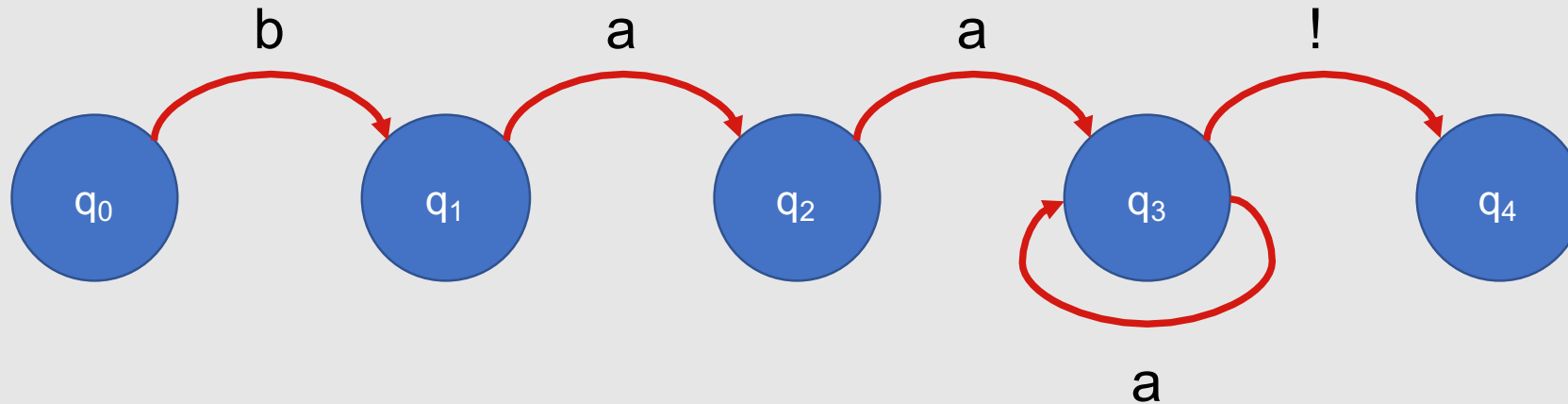
# State transitions in FSAs can be represented using tables.



		Next Item in Sequence			
		b	a	!	<end>
Currently in State	$q_0$	$q_1$			
	$q_1$				
	$q_2$				
	$q_3$				
	$q_4$				

Go to State

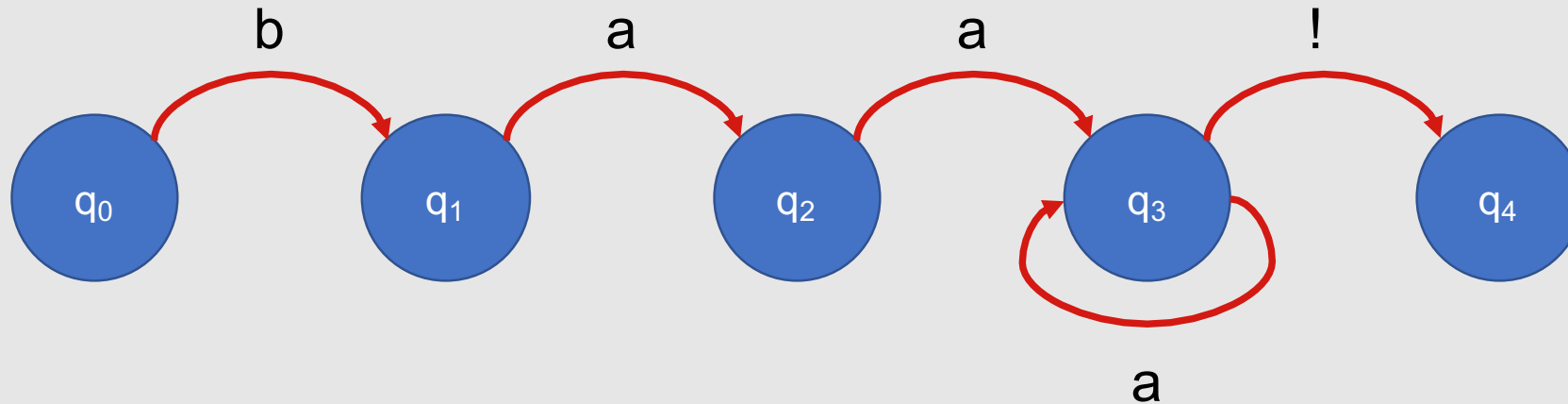
# State transitions in FSAs can be represented using tables.



		Next Item in Sequence			
Currently in State		b	a	!	<end>
	q <sub>0</sub>	q <sub>1</sub>	☹	☹	☹
	q <sub>1</sub>				
	q <sub>2</sub>				
	q <sub>3</sub>				
	q <sub>4</sub>				

Go to State

# State transitions in FSAs can be represented using tables.



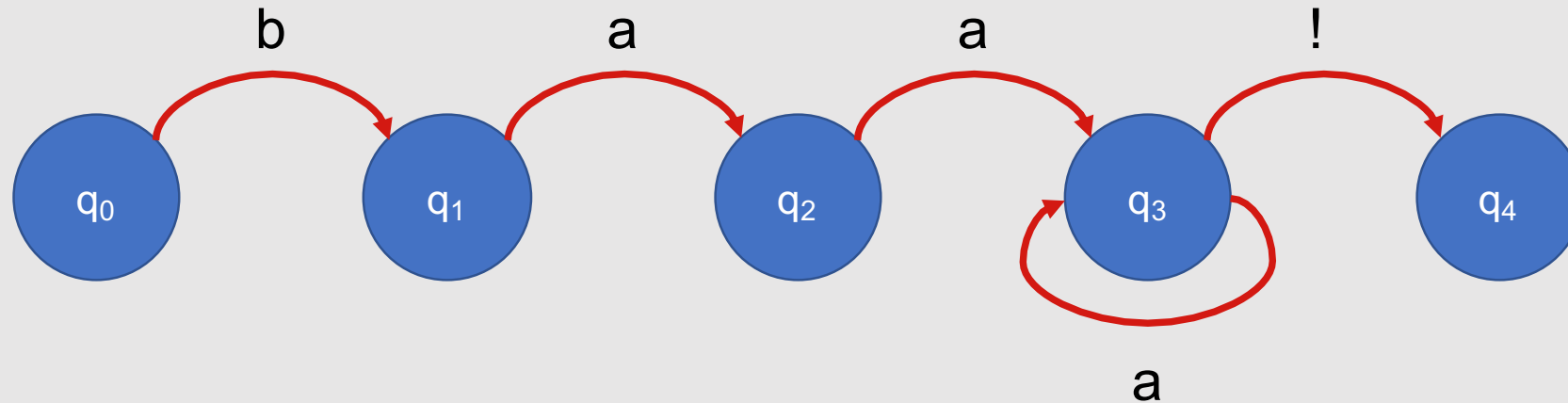
Next Item in Sequence

	b	a	!	<end>
q <sub>0</sub>	q <sub>1</sub>	☹	☹	☹
q <sub>1</sub>	☹	q <sub>2</sub>		
q <sub>2</sub>				
q <sub>3</sub>				
q <sub>4</sub>				

Currently in State

Go to State

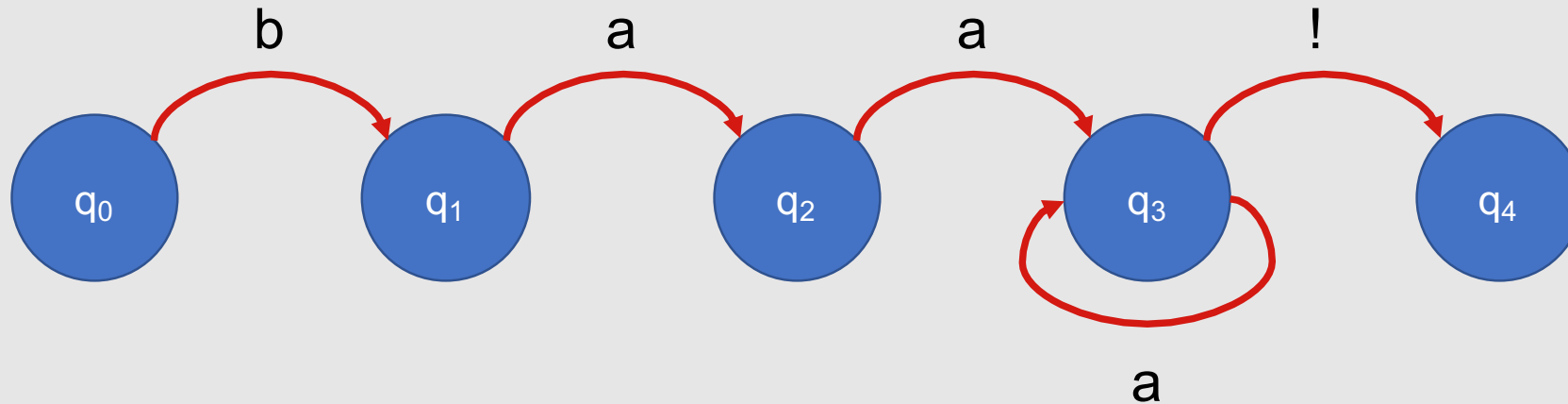
# State transitions in FSAs can be represented using tables.



		Next Item in Sequence			
		b	a	!	<end>
Currently in State	q <sub>0</sub>	q <sub>1</sub>	☹	☹	☹
	q <sub>1</sub>	☹	q <sub>2</sub>	☹	☹
	q <sub>2</sub>	☹	q <sub>3</sub>		
	q <sub>3</sub>				
	q <sub>4</sub>				

Go to State

# State transitions in FSAs can be represented using tables.



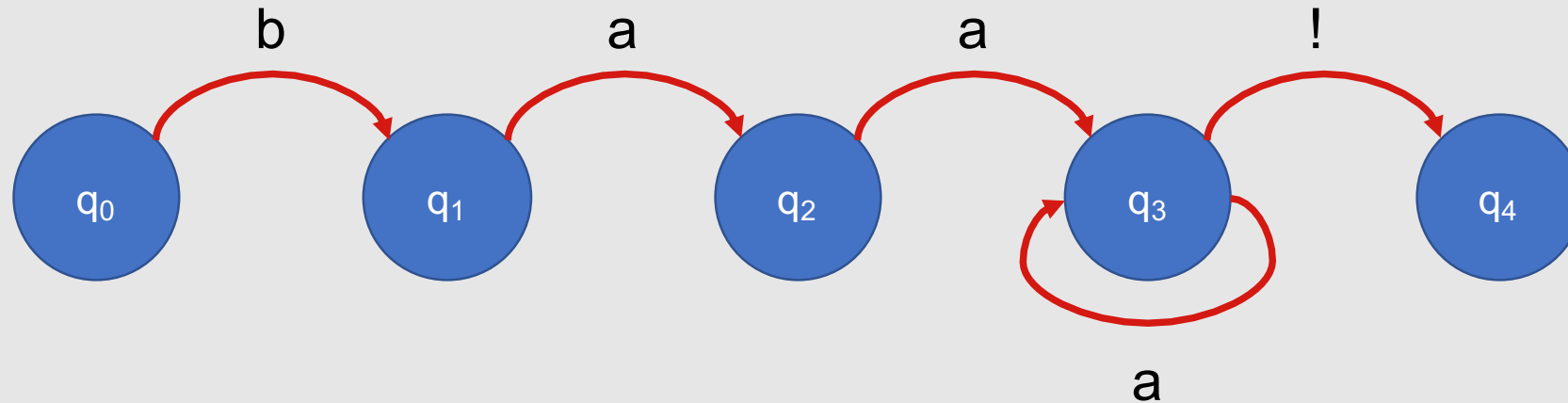
Next Item in Sequence

	b	a	!	<end>
q <sub>0</sub>	q <sub>1</sub>	☹	☹	☹
q <sub>1</sub>	☹	q <sub>2</sub>	☹	☹
q <sub>2</sub>	☹	q <sub>3</sub>	☹	☹
q <sub>3</sub>	☹	q <sub>3</sub>		
q <sub>4</sub>				

Currently in State

Go to State

# State transitions in FSAs can be represented using tables.



Next Item in Sequence

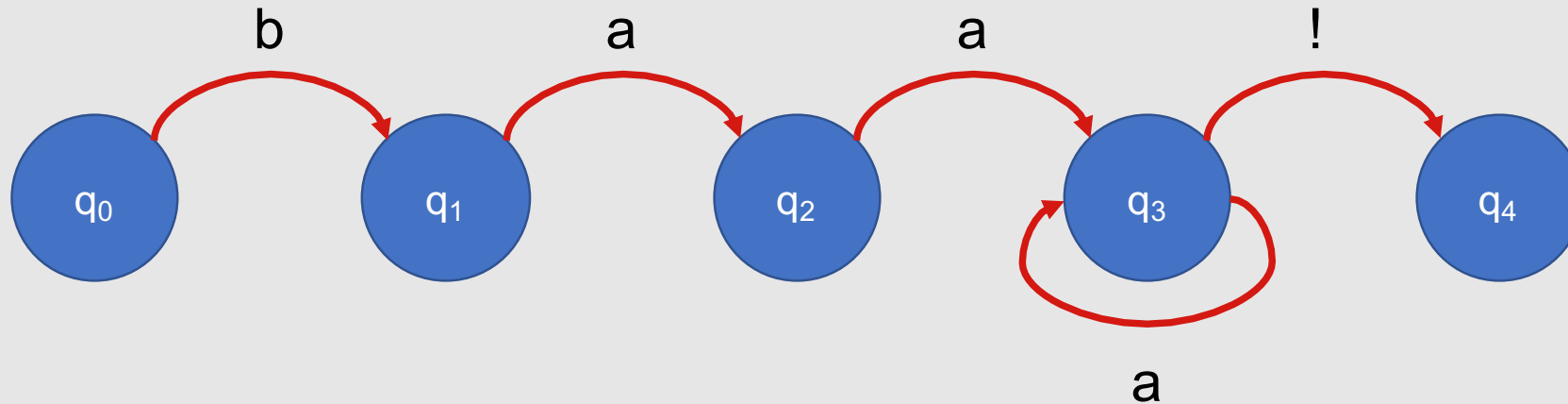
	b	a	!	<end>
q <sub>0</sub>	q <sub>1</sub>	☹	☹	☹
q <sub>1</sub>	☹	q <sub>2</sub>	☹	☹
q <sub>2</sub>	☹	q <sub>3</sub>	☹	☹
q <sub>3</sub>	☹	q <sub>3</sub>	q <sub>4</sub>	
q <sub>4</sub>				

Currently in State

Go to State



# State transitions in FSAs can be represented using tables.



		Next Item in Sequence			
Currently in State		b	a	!	<end>
	q <sub>0</sub>	q <sub>1</sub>	☹	☹	☹
	q <sub>1</sub>	☹	q <sub>2</sub>	☹	☹
	q <sub>2</sub>	☹	q <sub>3</sub>	☹	☹
	q <sub>3</sub>	☹	q <sub>3</sub>	q <sub>4</sub>	☹
	q <sub>4</sub>	☹	☹	☹	☺

Accept!

# State transition tables simplify the process of determining whether your input will be accepted by the FSA.

- For a given sequence of items to match, **begin in the start state** with the first item in the sequence
- **Consult the table** ...is a transition to any other state permissible with the current item?
- If so, **move to the state indicated by the table**
- If you make it to the end of your sequence and to a final state, **accept**

# Formal Algorithm

```
index ← beginning of sequence
current_state ← initial state of FSA
loop:
    if end of sequence has been reached:
        if current_state is an accept state:
            return accept
        else:
            return reject
    else if transition_table[current_state, sequence[index]] is empty:
        return reject
    else:
        current_state ← transition_table[current_state, sequence[index]]
        index ← index + 1
end
```

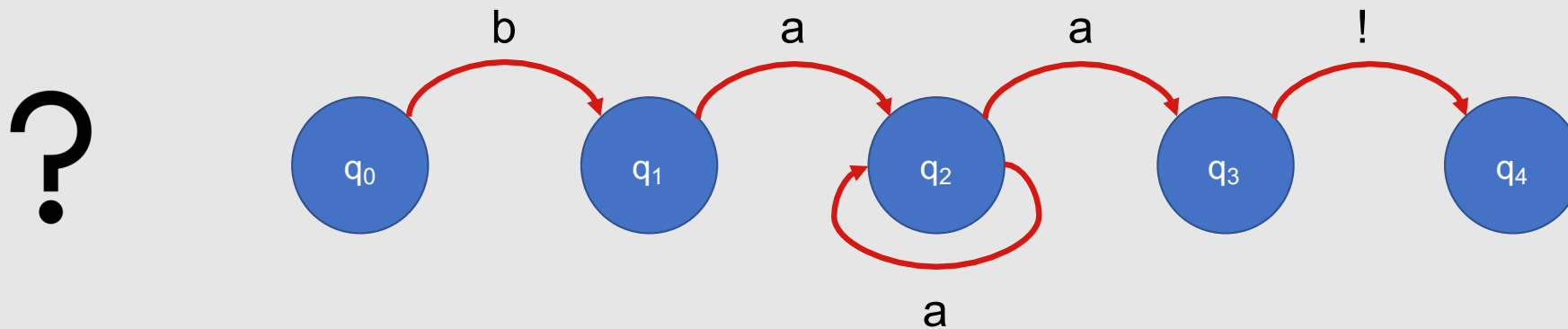
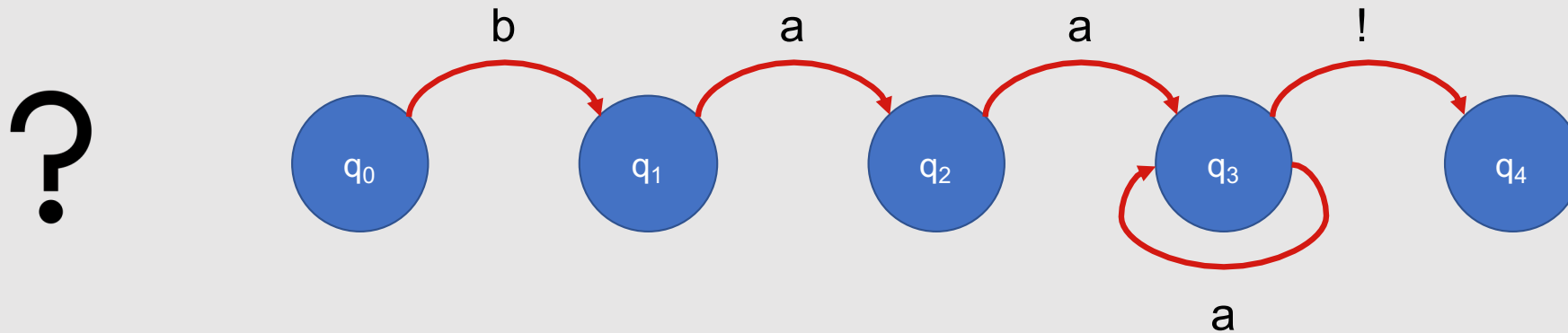
# Deterministic vs. Non-Deterministic FSAs

---

**Deterministic FSA:** At each point in processing a sequence, there is one unique thing to do (no choices!)

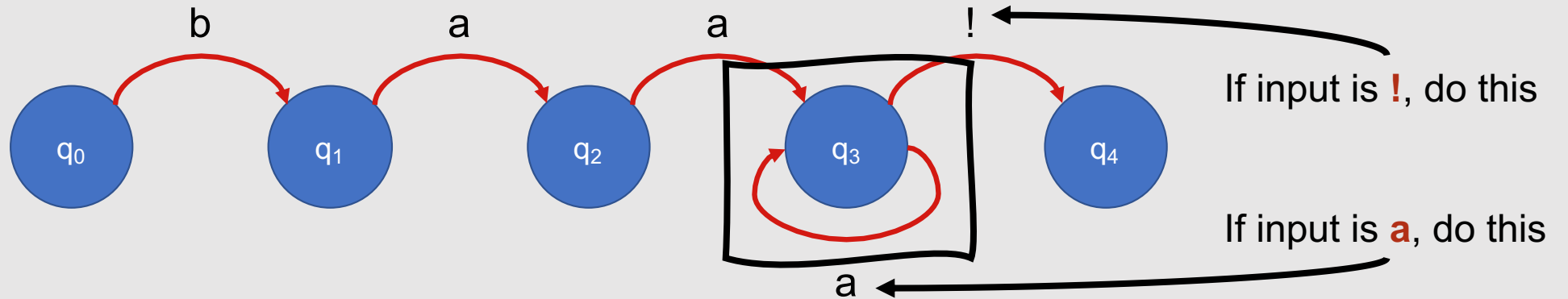
**Non-Deterministic FSA:** At one or more points in processing a sequence, there are multiple permissible next steps (choices!)

# Deterministic or Non-Deterministic?

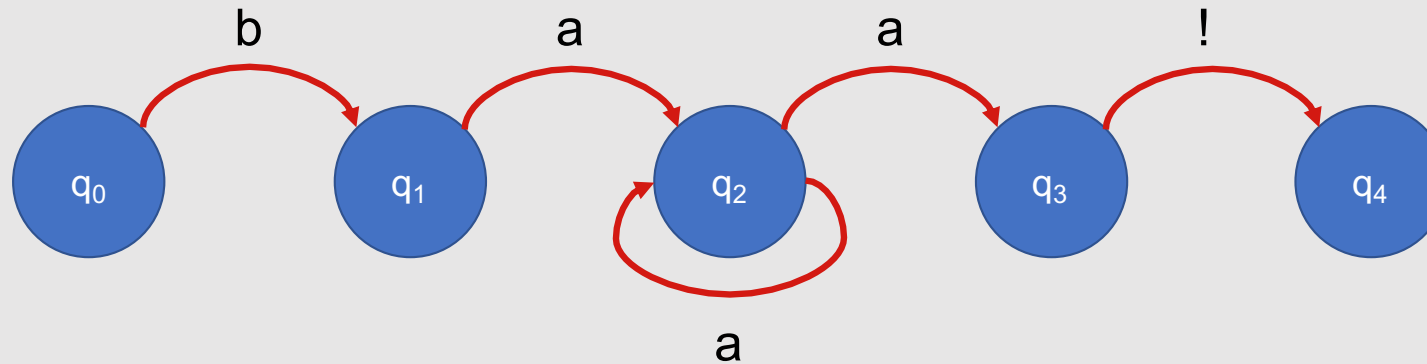


# Deterministic or Non-Deterministic?

Deterministic!

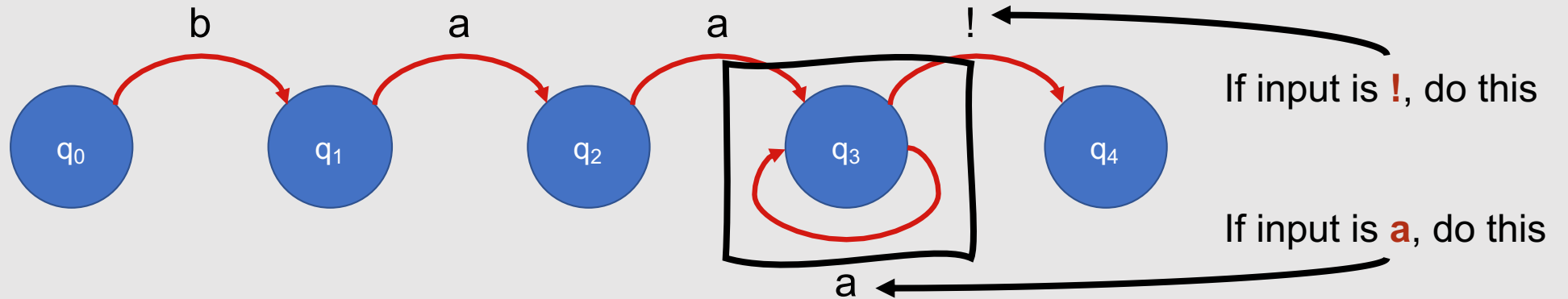


?

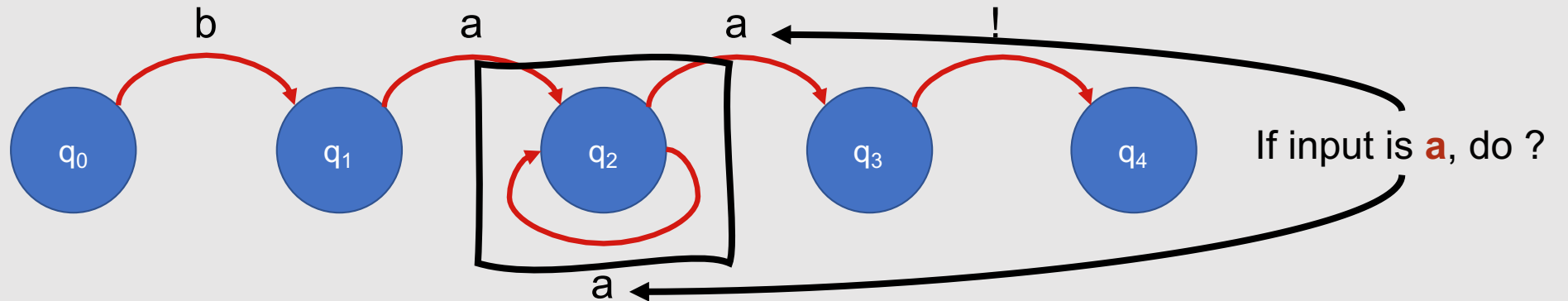


# Deterministic or Non-Deterministic?

Deterministic!



Non-Deterministic!



# Every non-deterministic FSA can be converted to a deterministic FSA.

---

- This means that both are equally powerful!
- Deterministic FSAs can accept as many languages as non-deterministic ones



## Non-Deterministic FSAs: How to check for input acceptance?

- Two approaches:
  1. Convert the non-deterministic FSA to a deterministic FSA and then check that version
  2. Manage the process as a state-space search

# Non-Deterministic FSA Search Assumptions

---

There exists at least one path through the FSA for an item that is part of the language defined by the machine

---

Not all paths directed through the FSA for an accept item lead to an accept state

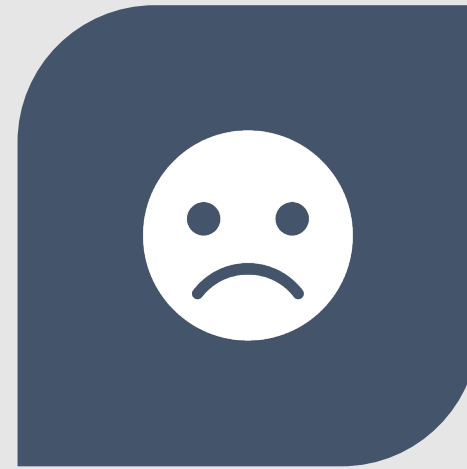
---

No paths through the FSA lead to an accept state for an item not in the language

# Non-Deterministic FSA Search Assumptions

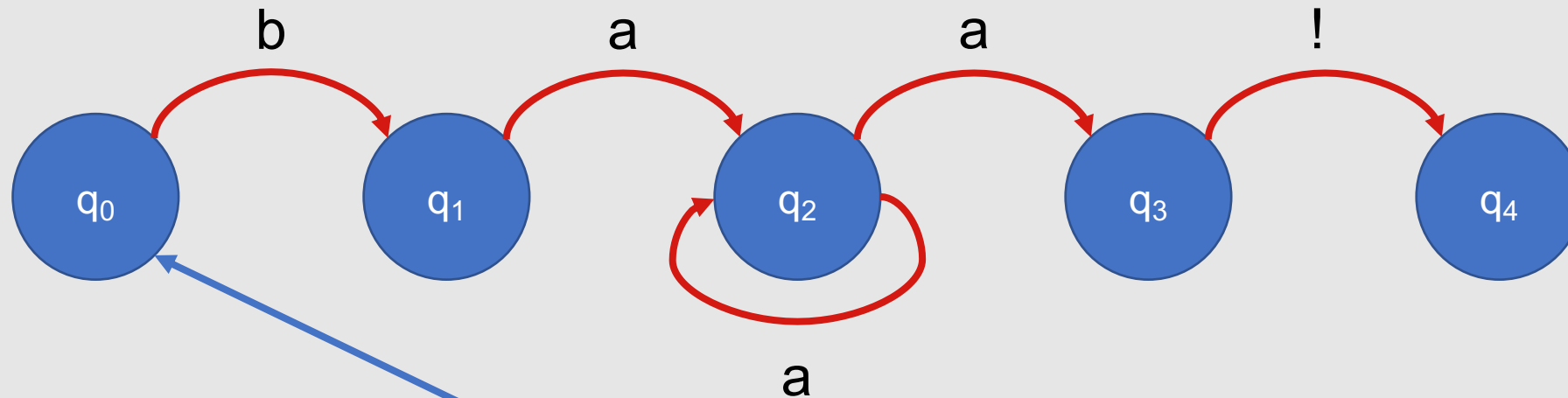


SUCCESS = PATH IS FOUND  
FOR A GIVEN ITEM THAT ENDS  
IN AN ACCEPT



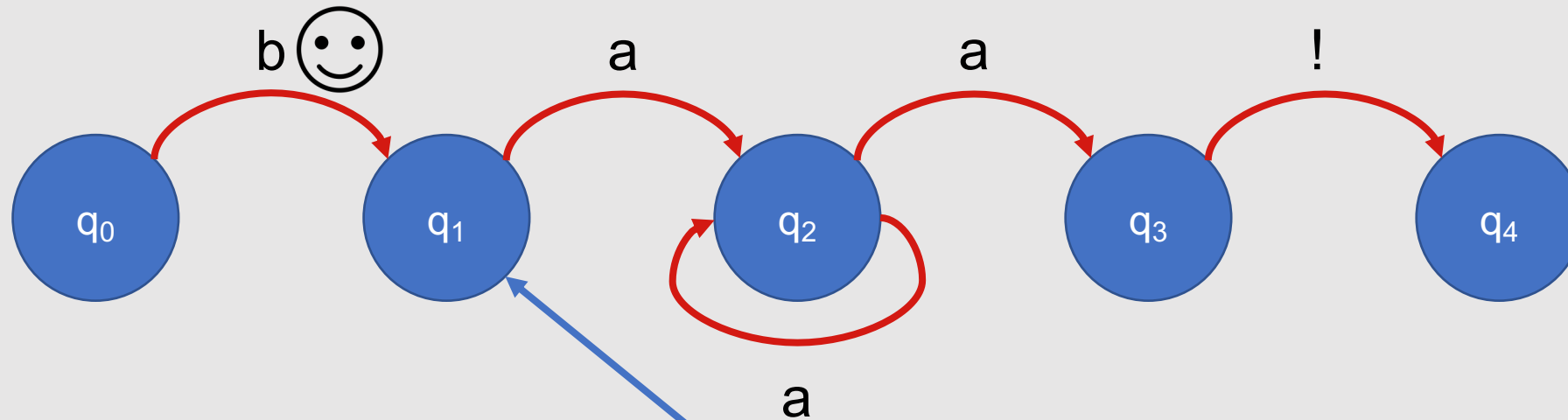
FAILURE = ALL POSSIBLE PATHS  
FOR A GIVEN ITEM LEAD TO  
FAILURE

# Example: Non-Deterministic FSA Search



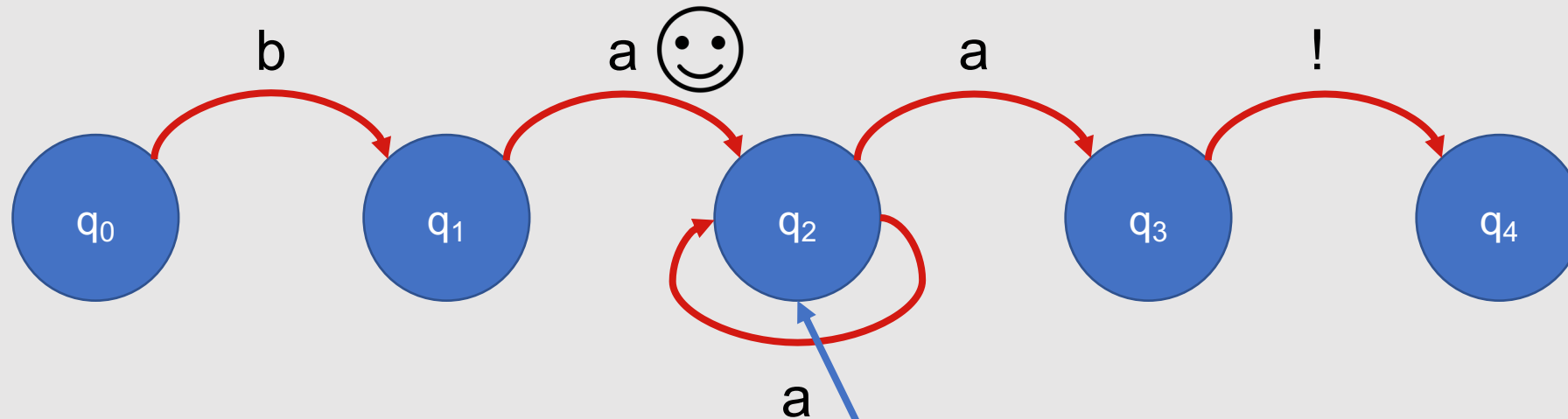
Test Input: baaa!

# Example: Non-Deterministic FSA Search



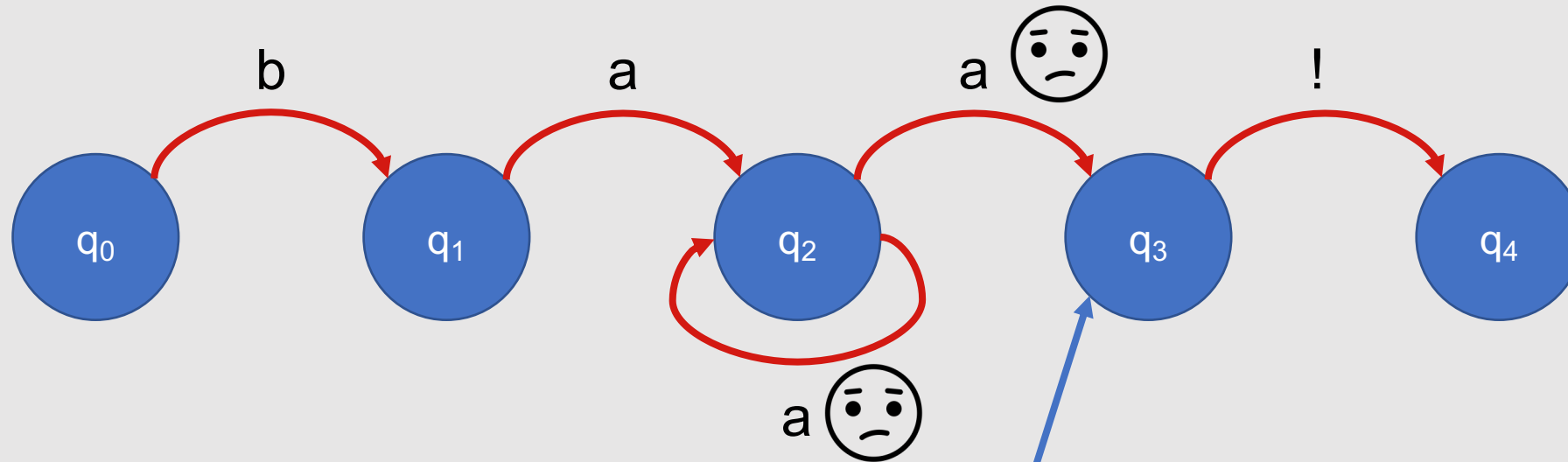
Test Input: **b**aaa!

# Example: Non-Deterministic FSA Search



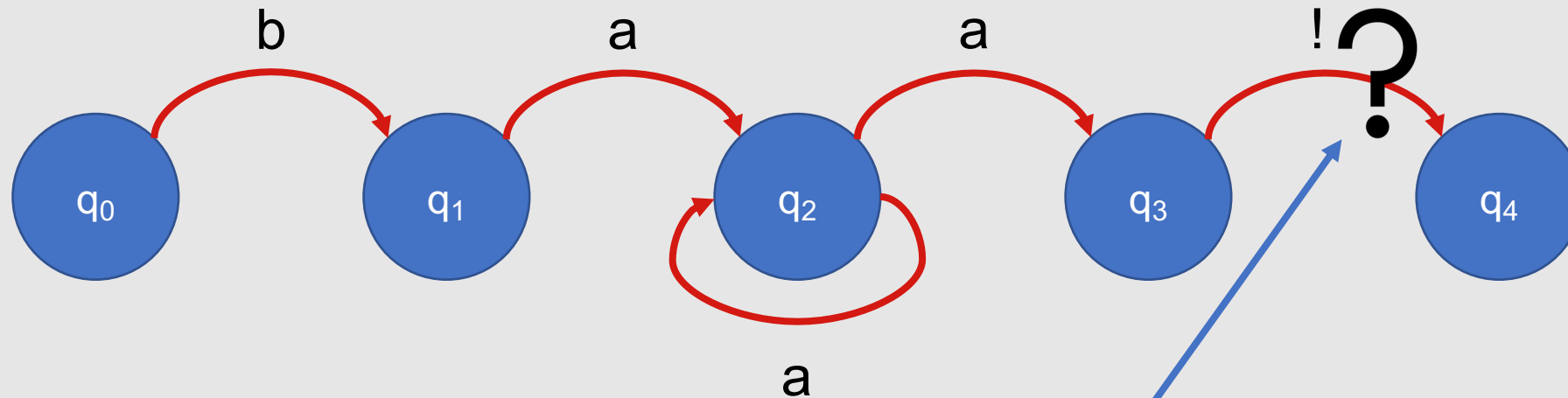
Test Input: baaa!

# Example: Non-Deterministic FSA Search



Test Input: ba aa!

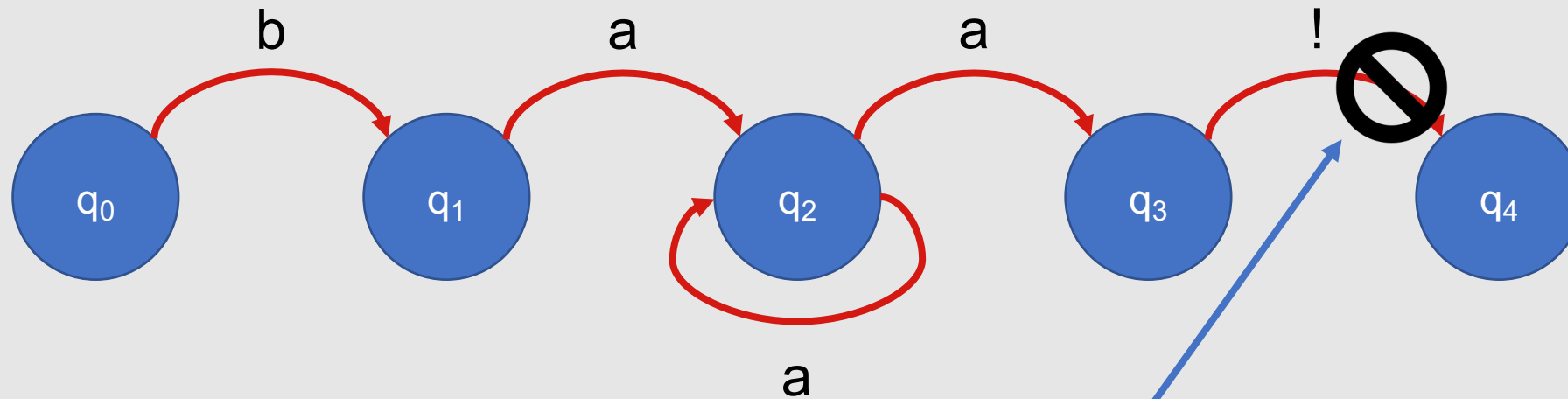
# Example: Non-Deterministic FSA Search



Test Input: baaba!

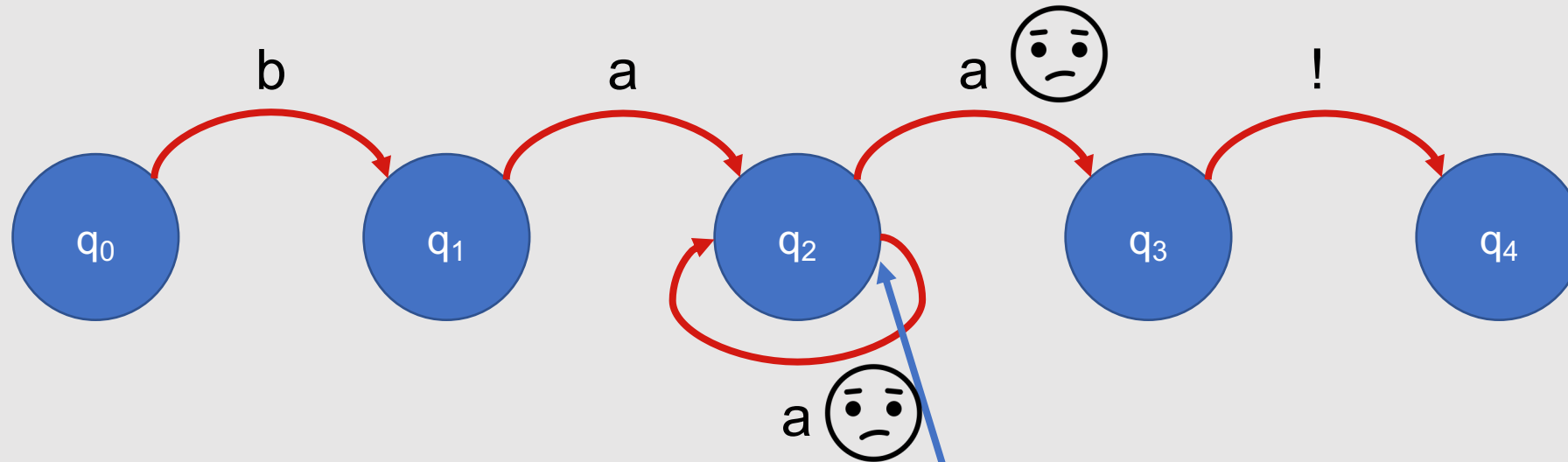


# Example: Non-Deterministic FSA Search



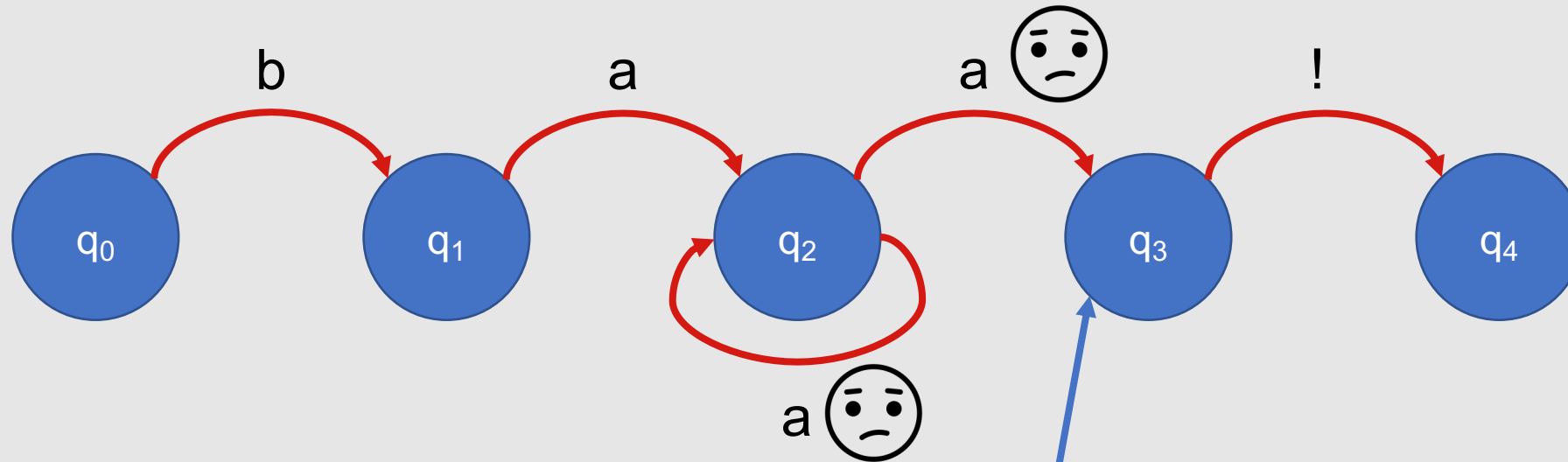
Test Input: ba**a**! ☹️

# Example: Non-Deterministic FSA Search



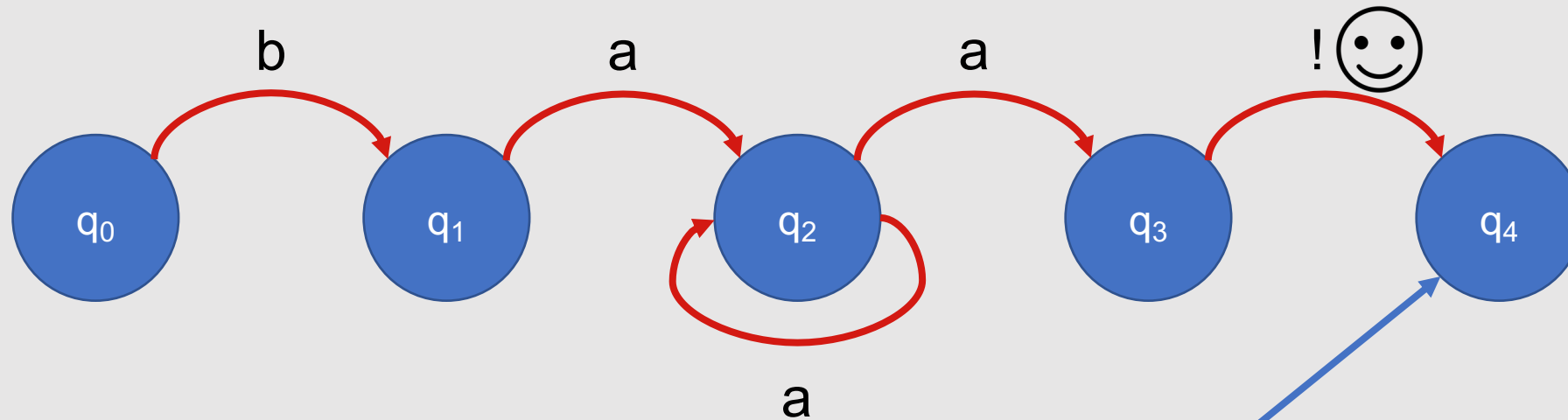
Test Input: ba aa!

# Example: Non-Deterministic FSA Search



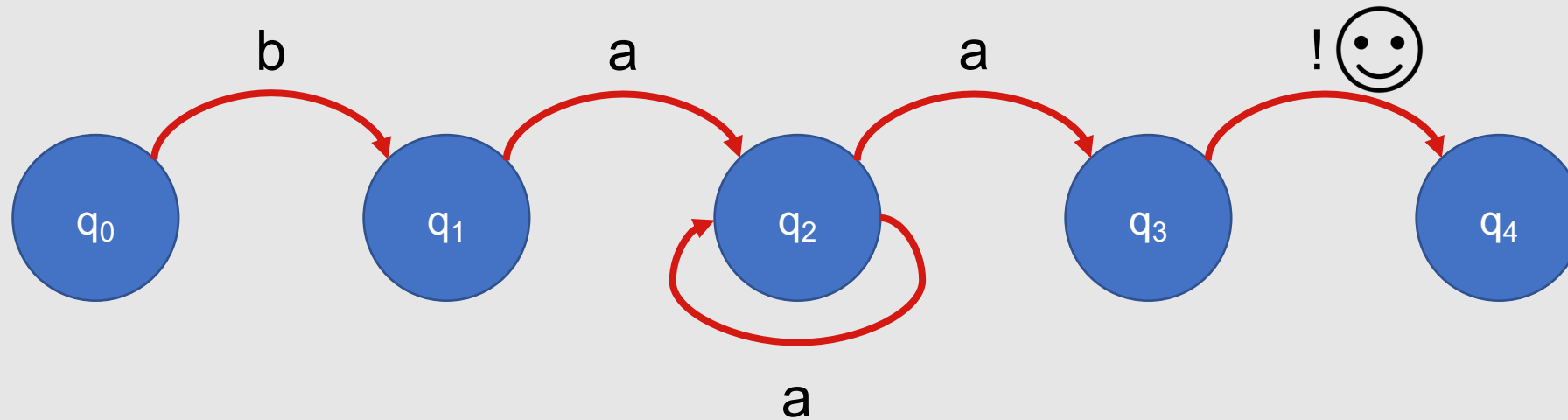
Test Input: baaba!

# Example: Non-Deterministic FSA Search



Test Input: baaa!

# Example: Non-Deterministic FSA Search



Test Input: **baaa!** 😊

# Non-Deterministic FSA Search

- States in the search space are pairings of sequence indices and states in the FSA
- By keeping track of which states have and have not been explored, we can systematically explore all the paths through an FSA given an input

# Compositional FSAs

- You can apply set operations to any FSA
  - Union
  - Concatenation
  - Negation
    - For non-deterministic FSAs, first convert to a deterministic FSA
  - Intersection
- To do so, you may need to utilize an  $\epsilon$  transition
  - $\epsilon$  transition: Move from one state to another without consuming an item from the input sequence

# Summary: Finite State Automata

- FSAs are computational models that describe regular languages
- To determine whether an input item is a member of an FSA's language, you can process it sequentially from the start to (hopefully) the final state
- State transitions in FSAs can be represented using tables
- FSAs can be either deterministic or non-deterministic



# What are finite state transducers?

---

**Finite State Transducer (FST):** A type of FSA that describes mappings between two sets of items

---

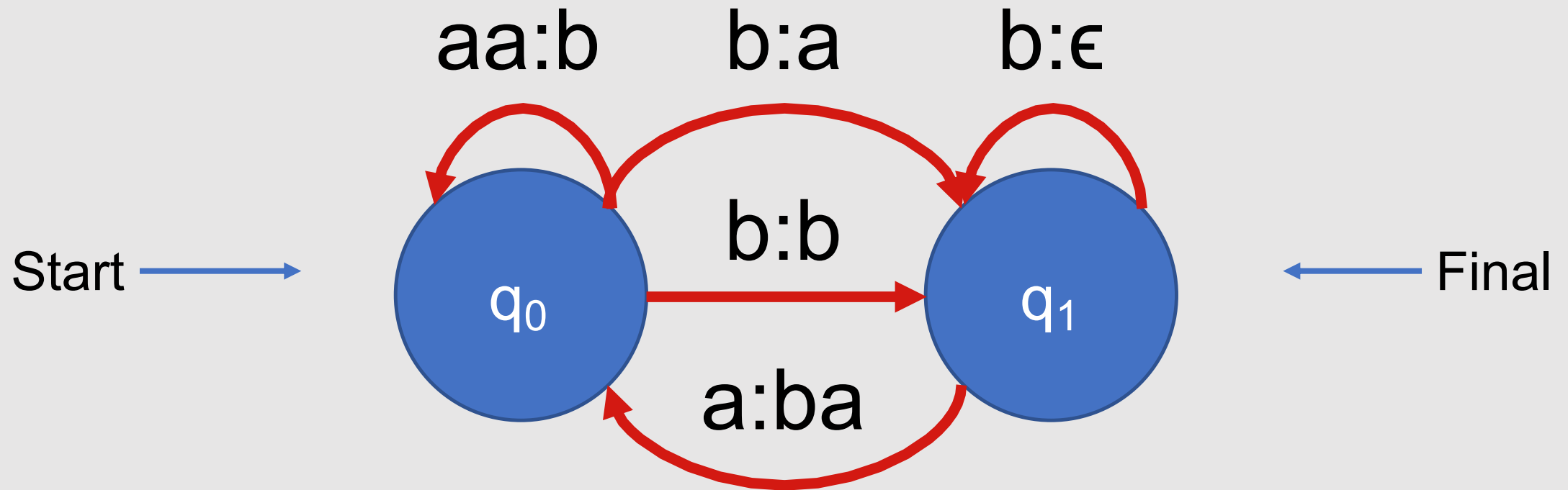
This means that FSTs recognize or generate pairs of items

---

FSAs can be converted to FSTs by labeling each arc with two items (e.g., **a:b** for an input of **a** and an output of **b**)

---

# Example: Simple FST



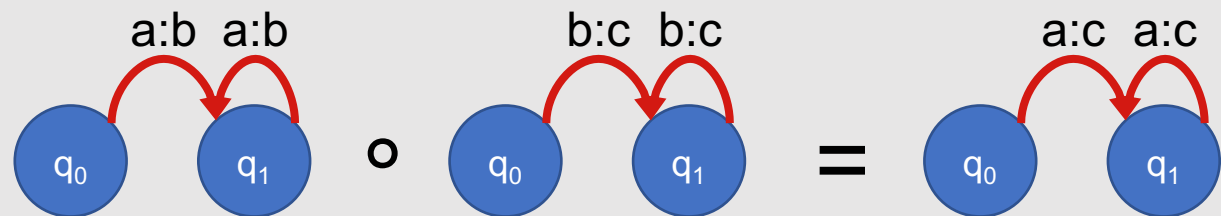
# Formal Definition

- A finite state transducer can be specified by enumerating the following properties:
  - The set of states,  $Q$
  - A finite input alphabet,  $\Sigma$
  - A finite output alphabet,  $\Delta$
  - A start state,  $q_0$
  - A set of accept/final states,  $F \subseteq Q$
  - A transition function or transition matrix between states,  $\delta(q,i)$
  - An output function giving the set of possible outputs for each state and input,  $\sigma(q,i)$
- $\delta(q,i)$ : Given a state  $q \in Q$  and input  $i \in \Sigma$ ,  $\delta(q,i)$  returns a new state  $q' \in Q$ .

# Formal Properties

**Composition:** Letting  $T_1$  be an FST from  $I_1$  to  $O_1$  and letting  $T_2$  be an FST from  $I_2$  to  $O_2$ , the two FSTs can be composed such that the resulting FST maps directly from  $I_1$  to  $O_2$ .

**Inversion:** Letting  $T$  be an FST that maps from  $I$  to  $O$ , its inversion ( $T^{-1}$ ) will map from  $O$  to  $I$ .



# Deterministic vs. Non- Deterministic FSTs

---

Just like FSAs, **FSTs can be non-deterministic** ...one input can be translated to many possible outputs!

---

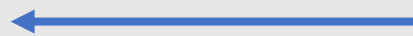
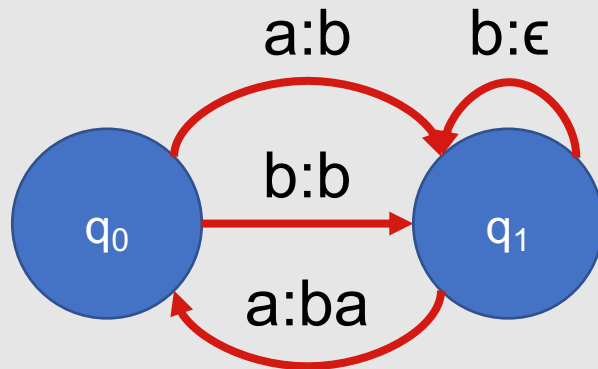
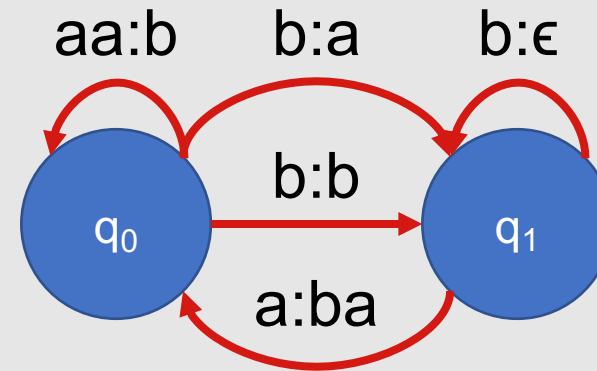
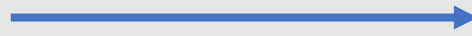
Unlike FSAs, **not all non-deterministic FSTs can be converted to deterministic FSTs**

---

FSTs with underlying deterministic FSAs (at any state, a given input maps to at most one transition out of the state) are called **sequential transducers**

# Examples: Non-Deterministic and Sequential Transducers

Non-Deterministic



Sequential

# Remember morphology?

- **Morphemes:**
  - Small meaningful units that make up words
  - **Stems:** The core meaning-bearing units
  - **Affixes:** Bits and pieces that adhere to stems and add additional information
    - -ed
    - -ing
    - -s
- Morphological parsing is a classic use case for FSTs

## Morphological Parsing

- The task of recognizing the component morphemes of words (e.g., foxes → fox + es) and building structured representations of those components



# Why is morphological parsing necessary?

## Morphemes can be **productive**

- Example: -ing attaches to almost every verb, including brand new words
  - “Why are you Instagramming that?”

## Some languages are very **morphologically complex**

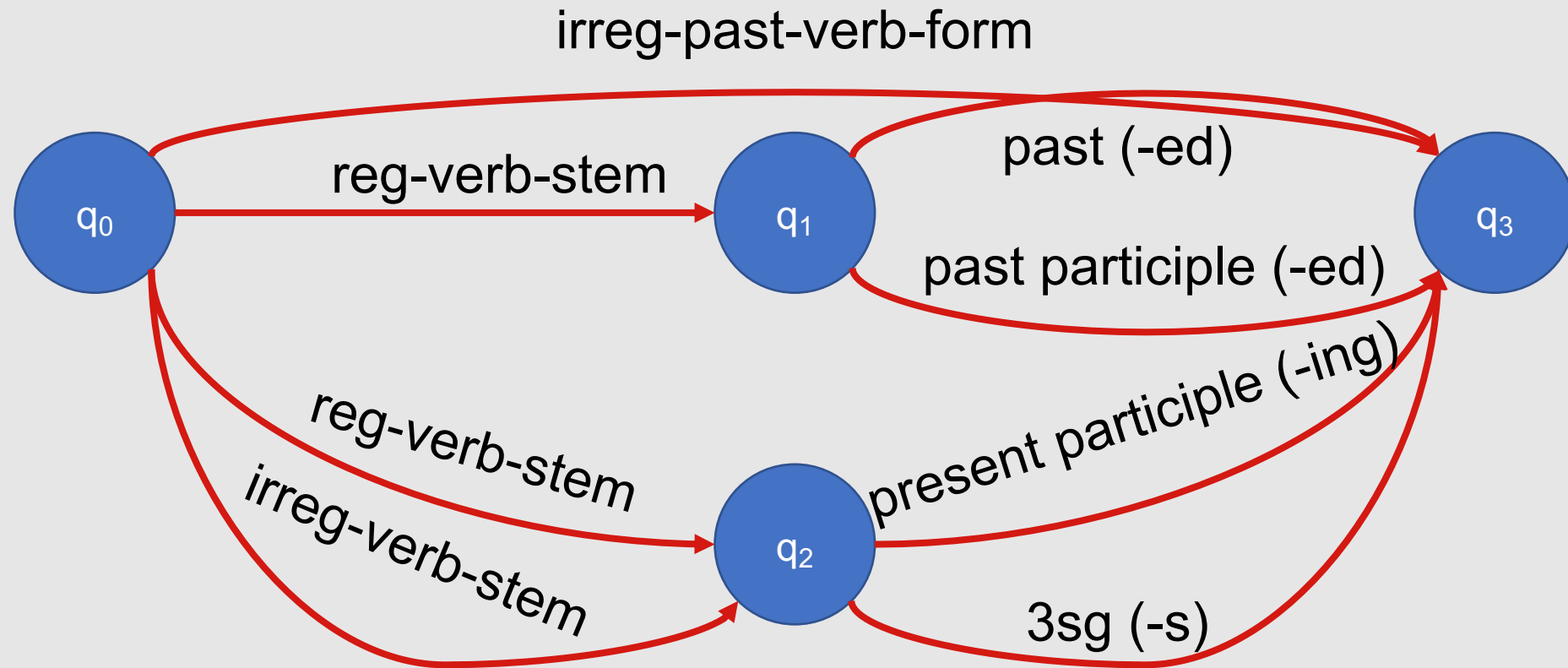
- Uygarlastiramadiklarimizdanmissinizcasina
  - Uygar ‘civilized’ + las ‘become’
  - + tir ‘cause’ + ama ‘not able’
  - + dik ‘past’ + lar ‘plural’
  - + imiz ‘p1pl’ + dan ‘abl’
  - + mis ‘past’ + siniz ‘2pl’ + casina ‘as if’

# Finite State Morphological Parsing

Goal: Take input surface realizations and produce morphological parses as output

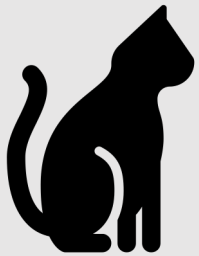
Surface Text	Morphological Parse
cats	cat +N +PL
cat	cat +N +SG
cities	city +N +PL
geese	goose +N +PL
goose	goose +N +SG
merging	merge +V +PresPart
caught	catch +V +Past

# Example Morphological Lexicon

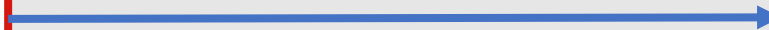


# Finite State Morphological Parsing

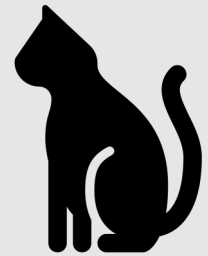
- Two sets of items:
  - Surface form (input text)
  - Lexical form (morphological parse)



cats

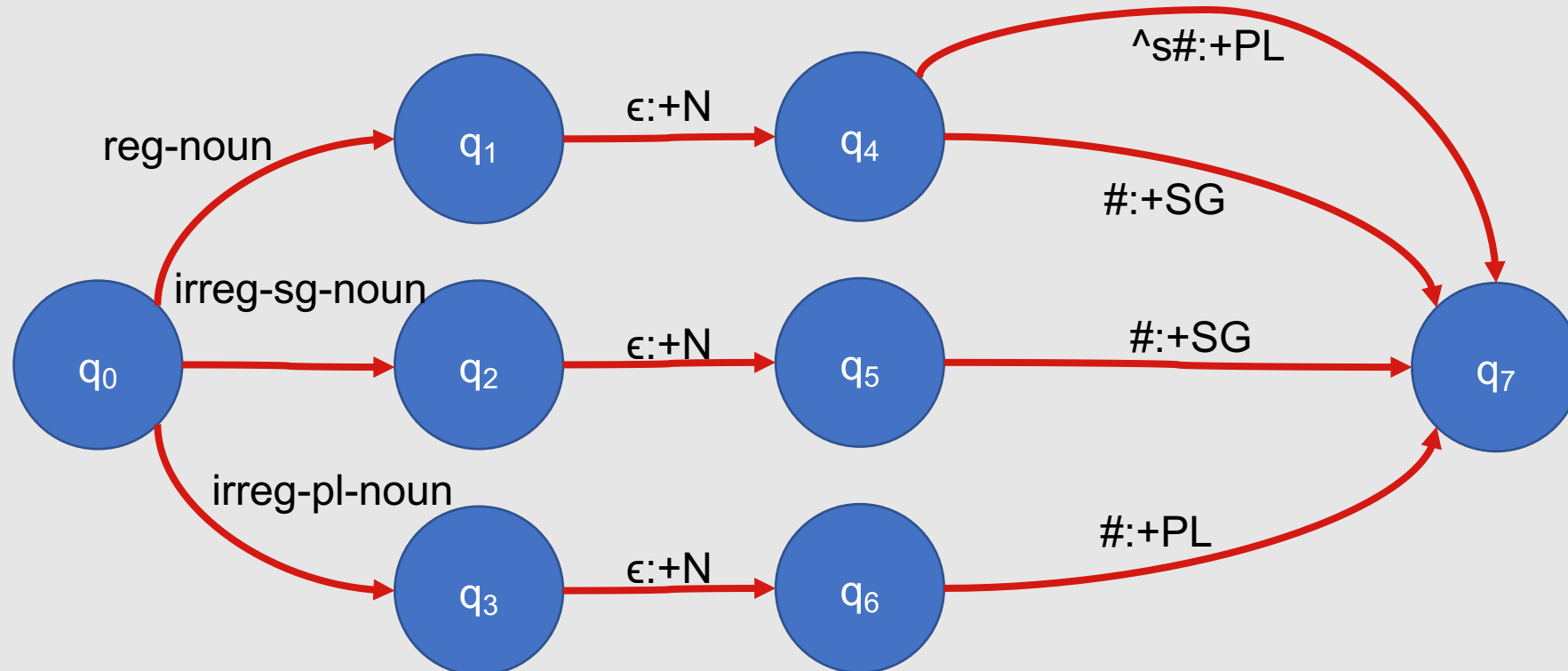


cat +N +PL



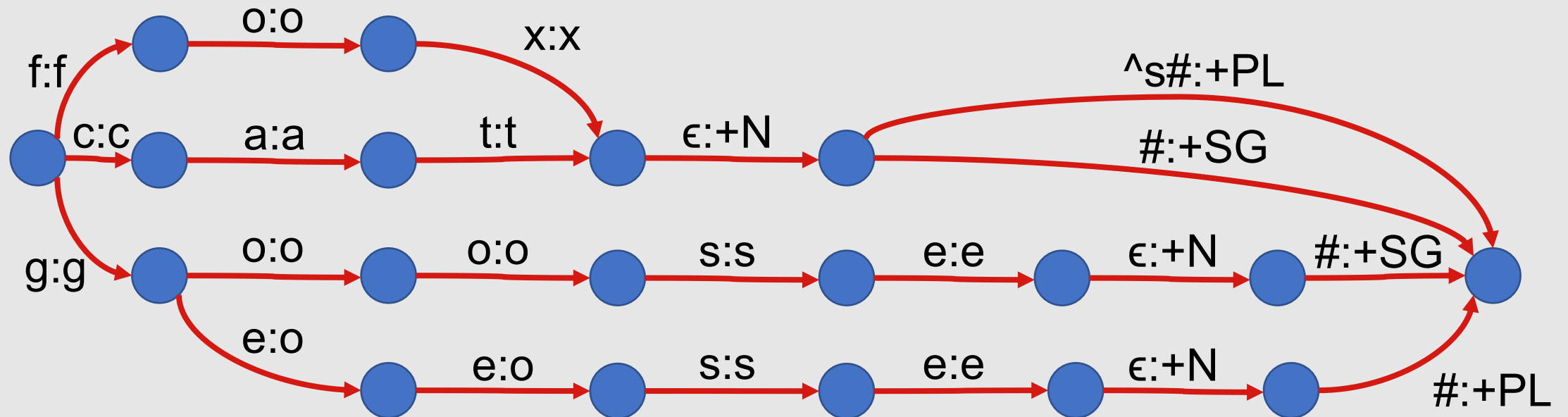
# Finite State Morphological Parsing

reg-noun	irreg-pl-noun	irreg-sg-noun
fox	g o:e o:e s e	goose
cat		



# Finite State Morphological Parsing

reg-noun	irreg-pl-noun	irreg-sg-noun
fox	g o:e o:e s e	goose
cat		



# Summary: Finite State Transducers

- FSTs are FSAs that describe mappings between two sets
- Although all non-deterministic FSAs can be converted to deterministic versions, all non-deterministic FSTs cannot
- FSTs with underlying deterministic FSAs are called sequential transducers
- FSTs are particularly useful for morphological parsing

# What are Hidden Markov Models (HMMs)?

---

**Probabilistic generative models** for  
**sequences**

---

Make predictions based on an  
underlying set of **hidden states**



How does sequence labeling differ from other types of classification?

- A lot of machine learning addresses the problem of classifying instances into a predefined number of classes
  - Decision Trees
  - Naïve Bayes
  - Logistic Regression
  - (Some) Neural Networks
  - Support Vector Machines

Spam



Not Spam

Dear Esteemed Professor Dr. **Natalie Parde**,  
I am interested in applying to **University of Illinois – Chicago** for a **Ph.D.** in **Computer Science** in the area of **Artificial Intelligence** and **Natural Language Processing**. I read your recent paper **“Enriching Neural Models with Targeted Features for Dementia Detection”** and see that you are interested in **Neural Models** and **Dementia Detection**....

Standard  
Classification  
Assumption:  
Individual  
cases are  
disconnected  
and  
independent.

---

However, many NLP problems do not satisfy this assumption.

---

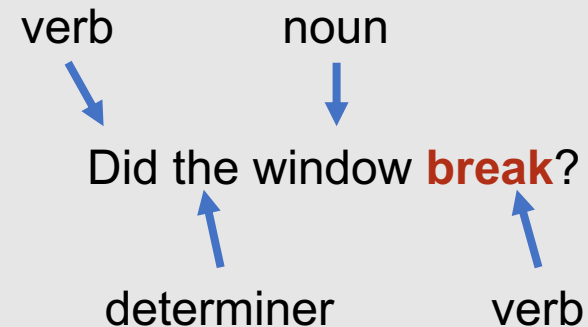
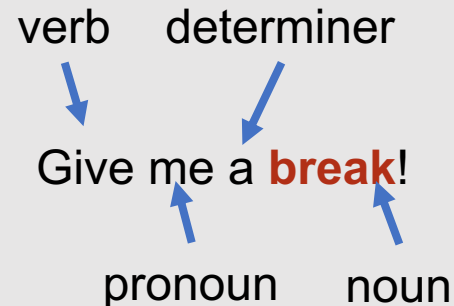
Instead, they involve many interconnected decisions, each of which resolve different ambiguities despite being mutually dependent.

---

For these problems, different learning and inference techniques are needed!

# Sequence Labeling

- Many NLP problems can be viewed as **sequence labeling** tasks.
- Objective: Find the label for the next item, based on the labels of other items in the sequence.



# Applications that can benefit from sequence labeling?

- Named entity recognition
- Semantic role labeling
- Genome analysis

person

organization

**Natalie Parde** works at the **University of Illinois at Chicago** and lives in **Chicago, Illinois**.

location

agent

source destination

**Natalie** drove for 15 hours from **Dallas** to **Chicago** in her trusty hail-damaged **Honda Accord**.

instrument

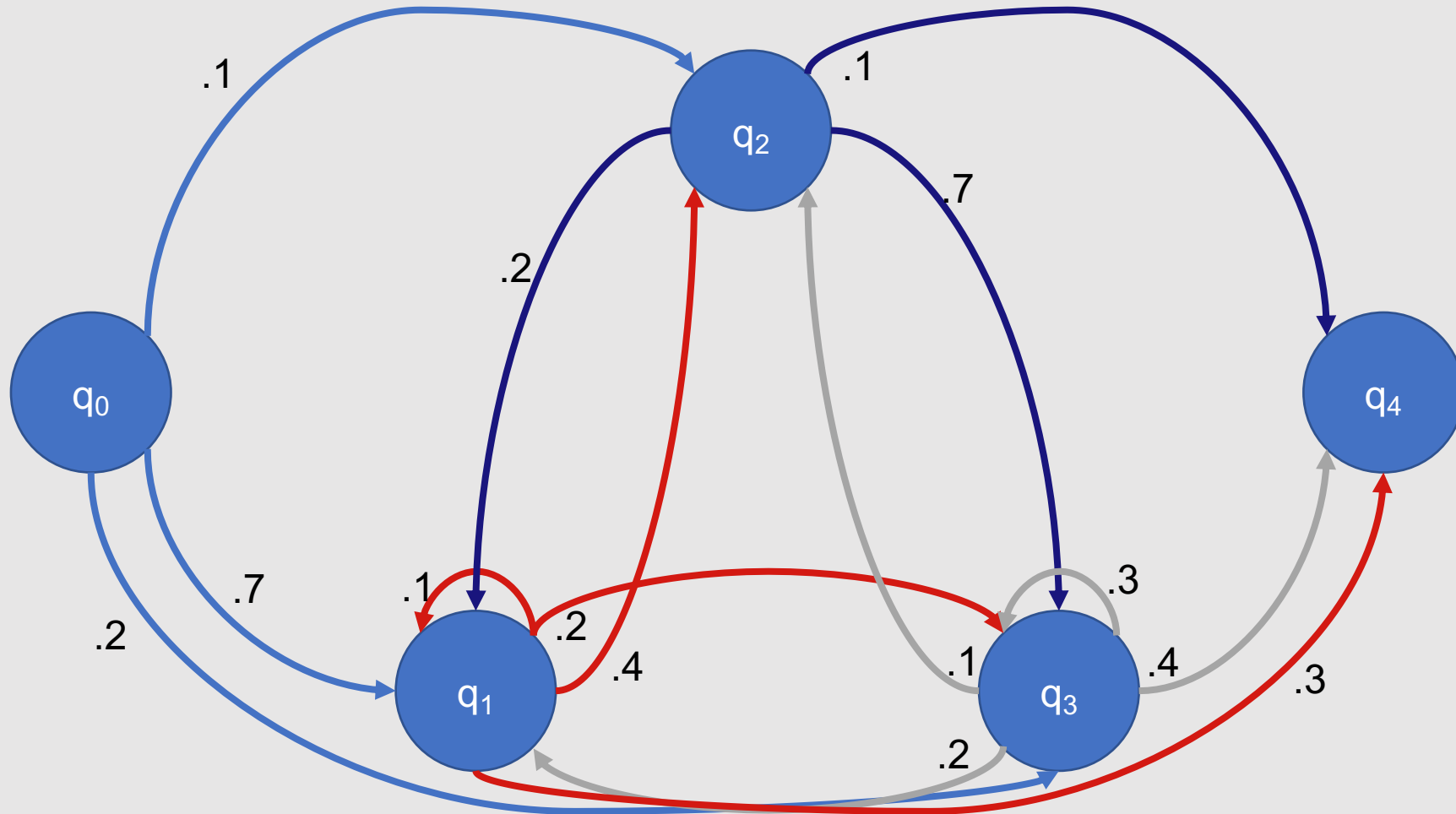
## Probabilistic Sequence Models

- Allow uncertainties to be integrated over multiple, interdependent classifications
- These classifications collectively determine the most likely global assignment
- Two standard models:
  - Hidden Markov Models
  - Conditional Random Fields

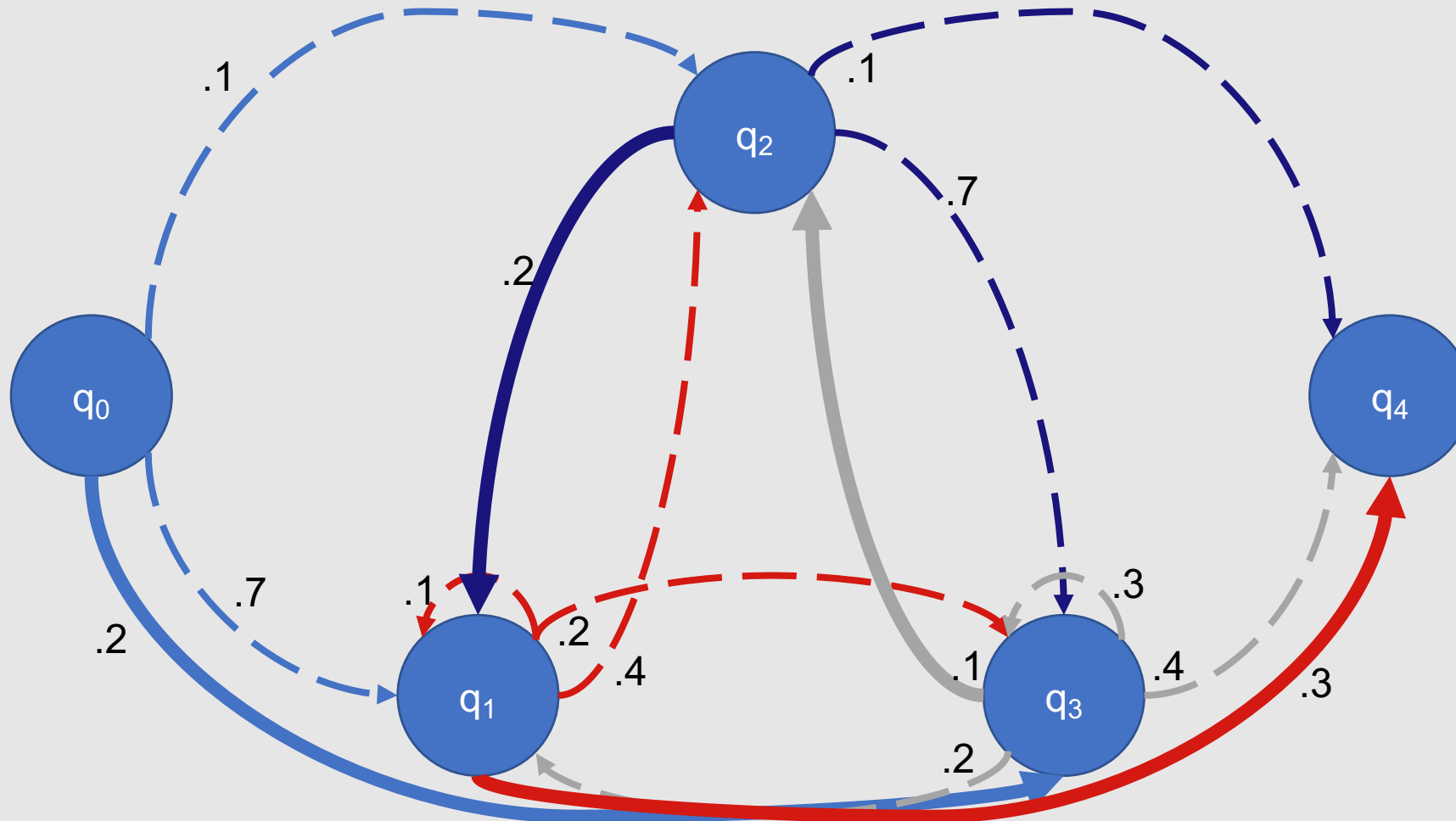
# What are Markov Models?

- **Finite state automata with probabilistic state transitions**
- Markov Property: The future is independent of the past, given the present.
  - In other words, the next state only depends on the current state ...it is independent of previous history.
- Also referred to as **Markov Chains**

# Sample Markov Model



# Sample Markov Model



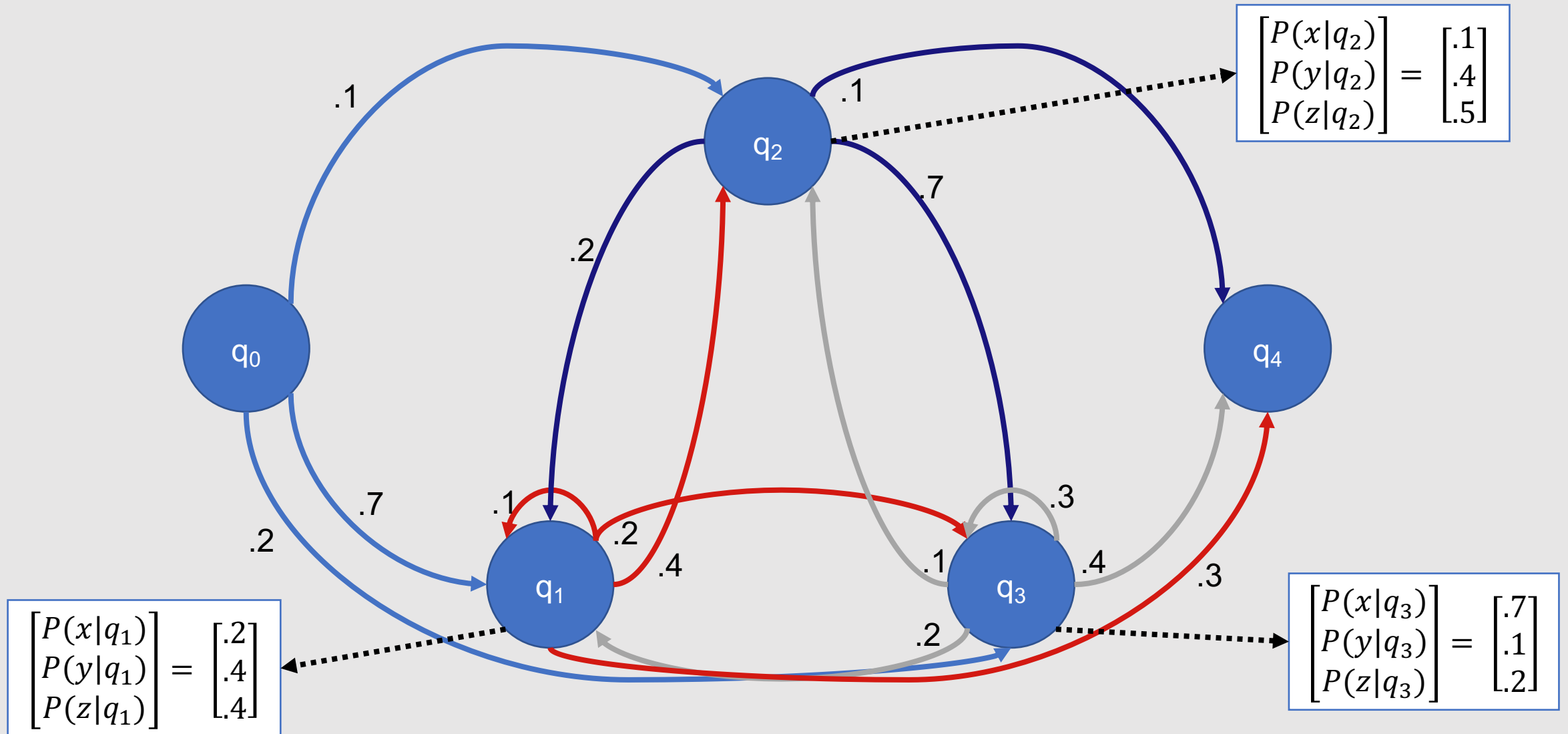
$$\begin{aligned} P(q_3 \ q_2 \ q_1 \ q_4) \\ &= .2 * .1 * .2 * .3 \\ &= .0012 \end{aligned}$$



# Hidden Markov Models

- Probabilistic generative models for sequences
- Assume an underlying set of hidden (unobserved) states in which the model can be
- Assume probabilistic transitions between states over time
- Assume probabilistic generation of items (e.g., tokens) from states

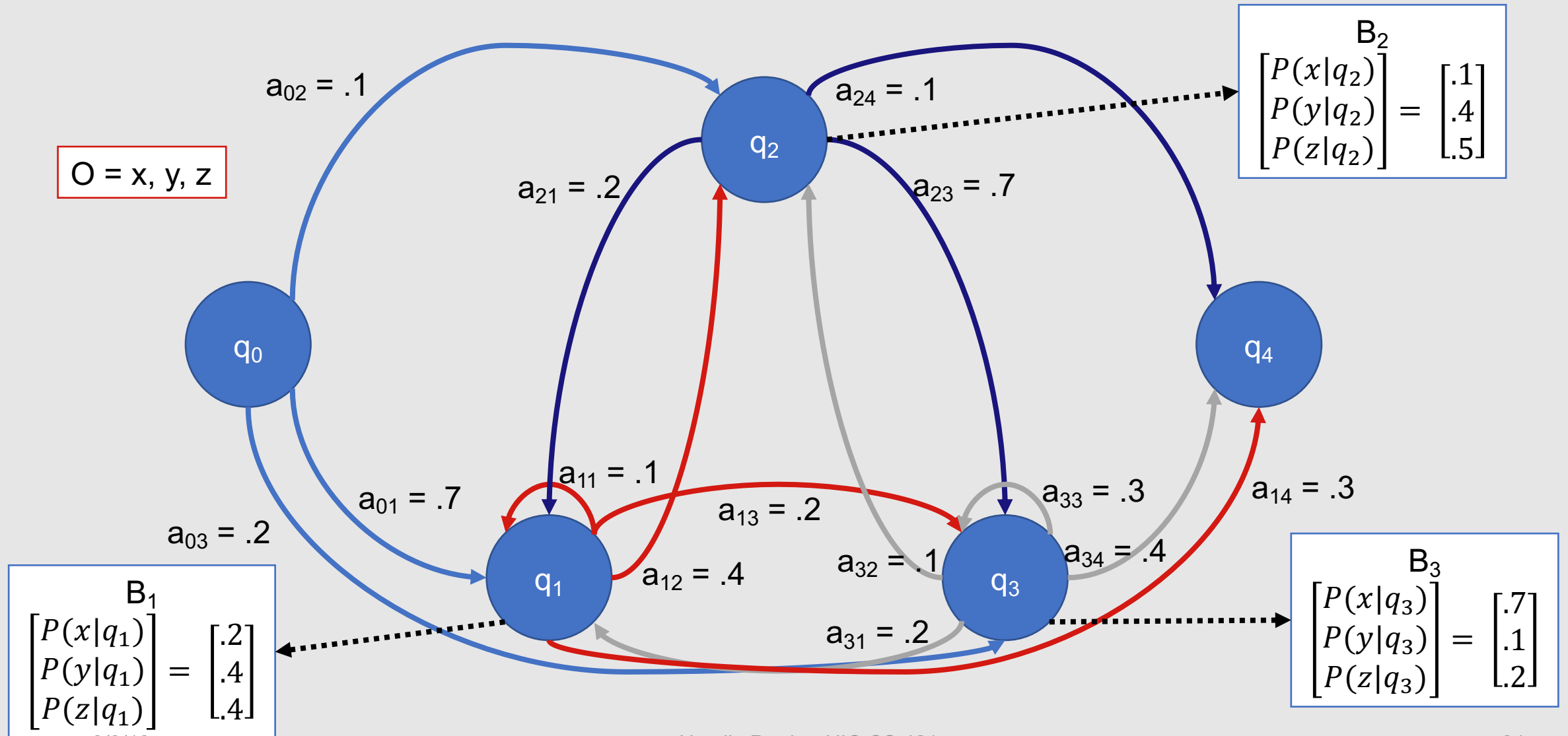
# Sample Hidden Markov Model



# Formal Definition

- A hidden Markov model can be specified by enumerating the following properties:
  - The set of states,  $Q$
  - A transition probability matrix,  $A$ , where each  $a_{ij}$  represents the probability of moving from state  $i$  to state  $j$ , such that  $\sum_{j=1}^n a_{ij} = 1 \forall i$
  - A sequence of  $T$  observations,  $O$ , each drawn from a vocabulary  $V = v_1, v_2, \dots, v_V$
  - A sequence of observation likelihoods,  $B$ , also called emission probabilities, each expressing the probability of an observation  $o_t$  being generated from a state  $i$
  - A start state,  $q_0$ , and final state,  $q_F$ , that are not associated with observations, together with transition probabilities out of  $q_0$  and into  $q_F$

# Sample Hidden Markov Model



# Corresponding Transition Matrix

---

	q0	q1	q2	q3	q4
q0	N/A	.7	.1	.2	N/A
q1	N/A	.1	.4	.2	.3
q2	N/A	.2	N/A	.7	.1
q3	N/A	.2	.1	.3	.4
q4	N/A	N/A	N/A	N/A	N/A

## Can we use HMMs to generate text?

- Sure!
- More generally, you can generate a sequence of  $T$  observations:  $O = o_1, o_2, \dots, o_T$

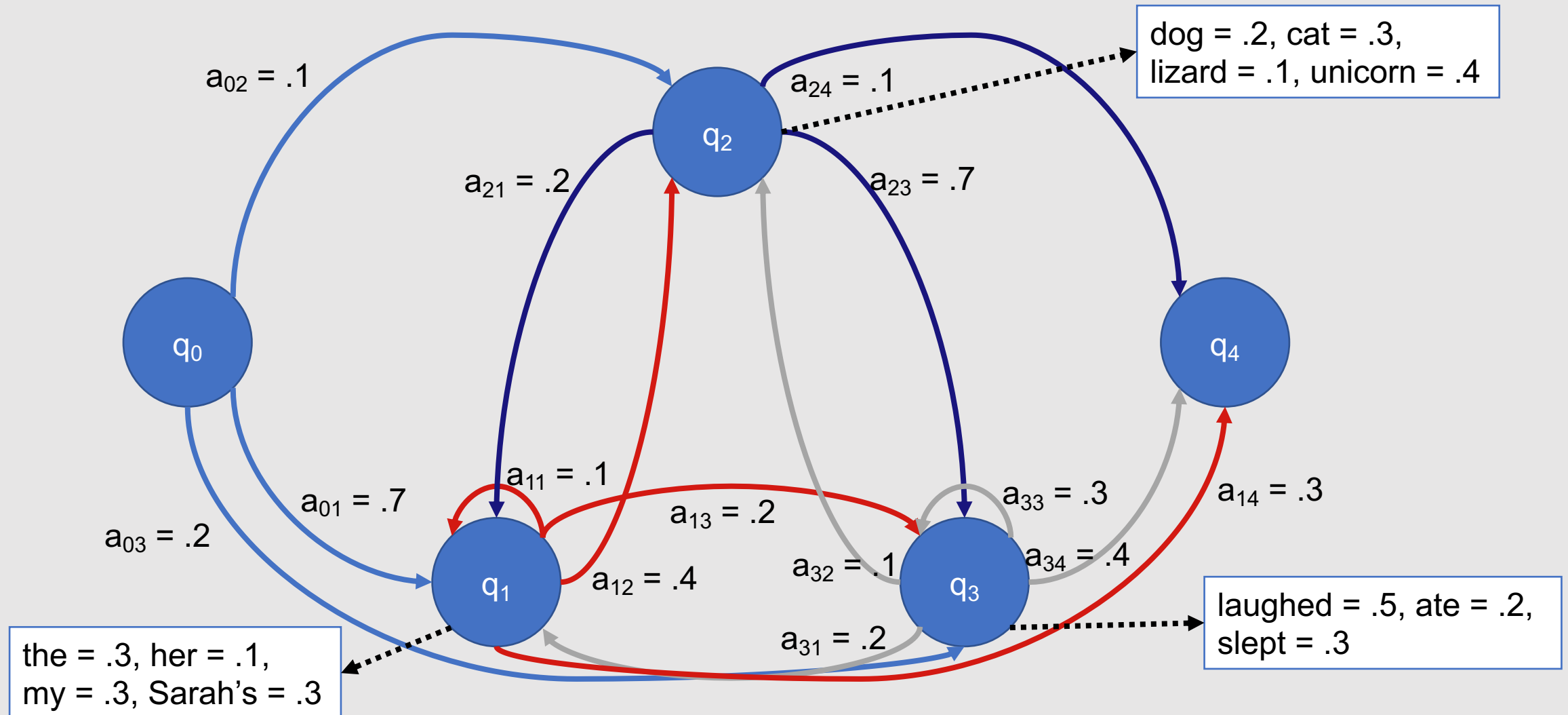
*Begin in the start state*

*For  $t$  in  $[0, \dots, T]$ :*

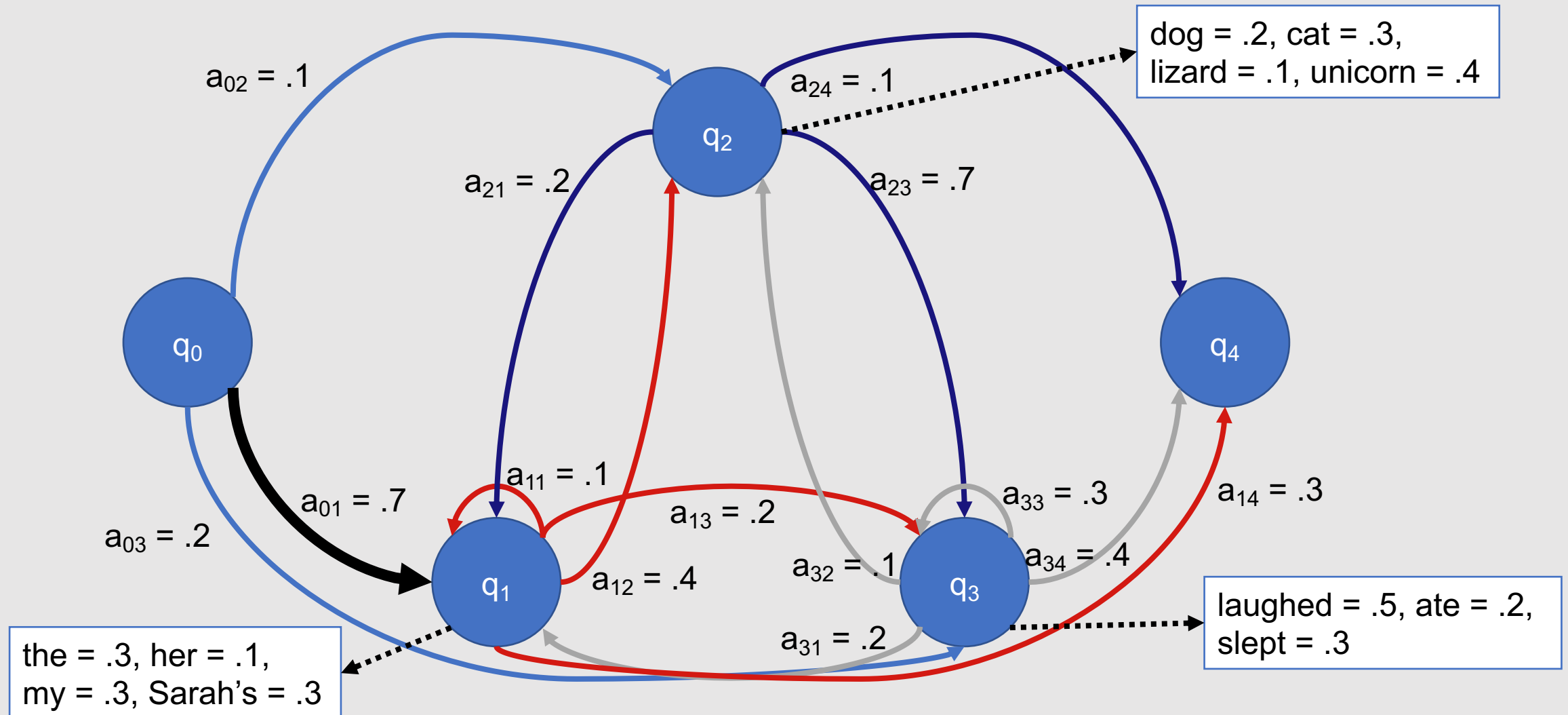
*Randomly select a new state based on the transition distribution for the current state*

*Randomly select an observation from the new state based on the observation distribution for that state*

# Sample Text Generation

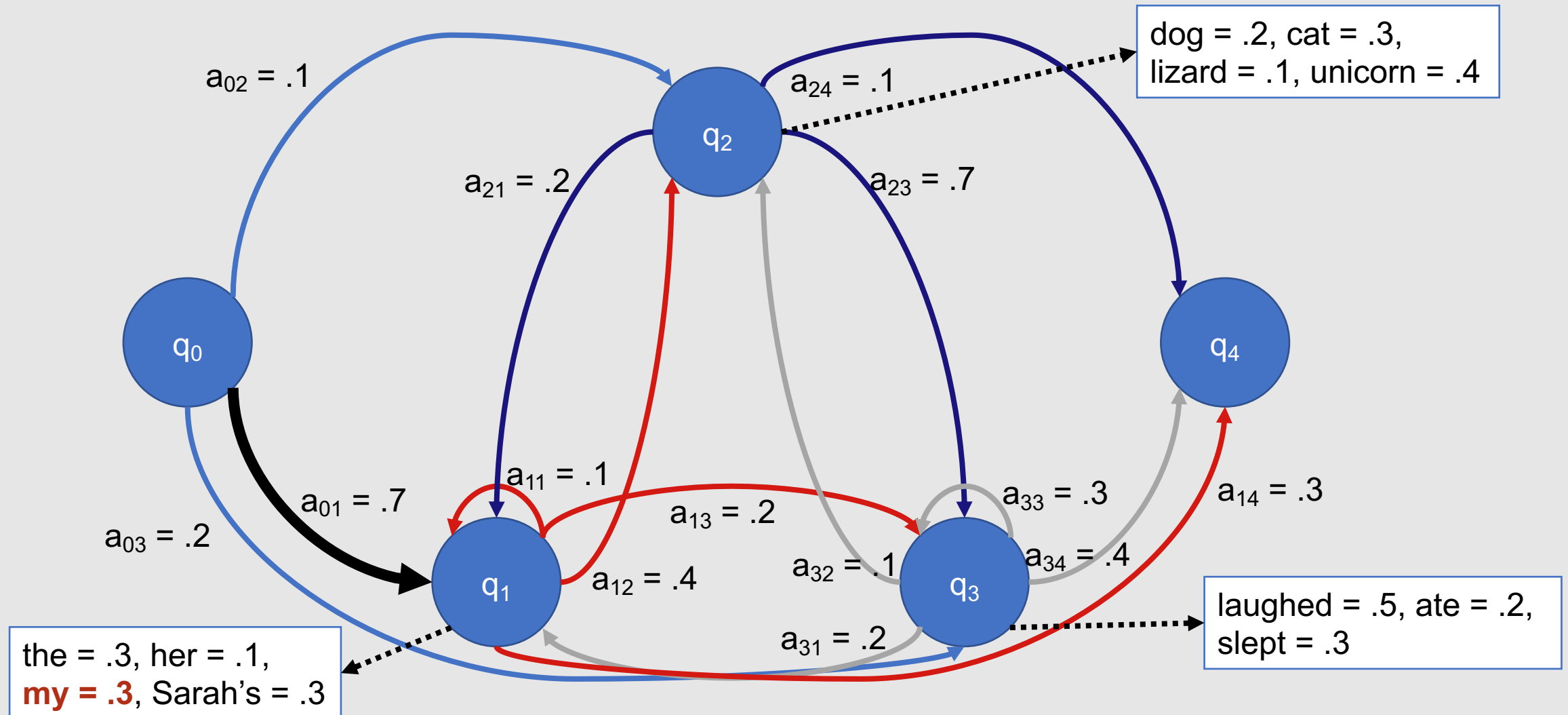


# Sample Text Generation

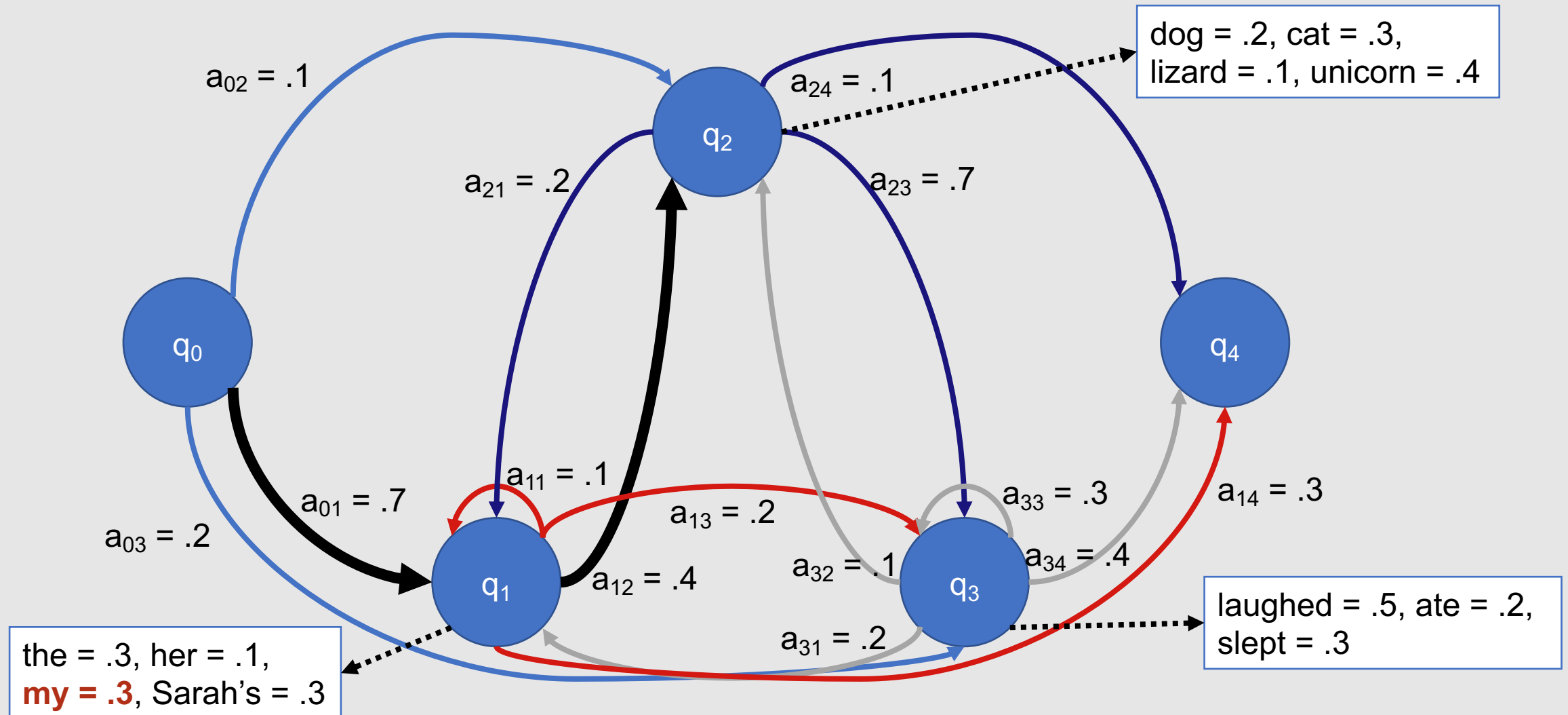




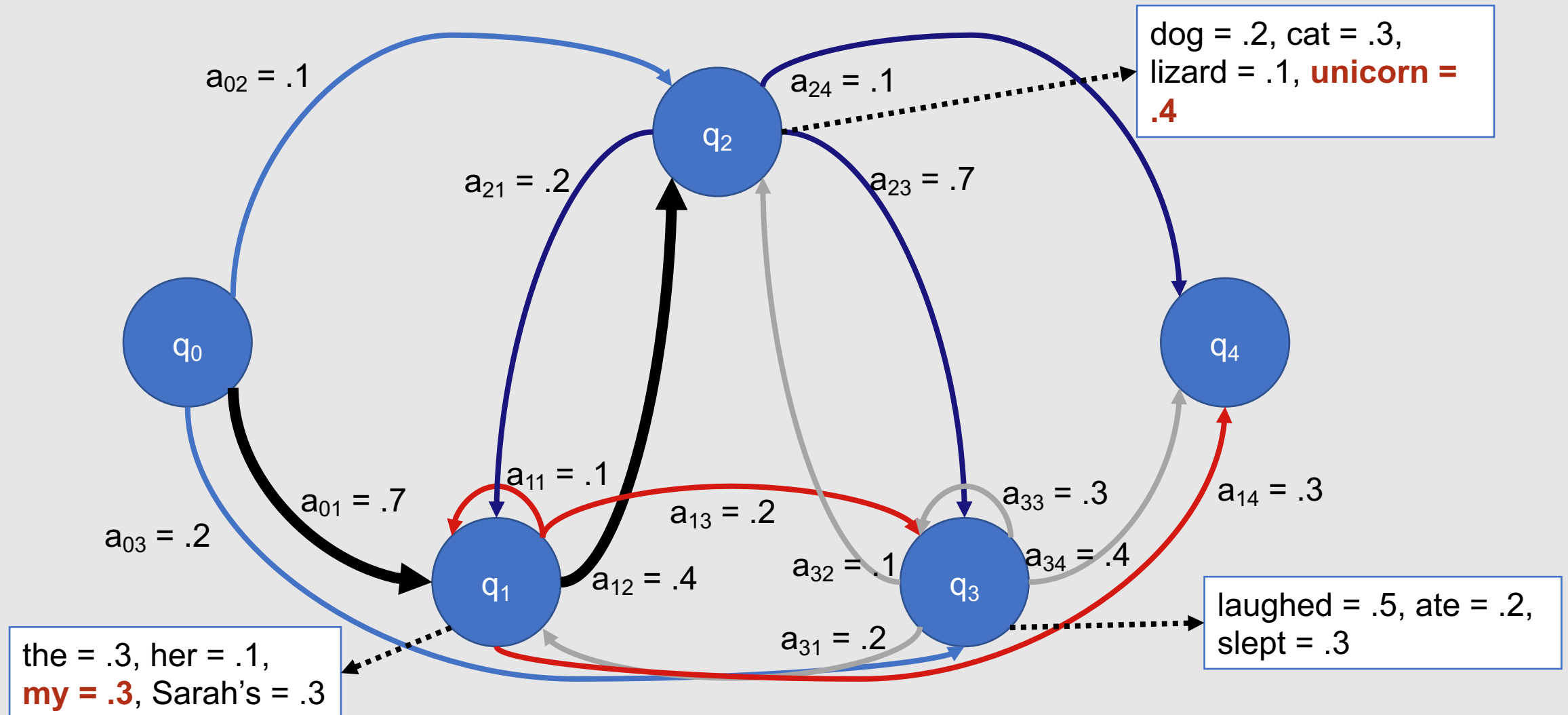
# Sample Text Generation



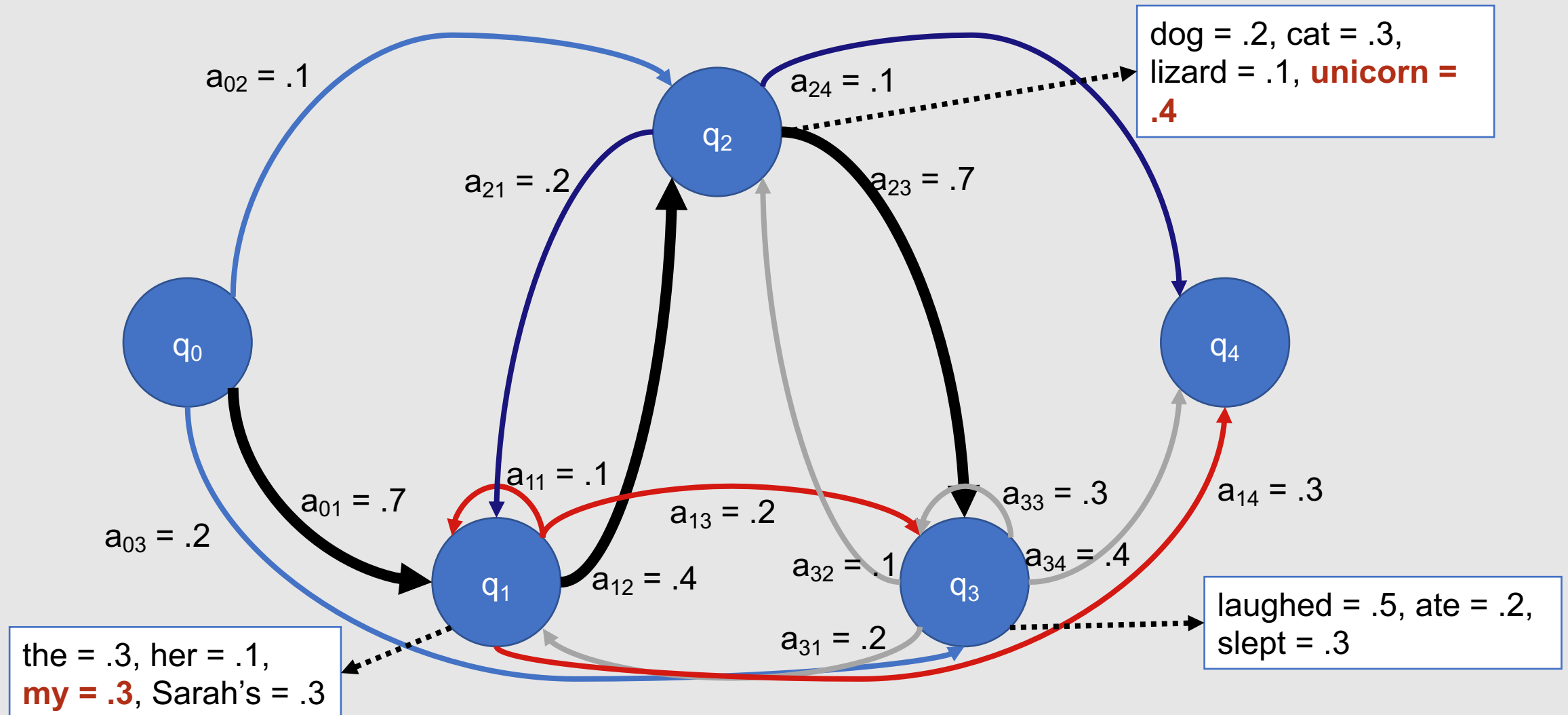
# Sample Text Generation



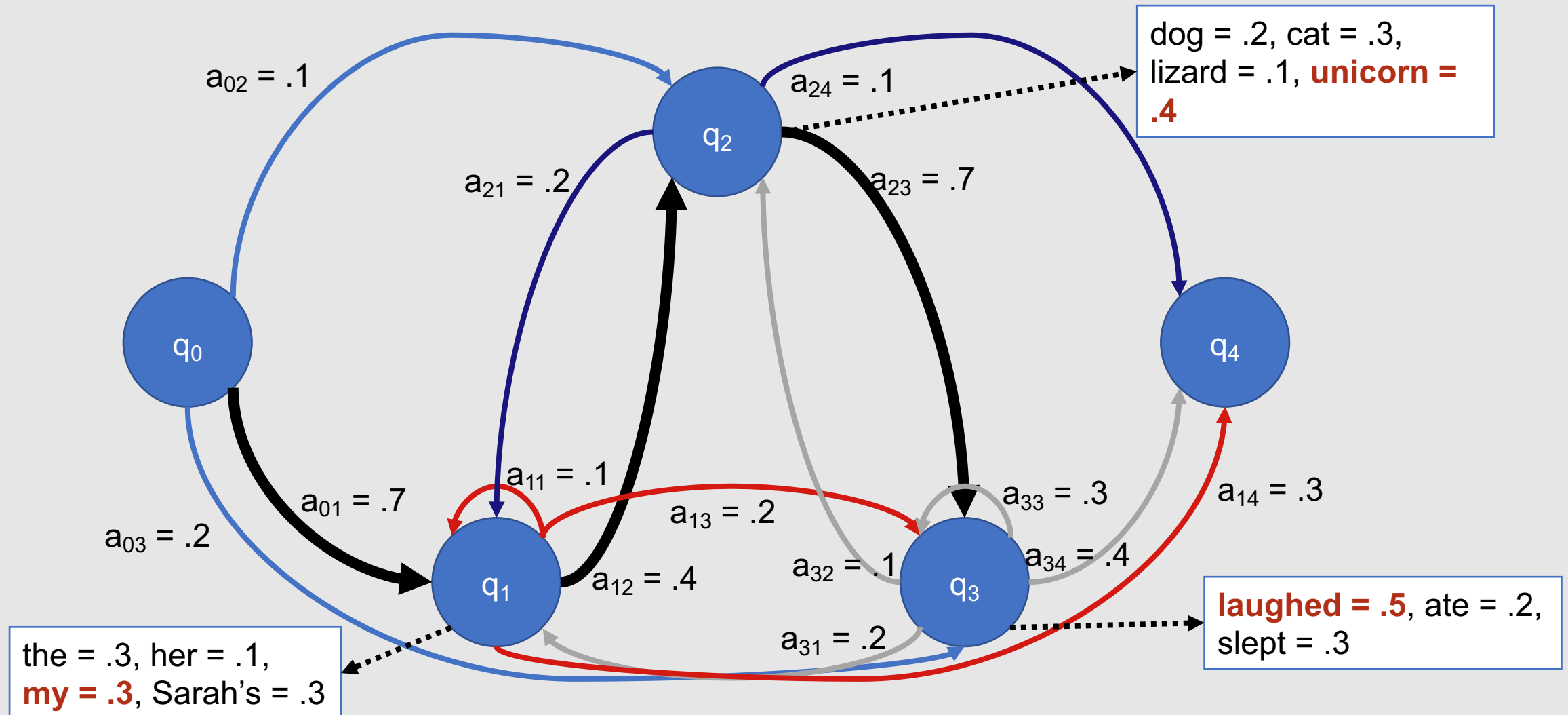
# Sample Text Generation



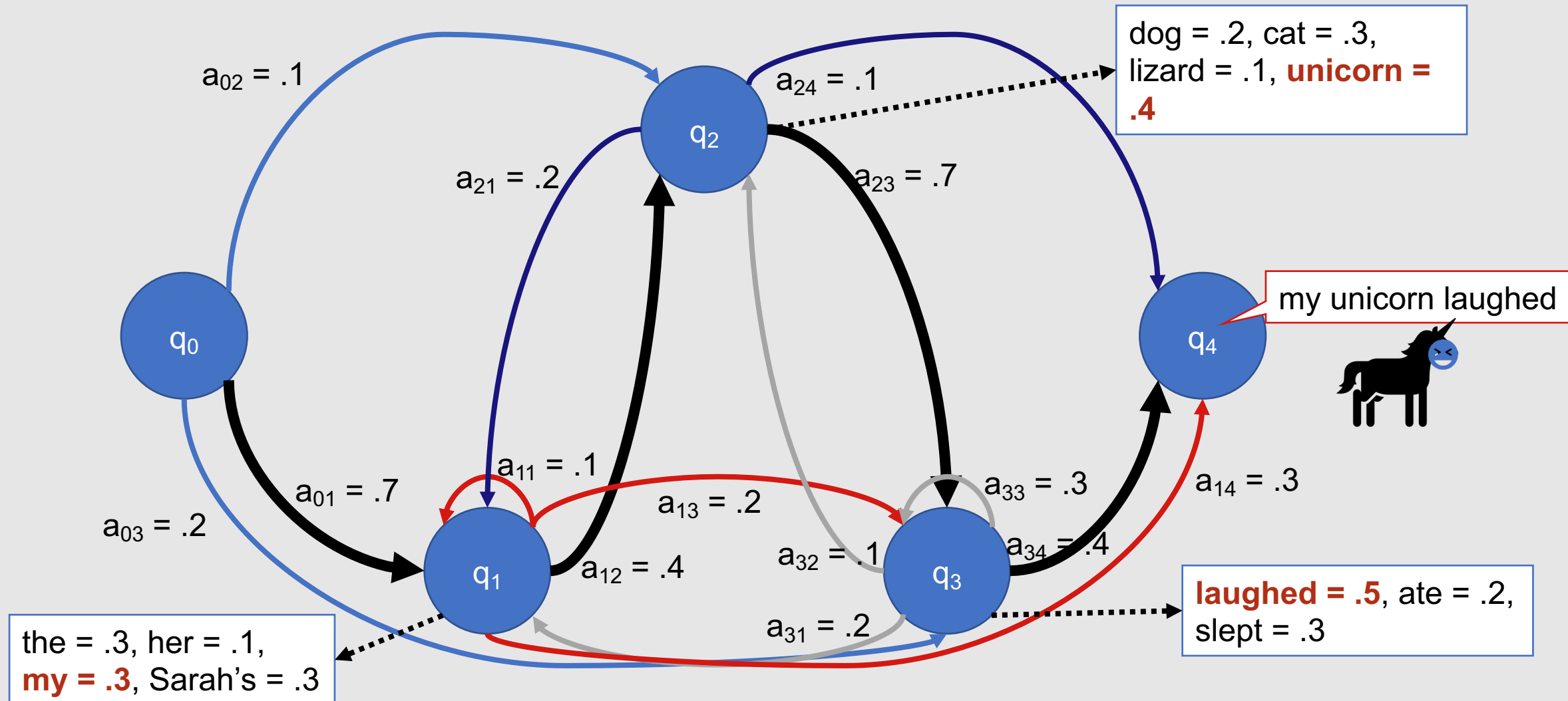
# Sample Text Generation



# Sample Text Generation



# Sample Text Generation



## Three Fundamental HMM Problems

- Observation Likelihood: How likely is a particular observation sequence to occur?
- Decoding: What is the best sequence of hidden states for an observed sequence?
  - What is the best sequence of labels for our test data?
- Learning: What are the transition probabilities and observation likelihoods that best fit the observation sequence and HMM states?
  - How do we empirically fit our training data?

# Observation Likelihood

- Given a sequence of observations and an HMM, what is the probability that this sequence was generated by the model?
- Allows the HMM to be used as a **language model**: A formal probabilistic model of a language that assigns a probability to each string by saying how likely that string was to have been generated by the language.
- Useful for two tasks:
  - Sequence classification
  - Selecting the most likely sequence



# Sequence Classification

- Assuming an HMM is available for every possible class, what is the most likely class for a given observation sequence?
  - Which HMM is most likely to have generated the sequence?
- HMMs are commonly used in automated speech recognition (ASR) for this purpose
  - Given a set of sounds, what is the most likely word?

# Most Likely Sequence

- Of two or more possible sequences, which one was most likely generated by a given HMM?
- Also useful for speech recognition
  - Rank alternative word sequence interpretations

How can we compute  
the observation  
likelihood?

- Naïve Solution:
  - Consider all possible state sequences,  $Q$ , of length  $T$  that the model,  $\lambda$ , could have traversed in generating the given observation sequence,  $O$
  - Compute the probability of a given state sequence from  $A$ , and multiply it by the probability of generating the given observation sequence for that state sequence
    - $P(O, Q \mid \lambda) = P(O \mid Q, \lambda) * P(Q \mid \lambda)$
  - Repeat for all possible state sequences, and sum over all to get  $P(O \mid \lambda)$
- But, this is computationally complex!
  - $O(TN^T)$

How can we compute  
the observation  
likelihood?

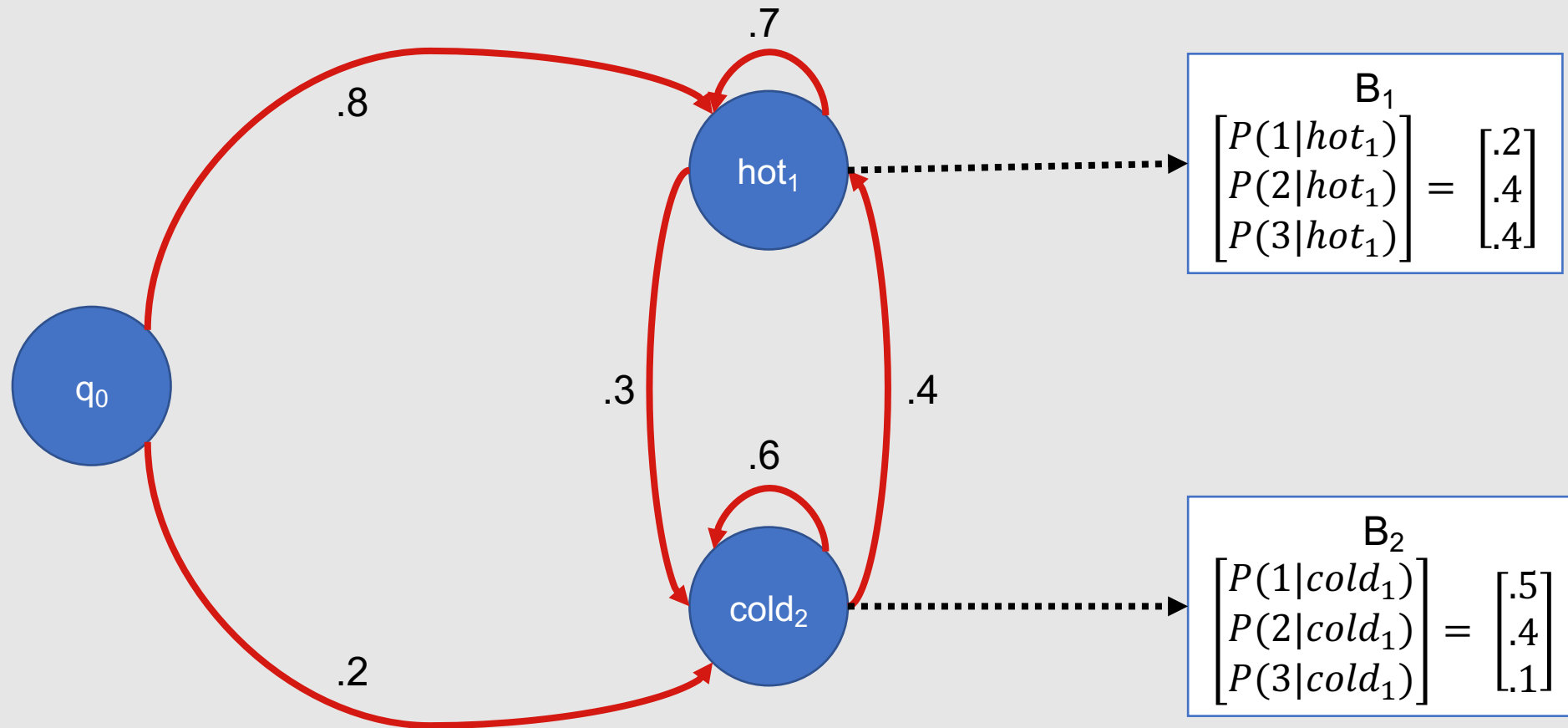
- Efficient Solution:
  - **Forward Algorithm:** Dynamic programming algorithm that computes the observation probability by summing over the probabilities of all possible hidden state paths that could generate the observation sequence.
  - Implicitly folds each of these paths into a single forward trellis
- Why does this work?
  - Markov assumption (the probability of being in any state at a given time  $t$  only relies on the probability of being in each possible state at time  $t-1$ ).
- Works in  $O(TN^2)$  time!

# Sample Problem

- It is 2799 and you are a climatologist studying the history of global warming
- Unfortunately, you have no records of the weather in Baltimore for the summer of 2007, although you do know how likely it was in general to move from a hot day to a cold day and so forth at that time
- Fortunately, a major breakthrough occurs: you find Jason Eisner's diary, which lists how many ice cream cones he ate every day that summer
- You decide to use those observations to estimate whether each day in a three-day sequence was hot or cold
  - Day 1: 3 ice cream cones
  - Day 2: 1 ice cream cone
  - Day 3: 3 ice cream cones



# Corresponding HMM



# How do you compute your forward probabilities?

- Let  $\alpha_i(j)$  be the probability of being in state  $j$  after seeing the first  $t$  observations, given your HMM  $\lambda$
- $\alpha_i(j)$  is computed by summing over the probabilities of every path that could lead you to this cell
  - $\alpha_i(j) = P(o_1, o_2 \dots o_t, q_t = j | \lambda) = \sum_{i=1}^N \alpha_{t-1}(i) a_{ij} b_j(o_t)$ 
    - $q_t = j$  is the probability that the  $t^{th}$  state in the sequence of states is state  $j$
  - $\alpha_{t-1}(i)$ : The previous forward path probability from the previous time step
  - $a_{ij}$ : The transition probability from previous state  $q_i$  to current state  $q_j$
  - $b_j(o_t)$ : The state observation likelihood of the observed item  $o_t$  given the current state  $j$

# Formal Algorithm

create a probability matrix  $forward[N+2, T]$

for each state  $q$  in  $[1, \dots, N]$  do:

$forward[q, 1] \leftarrow a_{0,q} * b_q(o_1)$

for each time step  $t$  from 2 to  $T$  do:

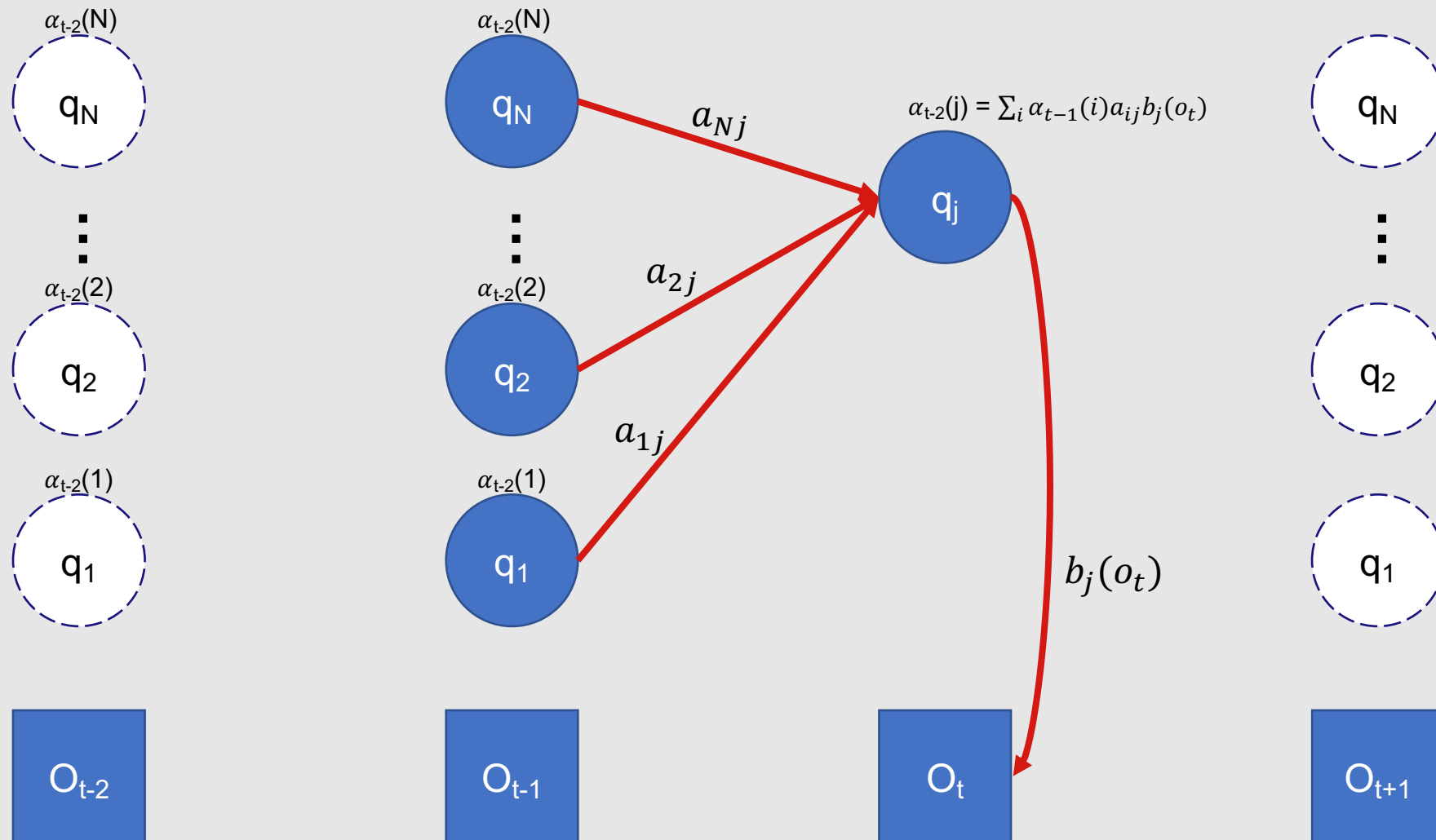
    for each state  $q$  from 1 to  $N$  do:

$forward[q, t] \leftarrow \sum_{q'=1}^N forward[q', t-1] * a_{q',q} * b_s(o_t)$

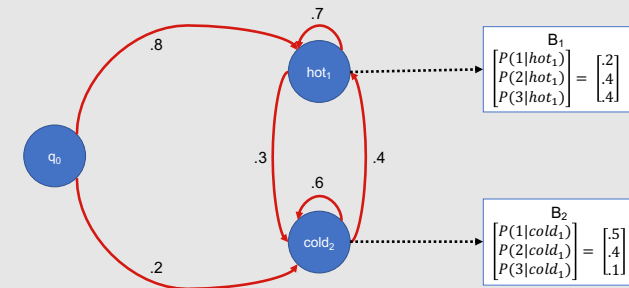
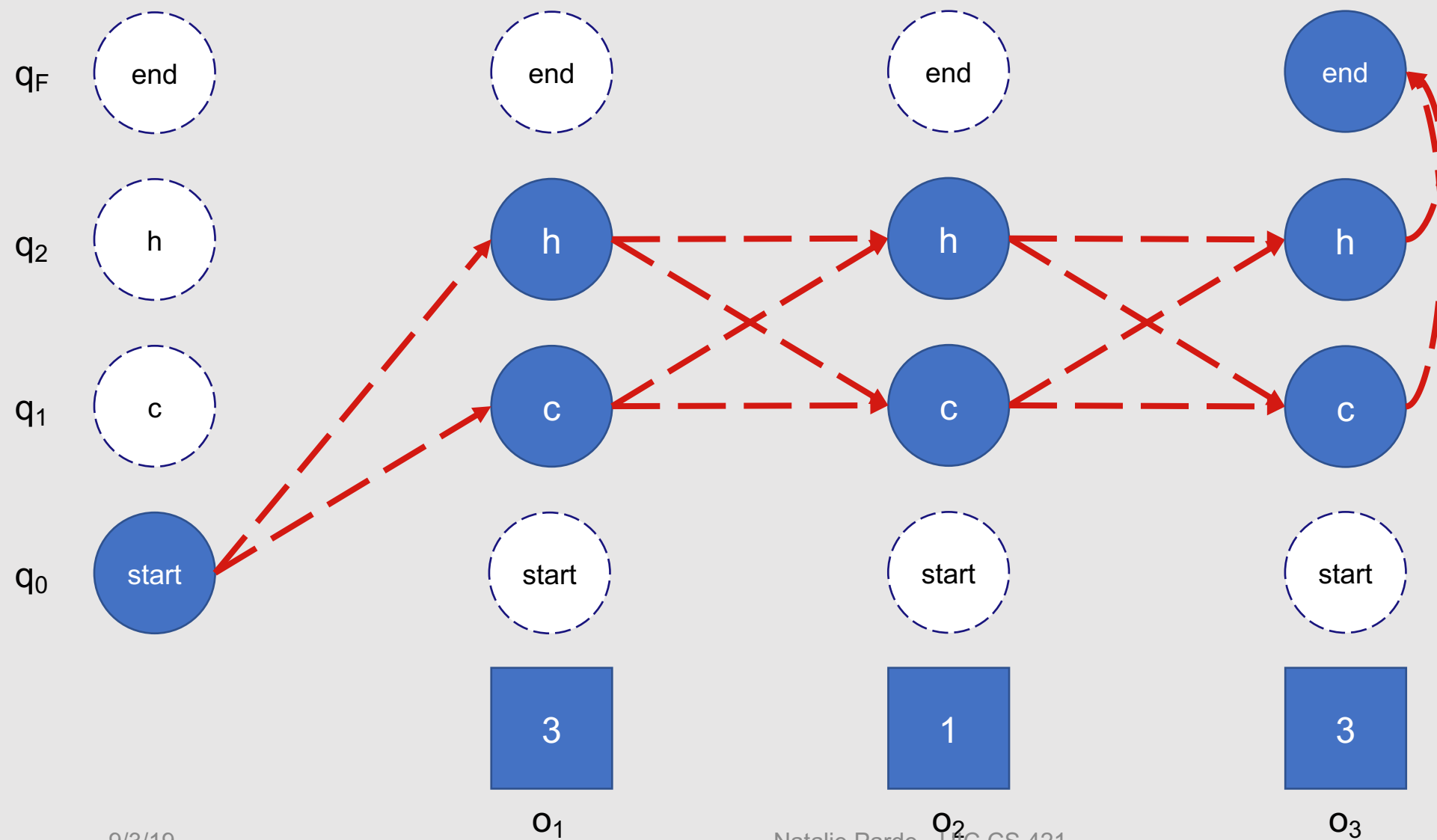
$forward[q_F, T] \leftarrow \sum_{q=1}^N forward[q, T] * a_{s,q_F}$



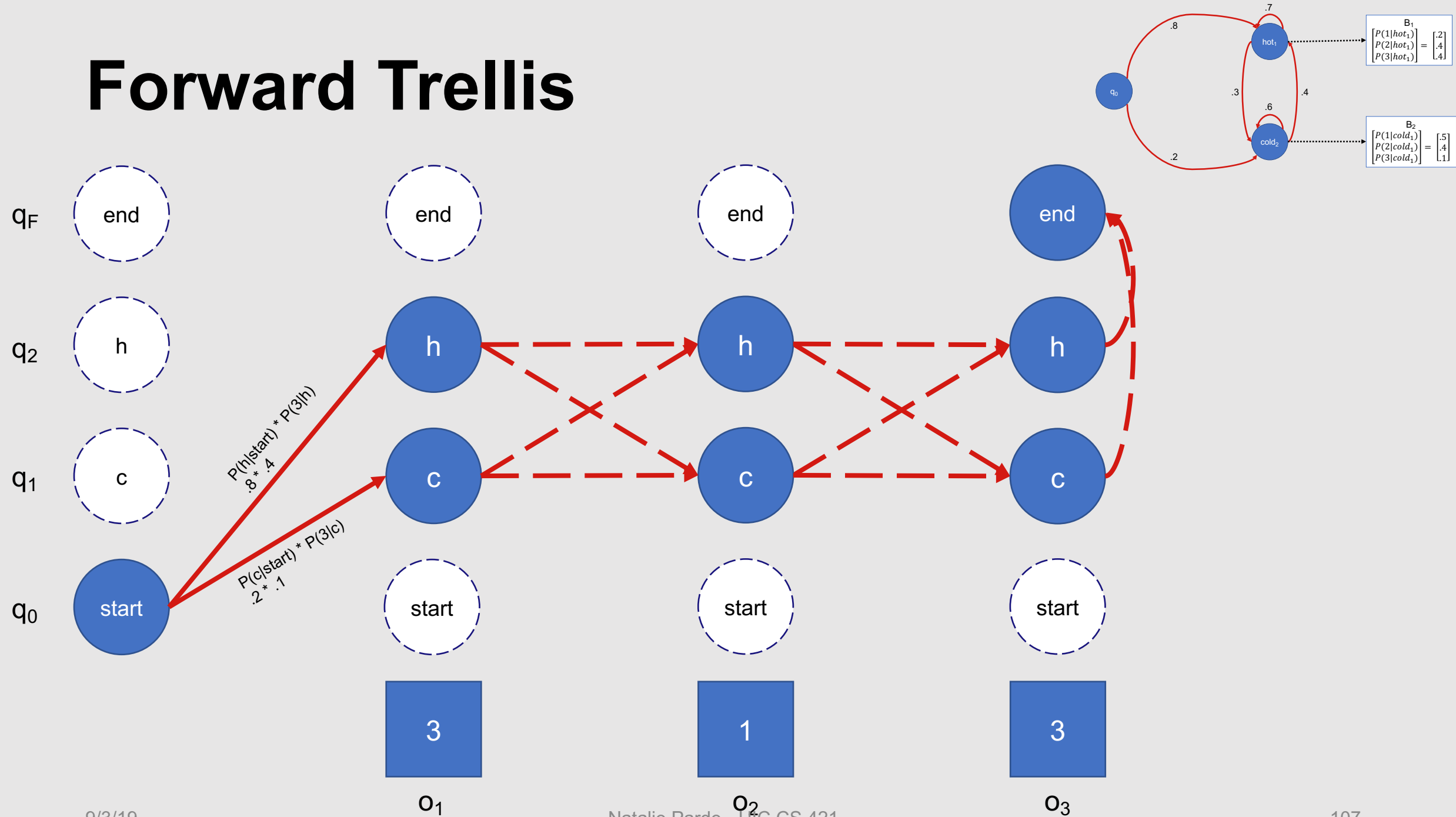
# Forward Step



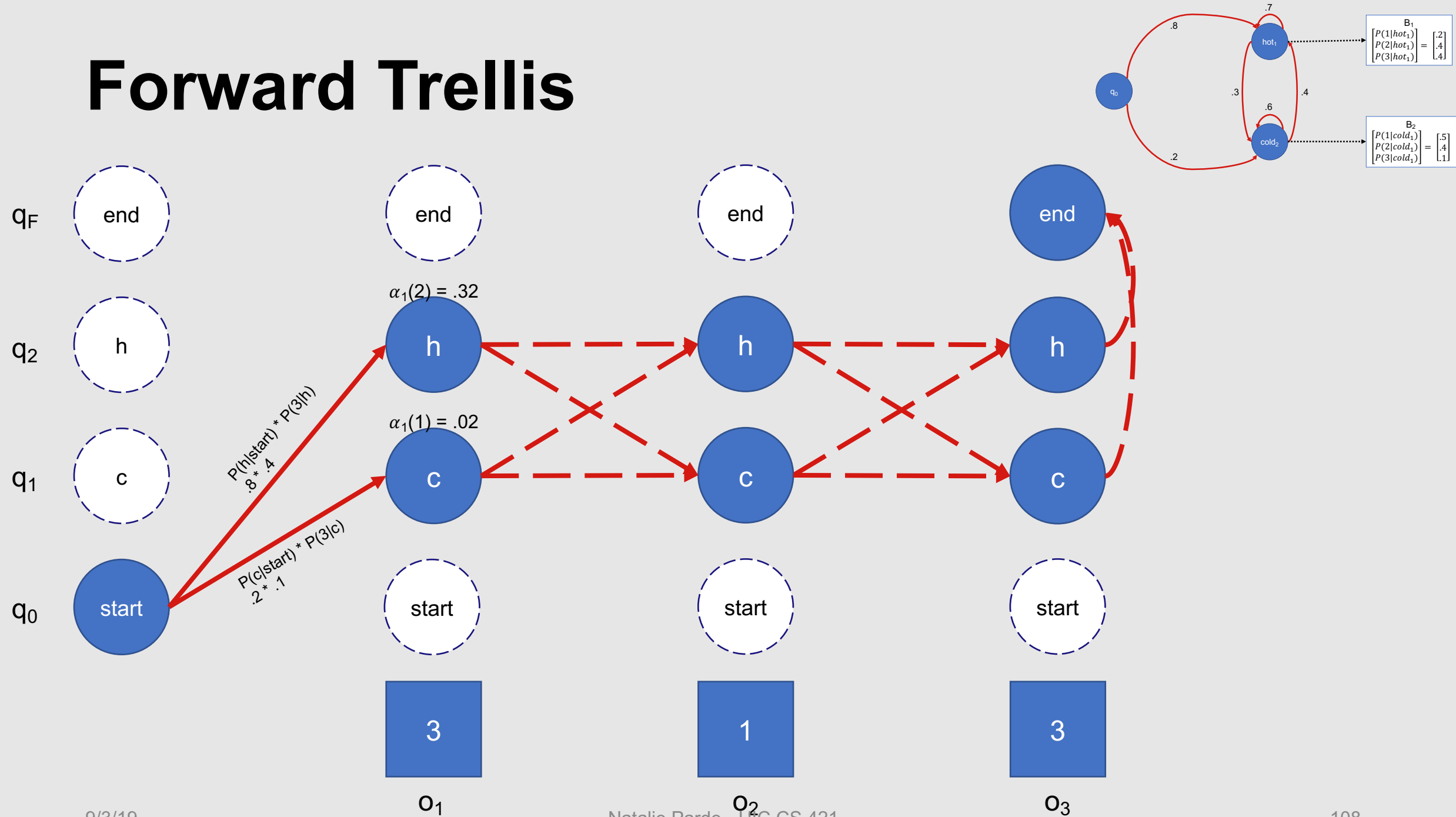
# Forward Trellis



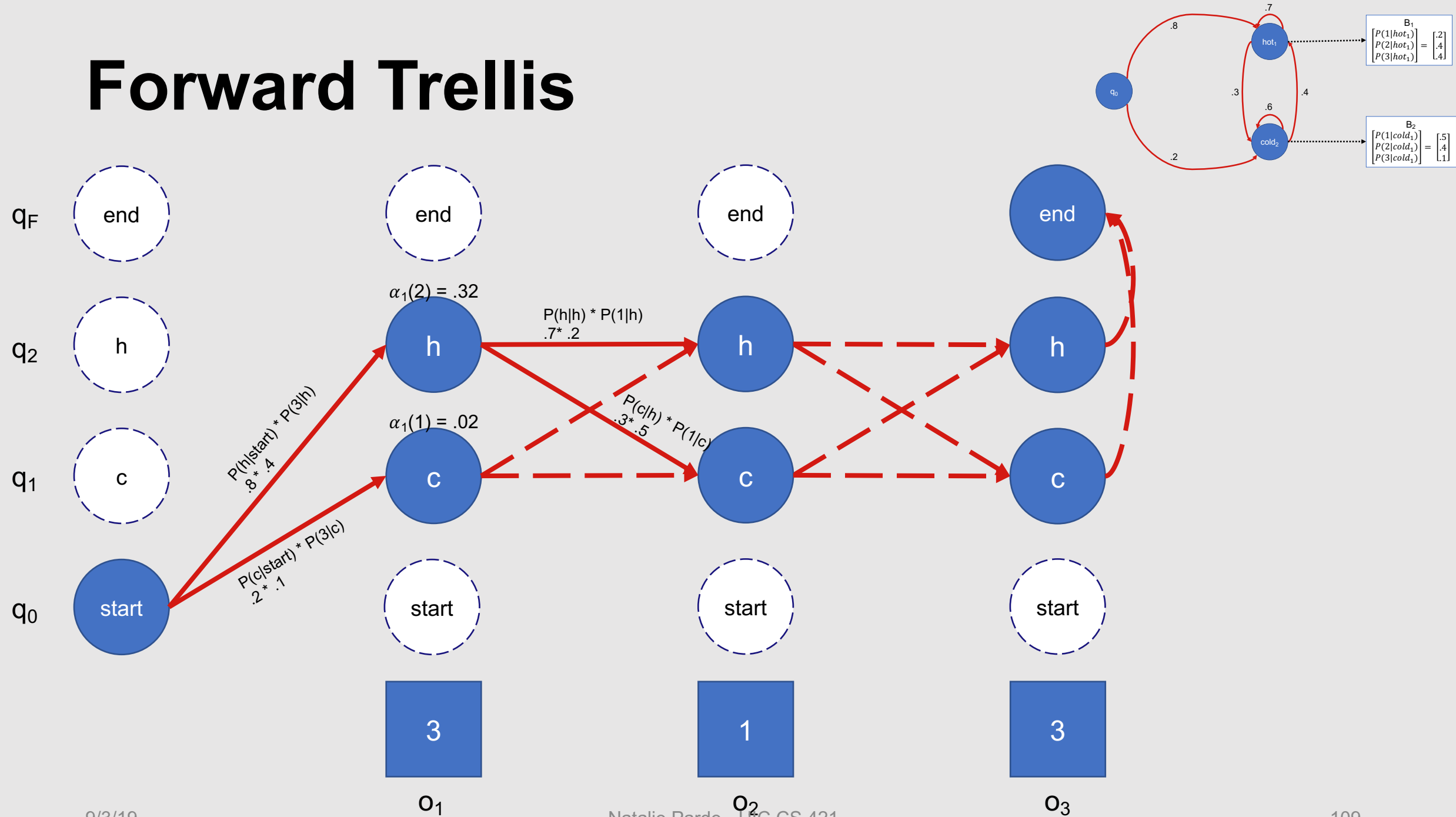
# Forward Trellis



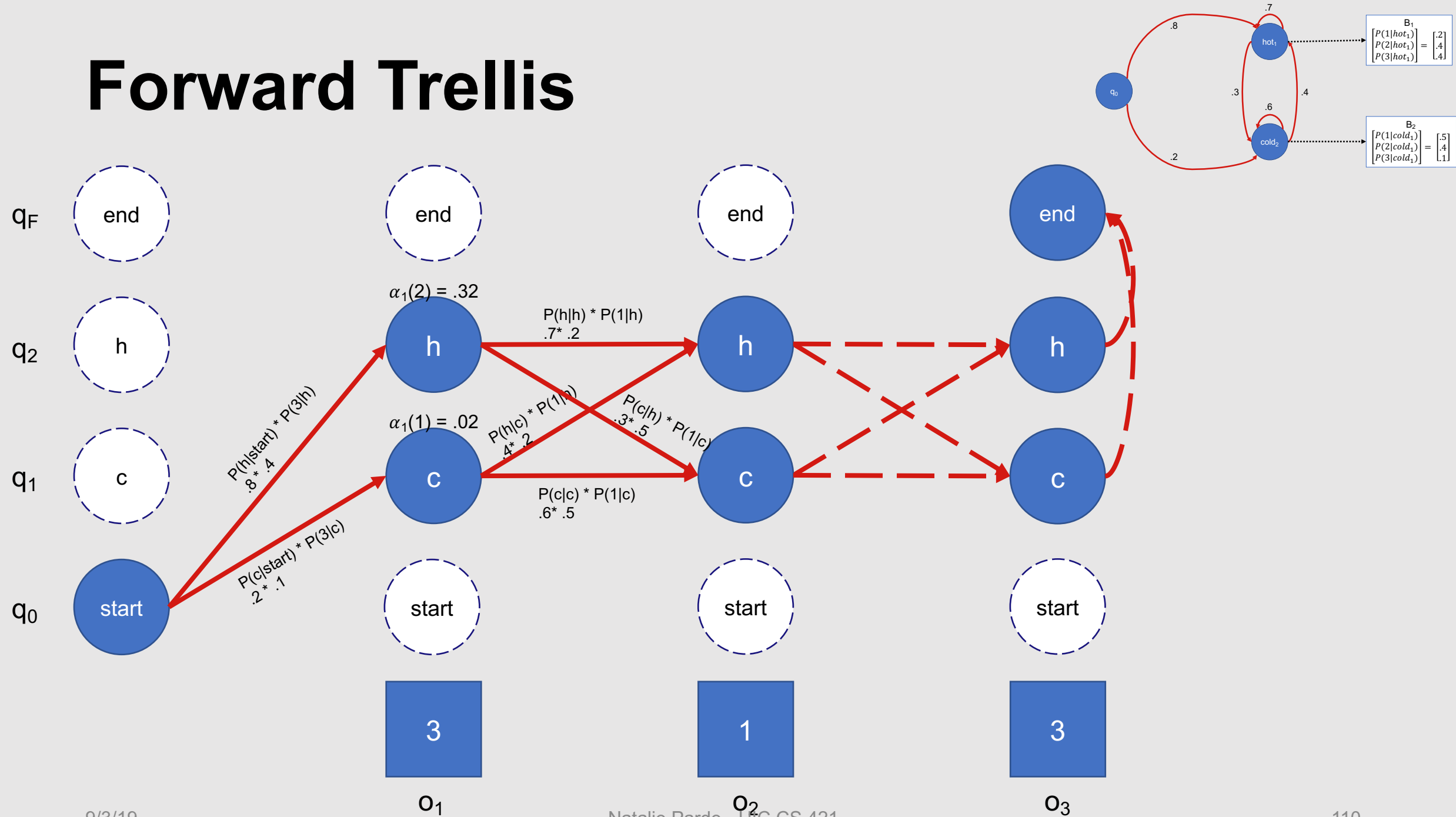
# Forward Trellis



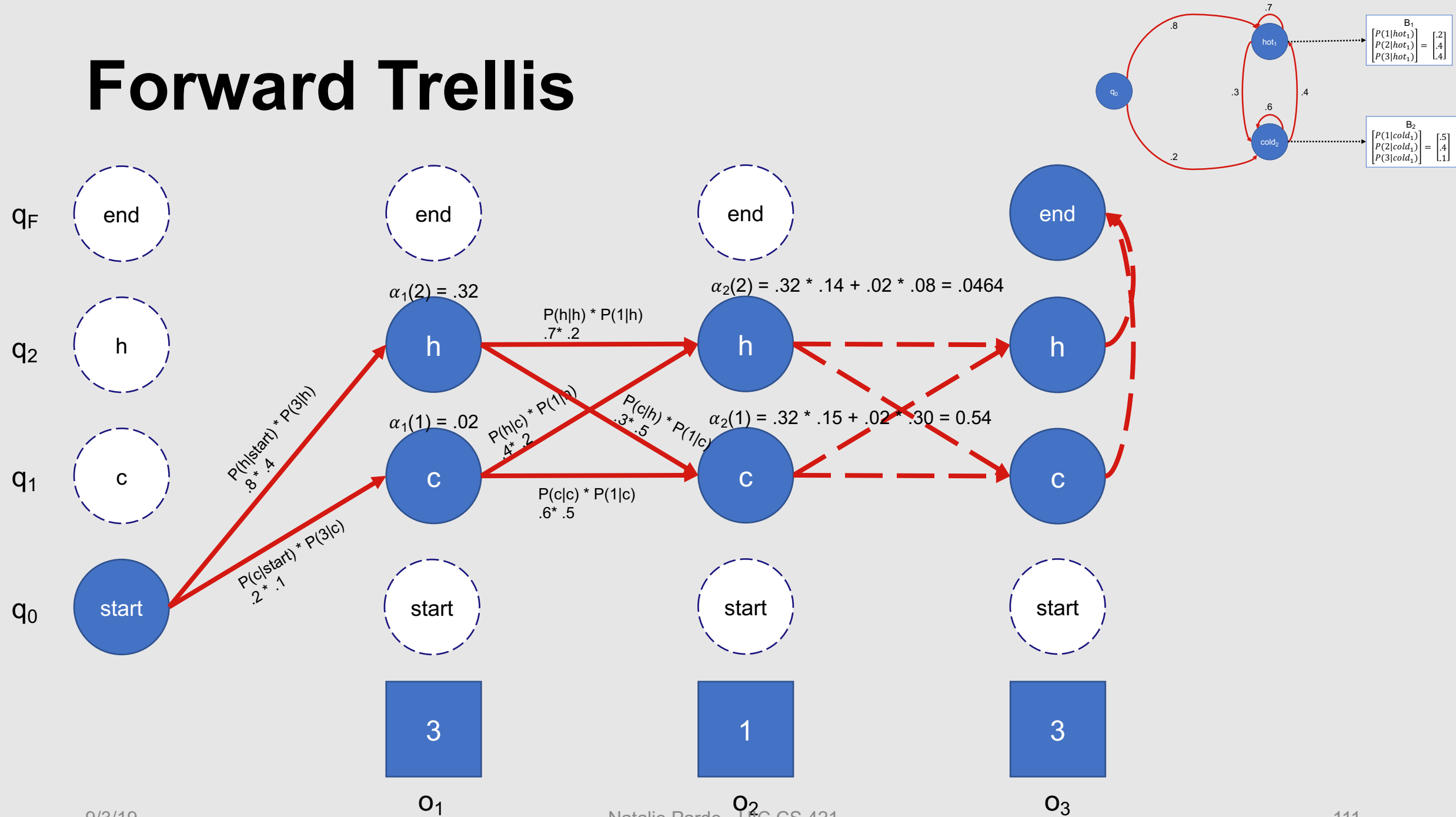
# Forward Trellis



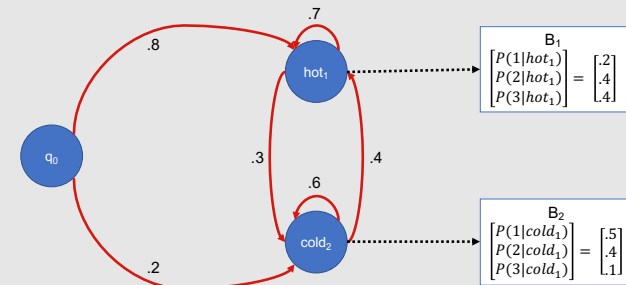
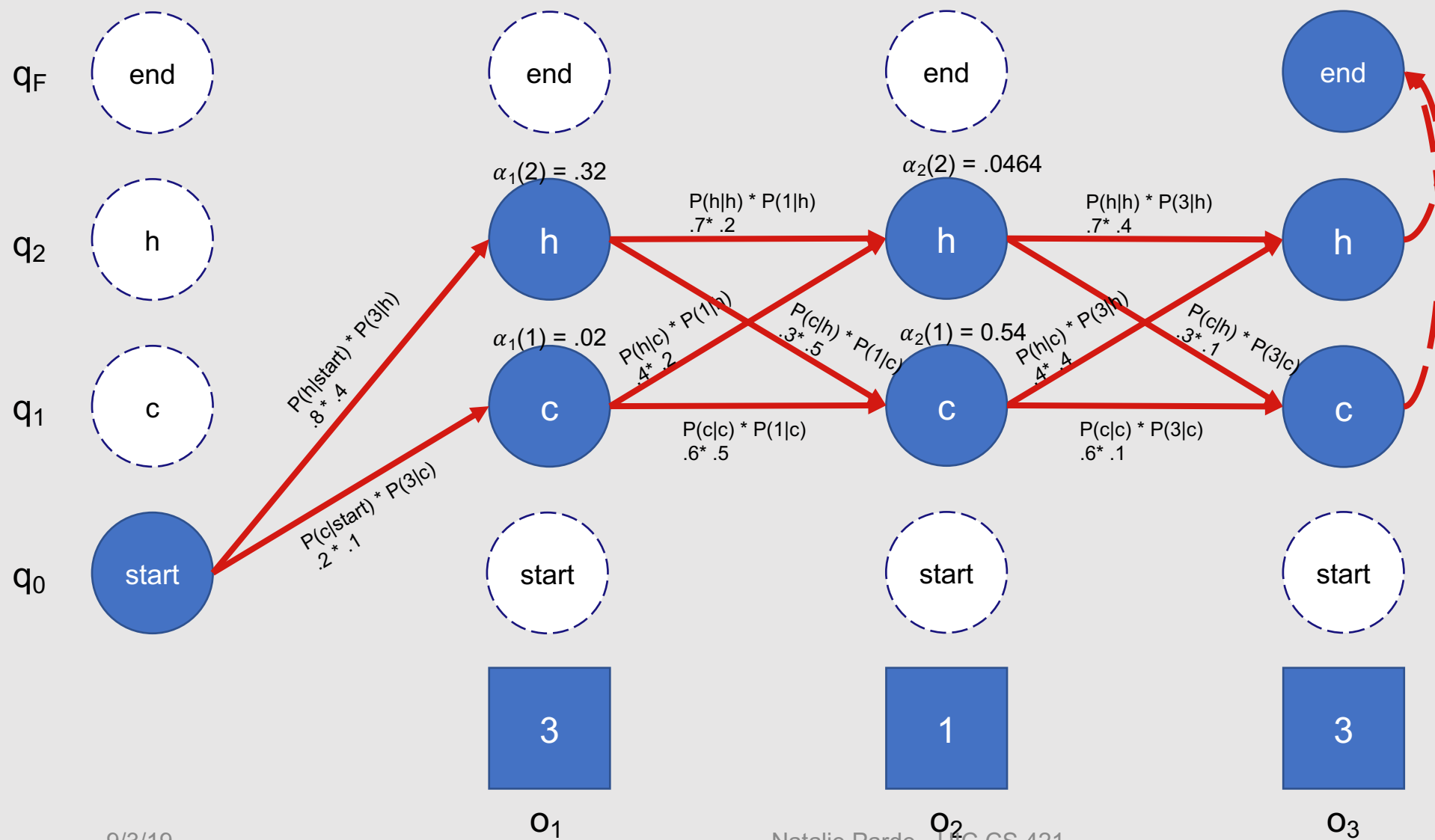
# Forward Trellis



# Forward Trellis

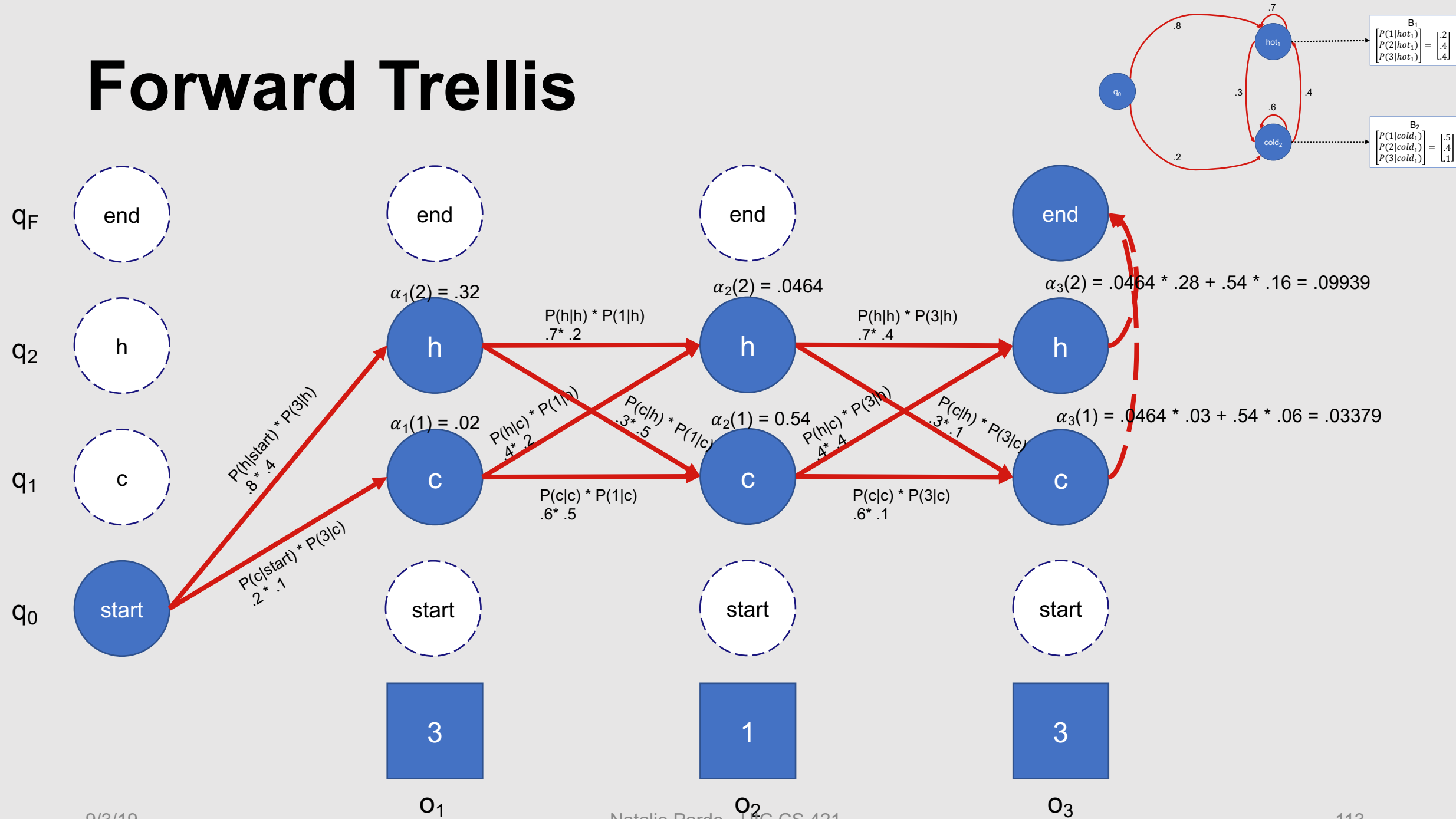


# Forward Trellis





# Forward Trellis



# Decoding

- Given an observation sequence and an HMM, what is the best hidden state sequence?
  - How do we choose a state sequence that is optimal in some sense (e.g., best explains the observations)?
- Very useful for sequence labeling!

## Naïve Approach:

- For each hidden state sequence  $Q$ , compute  $P(O|Q)$
- Pick the sequence with the highest probability

However, this is computationally inefficient!

- $O(N^T)$

## Decoding

# How can we decode sequences more efficiently?

- **Viterbi Algorithm**
  - Another dynamic programming algorithm
  - Uses a similar trellis to the Forward algorithm
- Viterbi time complexity:  $O(N^2T)$

# Viterbi Intuition

- **Goal:** Compute the joint probability of the observation sequence together with the best state sequence
- So, **recursively compute the probability of the most likely subsequence of states** that accounts for the first  $t$  observations and ends in state  $q_j$ .
  - $v_t(j) = \max_{q_0, q_1, \dots, q_{t-1}} P(q_0, q_1, \dots, q_{t-1}, o_1, \dots, o_t, q_t = q_j | \lambda)$
- Also **record backpointers** that subsequently allow you to backtrack the most probable state sequence
  - $bt_t(j)$  stores the state at time  $t-1$  that maximizes the probability that the system was in state  $q_j$  at time  $t$ , given the observed sequence

# Formal Algorithm

create a path probability matrix  $Viterbi[N+2, T]$

for each state  $q$  in  $[1, \dots, N]$  do:

$Viterbi[q, 1] \leftarrow a_{0,q} * b_q(o_1)$

$backpointer[q, 1] \leftarrow 0$

for each time step  $t$  in  $[2, \dots, T]$  do:

for each state  $q$  in  $[1, \dots, N]$  do:

$viterbi[q, t] \leftarrow \max_{q' \in [1, \dots, N]} viterbi[q', t-1] * a_{q',q} * b_q(o_t)$

$backpointer[q, t] \leftarrow \operatorname{argmax}_{q' \in [1, \dots, N]} viterbi[q', t-1] * a_{q',q}$

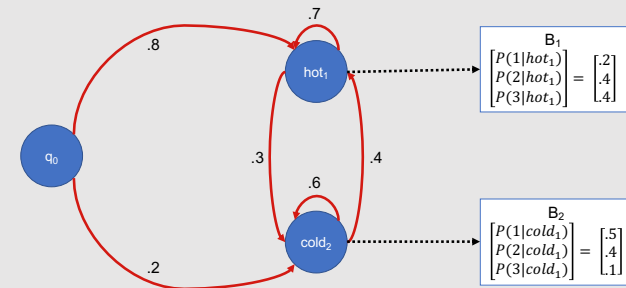
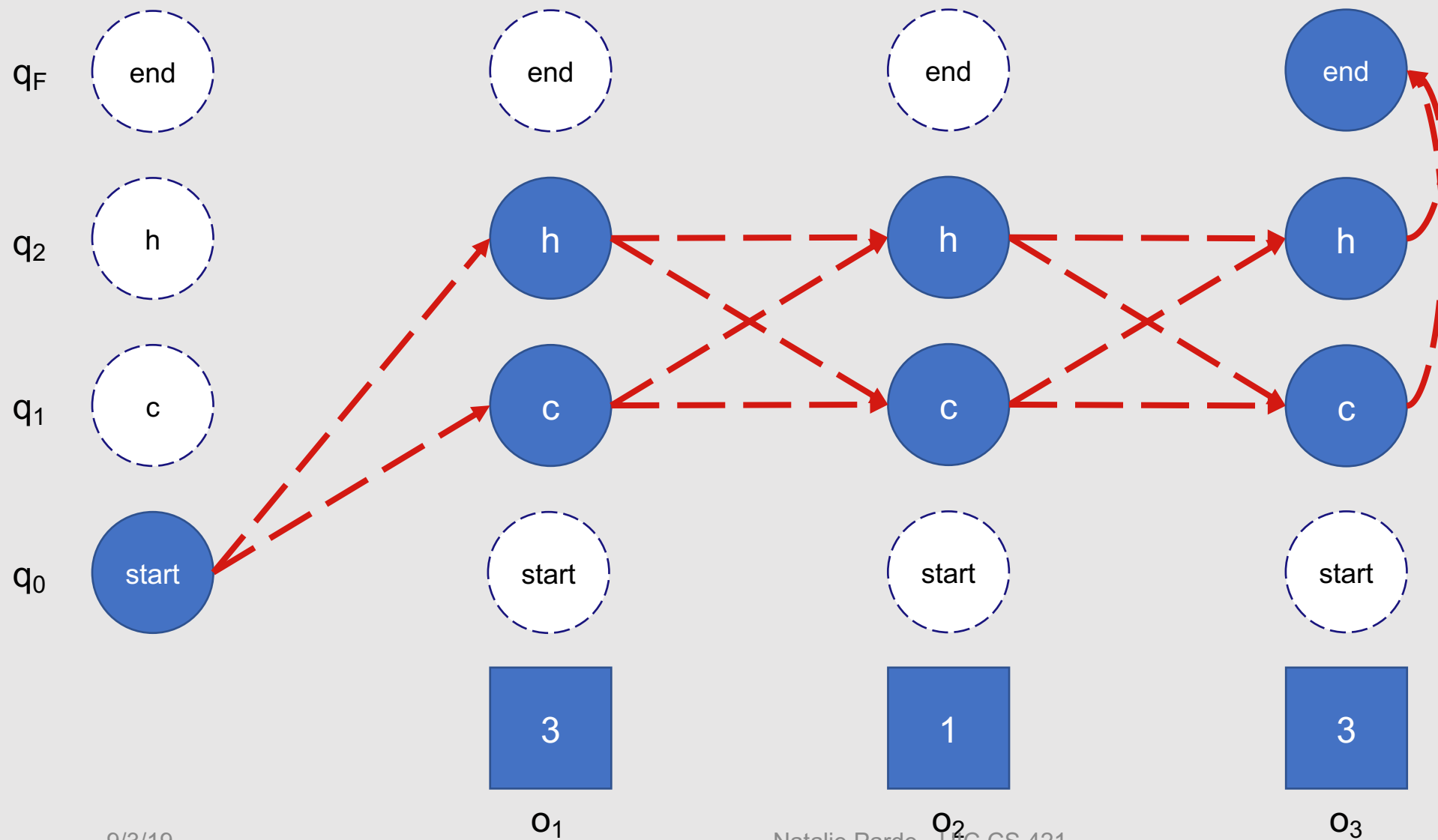
$viterbi[q_F, T] \leftarrow \max_{q' \in [1, \dots, N]} viterbi[q', T] * a_{q',q_F}$

$backpointer[q_F, T] \leftarrow \operatorname{argmax}_{q' \in [1, \dots, N]} viterbi[q', T] * a_{q',q_F}$

Seem familiar?

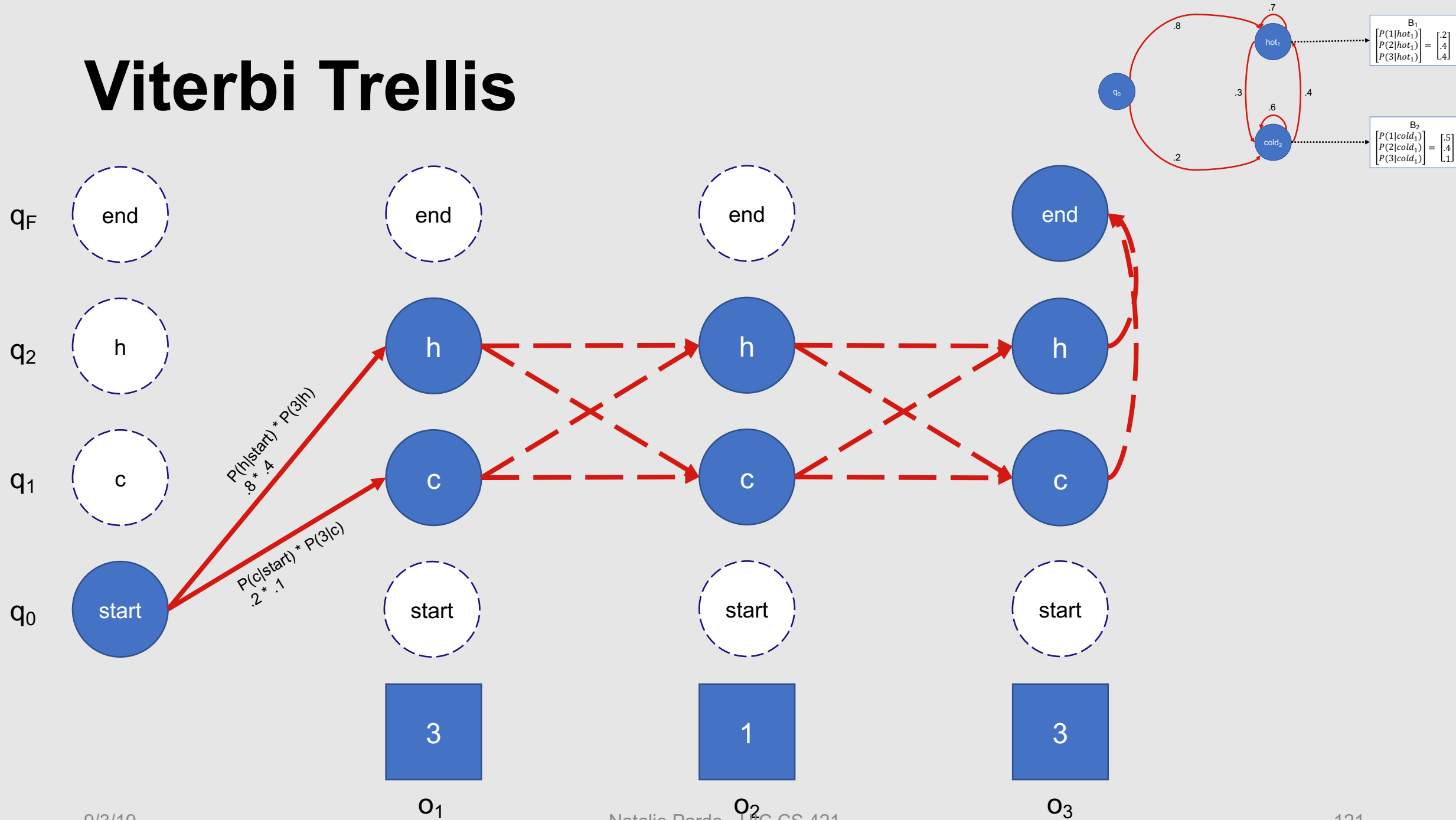
- Viterbi is basically the forward algorithm + backpointers, and substituting a max function for the summation operator

# Viterbi Trellis

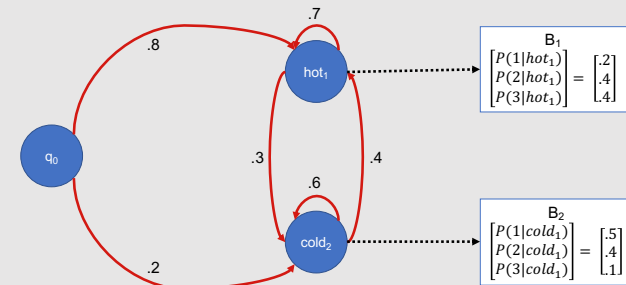
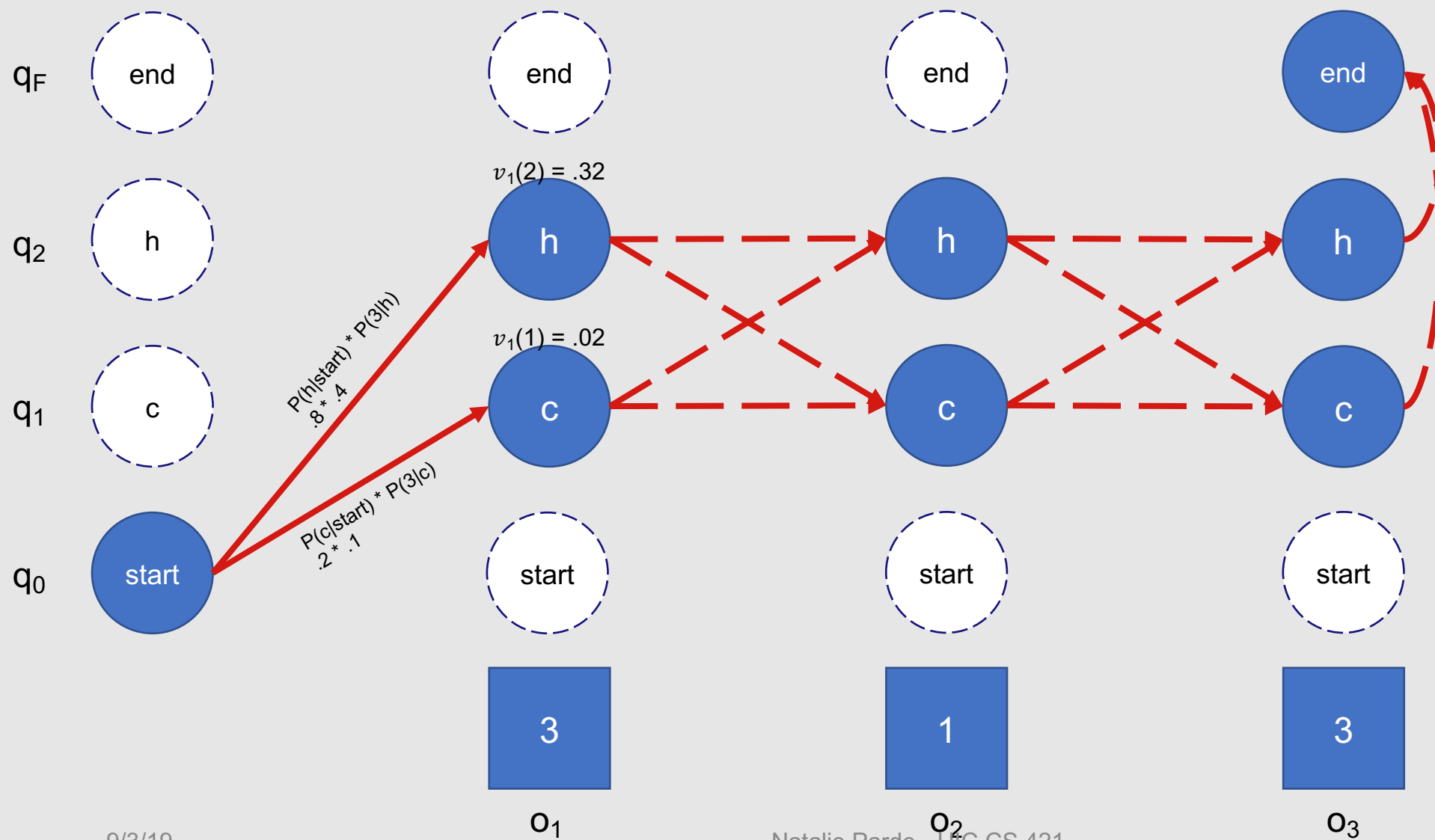




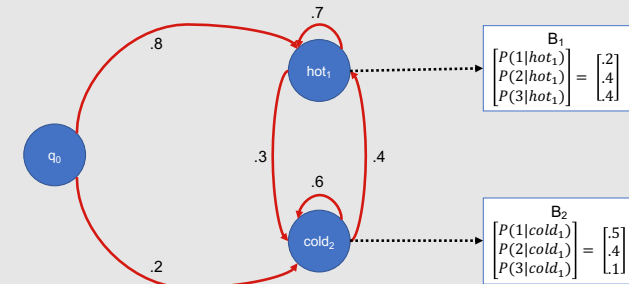
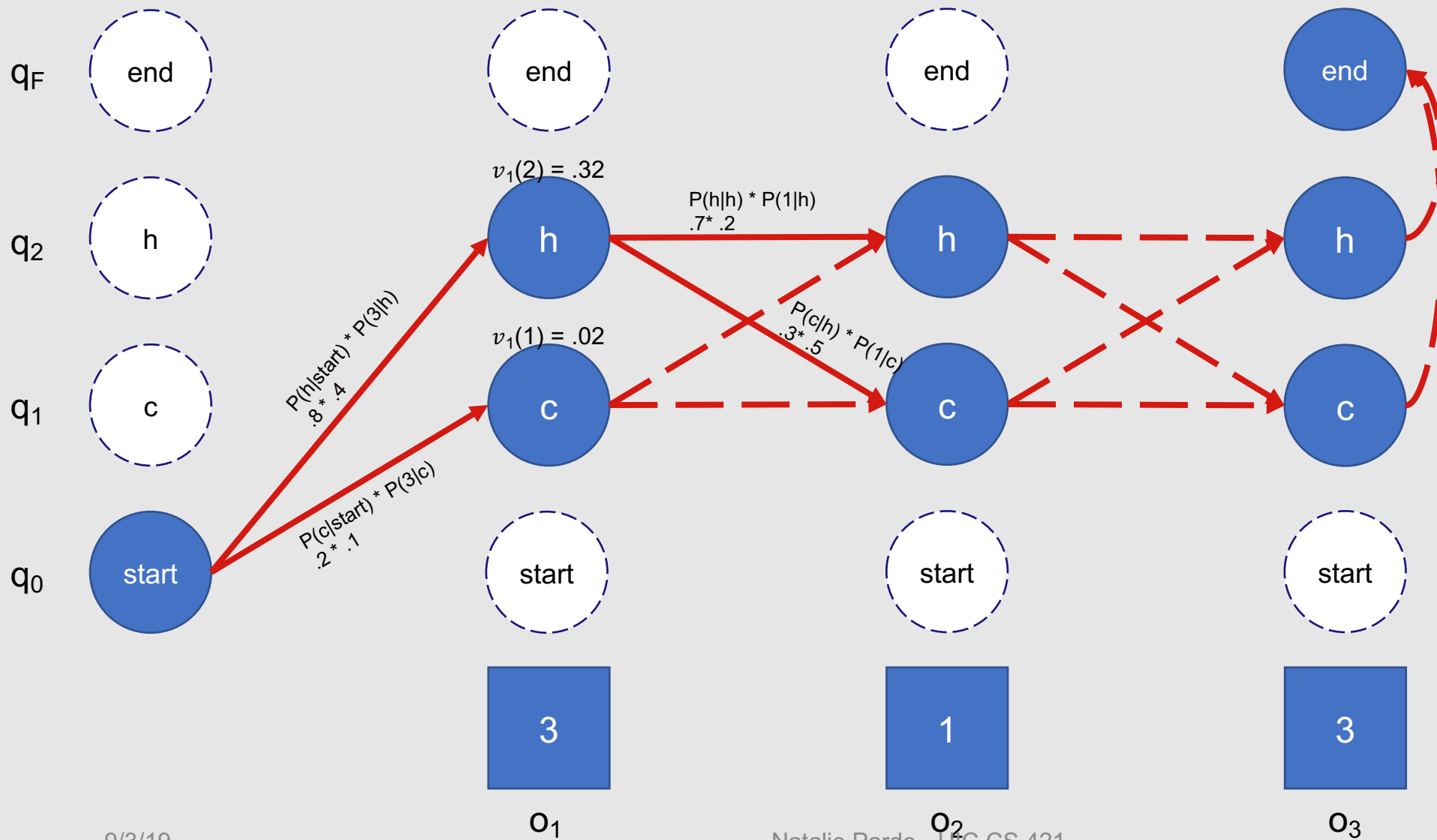
# Viterbi Trellis



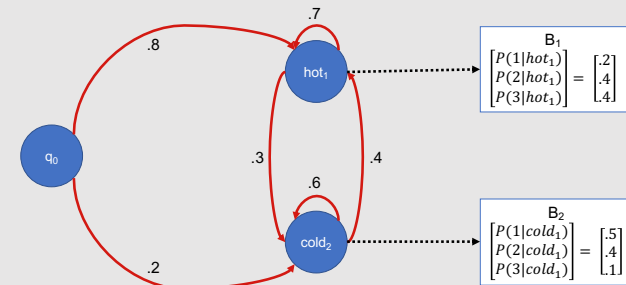
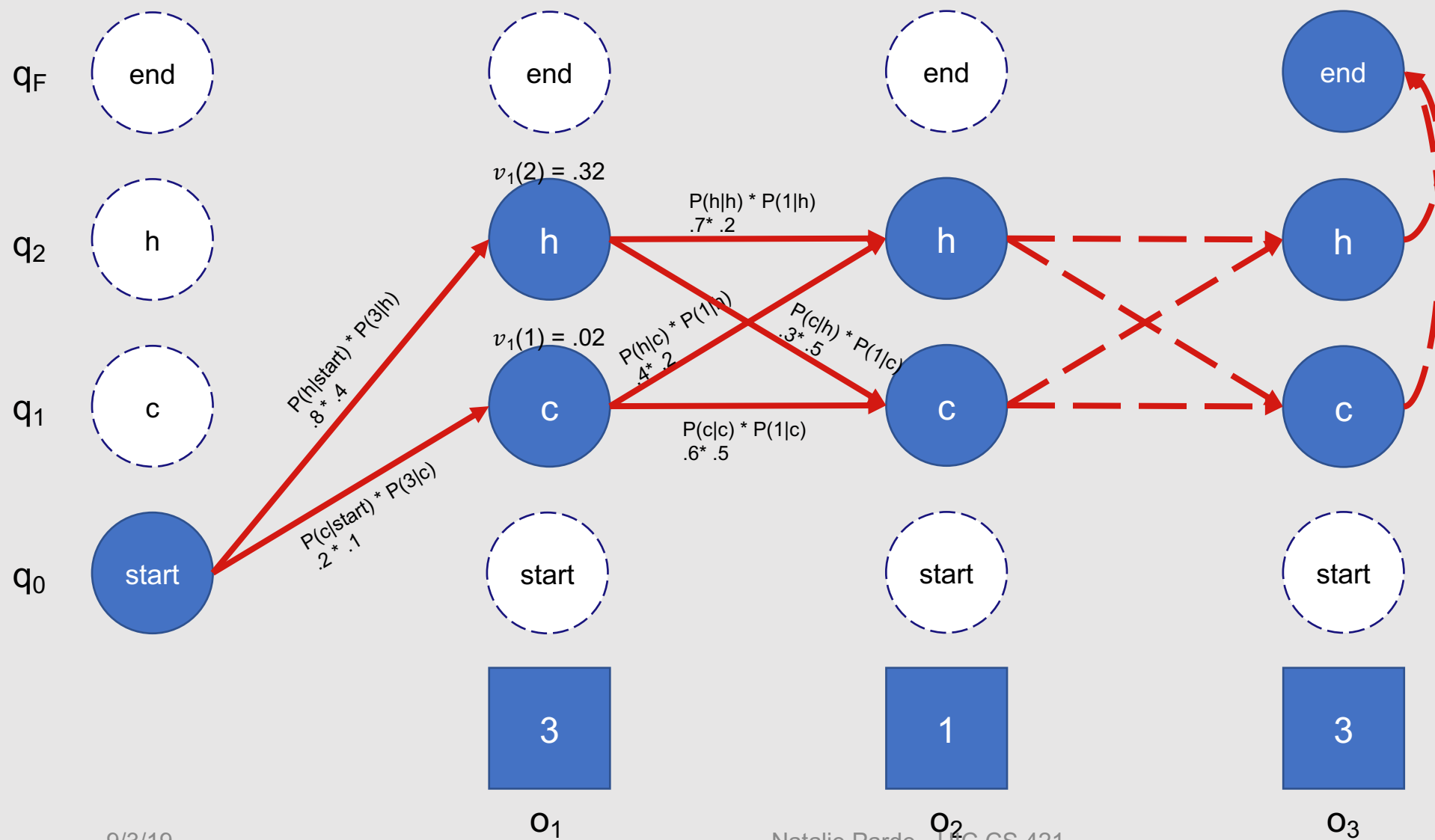
# Viterbi Trellis



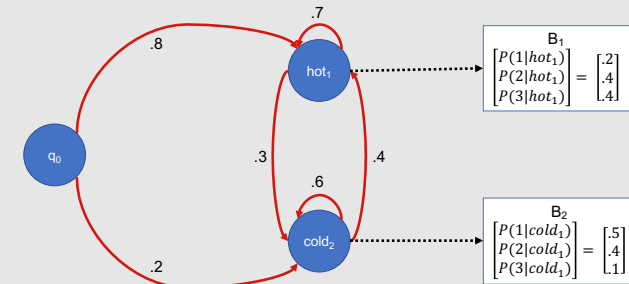
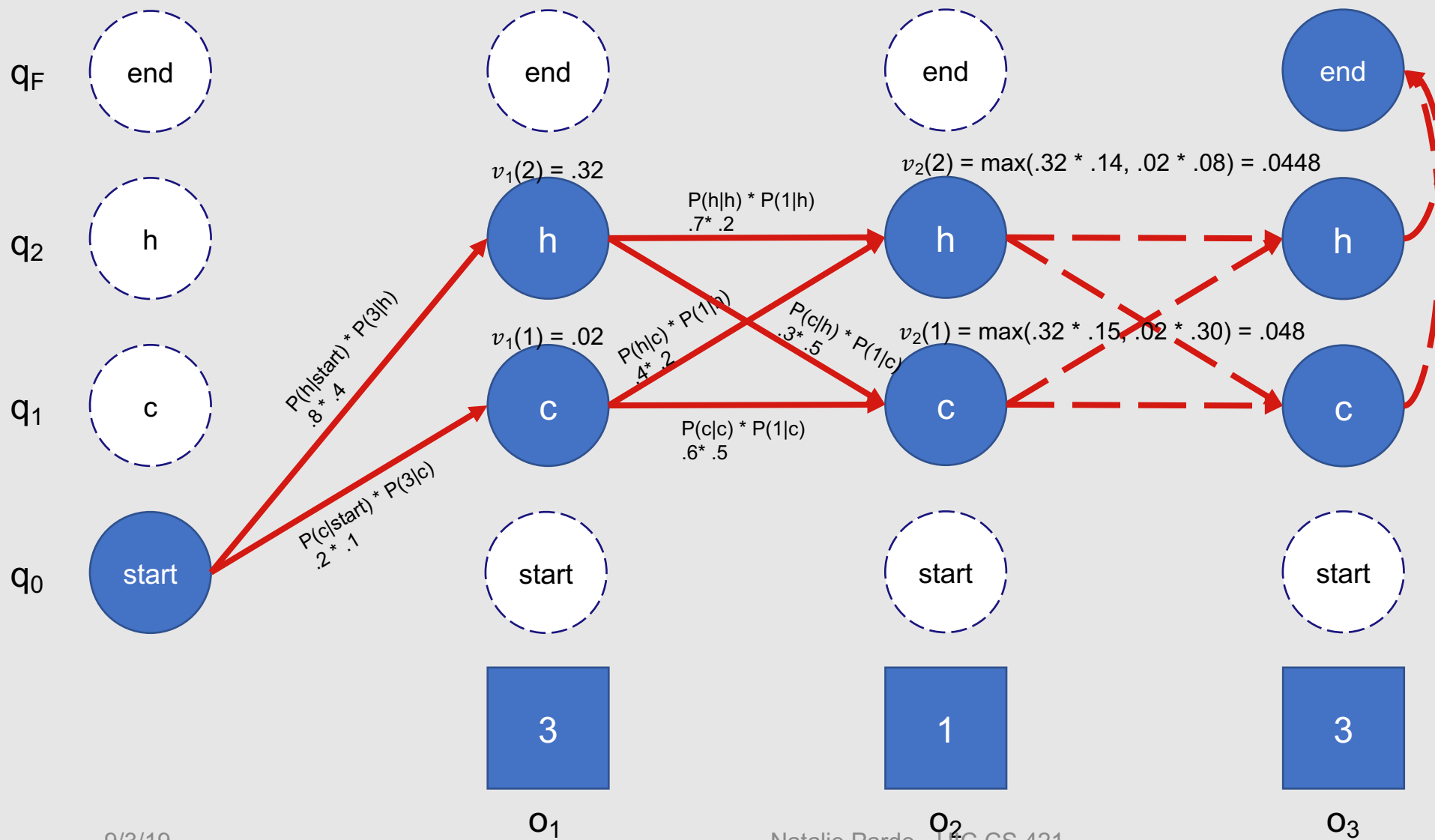
# Viterbi Trellis



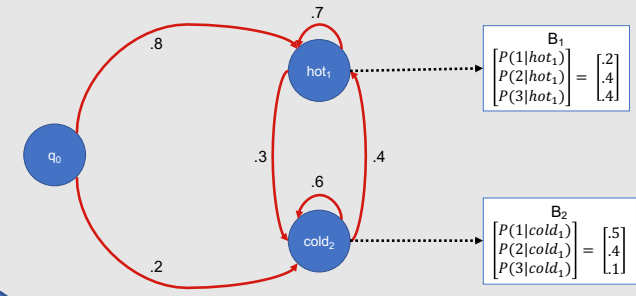
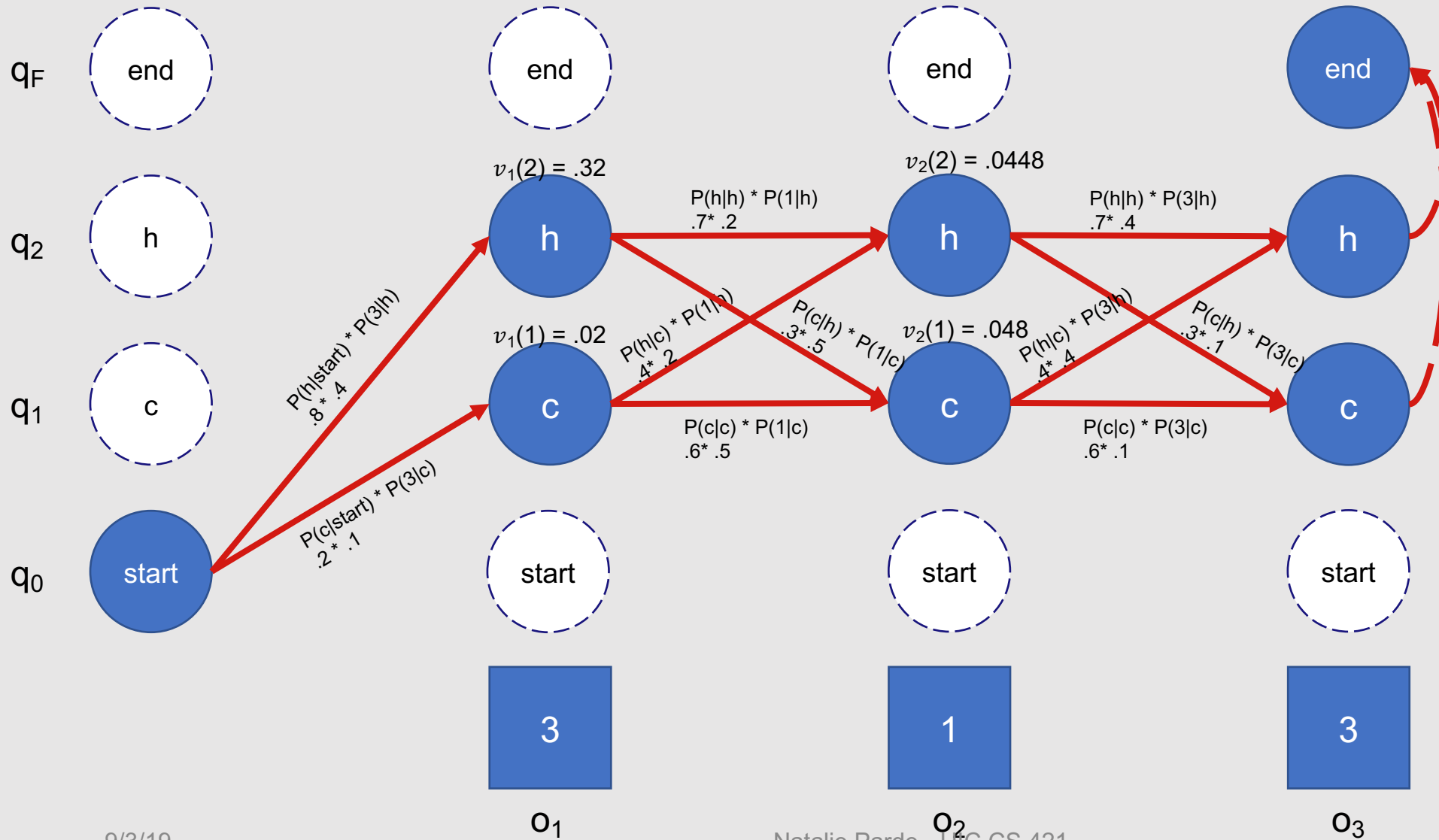
# Viterbi Trellis



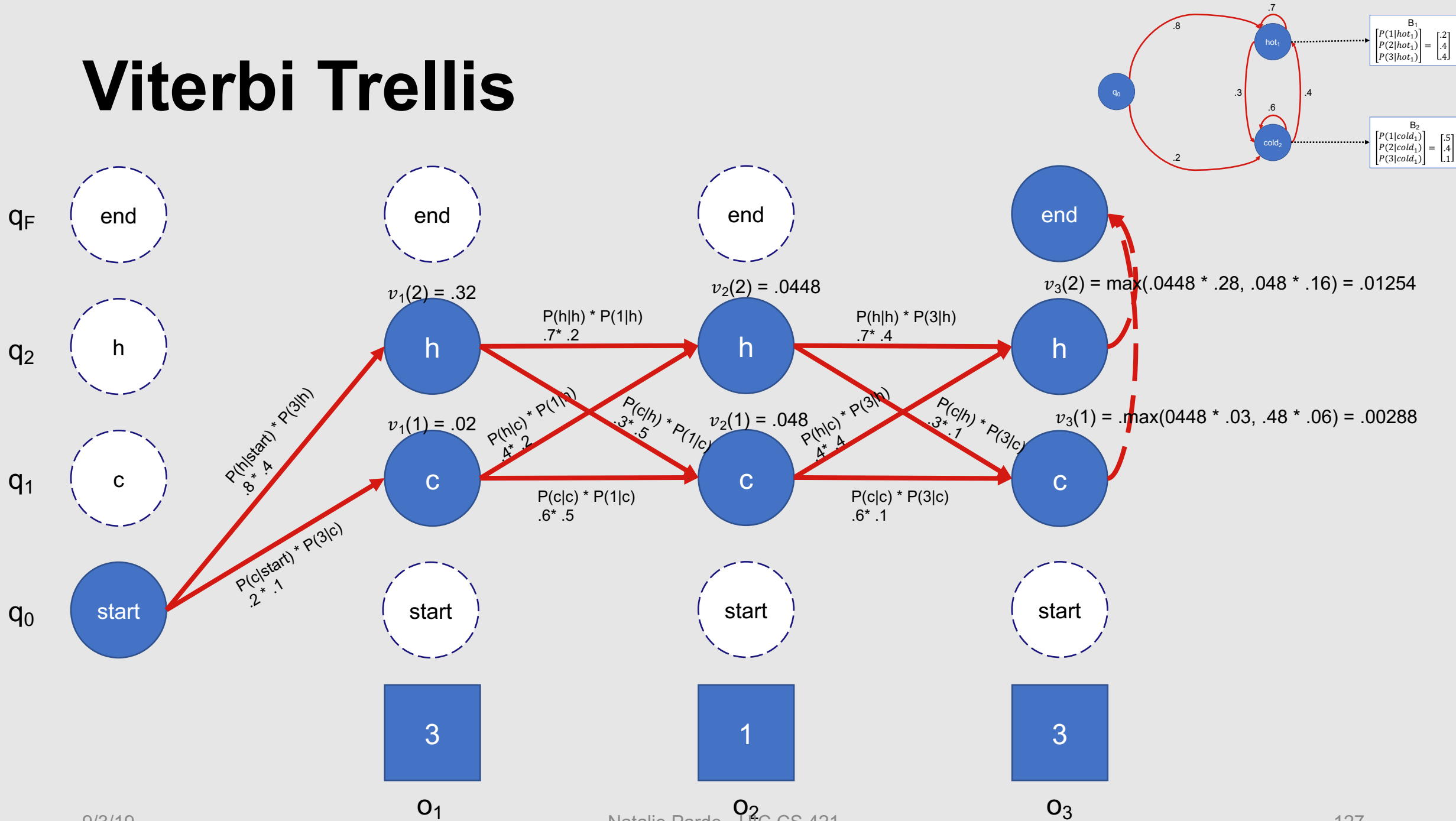
# Viterbi Trellis



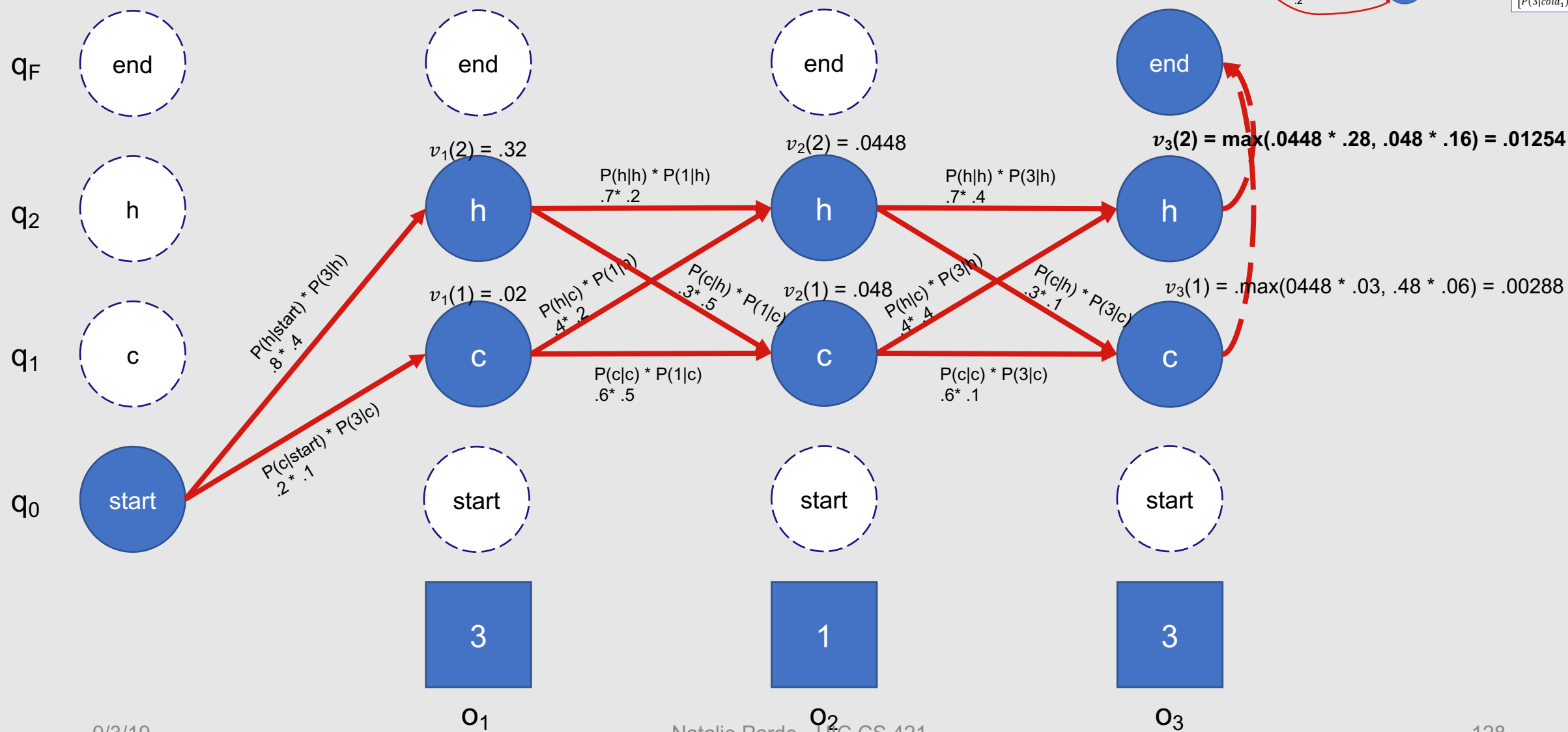
# Viterbi Trellis



# Viterbi Trellis

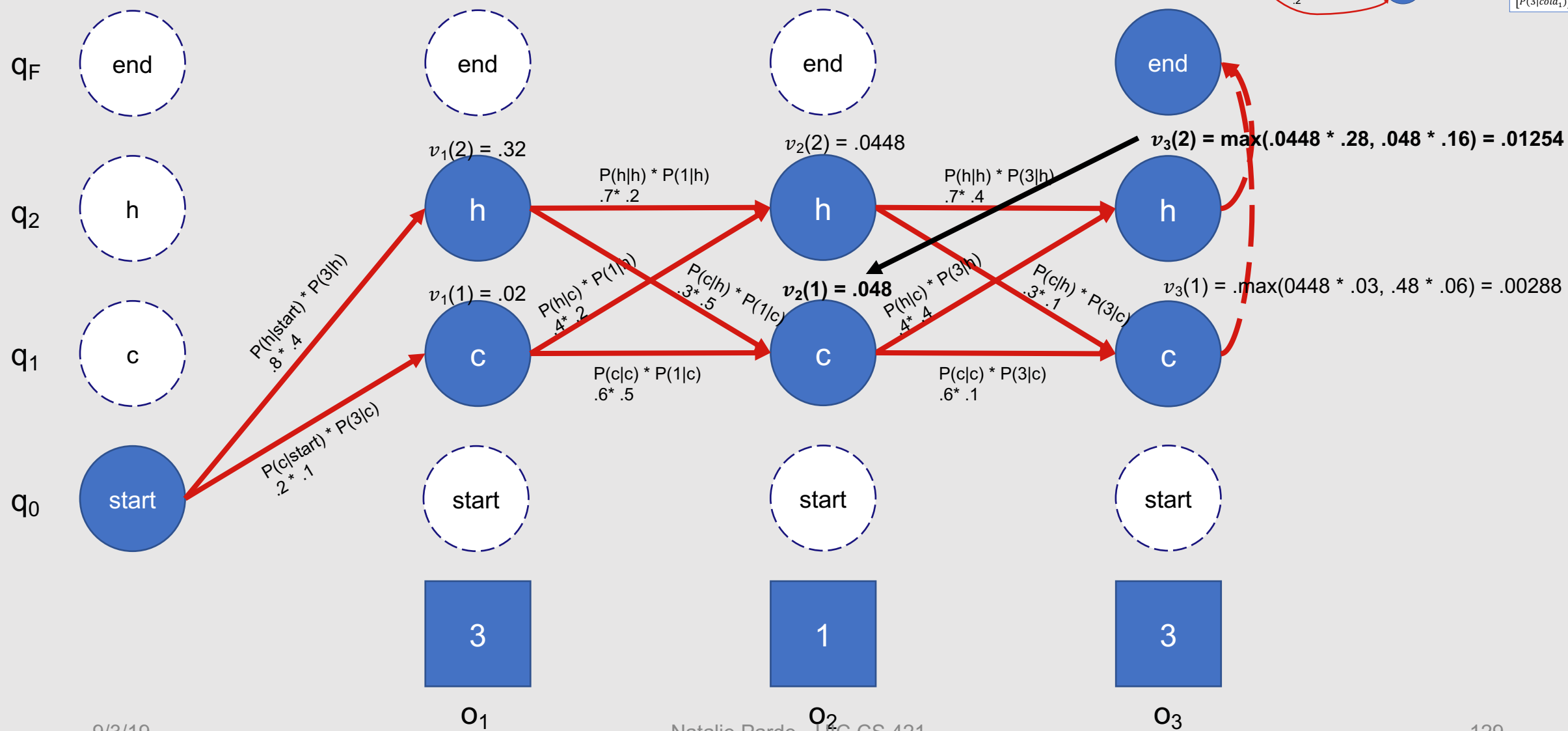


# Viterbi Backtrace

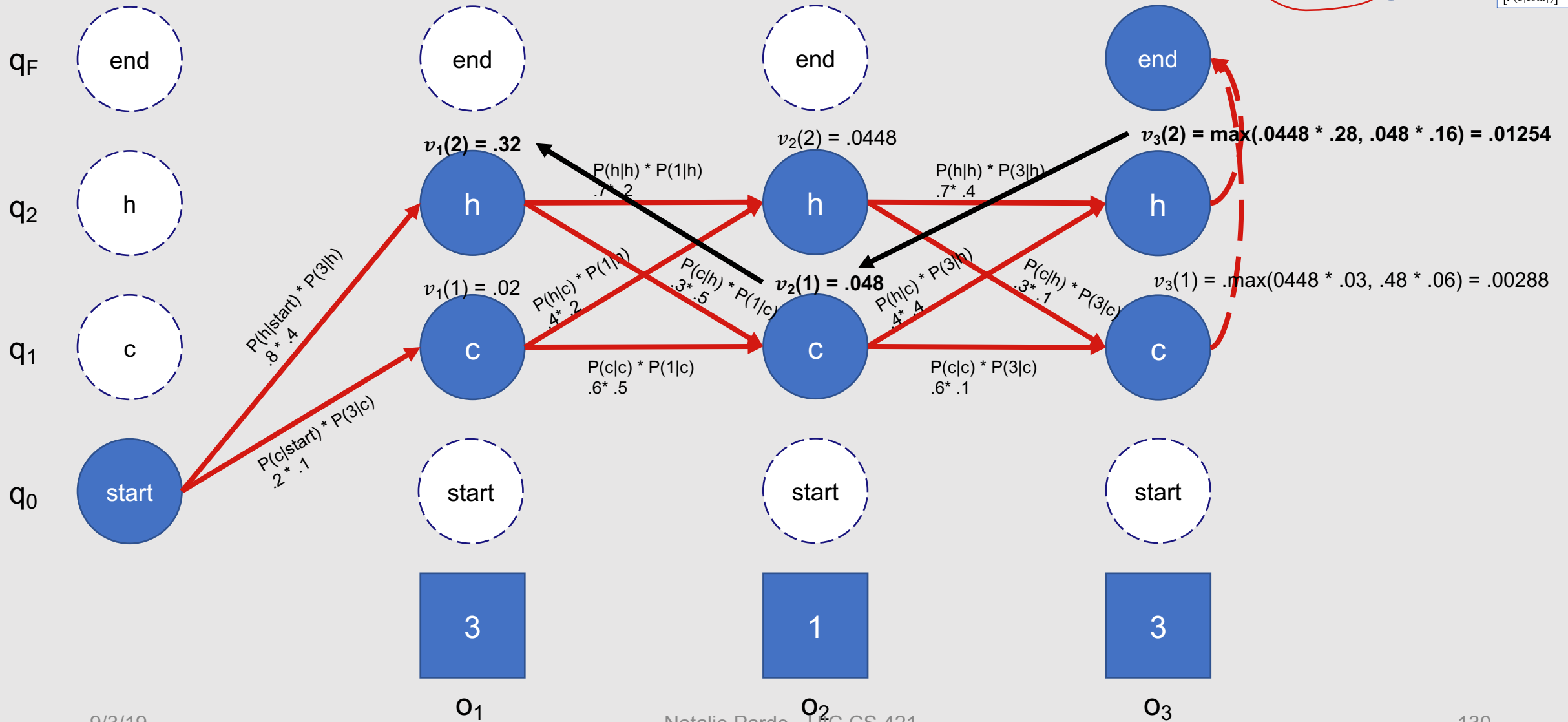




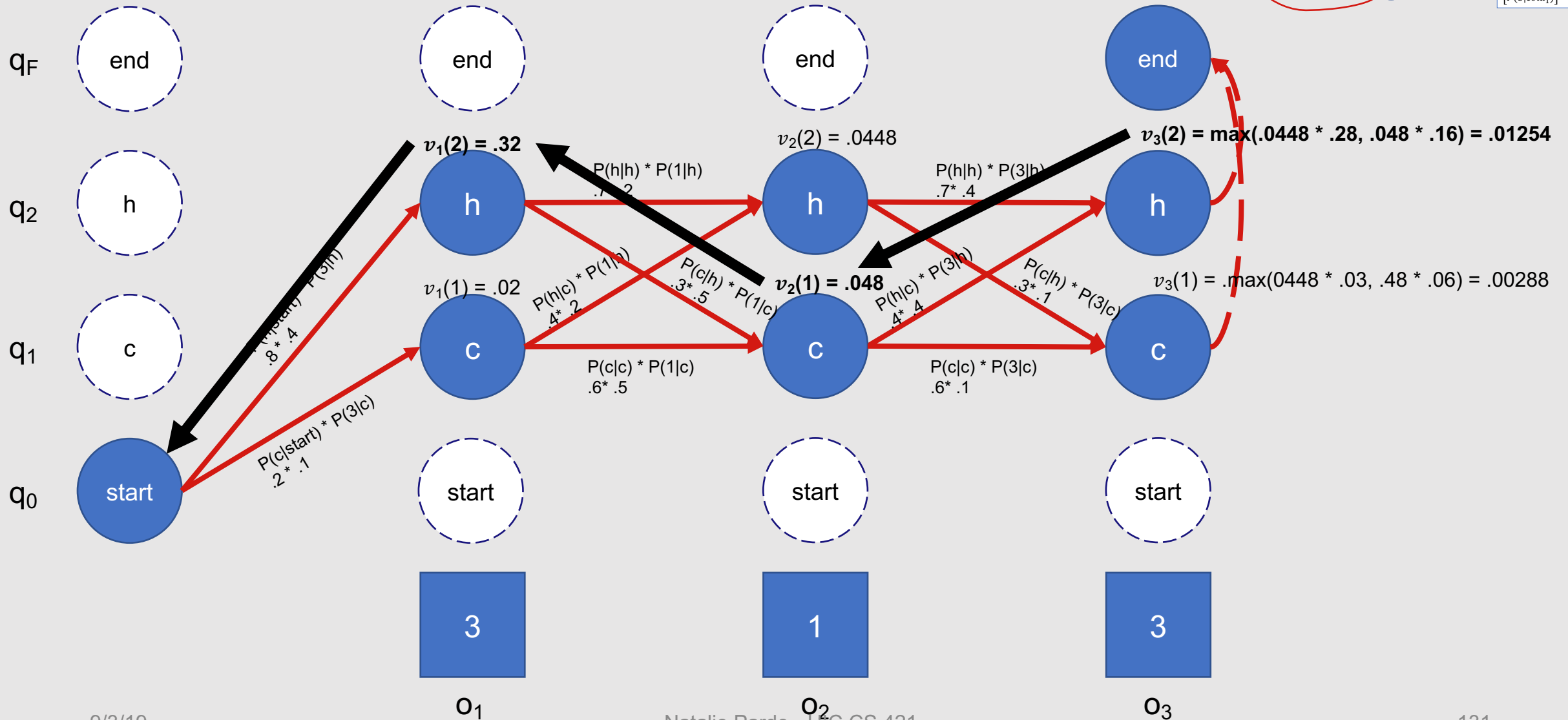
# Viterbi Backtrace



# Viterbi Backtrace



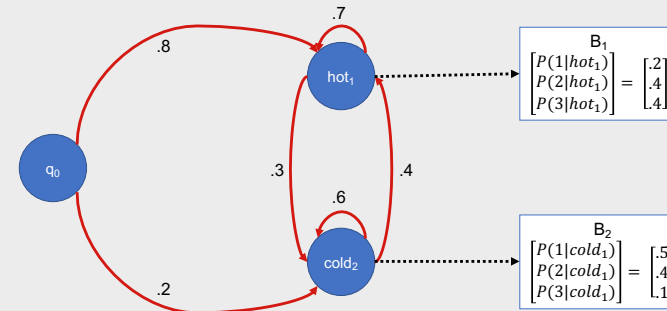
# Viterbi Backtrace



# Learning

- If we have a set of observations, can we learn the parameters (transition probabilities and observation likelihoods) directly?

3	1	3
2	1	3
3	3	3
3	2	2
1	1	2



# Forward-Backward Algorithm

---

- Special case of expectation-maximization (EM) algorithm
- Also known as the Baum-Welch algorithm
- Input:
  - Unlabeled sequence of observations,  $O$
  - Vocabulary of hidden states,  $Q$
- Example:
  - $O = \{3, 1, 3\}$
  - $Q = \{H, C\}$

# How would this work with observable Markov models?

- Run the model on observation sequence  $O$
- Since it's not hidden, we know which states we went through, and therefore which transitions and observations were used
- Given that information:
  - $B = \{b_j(o_t)\}$ : Since every state can only generate one observation symbol, observation likelihoods are all 1.0
  - $A = \{a_{ij}\}$ :  $a_{ij} = \frac{C(i \rightarrow j)}{\sum_{q \in Q} C(i \rightarrow q)}$

# Extending this intuition to HMMs....

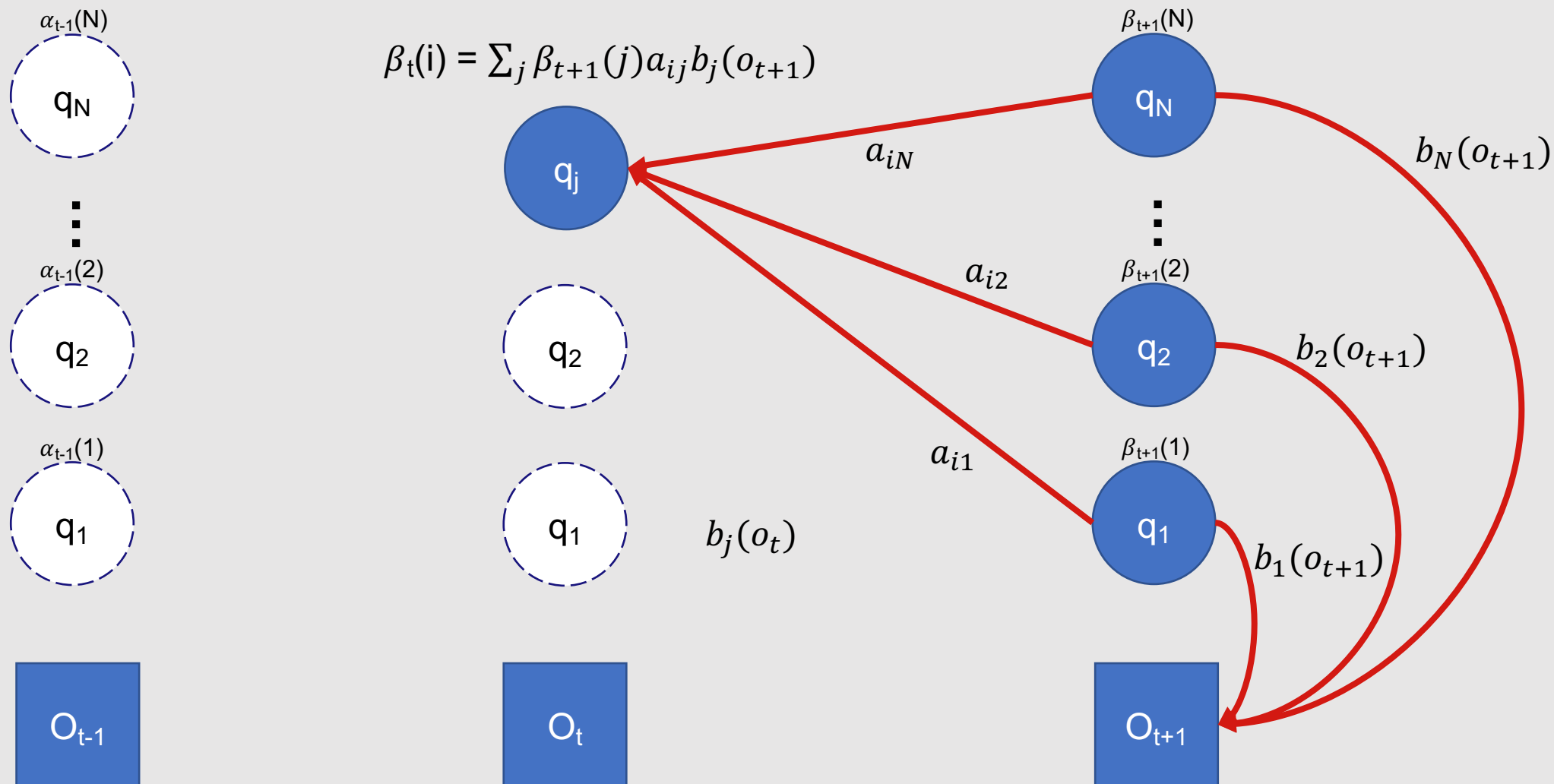
- We can't compute the counts directly from observed sequences
- Instead, we:
  - **Iteratively estimate the counts**
    - Start with base estimates for  $a_{ij}$  and  $b_j$ , and iteratively improve those estimates
  - **Get estimated probabilities** by:
    - Computing the forward probability for an observation
    - Dividing that probability mass among all the different paths that contributed to this forward probability

## Backward Algorithm

- We define the backward probability as follows:
  - $\beta_t(i) = P(o_{t+1}, o_{t+2}, \dots, o_T | q_t = i, \lambda)$
- This is the probability of generating partial observations from time  $t+1$  until the end of the sequence, given that the HMM is in state  $i$  at time  $t$



# Backward Step



# Re-Estimating $a_{ij}$

- We re-estimate  $a_{ij}$  as follows:
  - $a_{ij}'$  = expected number of transitions from state  $i$  to state  $j$ , divided by expected number of transitions from state  $i$
- More formally, we first define  $\xi$  as the probability of being in state  $i$  at time  $t$  and state  $j$  at time  $t+1$ , given the observation sequence and the HMM:
  - $\xi(i, j) = P(q_t = i, q_{t+1} = j | O, \lambda)$
- To compute  $\xi$ , we first define not-quite- $\xi$  as a very similar probability with different conditioning of  $O$ :
  - not-quite- $\xi(i, j) = P(q_t = i, q_{t+1} = j, O | \lambda)$

# Re-Estimating $a_{ij}$

- From not-quite-  $\xi$ , we can use Bayes rule ( $P(X|Y, Z) = \frac{P(X, Y|Z)}{P(Y|Z)}$ ) to compute  $\xi$ :
  - $\xi_t(i, j) = \frac{\text{not-quite-}\xi_t(i, j)}{P(O|\lambda)}$
- This ends up being equivalent to:
  - $\xi_t(i, j) = \frac{\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{\alpha_T(N)}$
- Finally, we can use this then to re-estimate  $a_{ij}$ :
  - $a'_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i, j)}{\sum_{t=1}^{T-1} \sum_{j=1}^N \xi_t(i, j)}$

# Re-Estimating Observation Likelihood

- We re-estimate  $b_j$  as follows:
  - $b'_j(v_k)$  = expected number of times in state  $j$  and observing vocabulary symbol  $v_k$ , divided by the expected number of times in state  $j$
- Letting  $\gamma_t(j)$  represent the probability of being in state  $j$  at time  $t$ , we can formally define the re-estimation as:
  - $$b'_j(v_k) = \frac{\sum_{t=1}^T \text{s.t. } o_t = v_k \gamma_t(j)}{\sum_{t=1}^T \gamma_t(j)}$$

# Forward-Backward Algorithm

initialize A and B

iterate until convergence:

  # Expectation Step

  compute  $\gamma_t(j)$  for all t and j

  compute  $\xi_t(i, j)$  for all t, i, and j

  # Maximization Step

  recompute  $a_{ij}$

  recompute  $b_j(v_k)$

# Summary: Hidden Markov Models

- HMMs are probabilistic generative models for sequences
- They make predictions based on underlying hidden states
- Three fundamental HMM problems include:
  - Computing the likelihood of a sequence of observations
  - Determining the best sequence of hidden states for an observed sequence
  - Learning HMM parameters given an observation sequence and a set of hidden states
- Observation likelihood can be computed using the forward algorithm
- Sequences of hidden states can be decoded using the Viterbi algorithm
- HMM parameters can be learned using the forward-backward algorithm