Hidden Markov Models and Language Modeling with N-Grams

Natalie Parde

UIC CS 421

Language is inherently contextual.

- Words or characters in language are dependent upon one another!
- **Sequence modeling** allows us to make use of sequential information in language
- There are many ways to implement sequence models, including but not limited to:
  - **Hidden Markov models**
  - Neural networks

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?

- Other types of classification: Often classify entire text samples into discrete, predefined groups

Spam

Not Spam

ARE YOU LOOKING FOR WAYS TO ORGANIZE YOUR **INBOX**????

Check out our product now!!!!!!! Follow the link here to take your first step towards the ibnox of your dreams~~~

In these scenarios, models assume that the individual datapoints being classified are disconnected and independent.

- Many NLP problems do not satisfy this assumption
- Often problems involve:
  - Interconnected decisions
  - Each of which are mutually dependent
  - Each of which resolve different ambiguities
- For these problems, different learning and inference techniques are needed!

Sequence Labeling

- One example: **sequence labeling** tasks.
- Objective: Find the label for the next item, based on the labels of other items in the sequence.

verb determiner

Give me a **break**!

pronoun noun

verb noun

Did the window **break**?

determiner verb

Example Sequence Labeling Applications

- Named entity recognition
- Semantic role labeling

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 hail-damaged **Honda Accord**.

instrument

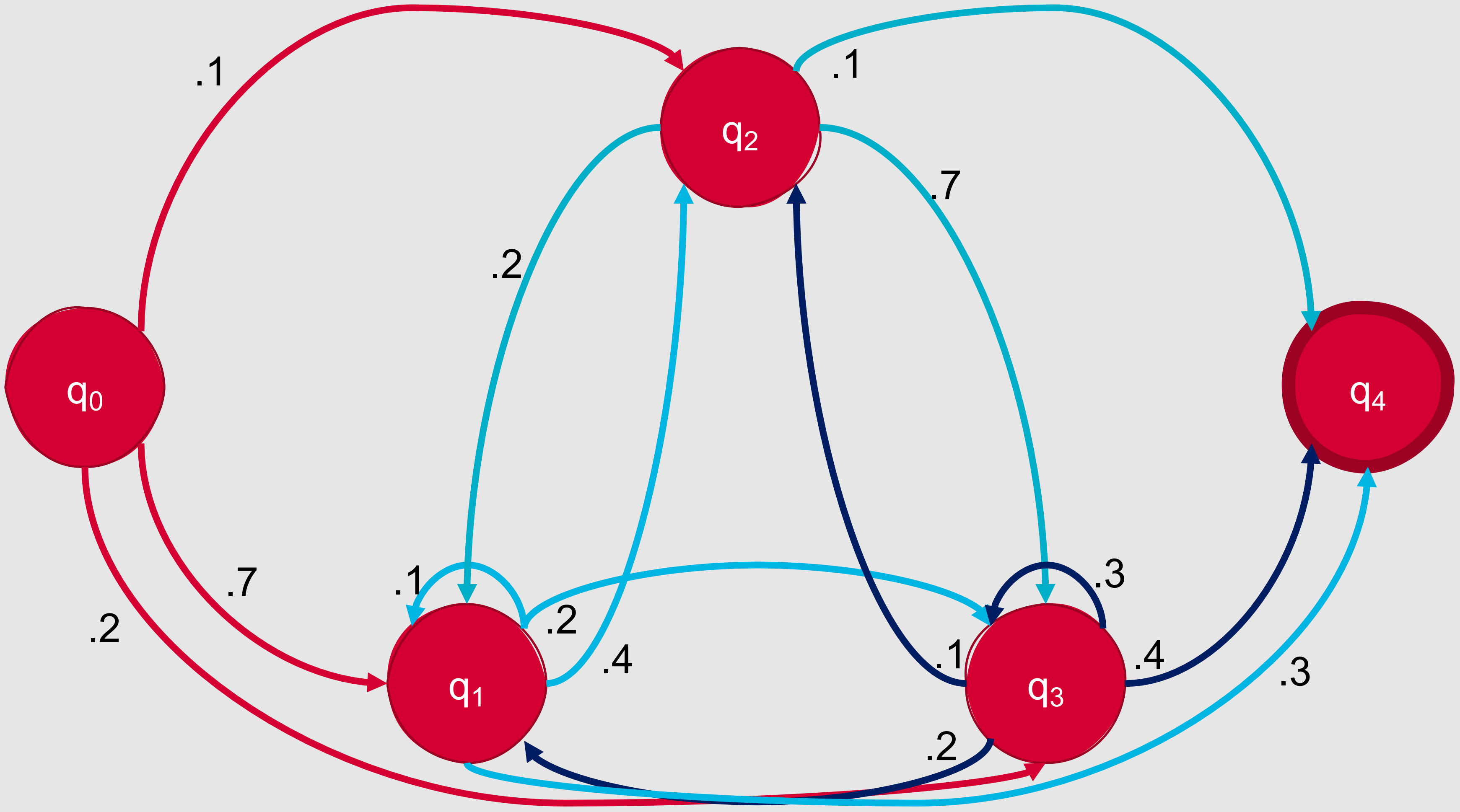
Probabilistic Sequence Models

- Resolving uncertainties involves multiple, interdependent classifications
- Two standard models:
  - Hidden Markov Models
  - Conditional Random Fields

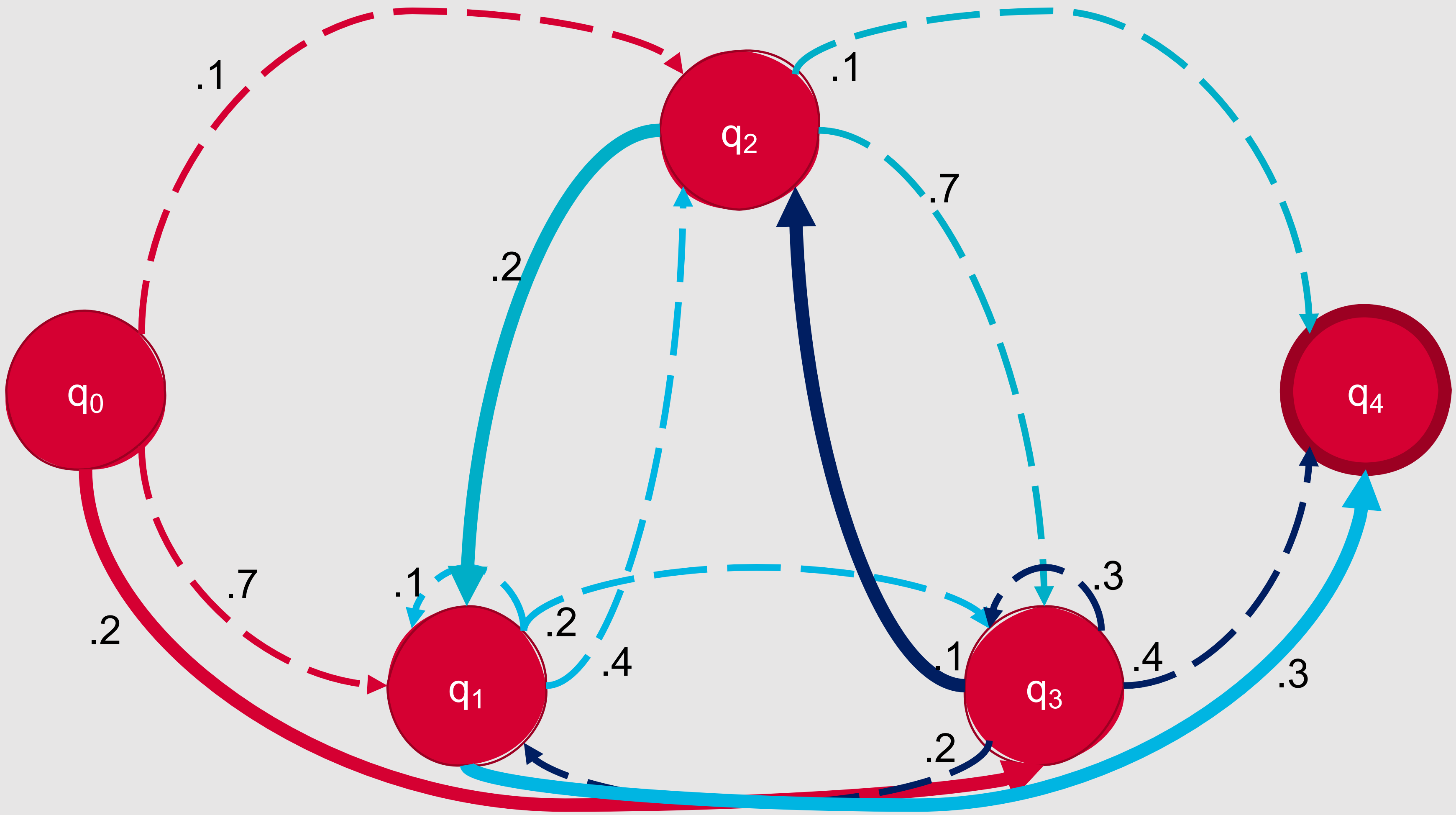
What are Markov Models?

- **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

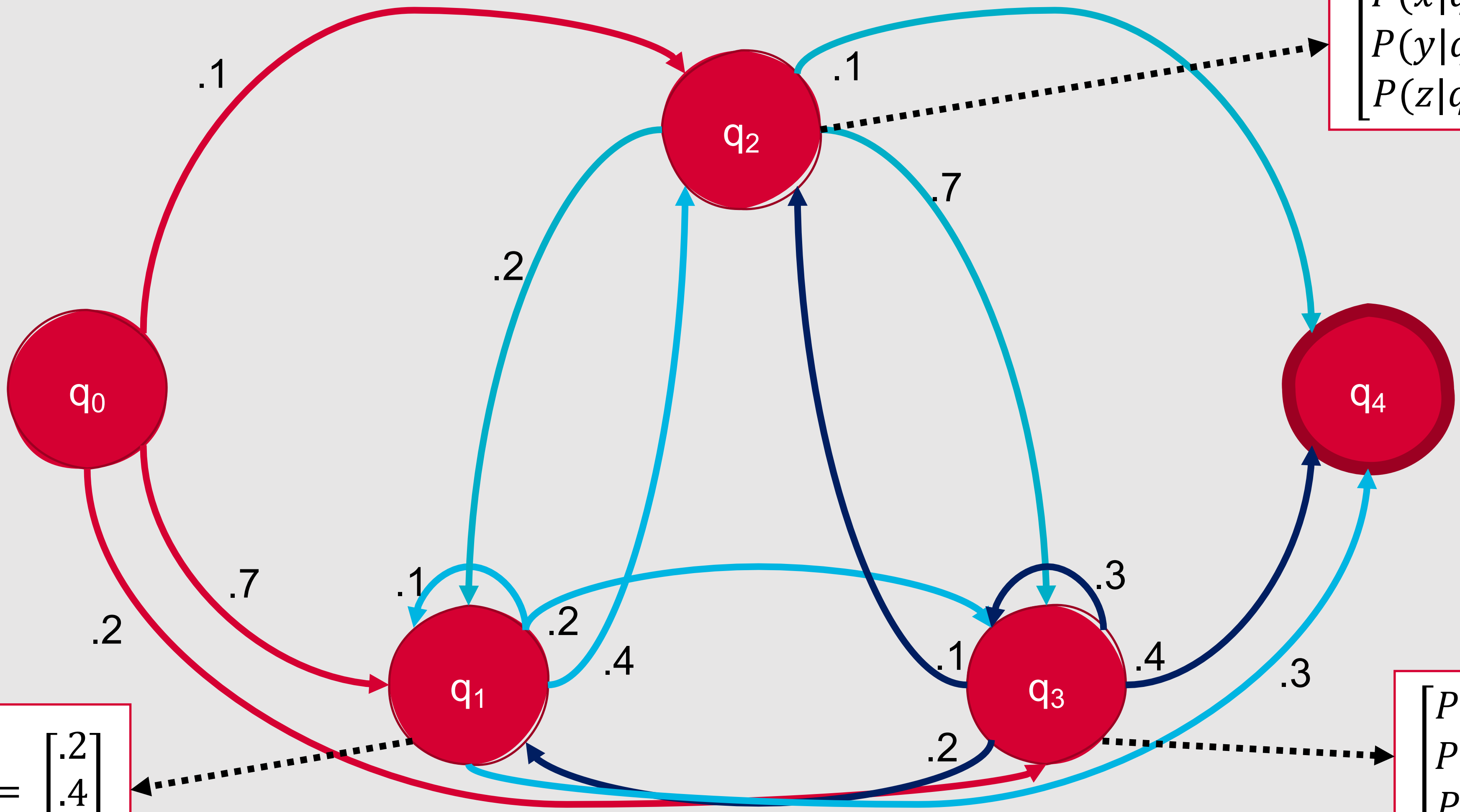


$P(q_3\ q_2\ q_1\ q_4)$
$= .2 * .1 * .2 * .3$
$= .0012$

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



$$\begin{bmatrix} P(x|q_2) \\ P(y|q_2) \\ P(z|q_2) \end{bmatrix} = \begin{bmatrix} .1 \\ .4 \\ .5 \end{bmatrix}$$

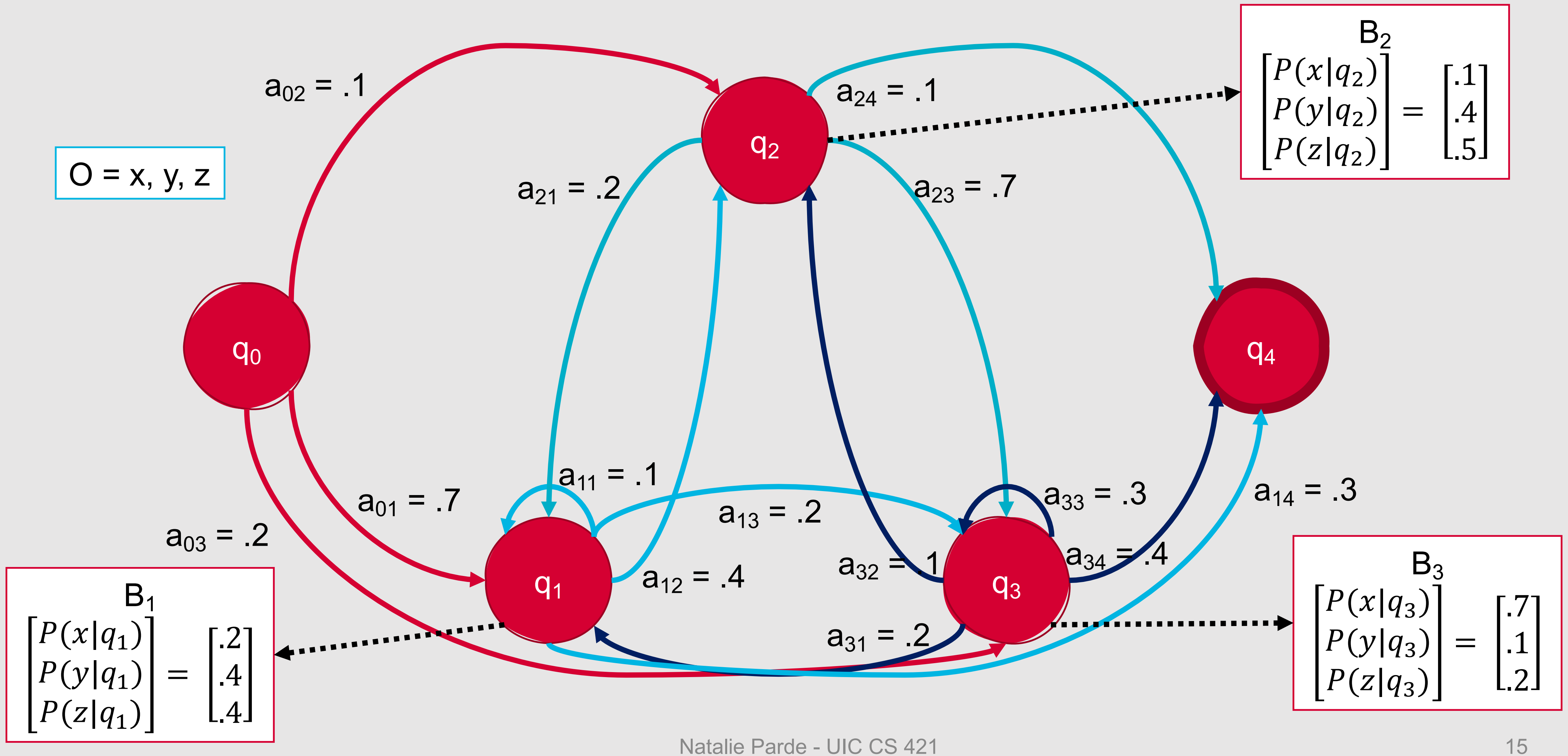
$$\begin{bmatrix} P(x|q_1) \\ P(y|q_1) \\ P(z|q_1) \end{bmatrix} = \begin{bmatrix} .2 \\ .4 \\ .4 \end{bmatrix}$$

$$\begin{bmatrix} P(x|q_3) \\ P(y|q_3) \\ P(z|q_3) \end{bmatrix} = \begin{bmatrix} .7 \\ .1 \\ .2 \end{bmatrix}$$

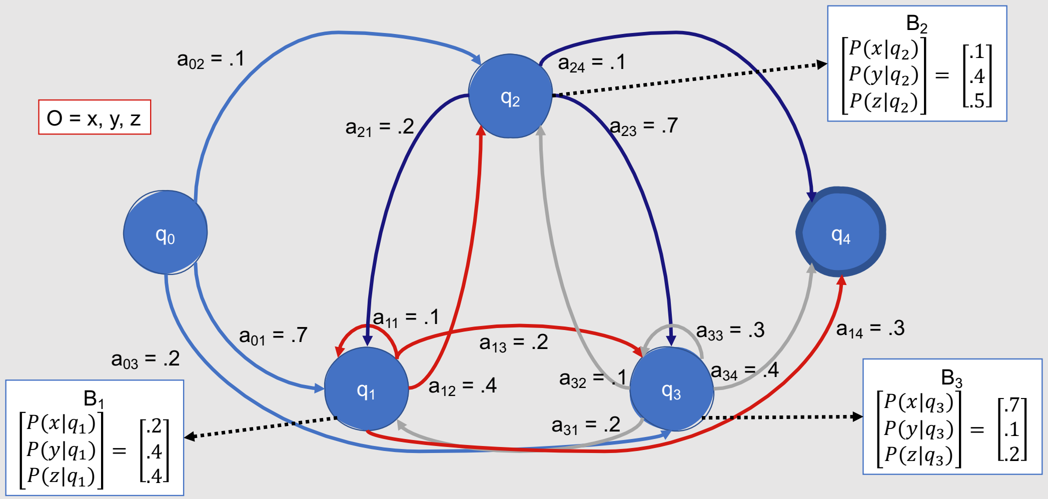
Formal Definition

- 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, …, 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

Practical Applications of HMMs

- One intuitive application: Text generation
- More generally, you can generate a sequence of T observations: $O = o_1, o_2, \ldots, o_T$

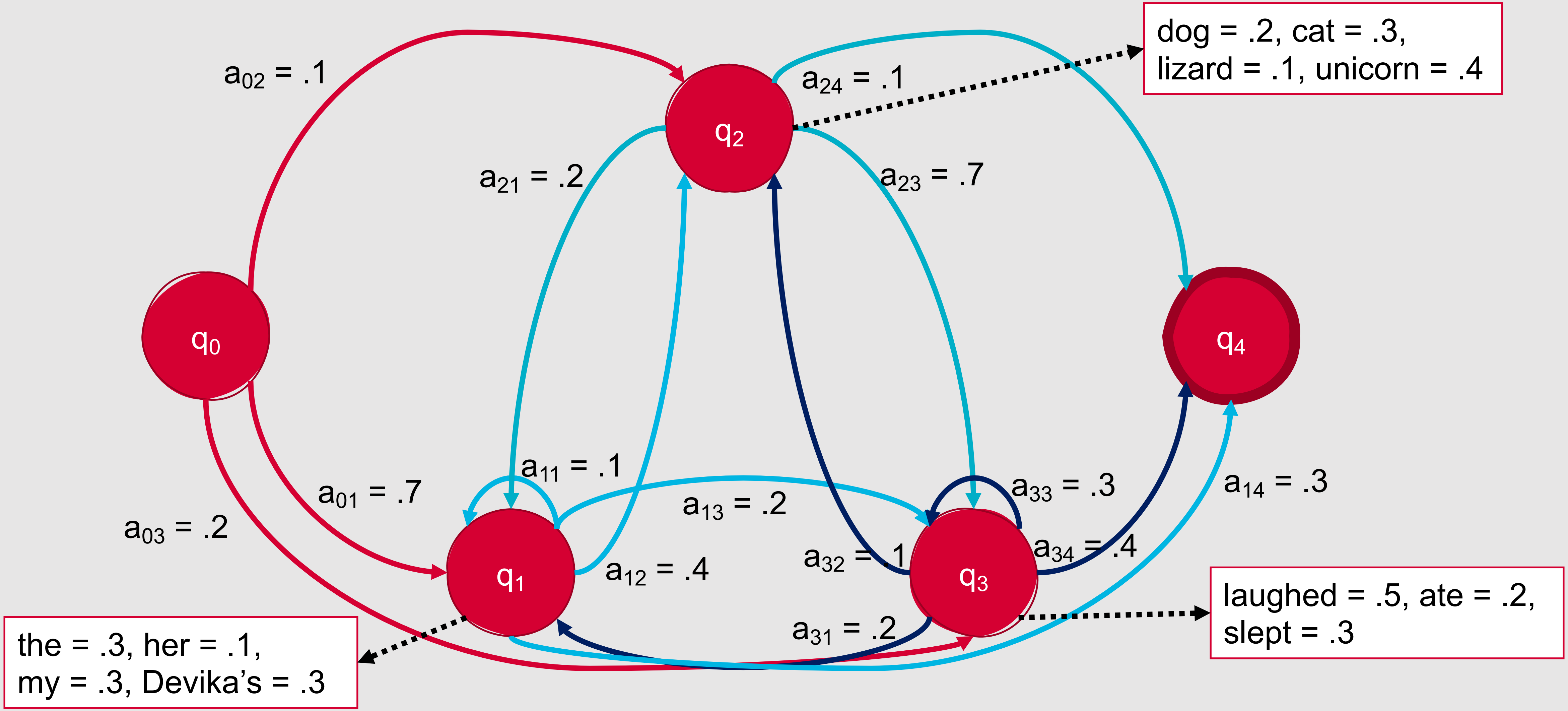
Begin in the start state

For t in [0, …, T]:

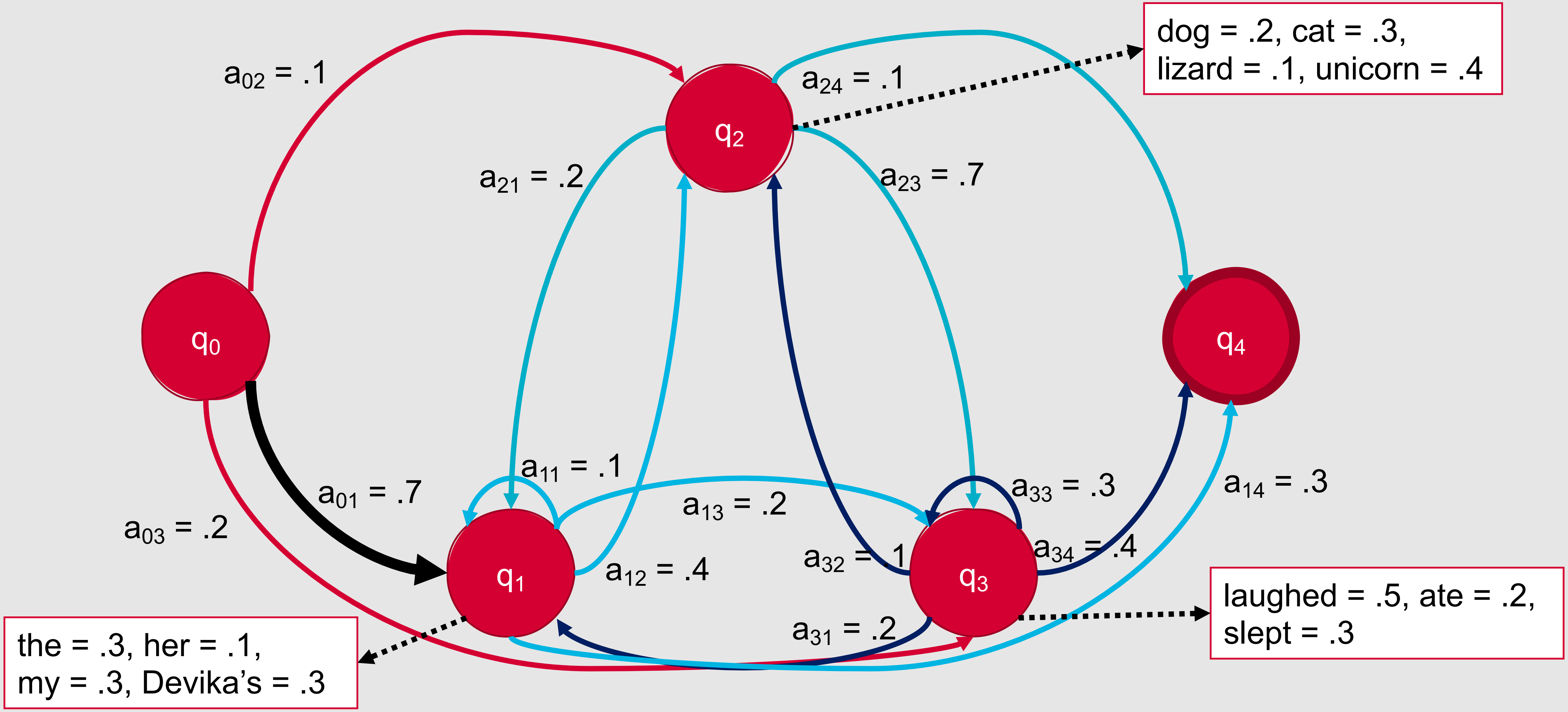
Randomly select a new state based on the transition distribution for the current state

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

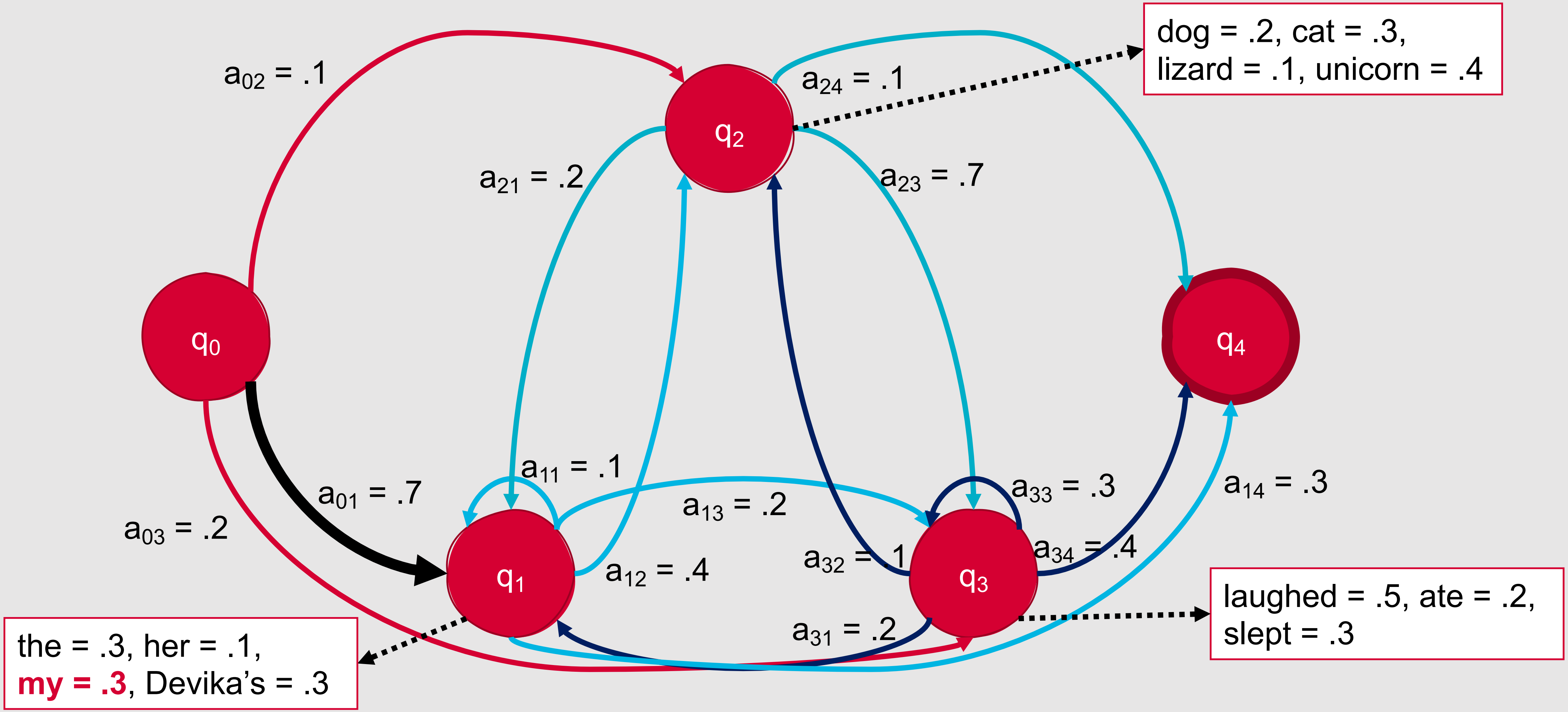
Sample Text Generation



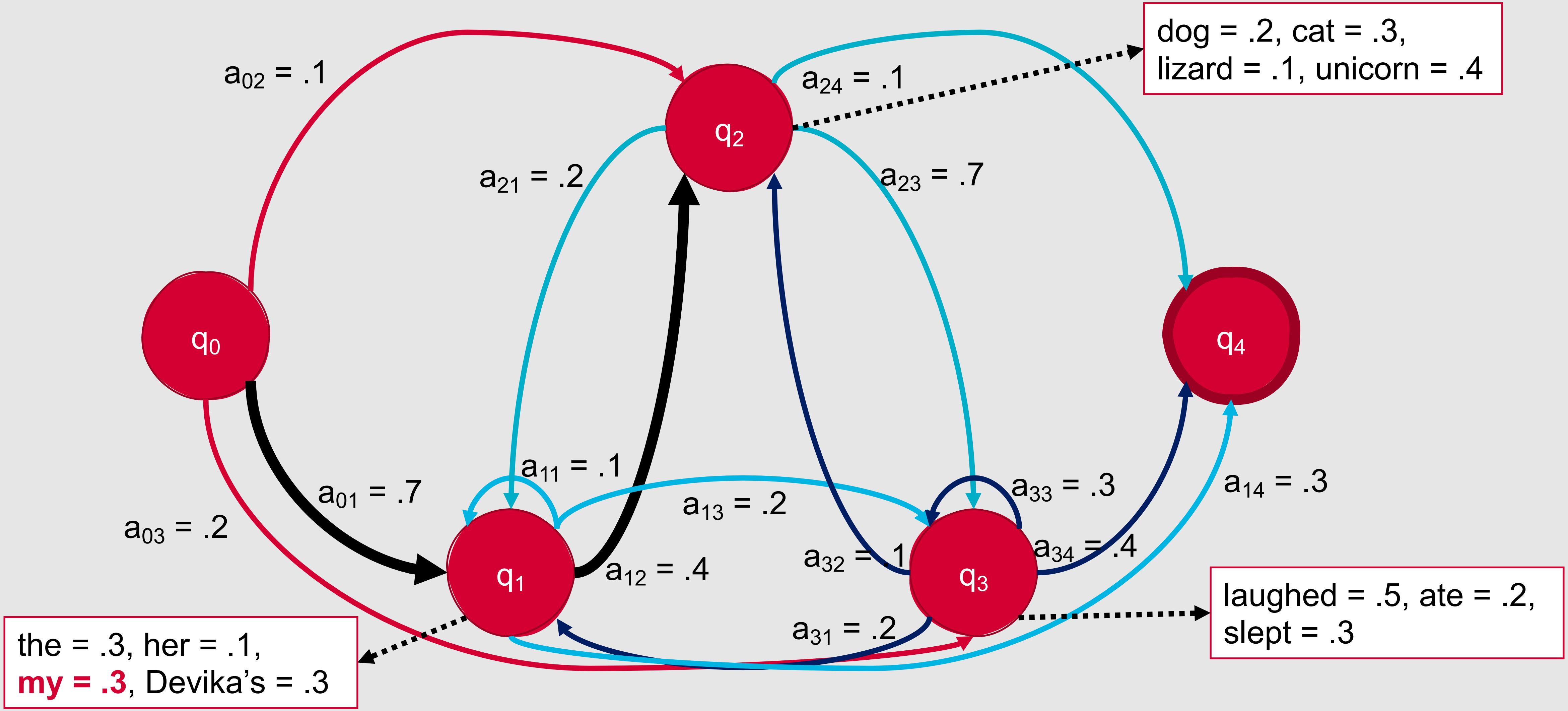
Sample Text Generation



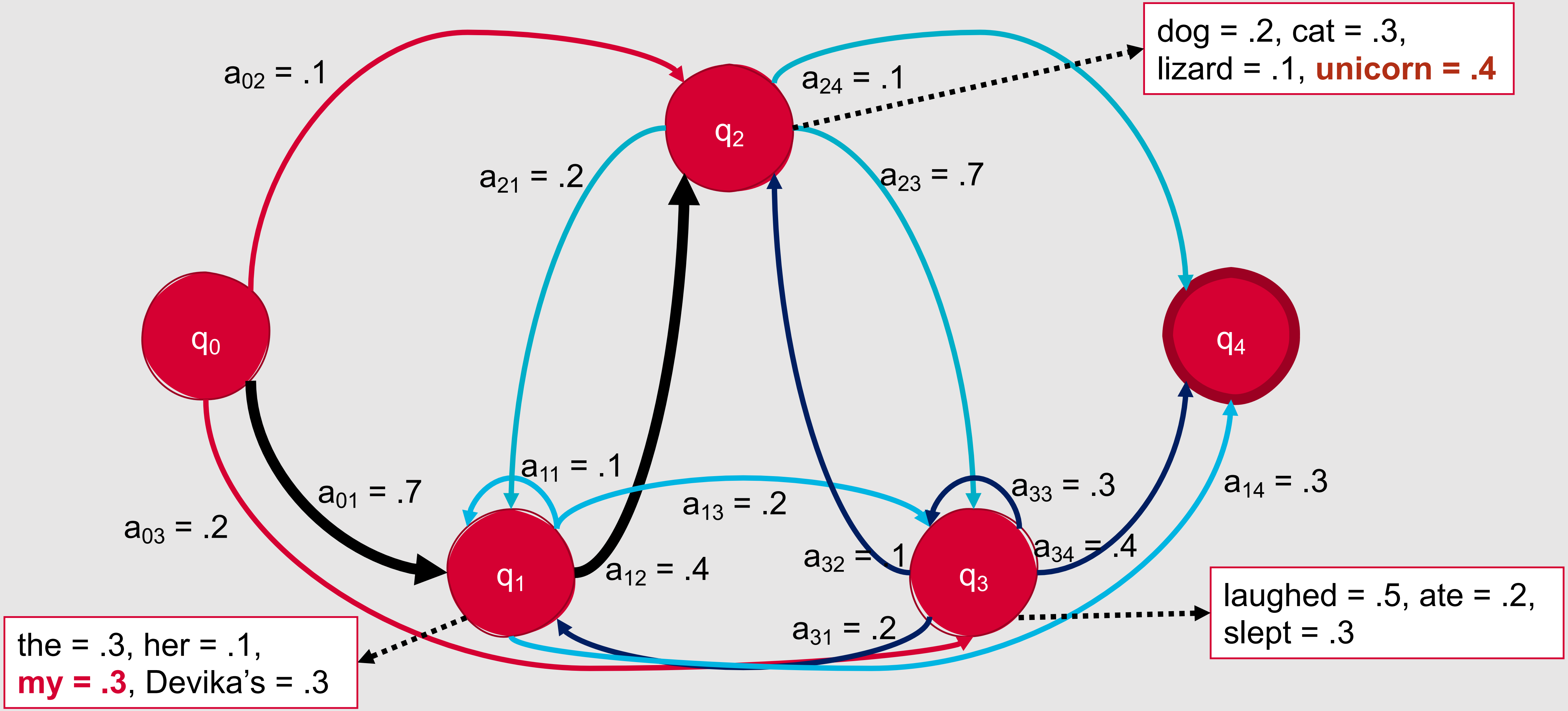
Sample Text Generation



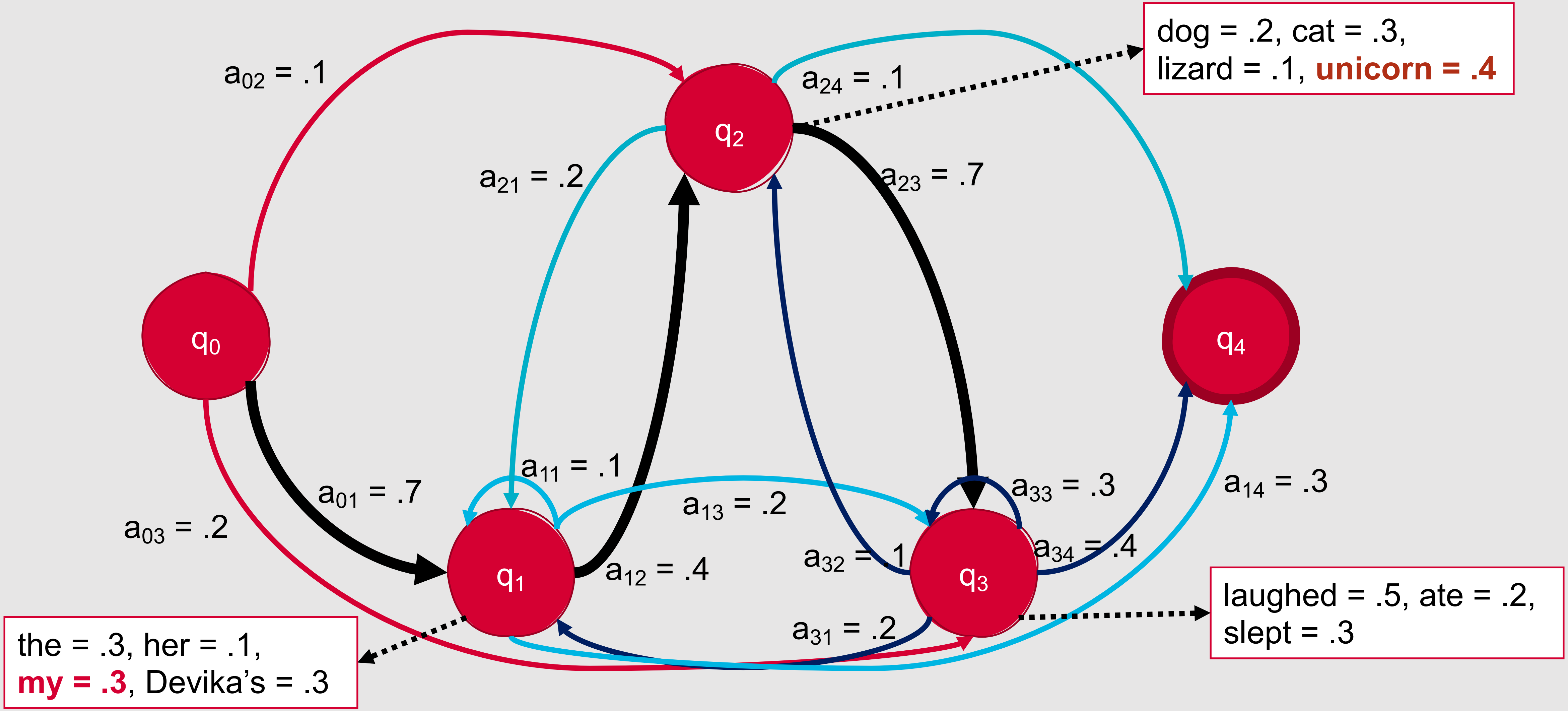
Sample Text Generation



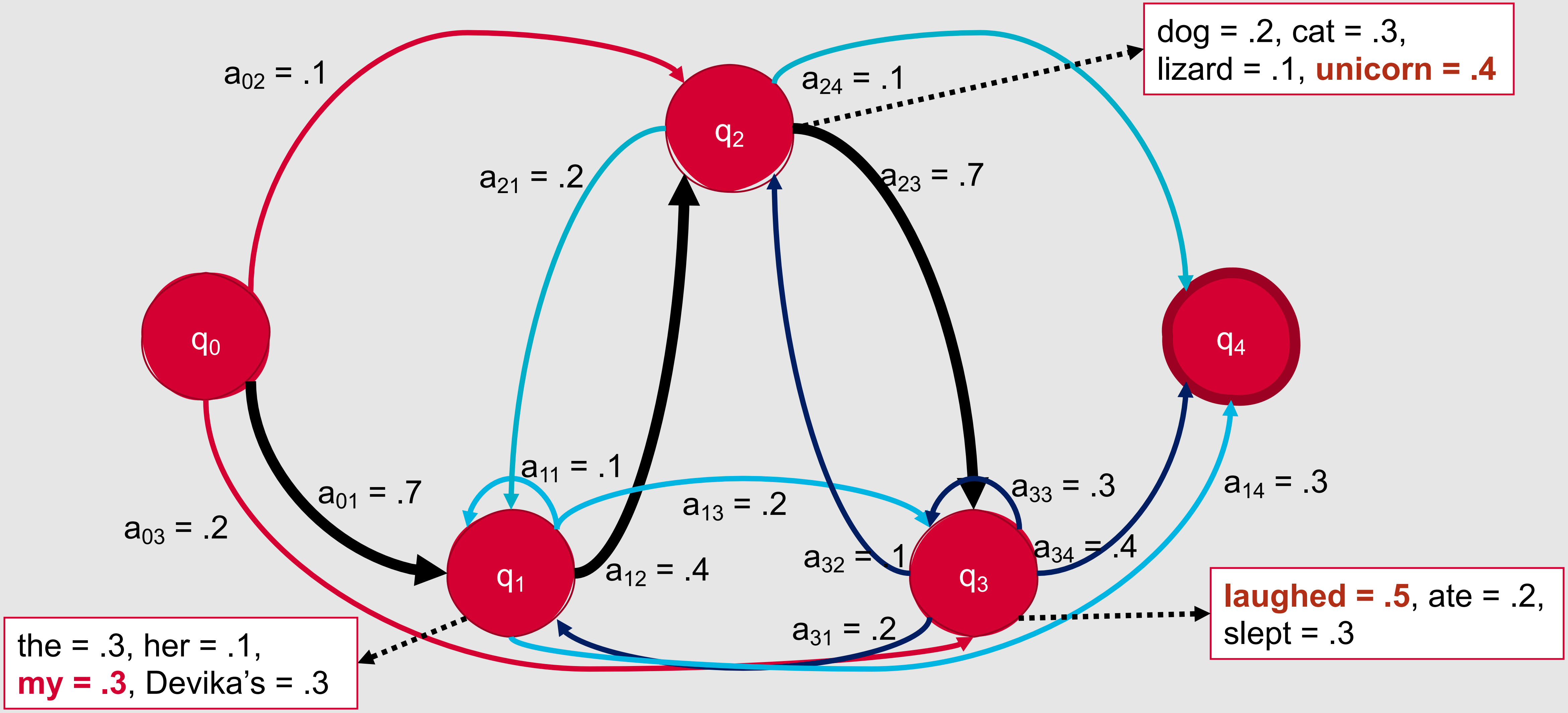
Sample Text Generation



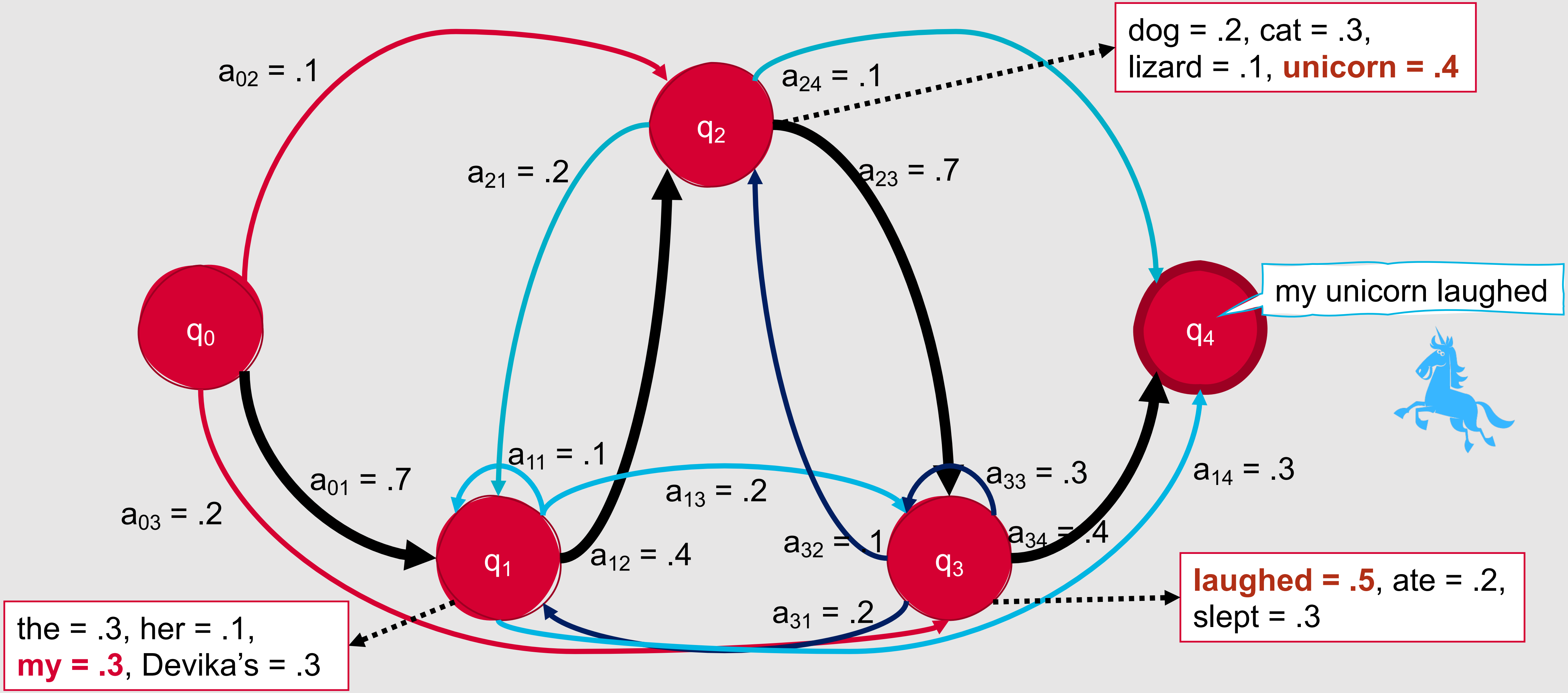
Sample Text Generation



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?

One way to compute observation likelihood is by using forward probabilities.

Observation Likelihood

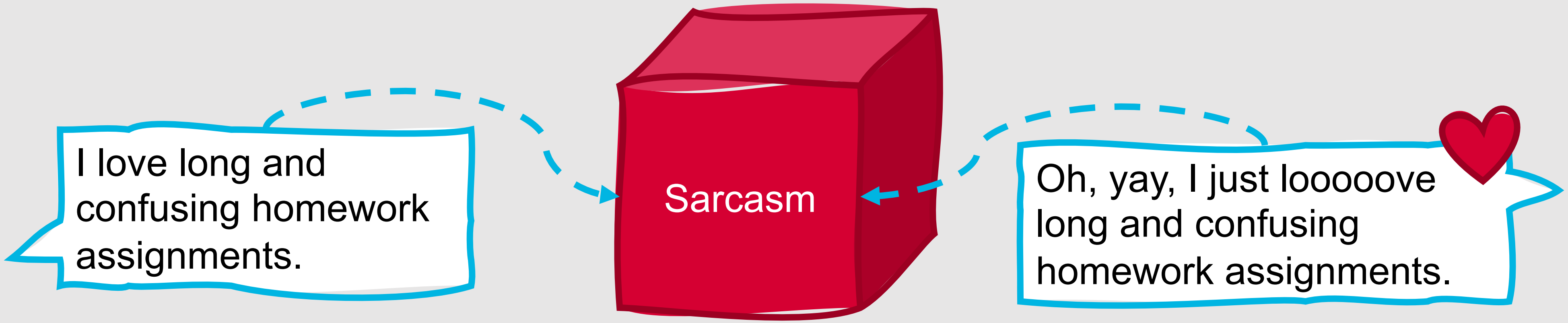
- Given a sequence of observations and an HMM, what is the probability that this sequence was generated by the model?
- Useful for two tasks:
  - Sequence classification
  - Selecting the most likely sequence

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?

Most Likely Sequence

- Of two or more possible sequences, which one was most likely generated by a given HMM?



How can we compute the observation likelihood?

- Naïve Solution:
  - Consider all possible state sequences, Q, of length T that the model, λ, 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!
  - $O(TN^T)$

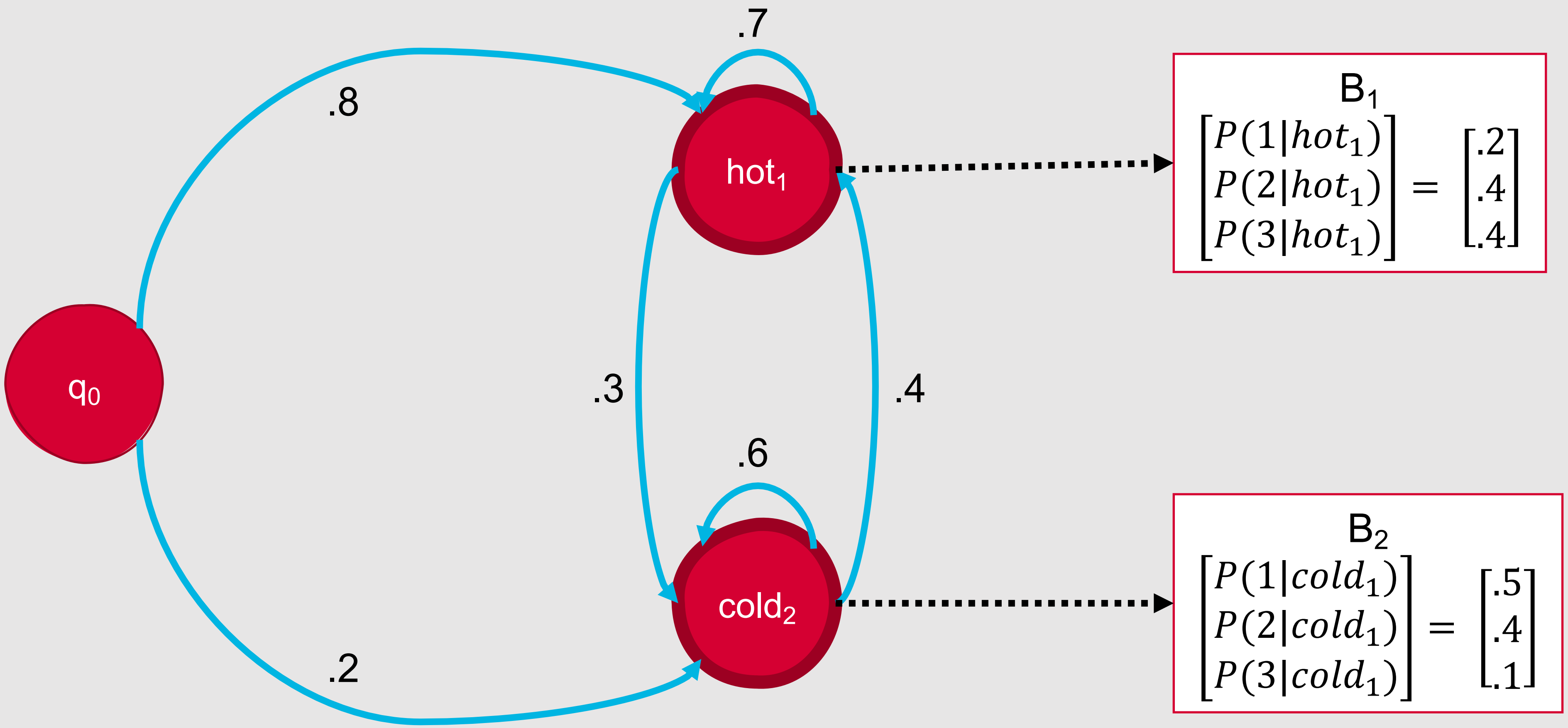
How can we compute the observation likelihood?

- 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 official records of the weather in Baltimore for the summer of 2007, although you know some key weather patterns, which you're representing using HMMs
- 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 focus on a three-day sequence:
  - Day 1: 3 ice cream cones
  - Day 2: 1 ice cream cone
  - Day 3: 3 ice cream cones

Current Leading HMM



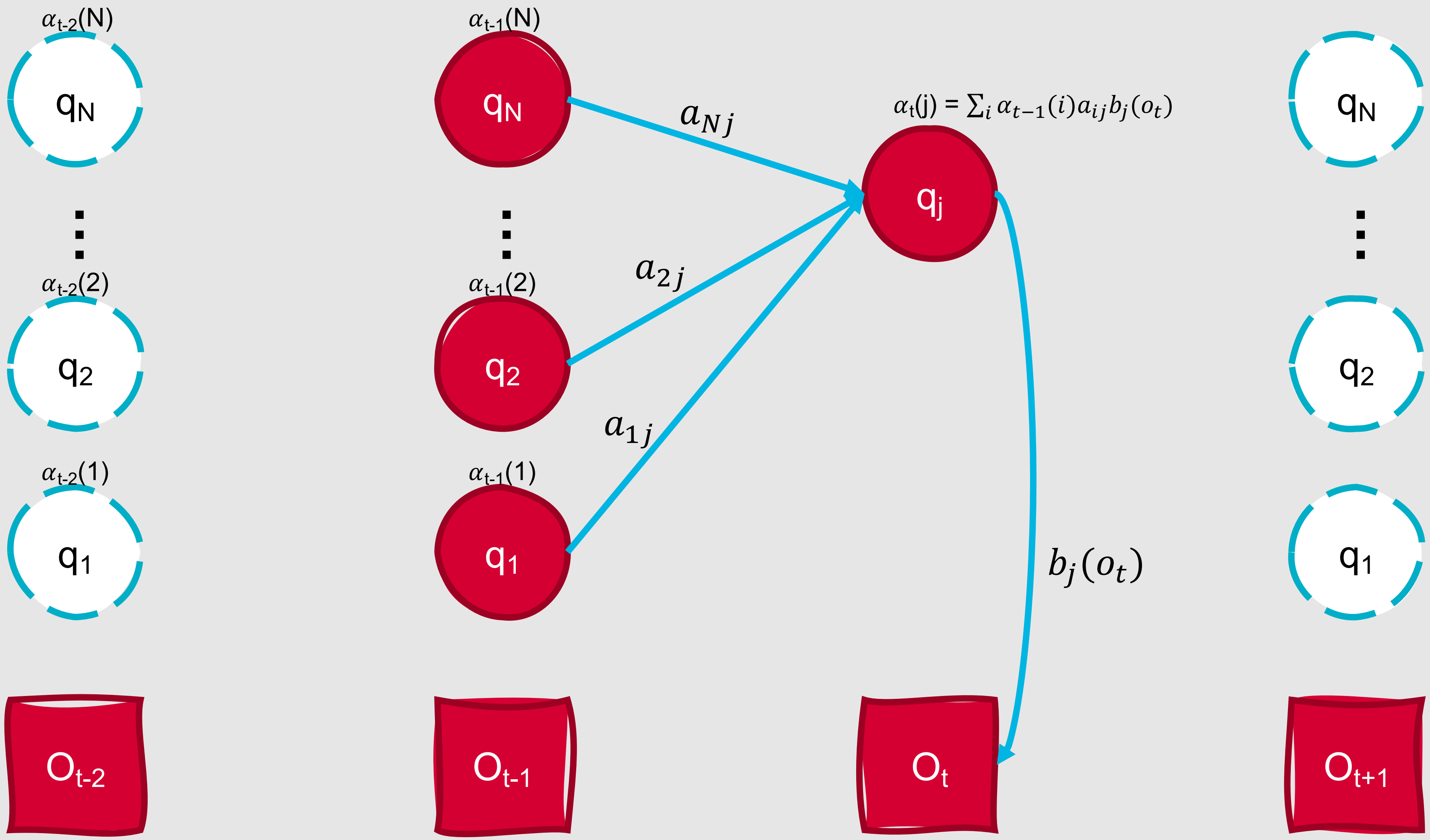
How do you compute your forward probabilities?

- Let $\alpha_i(j)$ be the probability of being in state *j* after seeing the first *t* observations, given your HMM λ
- $\alpha_i(j)$ 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*

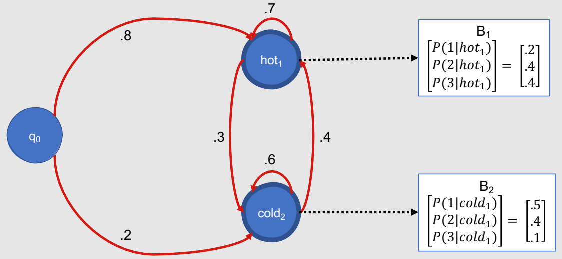
Formal Algorithm

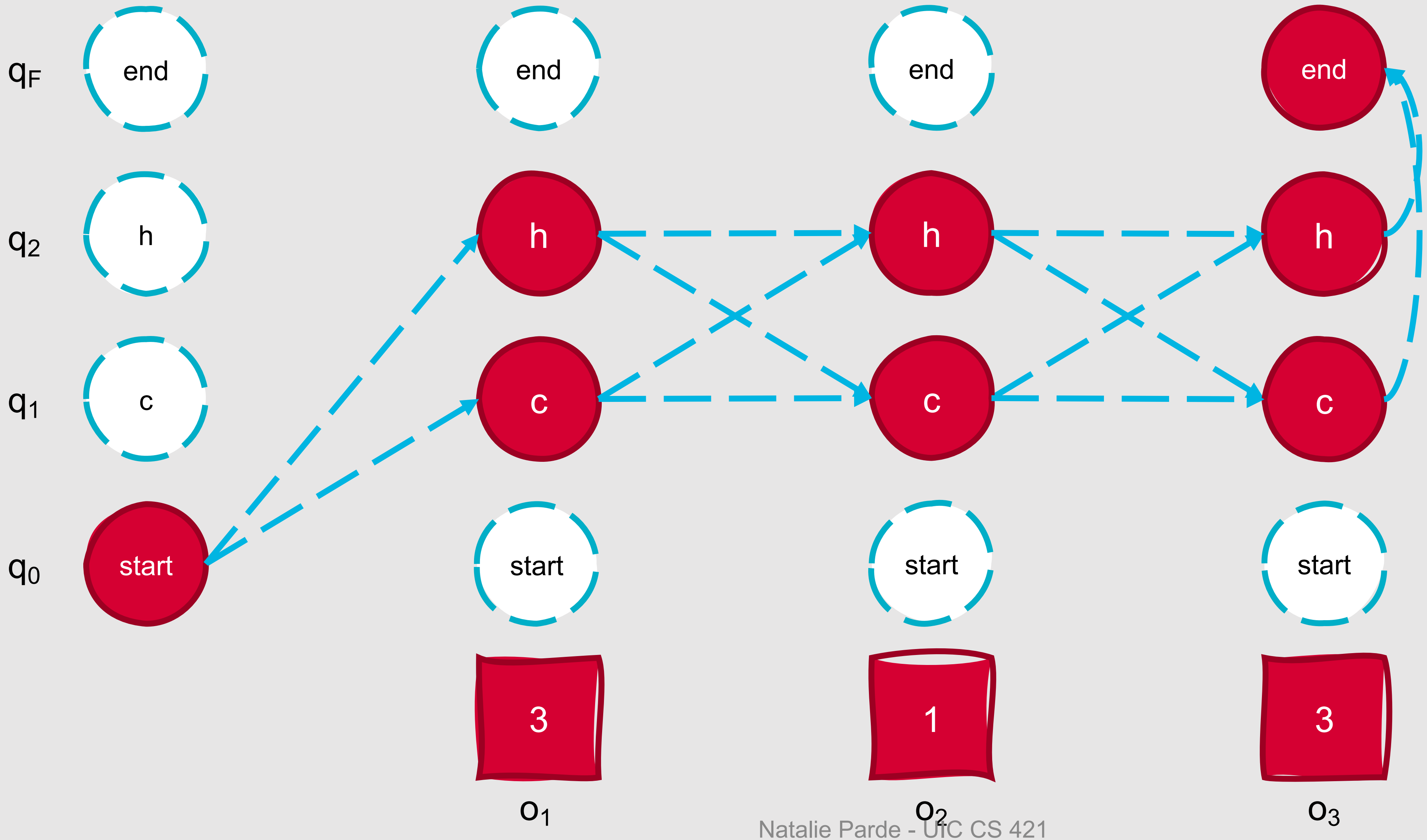
```
create a probability matrix forward[N+2,T]

for each state q in [1, …, N] do:
     forward[q,1] ← a_{0,q} * b_q(o_1)
for each time step t from 2 to T do:
     for each state q in [1, …, N] do:
          forward[q,t] ← Σ_{q'=1}^{N} forward[q',t−1] * a_{q',q} * b_q(o_t)
forwardprob ← Σ_{q=1}^{N} forward[q,T]
```

Forward Step

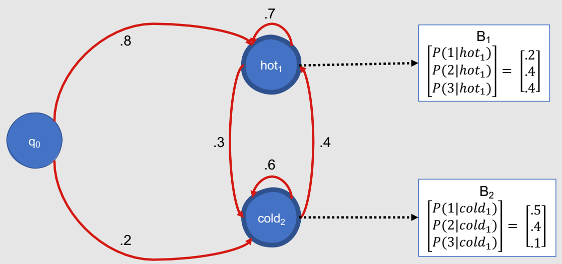


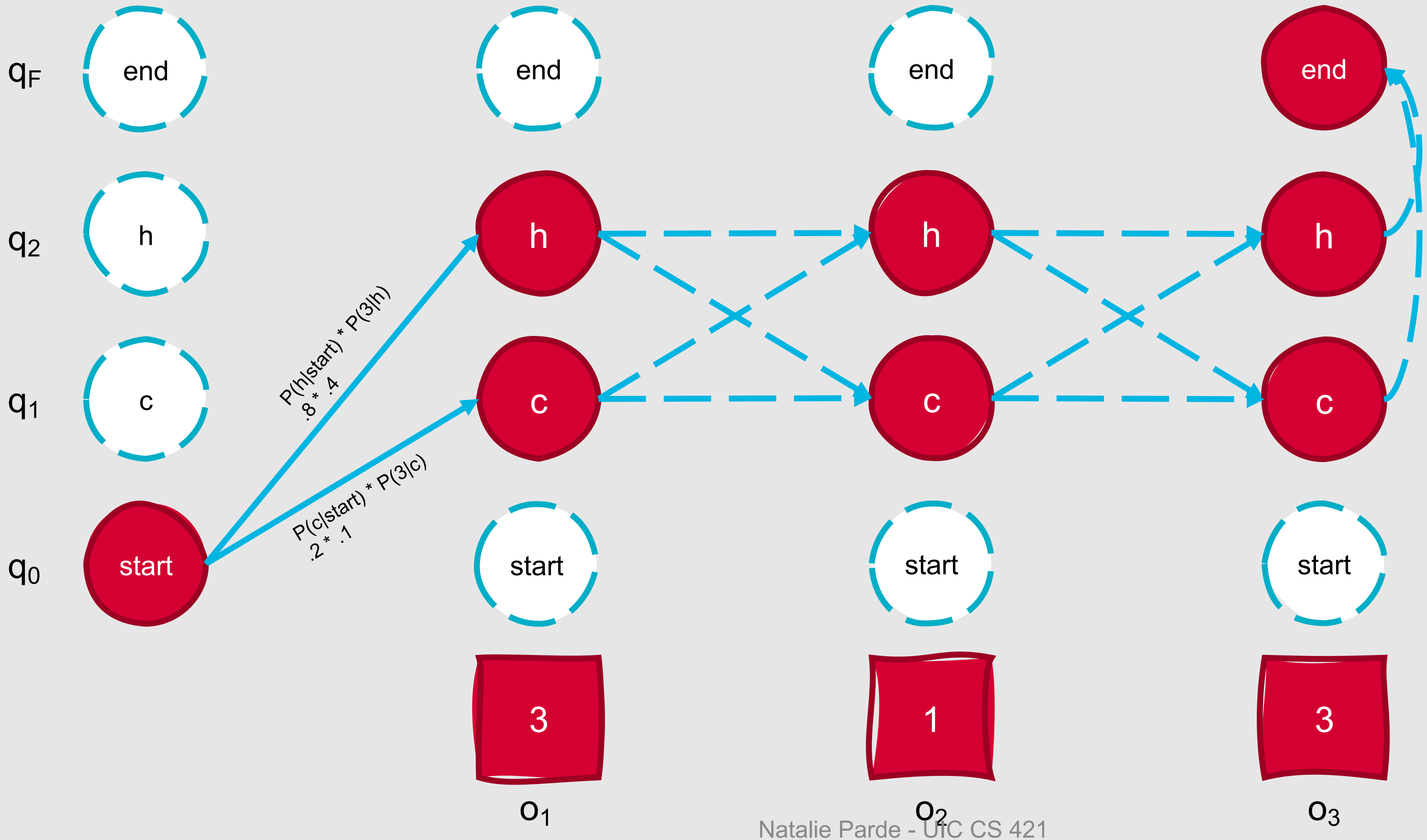
Forward Trellis



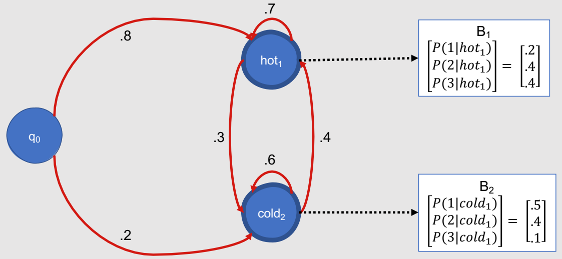


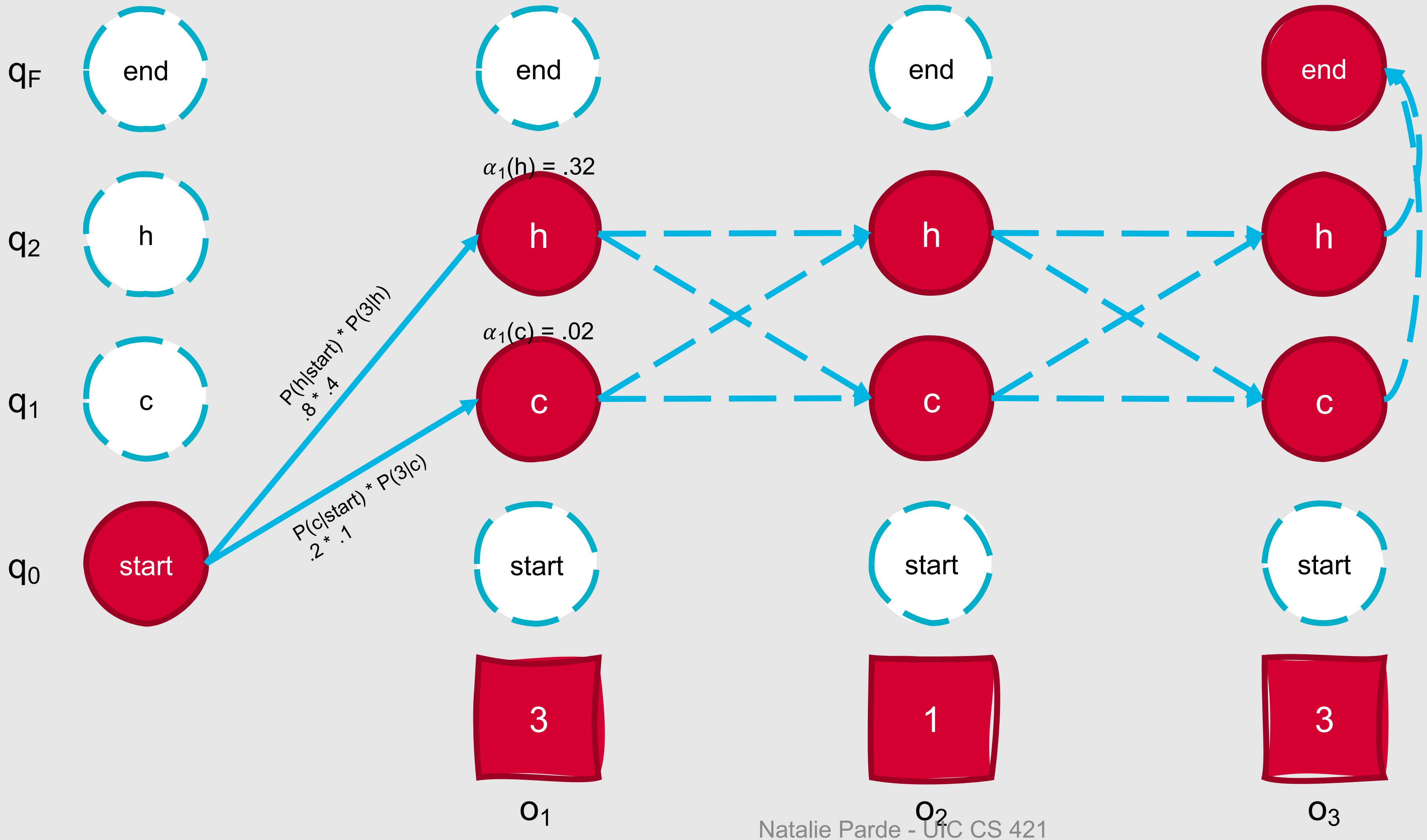
Forward Trellis



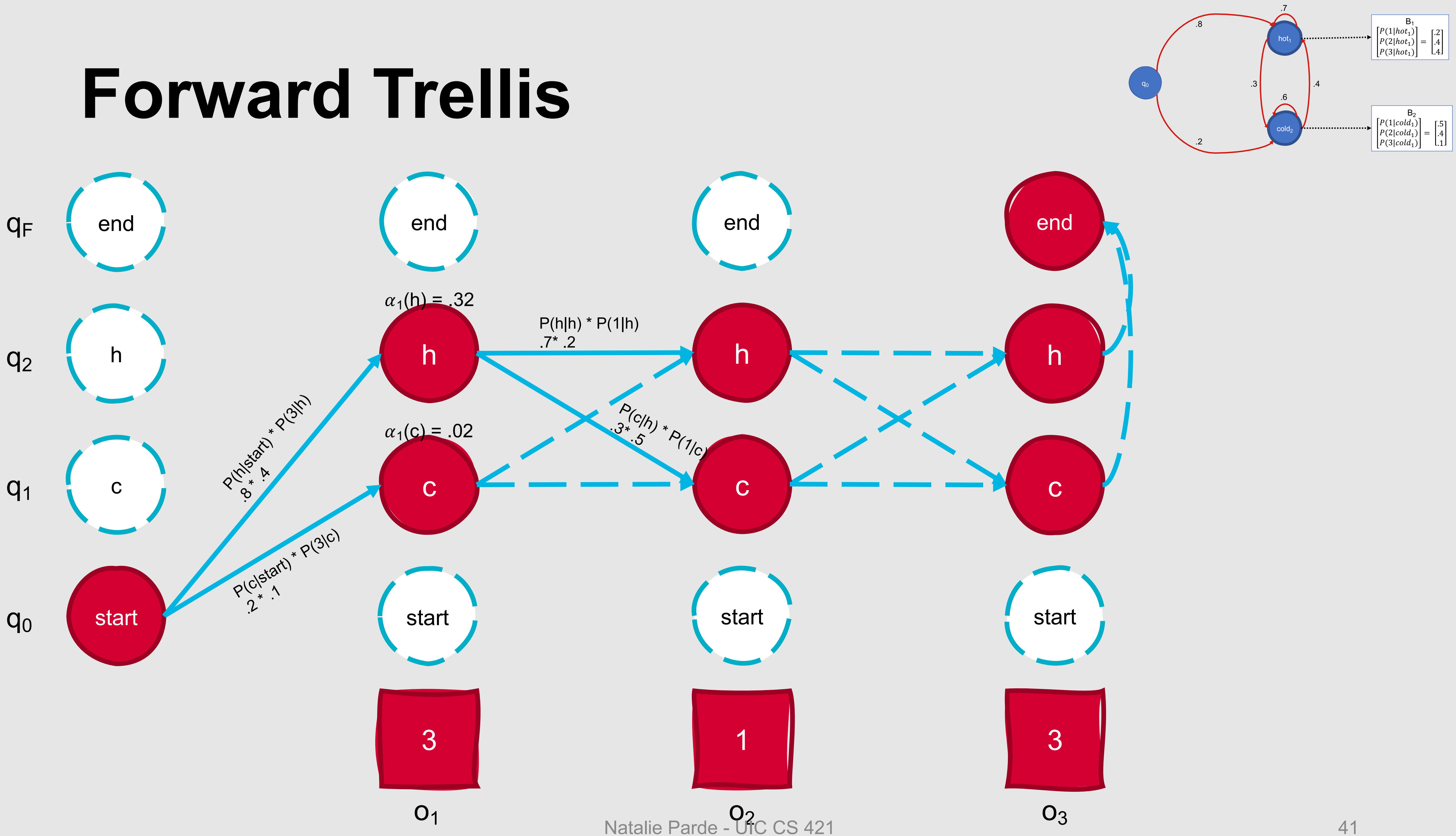


Forward Trellis

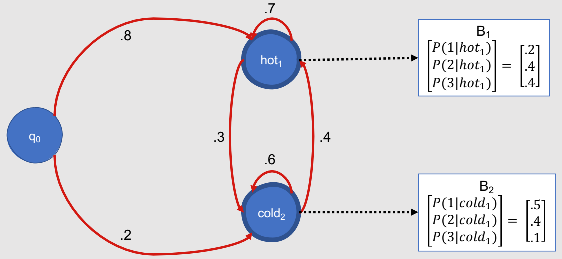


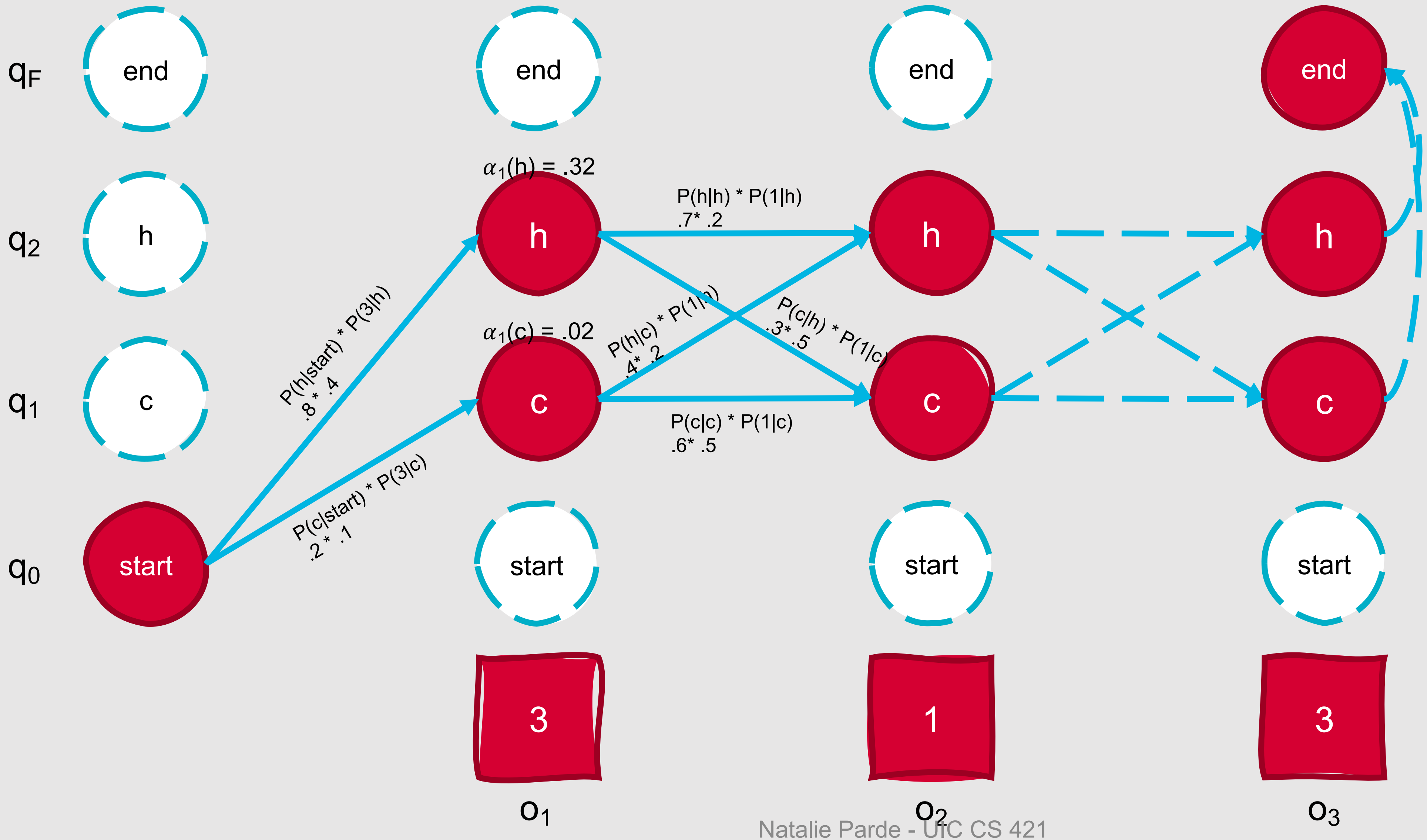


Forward Trellis

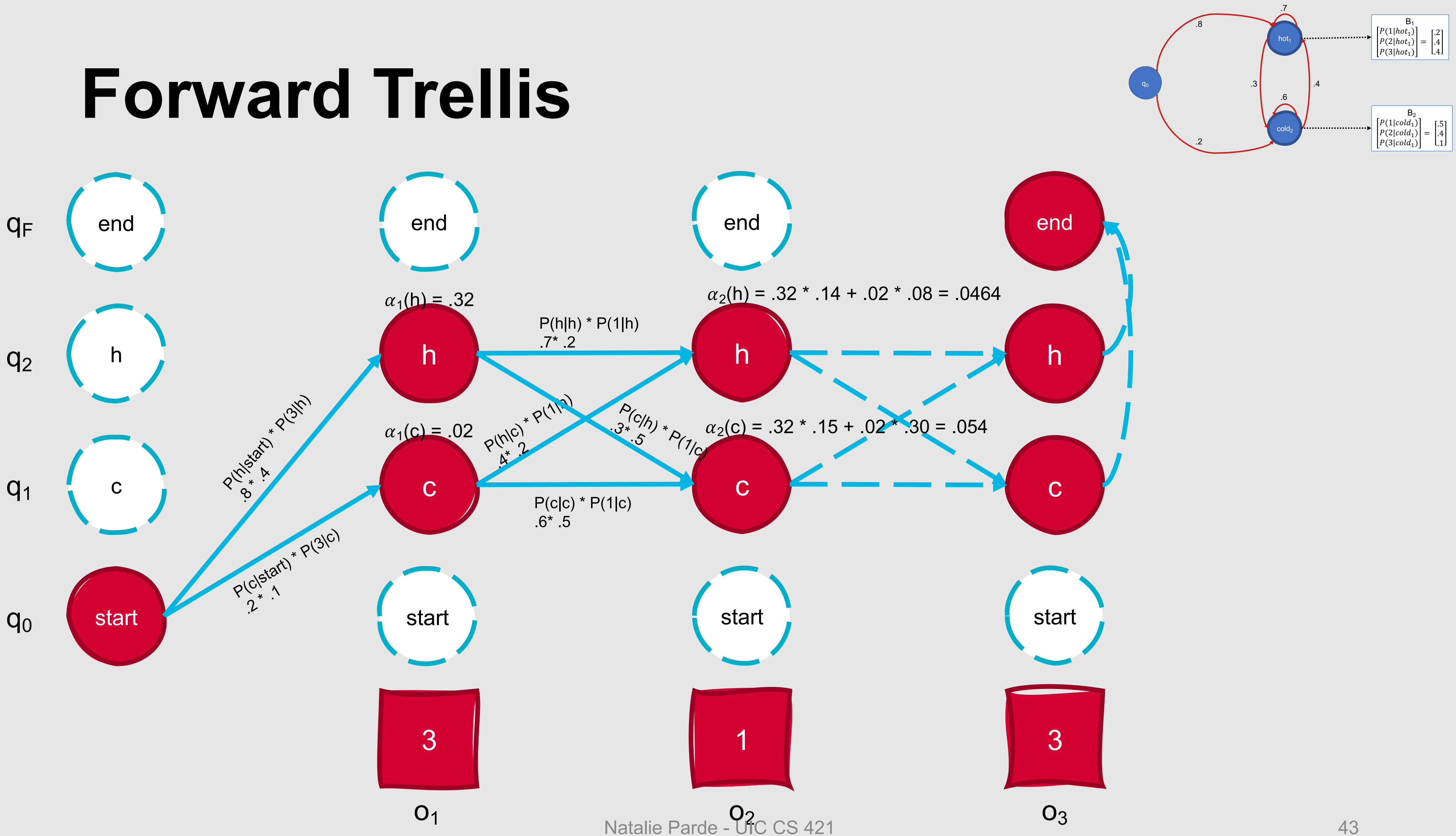


Forward Trellis

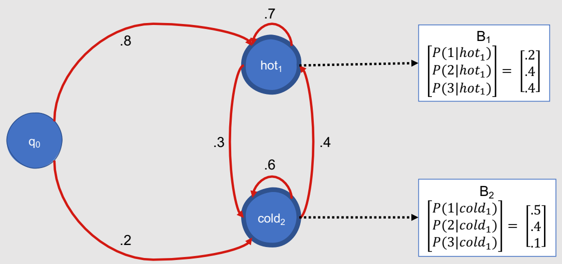


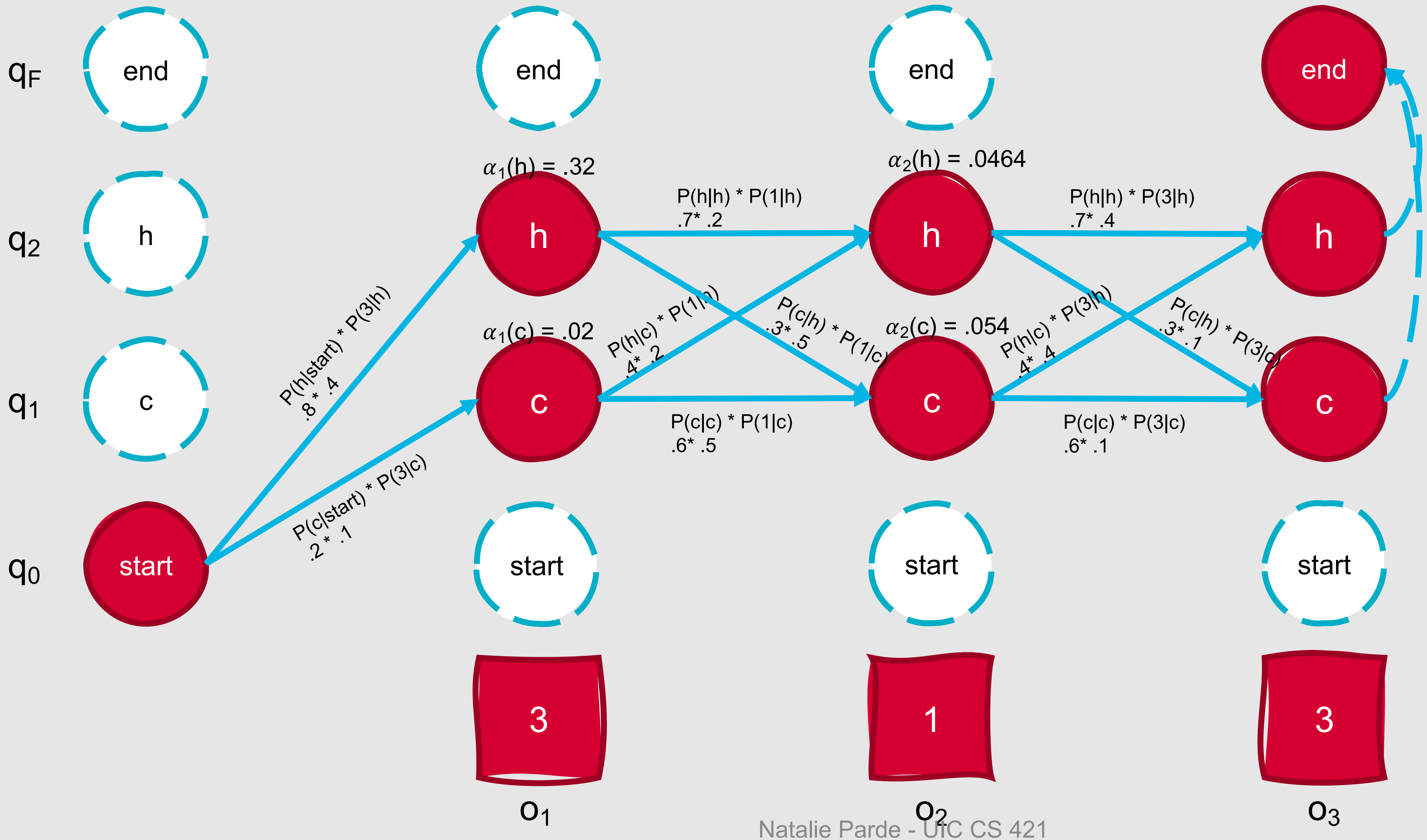


Forward Trellis

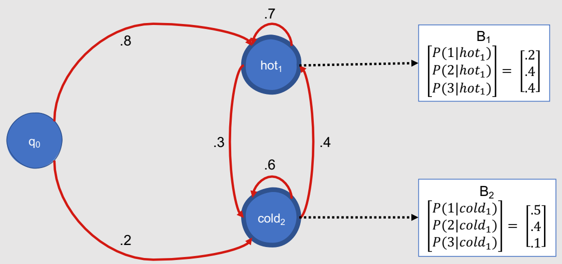


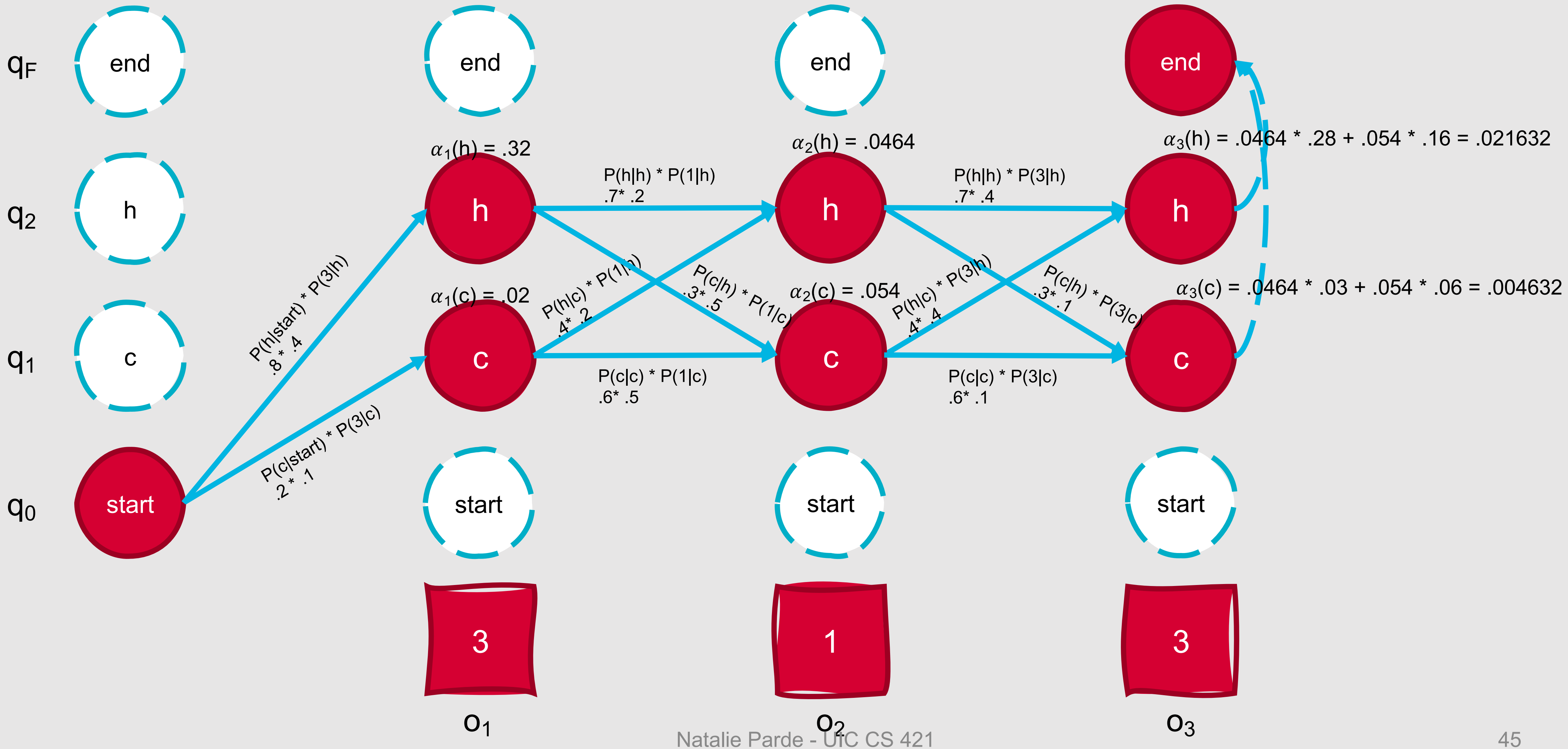
Forward Trellis



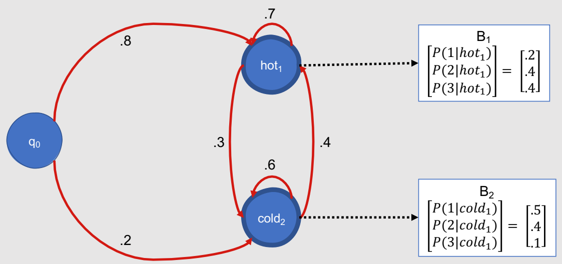


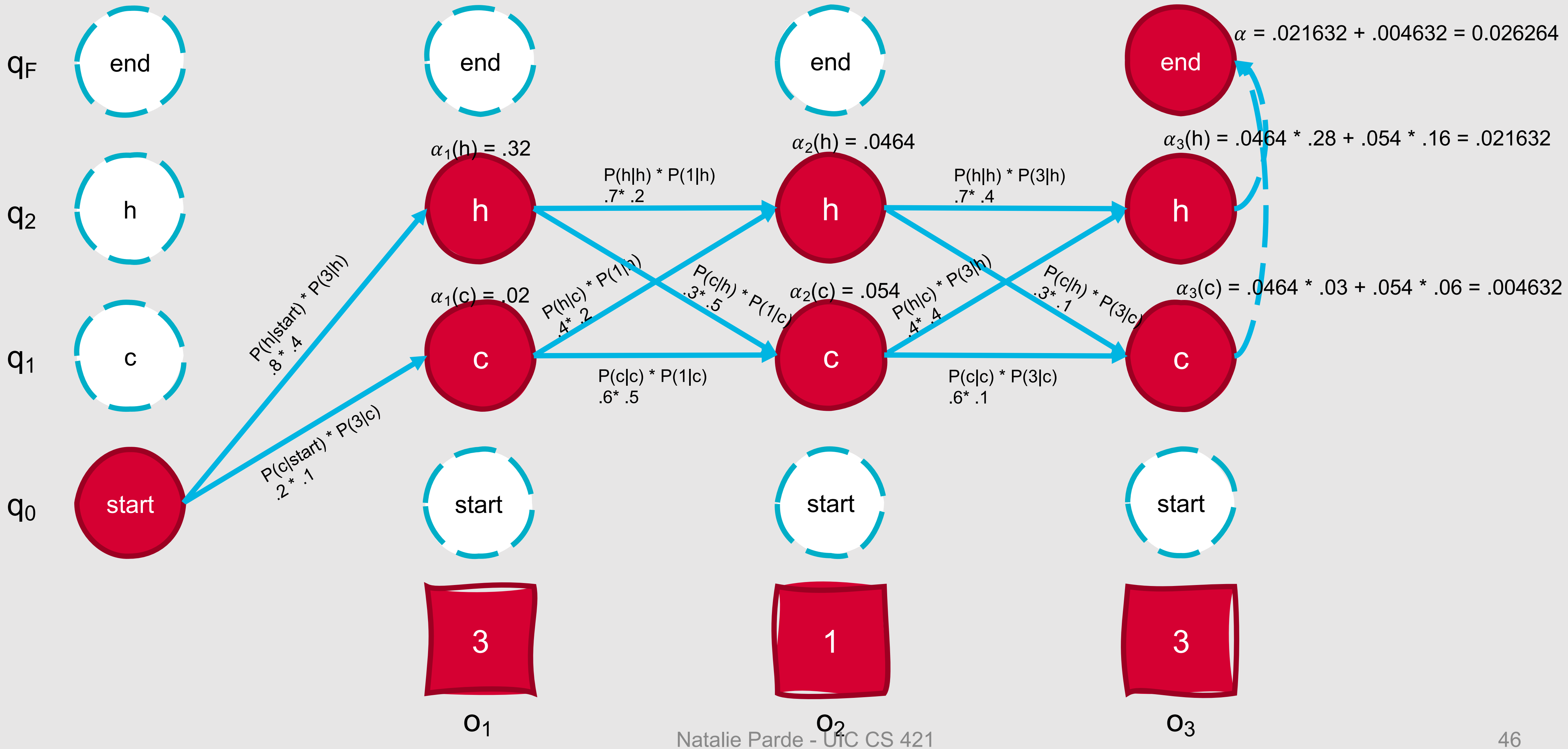
Forward Trellis





Forward Trellis





We've so far tackled one of the fundamental HMM tasks.

- What is the probability that a sequence of observations fits a given HMM?
- However, there are still two remaining tasks to explore….

Decoding

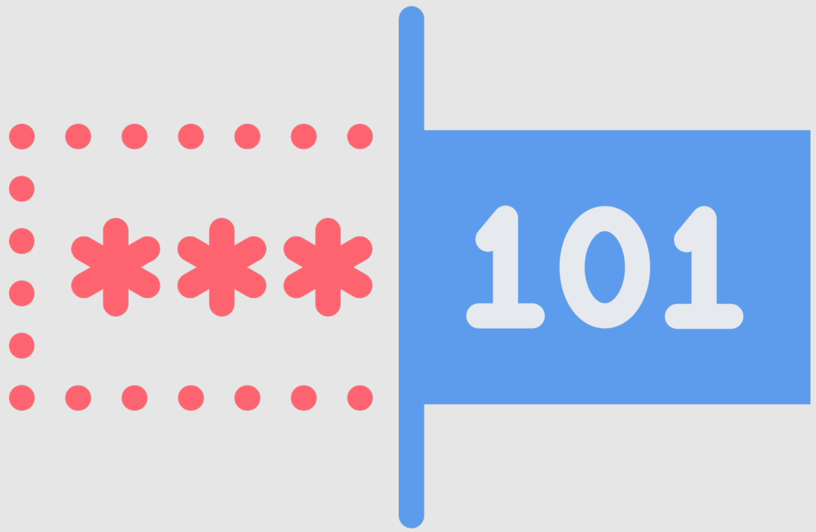
- Given an observation sequence and an HMM, what is the best hidden state sequence?
  - How do we choose a state sequence that is optimal in some sense (e.g., best explains the observations)?
- Very useful for sequence labeling!

Decoding

- Naïve Approach:
  - For each hidden state sequence Q, compute P(O|Q)
  - Pick the sequence with the highest probability
- However, this is computationally inefficient!
  - $O(N^T)$

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.
  - $v_t(j) = \max_{q_0,q_1,\ldots,q_{t-1}} P(q_0, q_1, \ldots, q_{t-1}, o_1, \ldots, o_t, q_t = q_j|\lambda)$
- Also **record backpointers** that subsequently allow you to backtrace 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,…,N] do:
      Viterbi[q,1] ← a0,q * bq(o1)
      backpointer[q,1] ← 0
for each time step t in [2,…,T] do:
      for each state q in [1,…,N] do:
```

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

$$backpointer[q,t] \leftarrow \underset{q' \in [1,\ldots,N]}{\operatorname{argmax}}\, viterbi[q',t-1] * a_{q',q} * b_q(o_t)$$

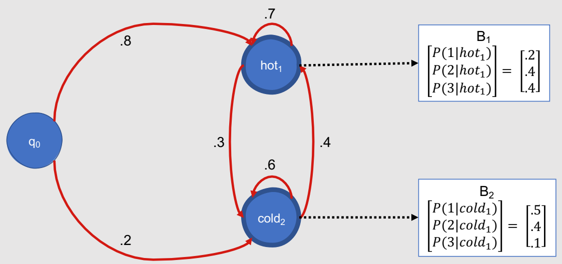
$$\text{bestpathprob} \leftarrow \max_{q' \in [1,\ldots,N]} viterbi[q',T]$$

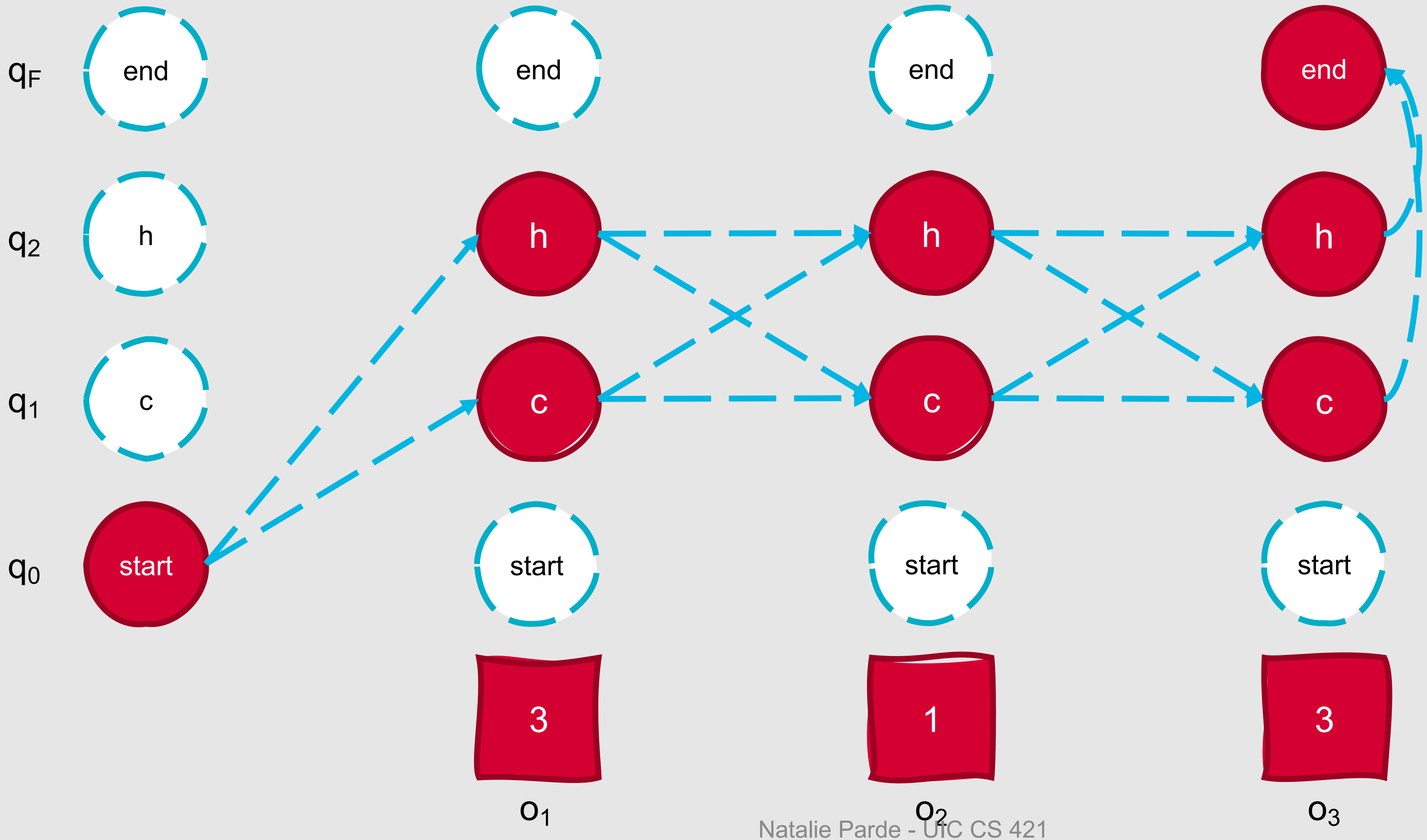
$$\text{bestpathpointer} \leftarrow \underset{q' \in [1,\ldots,N]}{\operatorname{argmax}}\, viterbi[q',T]$$

Seem familiar?

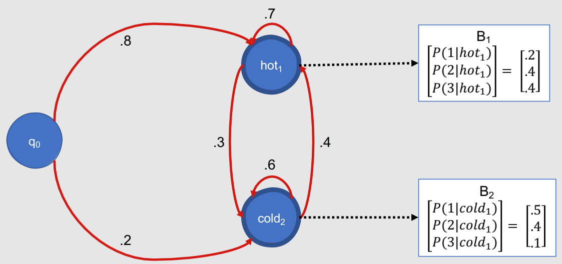
- Viterbi is basically the forward algorithm + backpointers, and substituting a max function for the summation operator

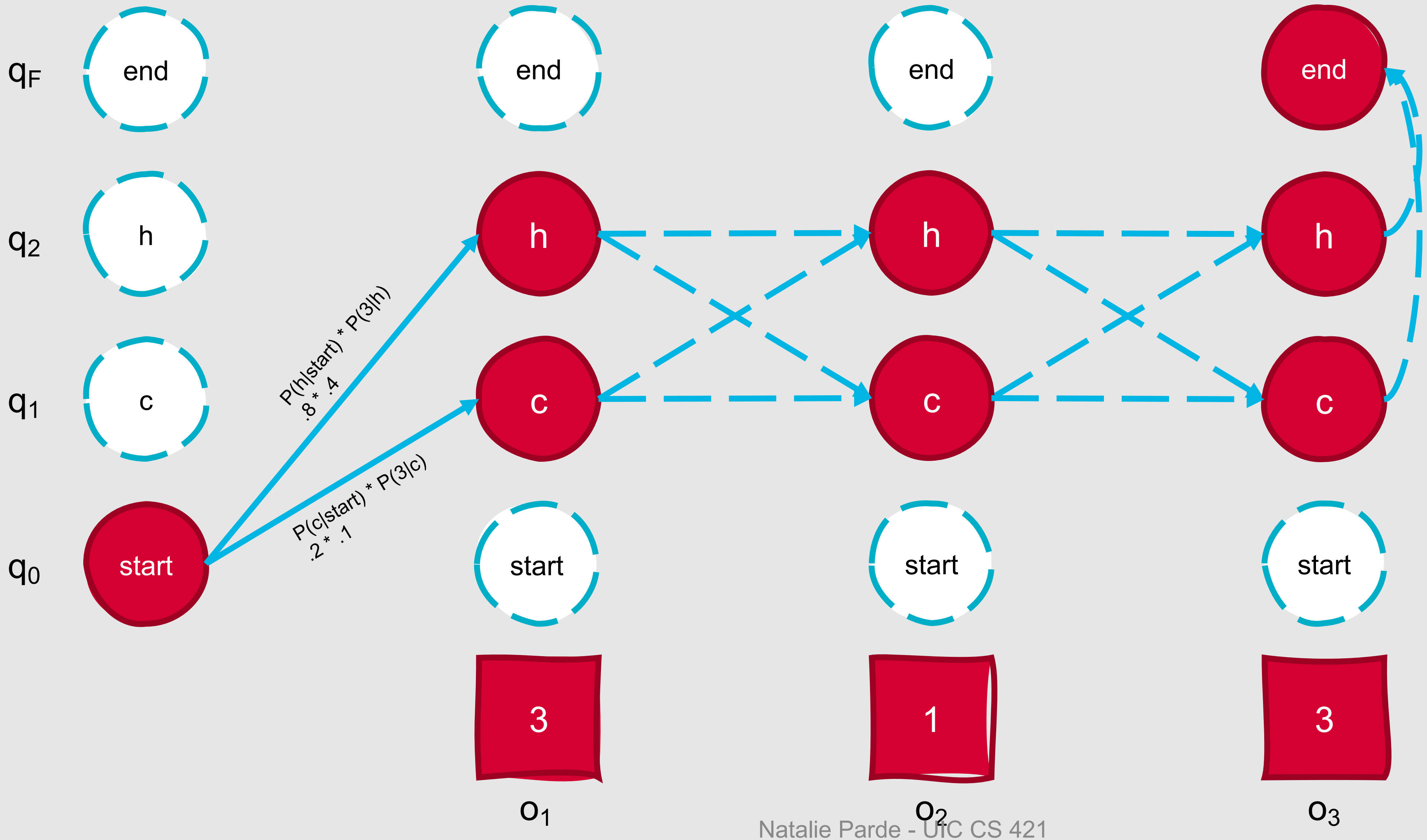
Viterbi Trellis



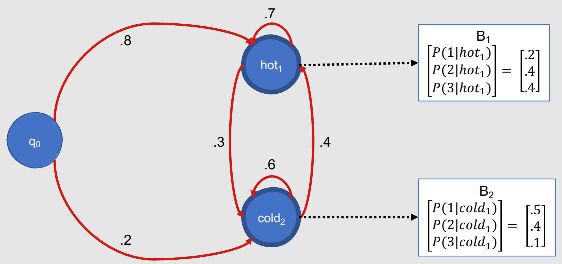


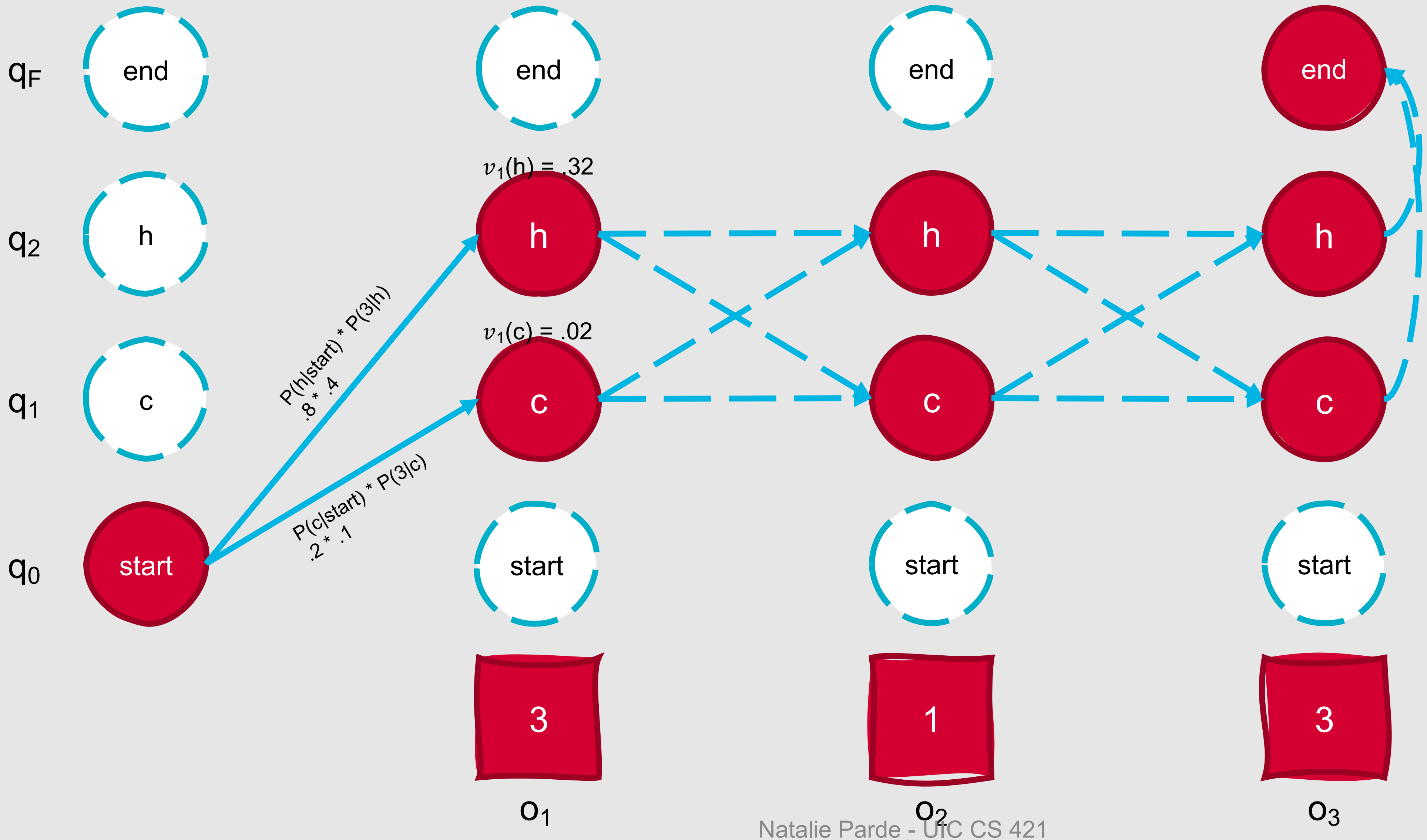
Viterbi Trellis



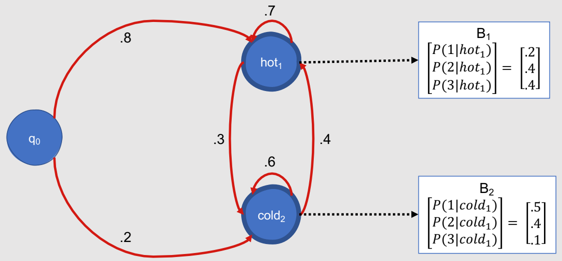


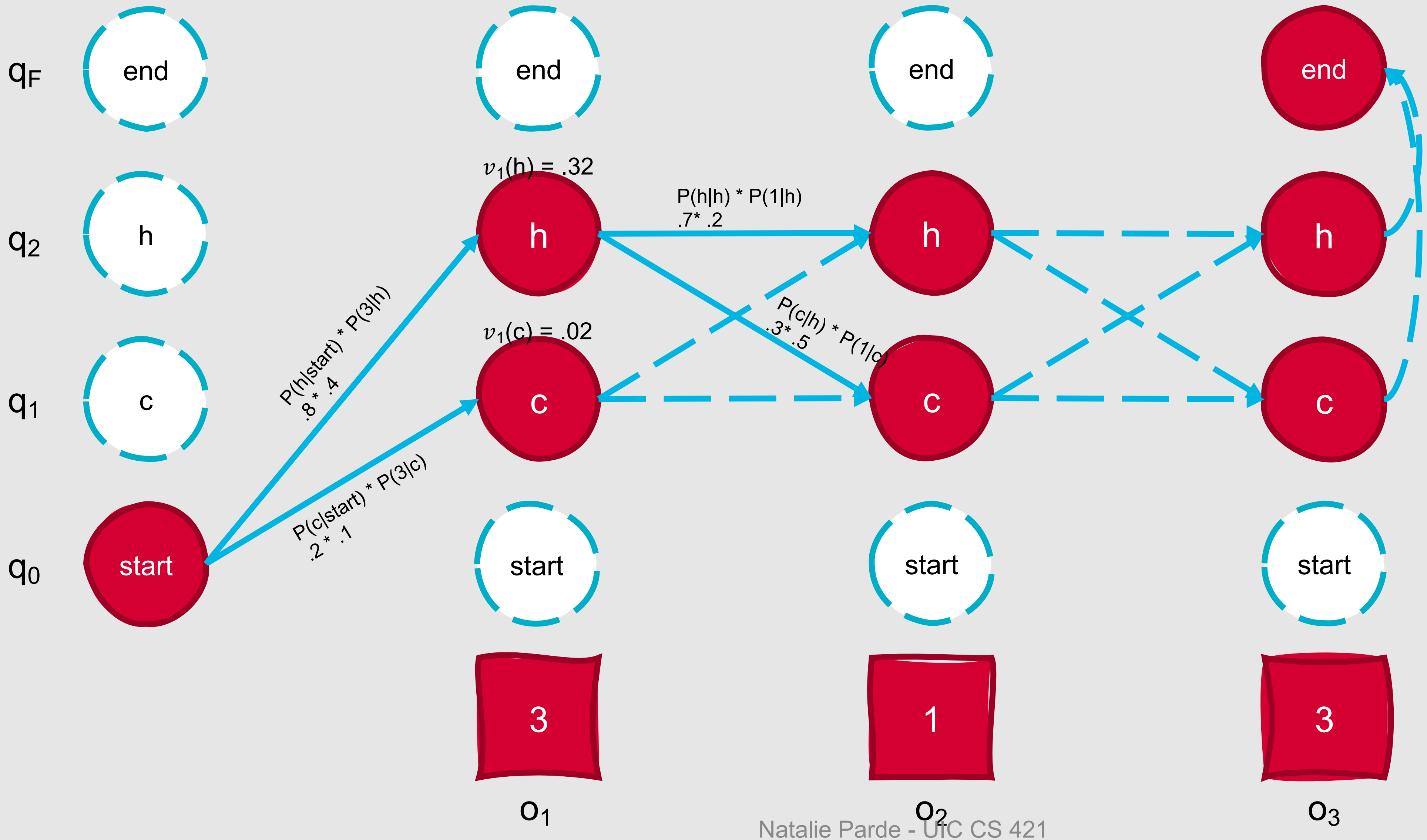
Viterbi Trellis



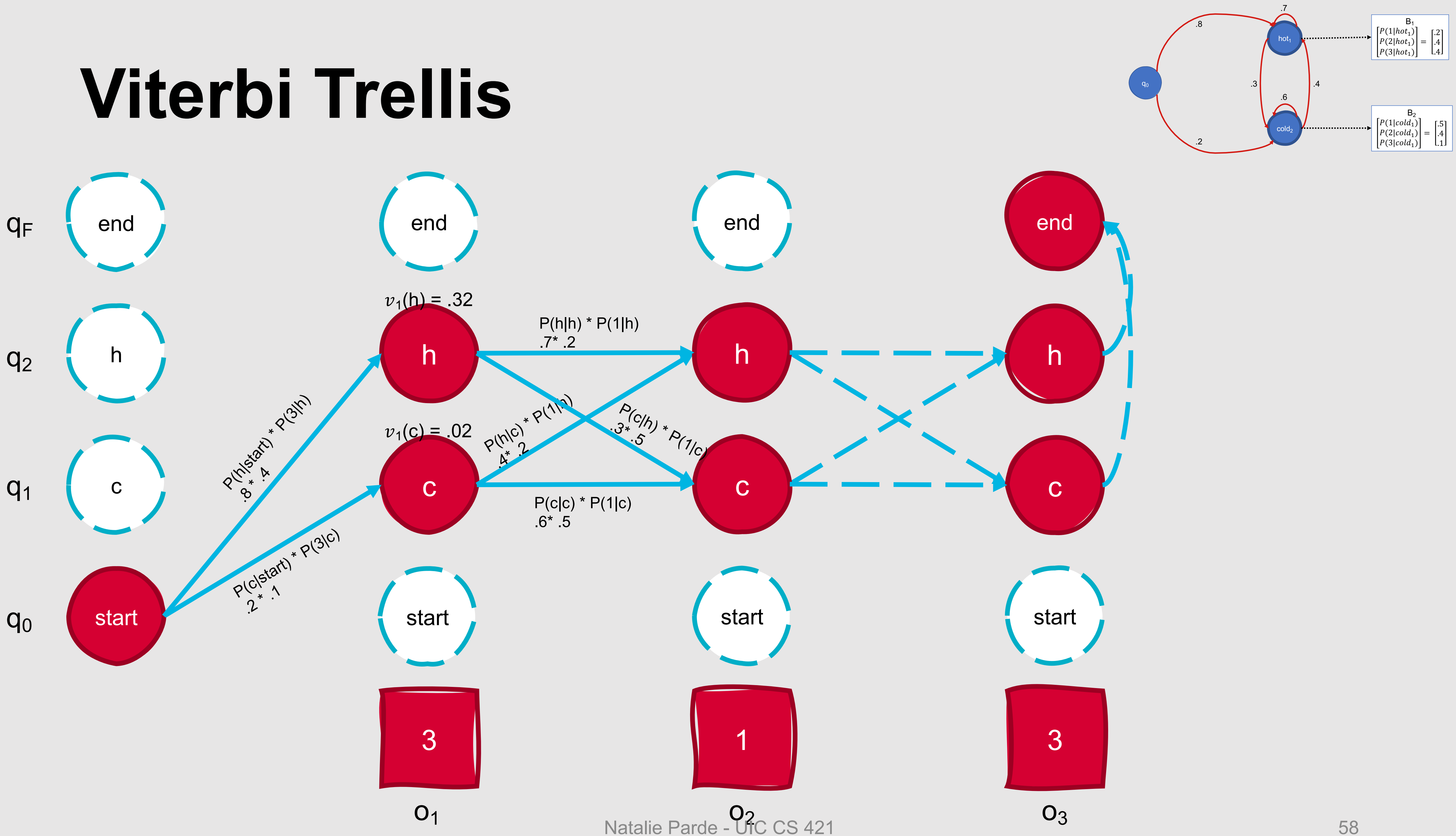


Viterbi Trellis

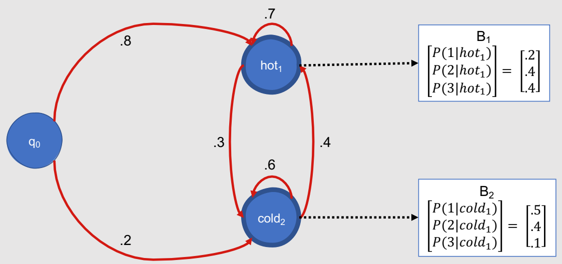


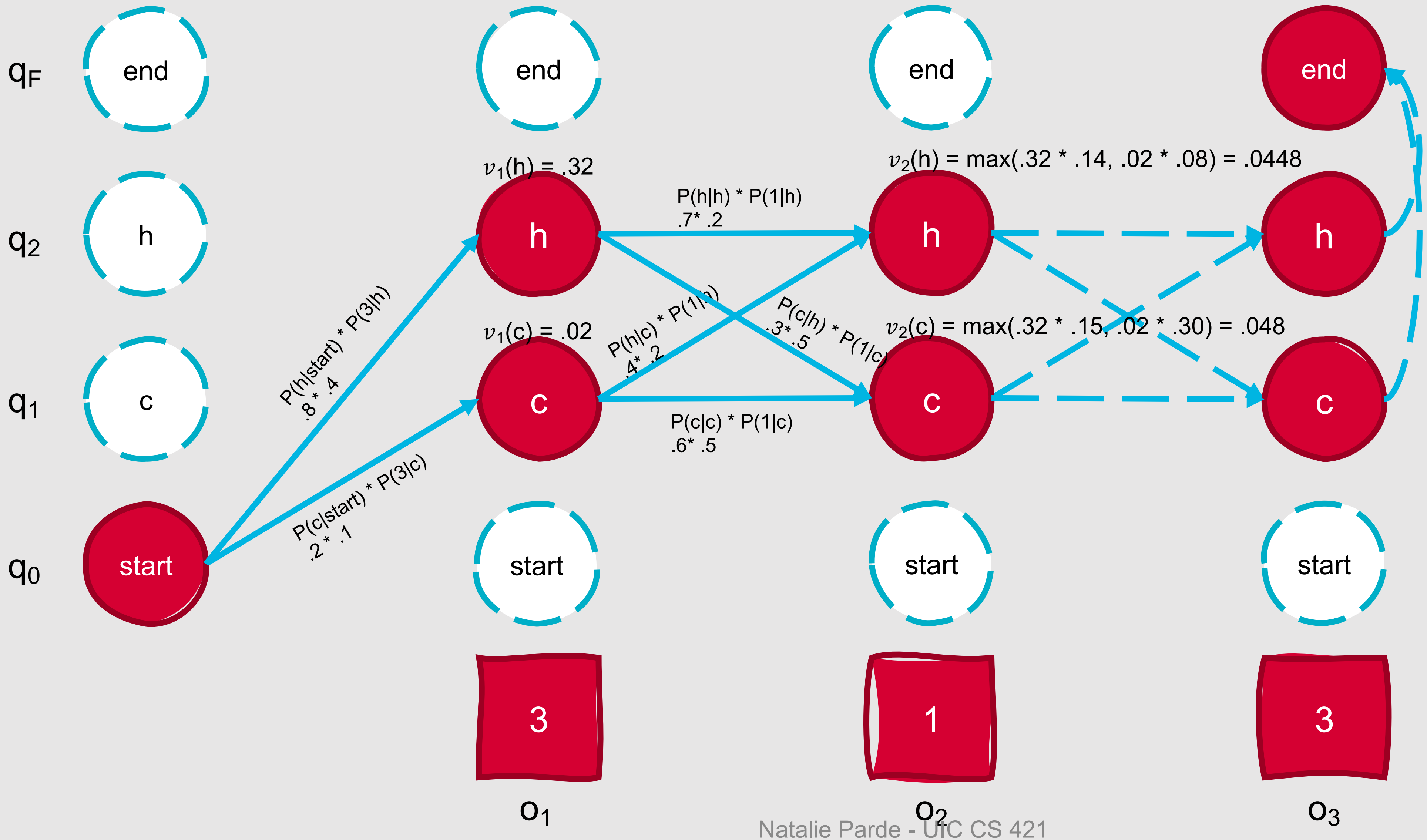


Viterbi Trellis

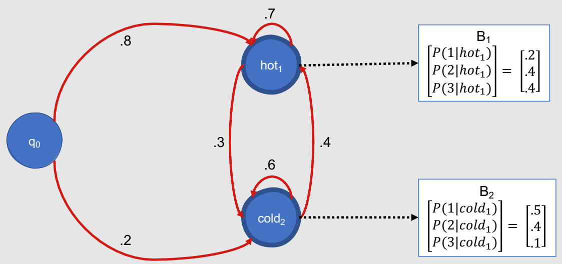


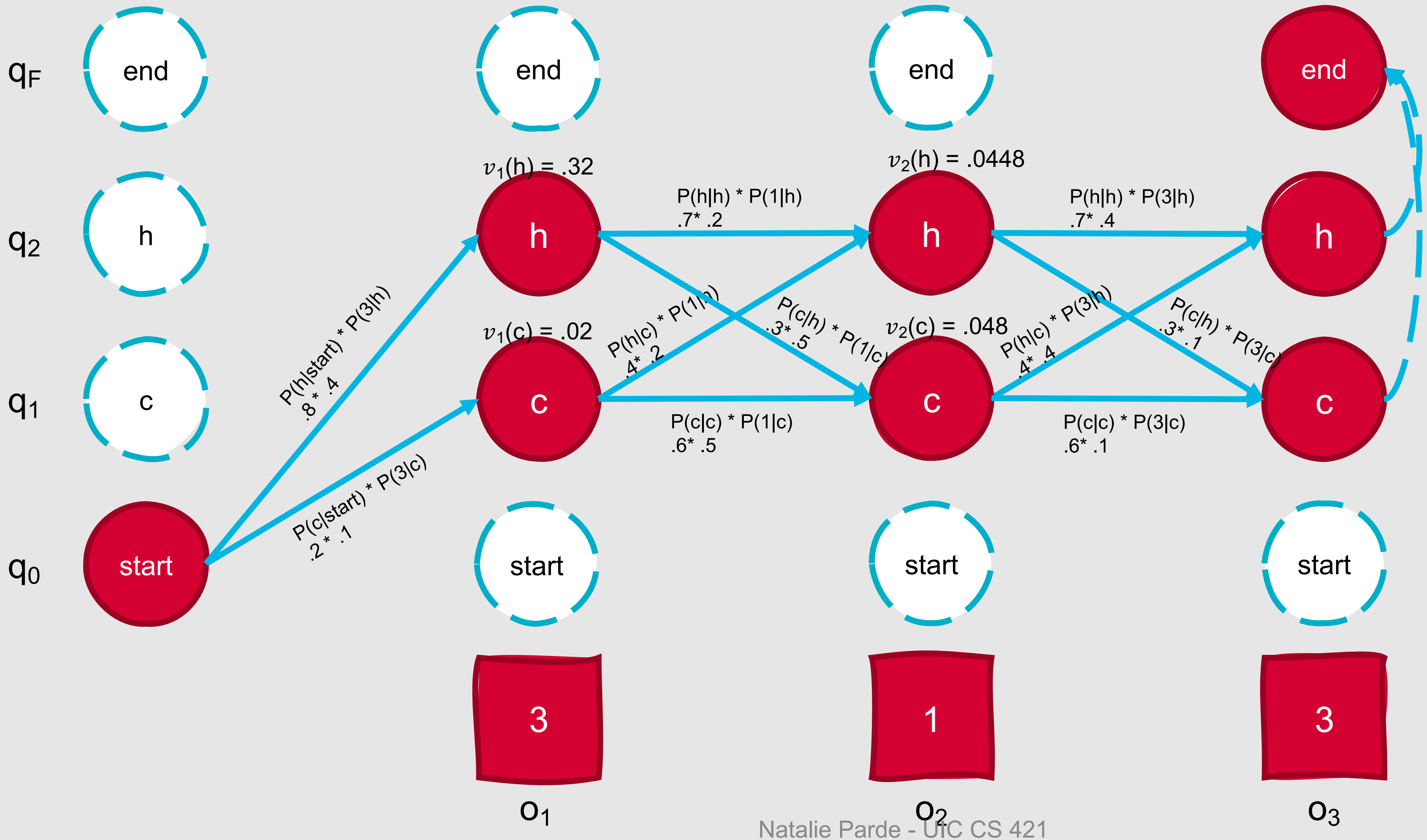
Viterbi Trellis



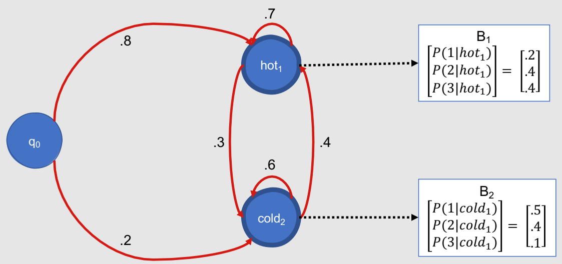


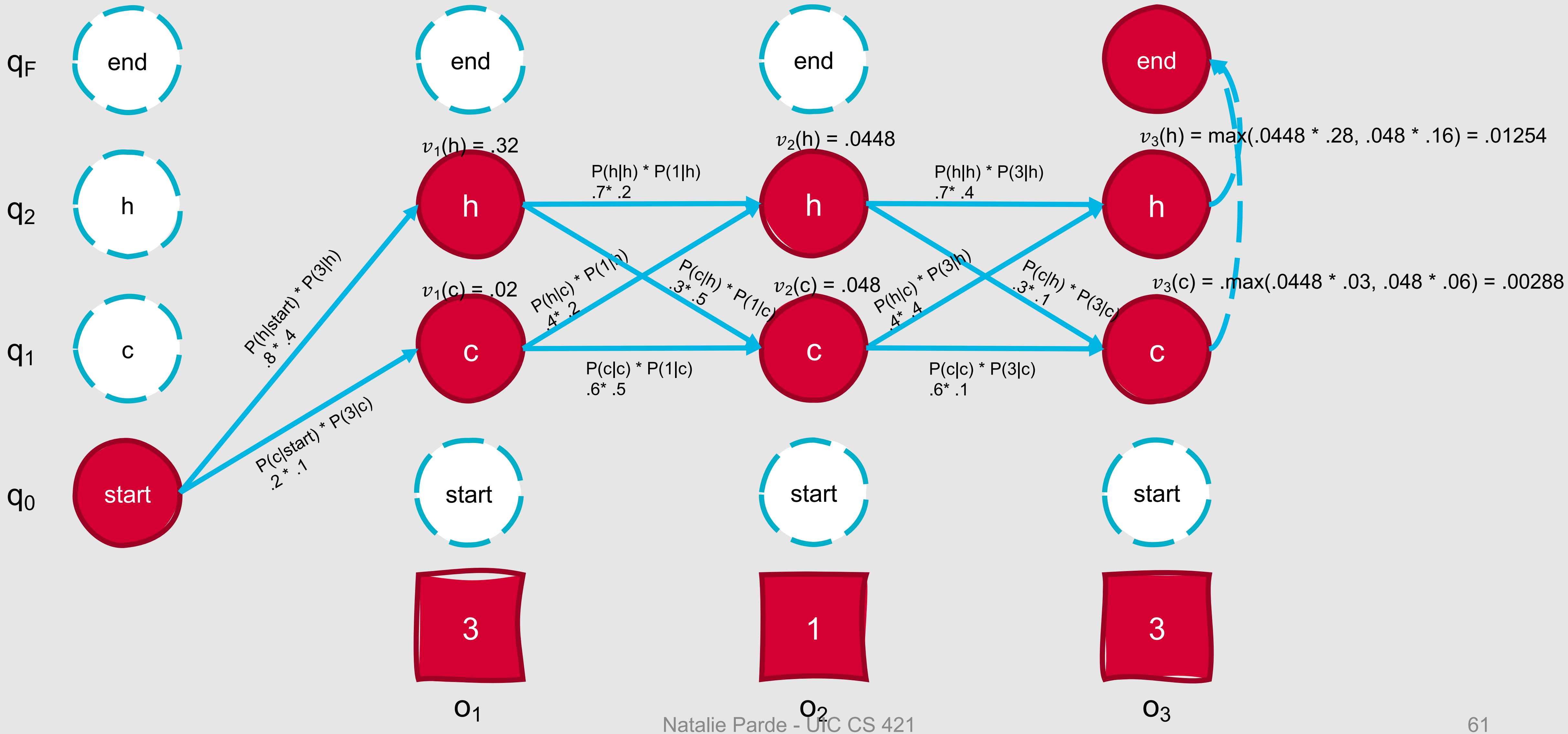
Viterbi Trellis



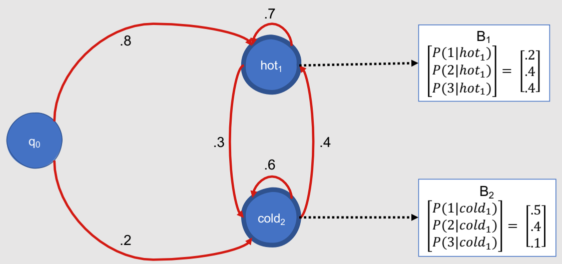


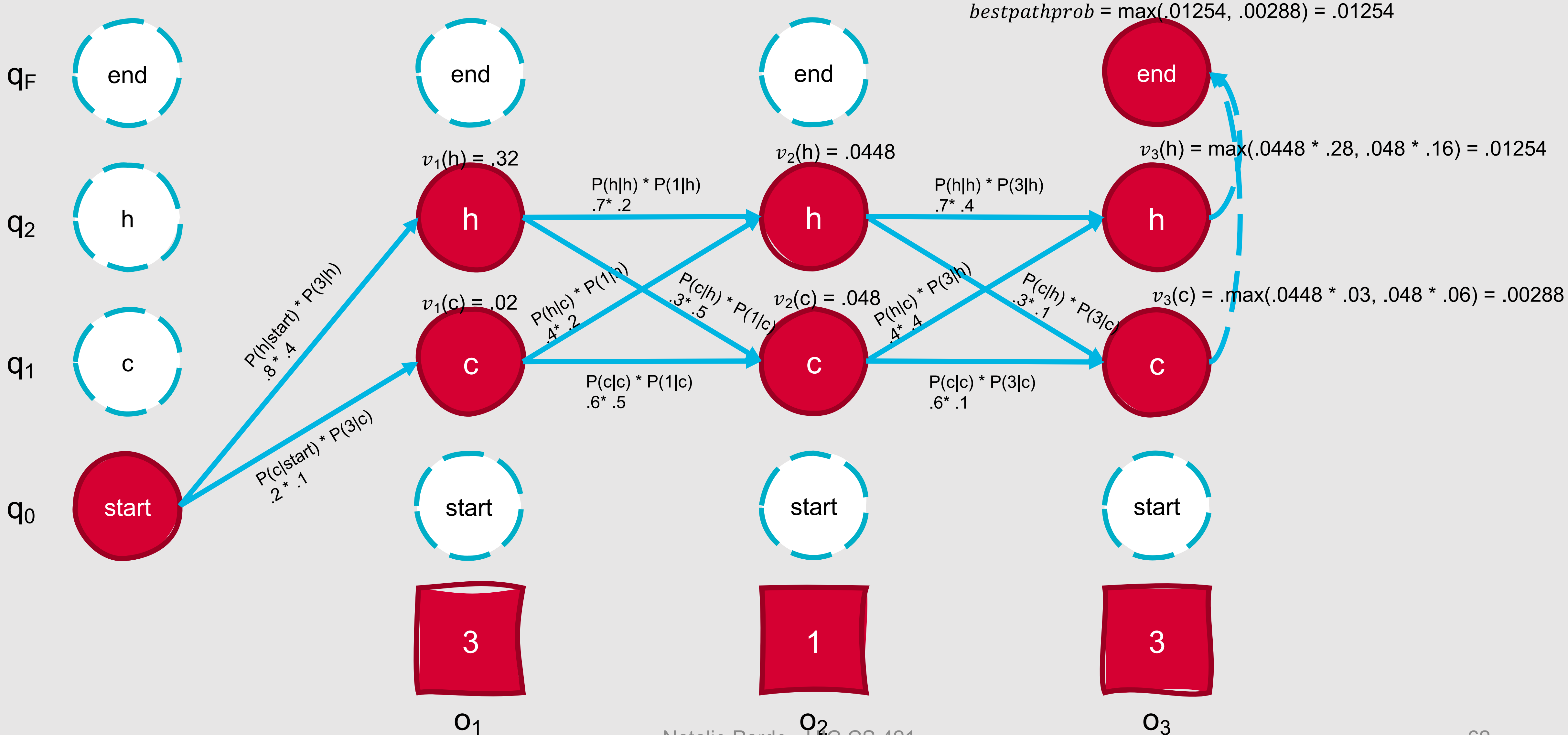
Viterbi Trellis



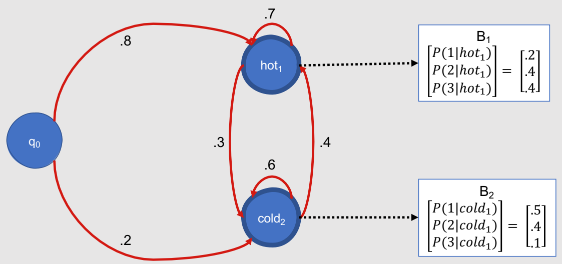


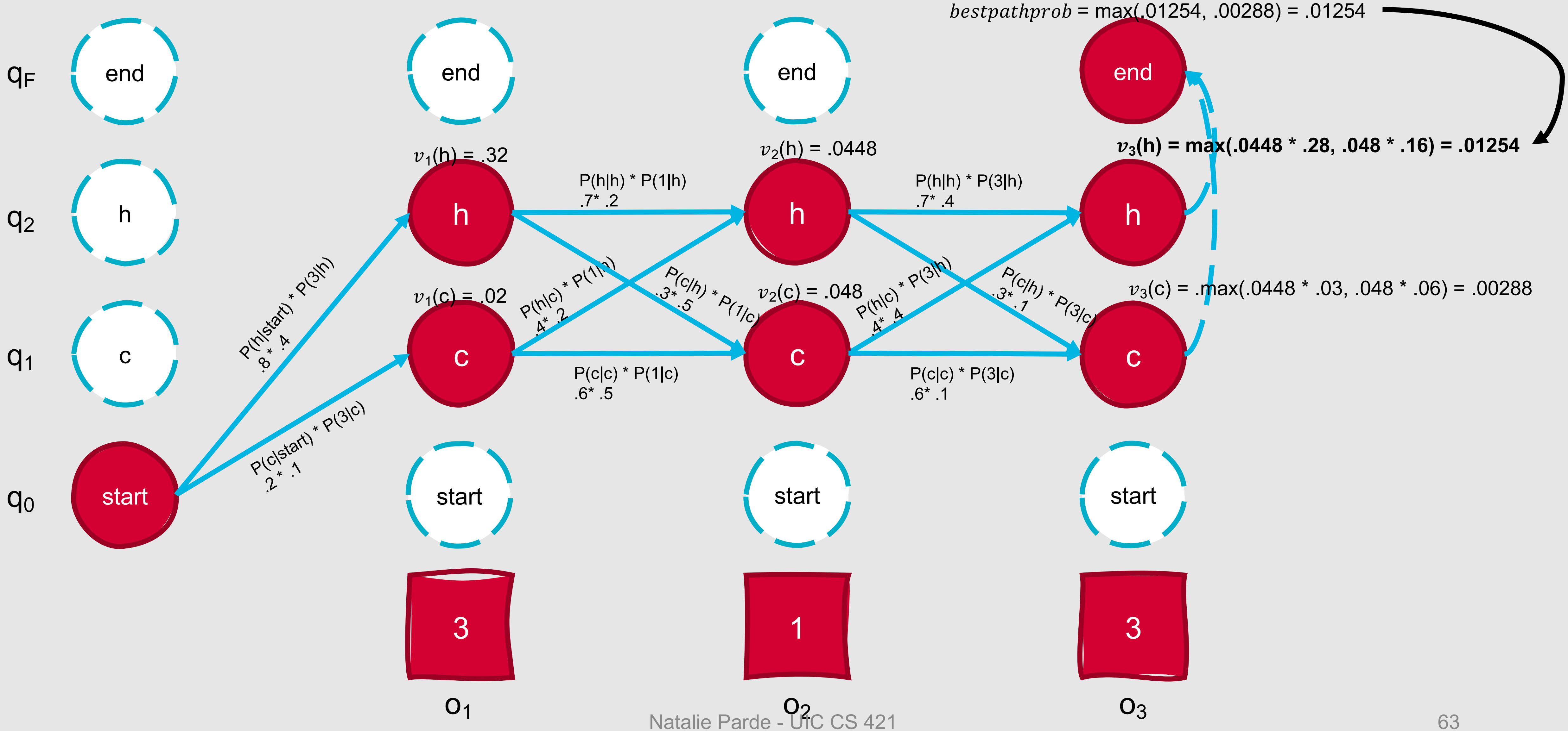
Viterbi Trellis



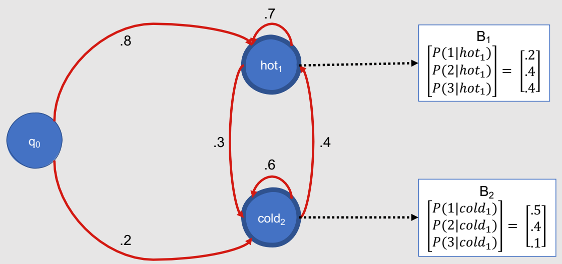


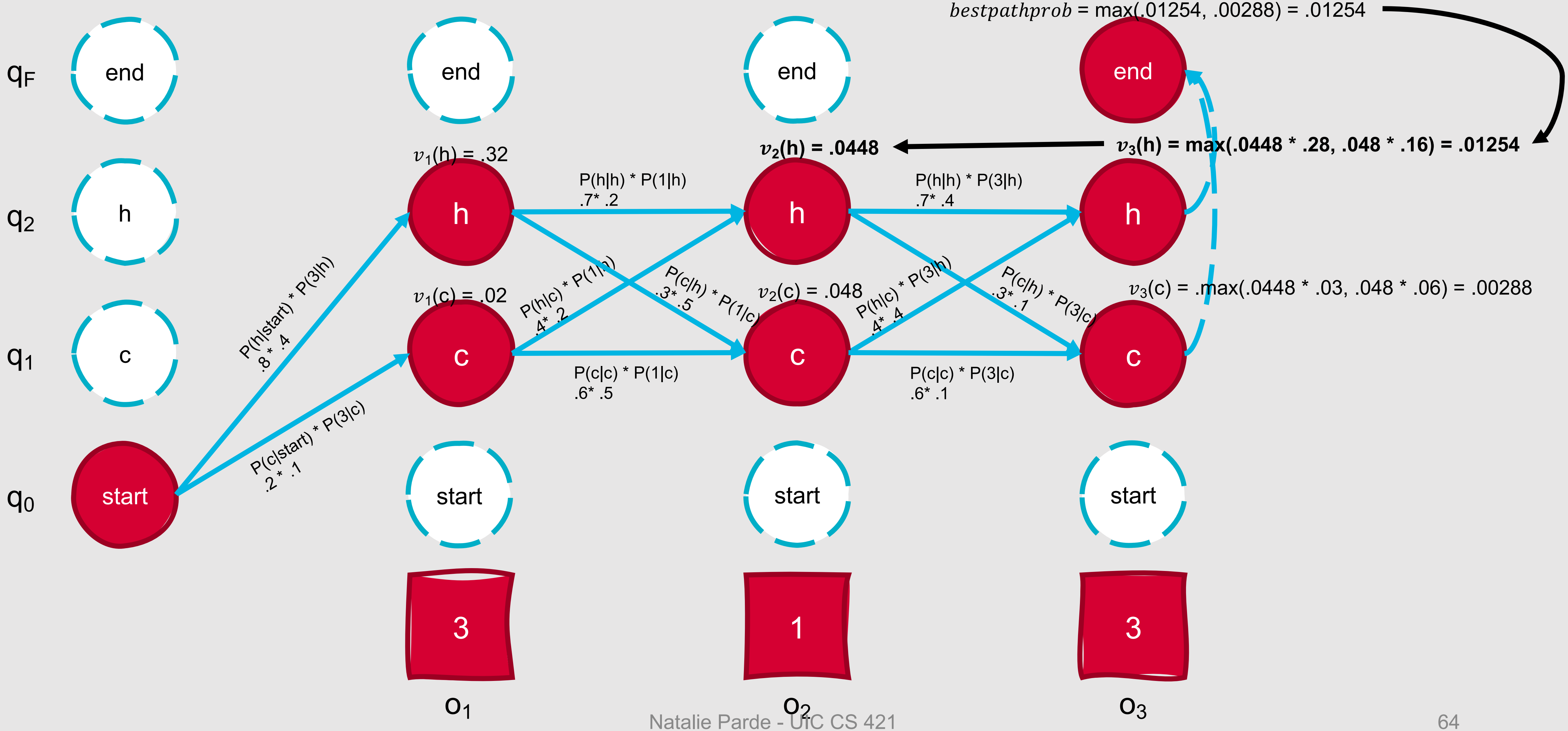
Viterbi Backtrace



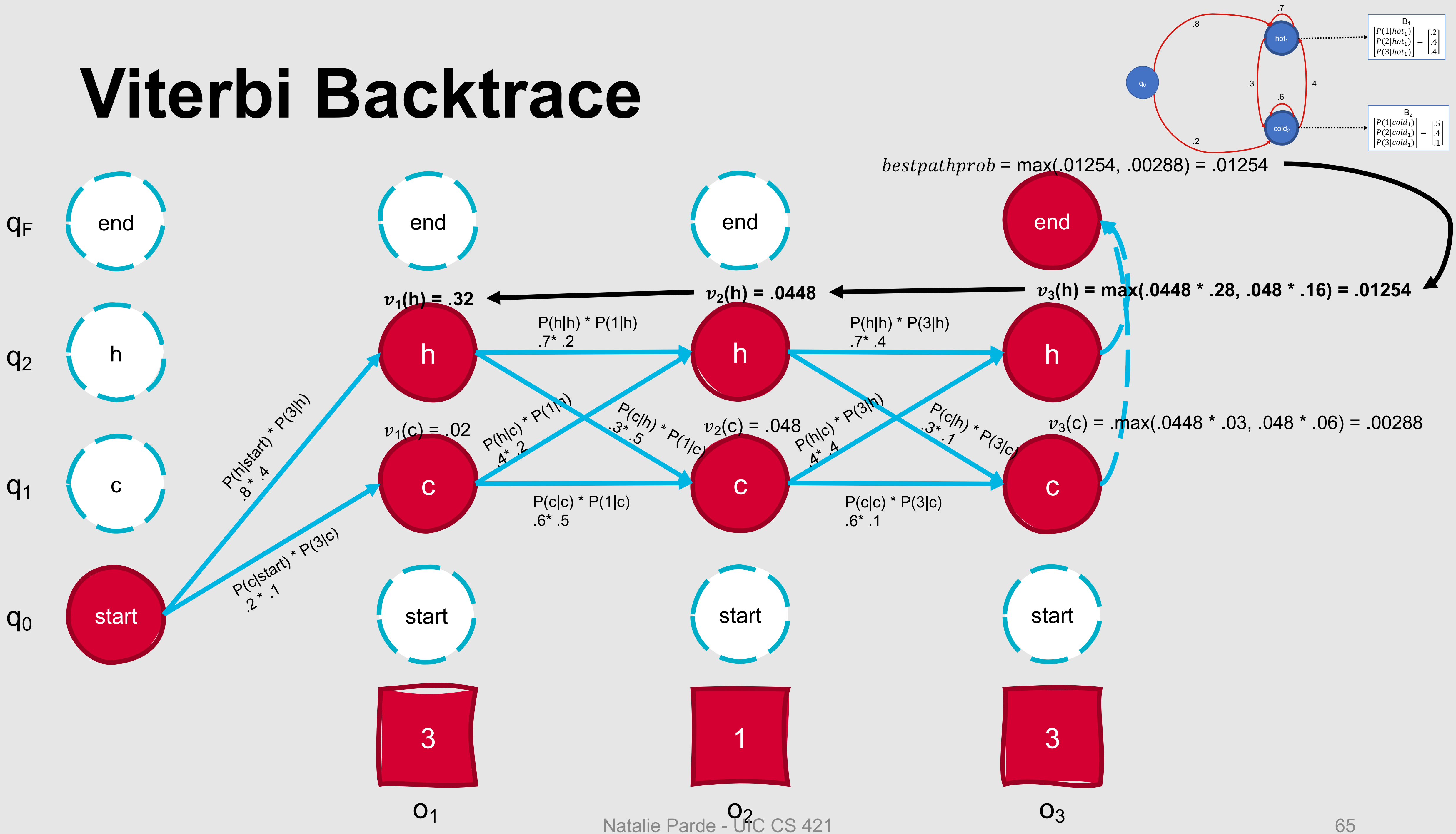


Viterbi Backtrace

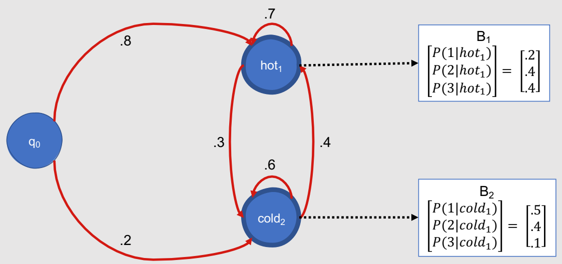


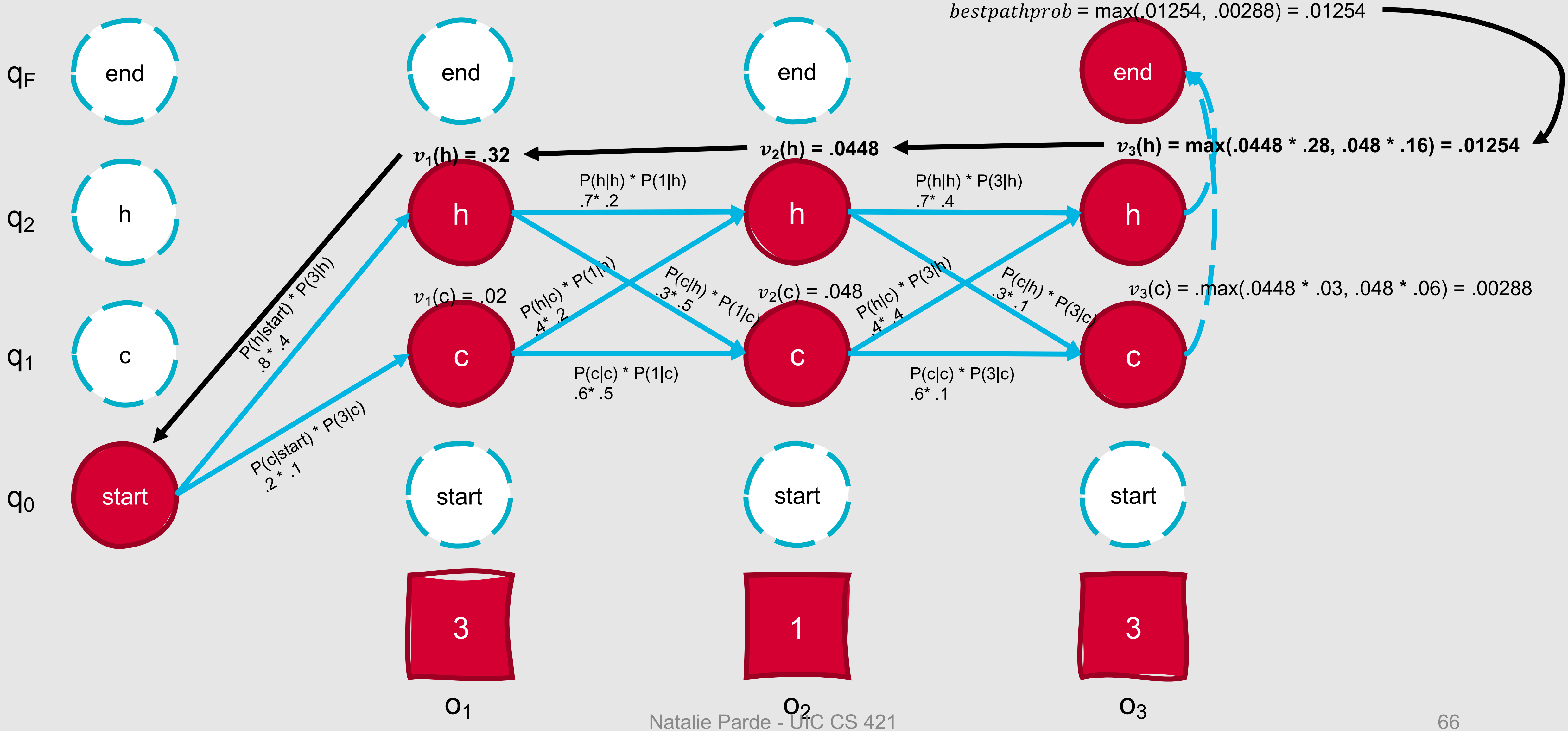


Viterbi Backtrace



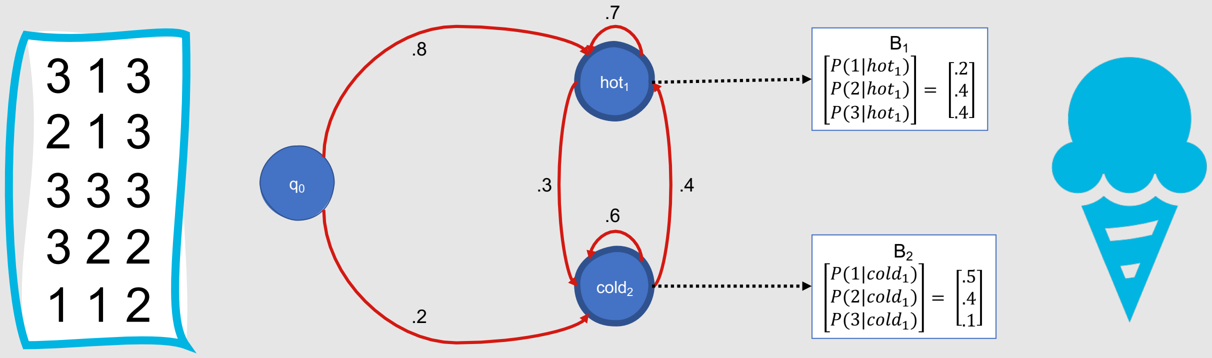
Viterbi Backtrace





Finally …learning!

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



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
- Output: Transition probabilities and observation likelihoods

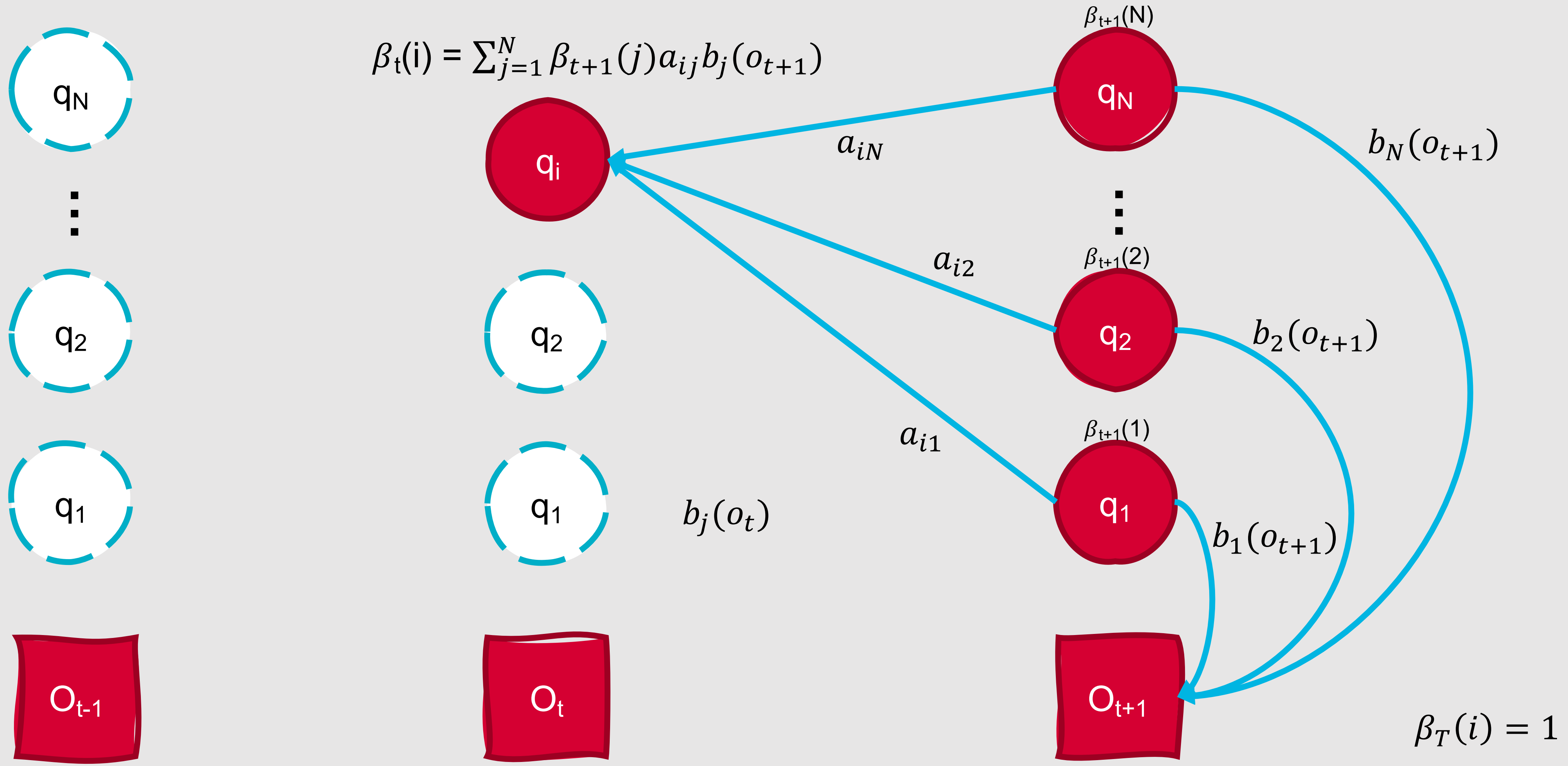
How does the algorithm compute these outputs?

- **Iteratively estimate the counts for transitions from one state to another**
  - 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

The second part of this process uses the **backward algorithm**.

- We define the backward probability as follows:
  - $\beta_t(i) = P(o_{t+1}, o_{t+2}, \ldots, o_T | q_t = i, \lambda)$
- This is the probability of generating partial observations from time *t*+1 until the end of the sequence, given that the HMM λ is in state *i* at time *t*
- Also computed using a trellis, but moves backwards instead

Backward Step



Once we compute backward probabilities, we can start estimating transition probabilities and observation likelihoods.

- We re-estimate transition probabilities, a_{ij}, as follows:
  - $\widehat{a_{ij}} = \frac{\text{expected \# transitions from state } i \text{ to state } j}{\text{expected \# transitions from state } i} = \frac{\sum_{t=1}^{T-1} \frac{a_t(i) a_{aij} b_j \beta_{t+1}(j)}{a_T(q_F)}}{\sum_{t=1}^{T-1} \sum_{j=1}^{N} \frac{a_t(i) a_{aij} b_j \beta_{t+1}(j)}{a_T(q_F)}}$
  - Check out the course textbook (Appendix A) for an in-depth discussion of how the numerator and denominator above are derived!
- It's common to simplify the inner portion of the summation in the equation above:
  - $\zeta_t(i,j) = \frac{a_t(i) a_{aij} b_j \beta_{t+1}(j)}{a_T(q_F)}$
- Therefore:
  - $\widehat{a_{ij}} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \sum_{j=1}^{N} \xi_t(i,j)}$

Re-Estimating Observation Likelihood

- We re-estimate b_j as follows:
  - $\hat{b}_j(v_k) = \frac{\text{expected \# of times in state } j \text{ and observing symbol } v_k}{\text{expected number of times in state } j} = \frac{\sum_{t=1,\ \text{s.t. } o_t=v_k}^{T} \frac{a_t(j)\beta_t(j)}{a_T(q_F)}}{\sum_{t=1}^{T} \frac{a_t(j)\beta_t(j)}{a_T(q_F)}}$
- Again, to simplify presentation we can abstract away the inner portion of the summation:
  - $\gamma_t(j) = \frac{a_t(j)\beta_t(j)}{a_T(q_F)}$
- Therefore:
  - $\hat{b}_j(v_k) = \frac{\sum_{t=1 \text{ s.t. } o_t=v_k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}$

Forward-Backward Algorithm

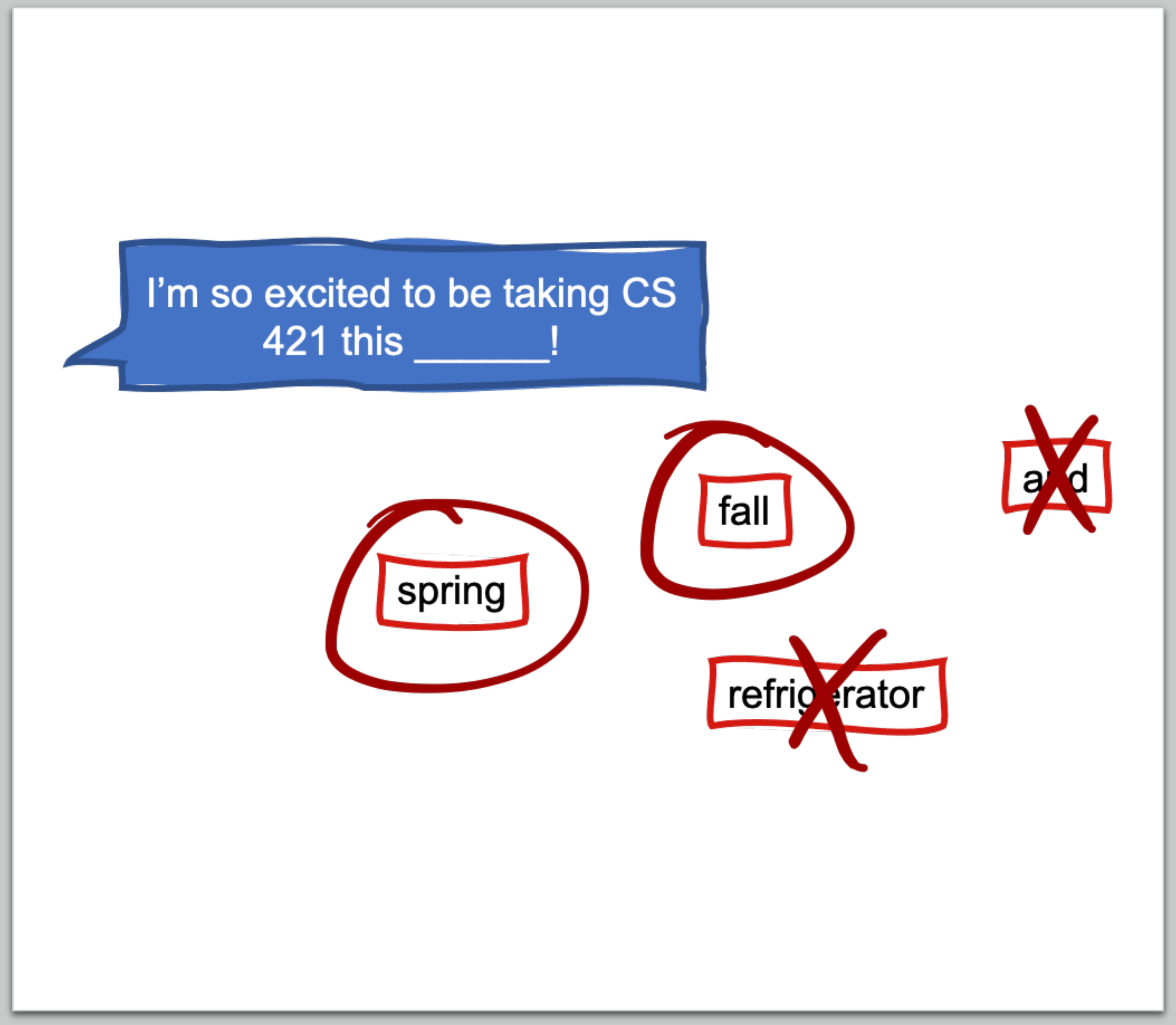
```
initialize A and B
iterate until convergence:

    # Expectation Step
    compute γ_t(j) for all t and j
    compute ζ_t(i,j) for all t, i, and j

    # Maximization Step
    α_ij = â_ij for all i and j
    b_j(v_k) = b̂_j(v_k) for all j, and all v_k in the output vocab V
```

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

Language Modeling

- The process of building models that predict the likelihood of word or character sequences in a language



Why is language modeling useful?

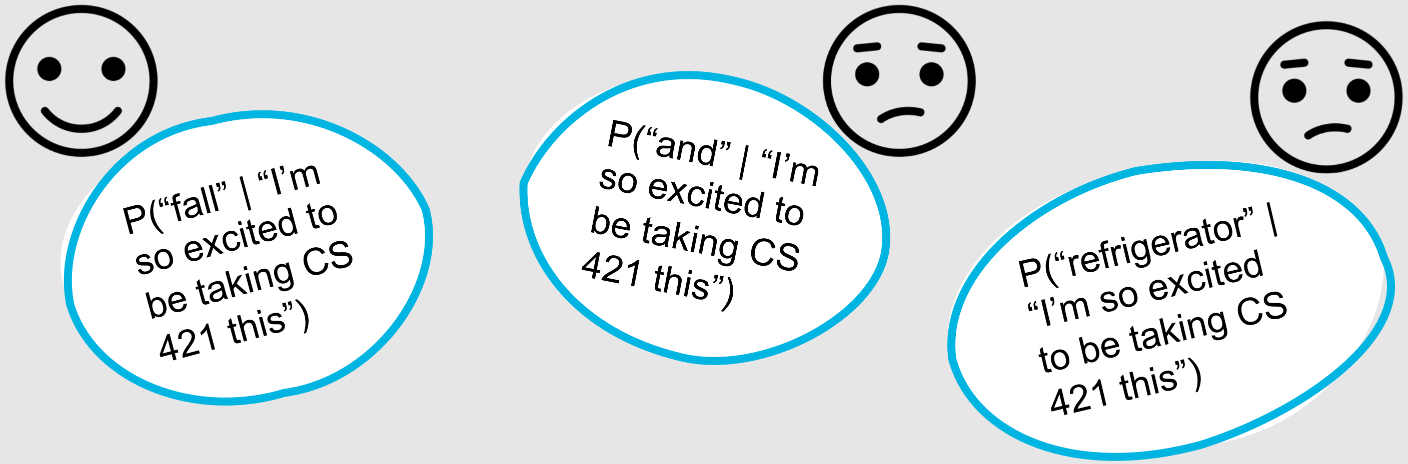
- Helps identify words in noisy, ambiguous input
  - Speech recognition or autocorrect
- Helps generate natural-sounding language
  - Machine translation or image captioning

Language models come in many forms!

- Less complex:
  - **N-gram language models**
- More sophisticated
  - Neural language models

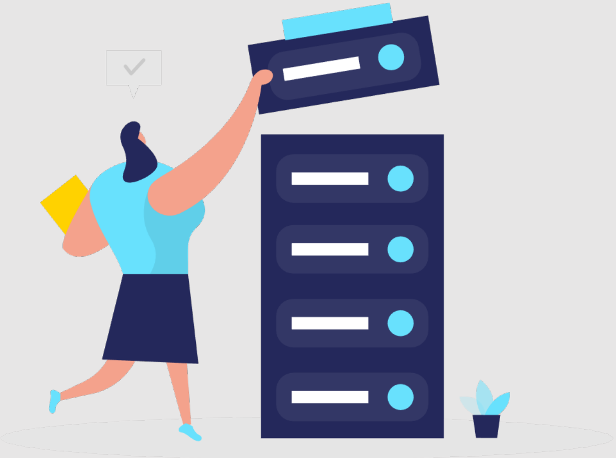
N-Gram Language Models

- Goal: Predict P(word|history)
  - P(“spring” | “I’m so excited to be taking CS 421 this”)



How do we predict these probabilities?

- One method: Estimate it from frequency counts
  - Take a large corpus
  - Count the number of times you see the history
  - Count the number of times the specified word follows the history



P("spring" | "I'm so excited to be taking CS 421 this")

= C("I'm so excited to be taking CS 421 this spring") / C("I'm so excited to be taking CS 421 this")

However, we don't necessarily want to use our *entire* history.

- What if our history contains uncommon words?
- What if we have limited computing resources?

P("spring" | "I'm so excited to be taking **Natalie Parde's** CS 421 this")

Out of all possible 11-word sequences on the web, how many are "I'm so excited to be taking Natalie Parde's CS 421 this"?

We need a better way to estimate P(word|history)!

- The solution: Instead of computing the probability of a word given its entire history, **approximate the history using the most recent few words**.
- We do this using fixed-length **n-grams**.

P("spring" | "taking CS 421 this")

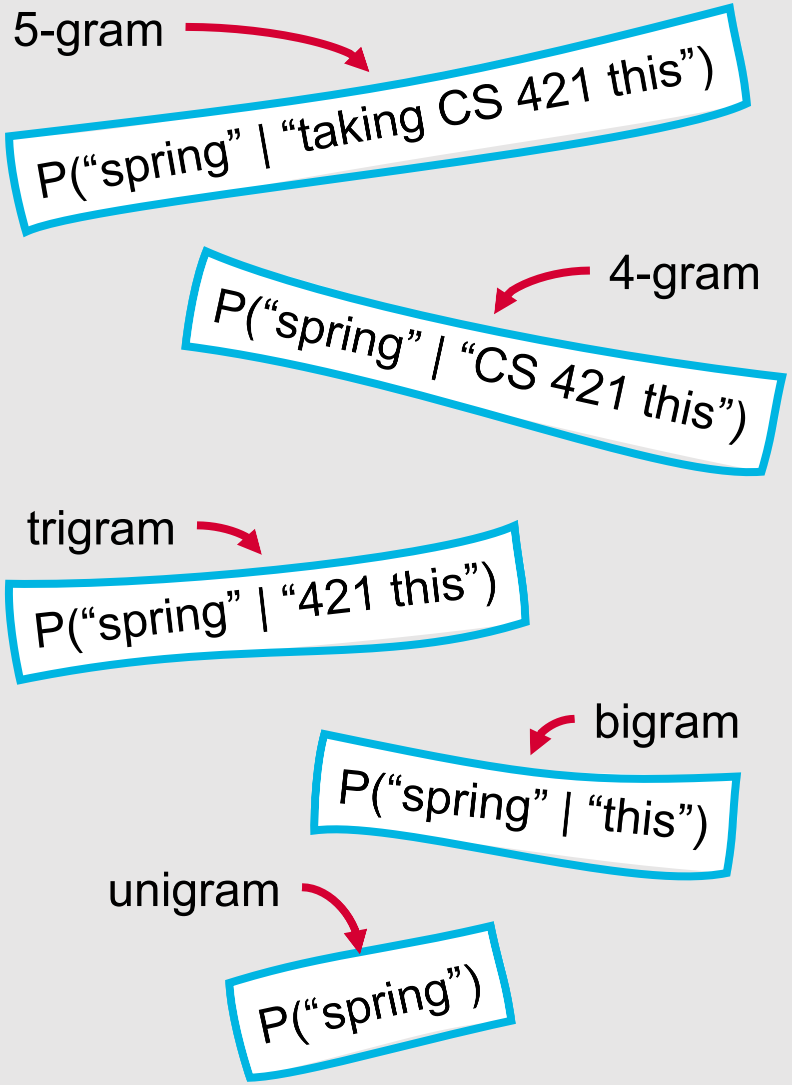
P("spring" | "CS 421 this")

P("spring" | "421 this")

P("spring" | "this")

Special N-Grams

- Most higher-order (n>3) n-grams are simply referred to using the value of *n*
  - 4-gram
  - 5-gram
- However, lower-order n-grams are often referred to using special terms:
  - Unigram (1-gram)
  - Bigram (2-gram)
  - Trigram (3-gram)



N-gram models follow the **Markov assumption**.

- We can predict the probability of some future unit without looking too far into the past
  - **Bigram language model**: Probability of a word depends only on the previous word
  - **Trigram language model**: Probability of a word depends only on the two previous words
  - **N-gram language model**: Probability of a word depends only on the *n*-1 previous words

More formally….

- $P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-N+1}^{n-1})$
- We can then multiply these individual word probabilities together to get the probability of a word sequence
  - $P(w_1^n) \approx \prod_{k=1}^{n} P(w_k|w_{k-N+1}^{k-1})$

P("Summer break is already over?")

P("over?" | "already") * P("already" | "is") * P("is" | "break") * P("break" | "Summer")

To compute n-gram probabilities, maximum likelihood estimation is often used.

- **Maximum Likelihood Estimation (MLE):**
  - Get the requisite n-gram frequency counts from a corpus
  - Normalize them to a 0-1 range
    - $P(w_n \mid w_{n-1})$ = # of occurrences of the bigram $w_{n-1}\ w_n$ / # of occurrences of the unigram w_{n-1}

Example: Maximum Likelihood Estimation

I am cold.

You are cold.

Everyone is cold.

This is Chicago.

Example: Maximum Likelihood Estimation

I am cold. → <s> I am cold. </s>

You are cold. → <s> You are cold. </s>

Everyone is cold. → <s> Everyone is cold. </s>

This is Chicago. → <s> This is Chicago. </s>

Example: Maximum Likelihood Estimation

I am cold. → <s> I am cold. </s>

You are cold. → <s> You are cold. </s>

Everyone is cold. → <s> Everyone is cold. </s>

This is Chicago. → <s> This is Chicago. </s>

Bigram	Frequency
<s> I	1
I am	1
am cold.	1
cold. </s>	3
…	…
is Chicago.	1
Chicago. </s>	1

Example: Maximum Likelihood Estimation

I am cold. → <s> I am cold. </s>

You are cold. → <s> You are cold. </s>

Everyone is cold. → <s> Everyone is cold. </s>

This is Chicago. → <s> This is Chicago. </s>

Bigram	Freq.
<s> I	1
I am	1
am cold.	1
cold. </s>	3
…	…
is Chicago.	1
Chicago. </s>	1

Unigram	Freq.
<s>	4
I	1
am	1
cold.	3
…	…
Chicago.	1
</s>	4

Example: Maximum Likelihood Estimation

I am cold. → <s> I am cold. </s>

You are cold. → <s> You are cold. </s>

Everyone is cold. → <s> Everyone is cold. </s>

This is Chicago. → <s> This is Chicago. </s>

Bigram	Freq.
<s> I	1
I am	1
am cold.	1
cold. </s>	3
…	…
is Chicago.	1
Chicago. </s>	1

Unigram	Freq.
<s>	4
I	1
am	1
cold.	3
…	…
Chicago.	1
</s>	4

P(“I” | “<s>”) = C(“<s> I”) / C(“<s>”) = 1 / 4 = 0.25

Example: Maximum Likelihood Estimation

I am cold. → <s> I am cold. </s>

You are cold. → <s> You are cold. </s>

Everyone is cold. → <s> Everyone is cold. </s>

This is Chicago. → <s> This is Chicago. </s>

Bigram	Freq.
<s> I	1
I am	1
am cold.	1
cold. </s>	3
…	…
is Chicago.	1
Chicago. </s>	1

Unigram	Freq.
<s>	4
I	1
am	1
cold.	3
…	…
Chicago.	1
</s>	4

P("I" | "<s>") = C("<s> I") / C("<s>") = 1 / 4 = 0.25

P("</s>" | "cold.") = C("cold. </s>") / C("cold.") = 3 / 3 = 1.00

Example: Maximum Likelihood Estimation

I am cold.	<s> I am cold. </s>
You are cold.	<s> You are cold. </s>
Everyone is cold.	<s> Everyone is cold. </s>
This is Chicago.	<s> This is Chicago. </s>

Bigram	Freq.
<s> I	1
I am	1
am cold.	1
cold. </s>	3
…	…
is Chicago.	1
Chicago. </s>	1

Unigram	Freq.
<s>	4
I	1
am	1
cold.	3
…	…
Chicago.	1
</s>	4

P(“I” | “<s>”) = C(“<s> I”) / C(“<s>”) = 1 / 4 = 0.25

P(“</s>” | “cold.”) = C(“cold. </s>”) / C(“cold.”) = 3 / 3 = 1.00 🤷‍♀️

We can learn a lot of useful things from n-gram statistics!

- Syntactic information
  - Do nouns often follow verbs?
  - Do verbs usually follow specific unigrams?
- Task-relevant information
  - Is it likely that virtual assistants will hear the word “I” in a user’s input?
- Cultural or sociological information
  - Are people likelier to want quesadillas than haggis?

Which type of n-gram is best?

- In general, the highest-order value of *n* that your data can support
- Sparsity increases with order, and sparse feature vectors are not very useful when training statistical models
- Make sure that your dataset is large enough to handle your selected n-gram size

We've learned how to build n-gram language models, but how do we evaluate them?

- Two types of evaluation paradigms:
  - Extrinsic
  - Intrinsic
- **Extrinsic evaluation:** Embed the language model in an application, and compute changes in task performance
- **Intrinsic evaluation:** Measure the quality of the model, independent of any application

Perplexity

- Intrinsic evaluation metric for language models
- Perplexity (PP) of a language model on a test set is the **inverse probability of the test set**, normalized by the number of words in the test set

More formally….

- $PP(W) = \sqrt[n]{\frac{1}{P(w_1 w_2 \ldots w_n)}} = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P(w_i | w_1 \ldots w_{i-1})}}$
  - Where *W* is a test set containing words w_1, w_2, …, w_n
  - History size depends on n-gram size
    - $P(w_i | w_{i-1})$ vs $P(w_i | w_{i-2} w_{i-1})$, etc.
- Higher conditional probability of a word sequence → lower perplexity
  - Minimizing perplexity = maximizing test set probability according to the language model

Example: Perplexity

Training Set

Word	Frequency
CS	10
421	10
Statistical	10
Natural	10
Language	10
Processing	10
University	10
of	10
Illinois	10
Chicago	10

Example: Perplexity

Training Set

Word	Frequency
CS	10
421	10
Statistical	10
Natural	10
Language	10
Processing	10
University	10
of	10
Illinois	10
Chicago	10

Test String

CS 421 Statistical Natural Language Processing University of Illinois Chicago

Example: Perplexity

Training Set

Word	Frequency
CS	10
421	10
Statistical	10
Natural	10
Language	10
Processing	10
University	10
of	10
Illinois	10
Chicago	10

Test String

CS 421 Statistical Natural Language Processing University of Illinois Chicago

$$PP(W) = \sqrt[n]{\frac{1}{P(w_1 w_2 \ldots w_n)}} = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P(w_i | w_1 \ldots w_{i-1})}}$$

Example: Perplexity

Training Set

Word	Frequency
CS	10
421	10
Statistical	10
Natural	10
Language	10
Processing	10
University	10
of	10
Illinois	10
Chicago	10

Test String

CS 421 Statistical Natural Language Processing University of Illinois Chicago

$$PP(W) = \sqrt[n]{\frac{1}{P(w_1 w_2 \dots w_n)}} = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P(w_i | w_1 \dots w_{i-1})}}$$

P("CS") = C("CS") / C(<all unigrams>) = 10/100 = 0.1

Example: Perplexity

Training Set

Word	Frequency
CS	10
421	10
Statistical	10
Natural	10
Language	10
Processing	10
University	10
of	10
Illinois	10
Chicago	10

Test String

CS 421 Statistical Natural Language Processing University of Illinois Chicago

$$PP(W) = \sqrt[n]{\frac{1}{P(w_1 w_2 \dots w_n)}} = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P(w_i | w_1 \dots w_{i-1})}}$$

P("CS") = C("CS") / C(<all unigrams>) = 10/100 = 0.1

P("421") = C("421") / C(<all unigrams>) = 10/100 = 0.1

Example: Perplexity

Training Set

Word	Frequency	P(Word)
CS	10	0.1
421	10	0.1
Statistical	10	0.1
Natural	10	0.1
Language	10	0.1
Processing	10	0.1
University	10	0.1
of	10	0.1
Illinois	10	0.1
Chicago	10	0.1

Test String

CS 421 Statistical Natural Language Processing University of Illinois Chicago

$$PP(W) = \sqrt[n]{\frac{1}{P(w_1 w_2 \dots w_n)}} = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P(w_i | w_1 \dots w_{i-1})}}$$

Example: Perplexity

Training Set

Word	Frequency	P(Word)
CS	10	0.1
421	10	0.1
Statistical	10	0.1
Natural	10	0.1
Language	10	0.1
Processing	10	0.1
University	10	0.1
of	10	0.1
Illinois	10	0.1
Chicago	10	0.1

Test String

CS 421 Statistical Natural Language Processing University of Illinois Chicago

$$PP(W) = \sqrt[n]{\frac{1}{P(w_1 w_2 \dots w_n)}} = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P(w_i | w_1 \dots w_{i-1})}}$$

PP("CS 421 Statistical Natural Language Processing University of Illinois Chicago")

$$= \sqrt[10]{\frac{1}{0.1*0.1*0.1*0.1*0.1*0.1*0.1*0.1*0.1*0.1}} = 10$$

Example: Perplexity

Training Set

Word	Frequency	P(Word)
CS	1	
421	1	
Statistical	1	
Natural	1	
Language	1	
Processing	1	
University	1	
of	1	
Illinois	1	
Chicago	91	

Test String

Illinois Chicago Chicago Chicago Chicago Chicago Chicago Chicago Chicago Chicago

$$PP(W) = \sqrt[n]{\frac{1}{P(w_1 w_2 \dots w_n)}} = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P(w_i | w_1 \dots w_{i-1})}}$$

Example: Perplexity

Training Set

Word	Frequency	P(Word)
CS	1	0.01
421	1	0.01
Statistical	1	0.01
Natural	1	0.01
Language	1	0.01
Processing	1	0.01
University	1	0.01
of	1	0.01
Illinois	1	0.01
Chicago	91	0.91

Test String

Illinois Chicago Chicago Chicago Chicago Chicago Chicago Chicago Chicago Chicago

$$PP(W) = \sqrt[n]{\frac{1}{P(w_1 w_2 \ldots w_n)}} = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P(w_i|w_1 \ldots w_{i-1})}}$$

Example: Perplexity

Training Set

Word	Frequency	P(Word)
CS	1	0.01
421	1	0.01
Statistical	1	0.01
Natural	1	0.01
Language	1	0.01
Processing	1	0.01
University	1	0.01
of	1	0.01
Illinois	1	0.01
Chicago	91	0.91

Test String

Illinois Chicago Chicago Chicago Chicago Chicago Chicago Chicago Chicago Chicago

$$PP(W) = \sqrt[n]{\frac{1}{P(w_1 w_2 \dots w_n)}} = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P(w_i | w_1 \dots w_{i-1})}}$$

PP("CS 521 Statistical Natural Language Processing University of Illinois Chicago")

$$= \sqrt[10]{\frac{1}{0.01*0.91*0.91*0.91*0.91*0.91*0.91*0.91*0.91*0.91}} = 1.73$$

Perplexity can be used to compare different language models.

Which language model is best?

- Model A: Perplexity = 962
- Model B: Perplexity = 170
- Model C: Perplexity = 109

Perplexity can be used to compare different language models.

Which language model is best?

- Model A: Perplexity = 962
- Model B: Perplexity = 170
- Model C: Perplexity = 109

A cautionary note….

- Improvements in perplexity do not guarantee improvements in task performance!
- However, the two are often correlated (and perplexity is quicker and easier to check)
- Strong language model evaluations also include an extrinsic evaluation component

Just like with HMMs, we can use n-gram language models to generate text.

1. Select an n-gram randomly from the distribution of all n-grams in the training corpus

2. Randomly select an n-gram from the same distribution, dependent on the previous n-gram
   - If we're using a bigram model and the previous bigram was "CS 421," our next bigram has to start with "421")

3. Repeat until the sentence-final token is reached

N-gram order affects generation output!

Unigram — No coherence between words

- To him swallowed confess hear both. Of save on trail for are ay device and rote life have
- Hill he late speaks; or! a more to leg less first you enter

Bigram — Minimal local coherence between words

- Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow.
- What means, sir. I confess she? then all sorts, he is trim, captain.

Trigram — More coherence….

- Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done.
- This shall forbid it should be branded, if renown made it empty.

4-gram — Direct quote from Shakespeare

- King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in;
- It cannot be but so.

In the case of a Shakespearean language model….

- Recall that the corpus of all Shakespearean text is relatively small (by modern NLP standards)
  - 29,066 vocabulary words
  - 884,647 tokens
- This means higher-order n-gram matrices are sparse!
  - Only five possible continuations for “It cannot be but” (“that,” “I,” “he,” “thou,” and “so”) …probability for all other continuations is assumed to be zero

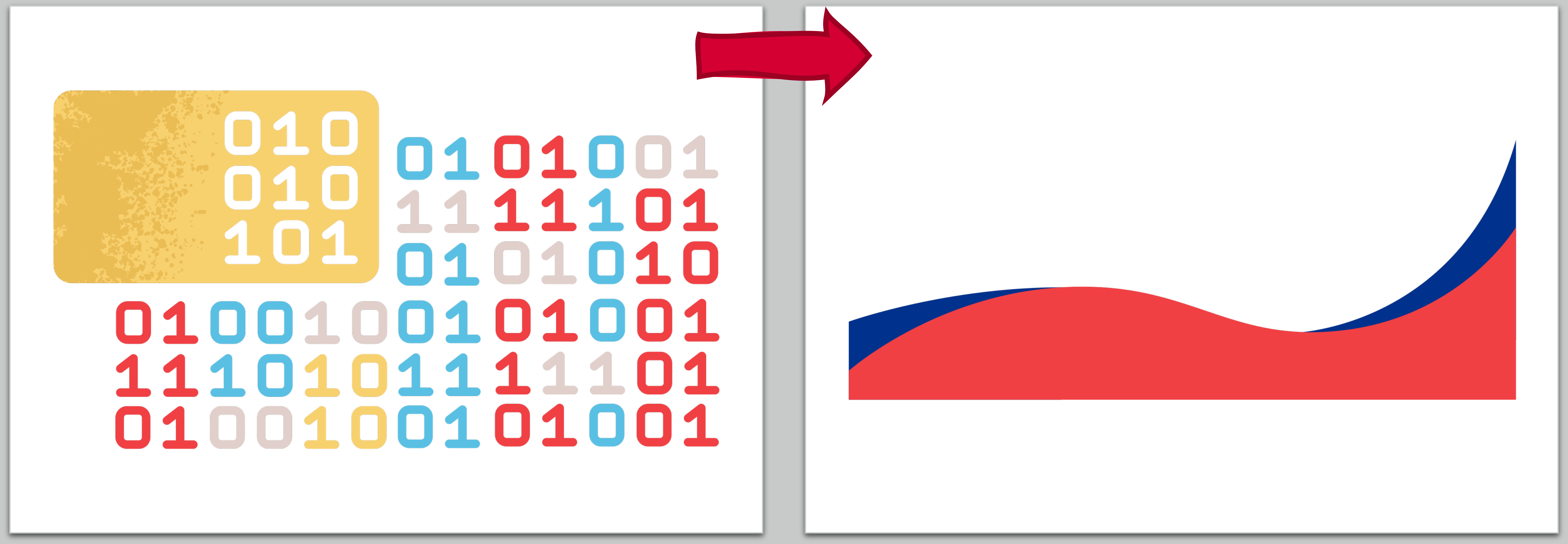
“Zero” probabilities create challenges for language models.

- In practice, language is varied and often unexpected---few combinations are truly impossible
- Zero probabilities also interfere with perplexity calculations
- Zero probabilities occur in two different scenarios:
  - **Unknown words** (**out-of-vocabulary** words)
  - Known words in **unseen contexts**

Modeling Unknown Words

- Add a pseudoword **<UNK>** to the vocabulary
- Then….
    - Option A:
        - Choose a fixed words list
        - Convert any words not in that list to <UNK>
        - Estimate the probabilities for <UNK> like any other word
    - Option B:
        - Replace all words occurring fewer than n times with <UNK>
        - Estimate the probabilities for <UNK> like any other word
    - Option C:
        - Replace the first occurrence of each word with <UNK>
        - Estimate the probabilities for <UNK> like any other word
- Beware: If <UNK> ends up with a high probability (e.g., because you have a small vocabulary), your language model will have artificially lower perplexity!
    - Make sure to compare to other language models using the same vocabulary to avoid gaming this metric

We can handle known words in previously unseen contexts by applying smoothing techniques.



Smoothing

- Taking a bit of the probability mass from more frequent events and giving it to unseen events.
  - Sometimes also called "discounting"
- Many different smoothing techniques:
  - Laplace (add-one)
  - Add-k
  - Stupid backoff
  - Kneser-Ney

Bigram	Frequency
CS 421	8
CS 590	5
CS 594	2
CS 521	0 😢

Bigram	Frequency
CS 421	7
CS 590	5
CS 594	2
CS 521	1 🥰

Laplace Smoothing

- Add one to all n-gram counts before they are normalized into probabilities
- Not the highest-performing technique for language modeling, but a useful baseline
  - Practical method for other text classification tasks
- $P(w_i) = \frac{c_i}{N} \rightarrow P_{\text{Laplace}}(w_i) = \frac{c_i+1}{N+V}$

Example: Laplace Smoothing

Corpus Statistics:

Unigram	Frequency
Chicago	4
is	8
cold	6
hot	0

Bigram	Frequency
Chicago is	2
is cold	4
is hot	0
…	0

Example: Laplace Smoothing

Corpus Statistics:

Unigram	Frequency
Chicago	4
is	8
cold	6
hot	0

Bigram	Frequency
Chicago is	2
is cold	4
is hot	0
…	0

$$P(w_i) = \frac{c_i}{N}$$

Unigram	Probability
Chicago	$\frac{4}{18} = 0.22$
is	$\frac{8}{18} = 0.44$
cold	$\frac{6}{18} = 0.33$
hot	$\frac{0}{18} = 0.00$

Bigram	Probability
Chicago is	
is cold	
is hot	

Example: Laplace Smoothing

Corpus Statistics:

Unigram	Frequency
Chicago	4
is	8
cold	6
hot	0

Bigram	Frequency
Chicago is	2
is cold	4
is hot	0
…	0

$$P(w_i) = \frac{c_i}{N}$$

Unigram	Probability
Chicago	$\frac{4}{18} = 0.22$
is	$\frac{8}{18} = 0.44$
cold	$\frac{6}{18} = 0.33$
hot	$\frac{0}{18} = 0.00$

Bigram	Probability
Chicago is	$\frac{2}{4} = 0.50$
is cold	$\frac{4}{8} = 0.50$
is hot	$\frac{0}{8} = 0.00$

Example: Laplace Smoothing

Corpus Statistics:

Unigram	Frequency
Chicago	4
is	8
cold	6
hot	0

Bigram	Frequency
Chicago is	2
is cold	4
is hot	0
…	0

$$P(w_i) = \frac{c_i}{N} \rightarrow P_{\text{Laplace}}(w_i) = \frac{c_i+1}{N+V}$$

Unigram	Probability
Chicago	
is	
cold	
hot	

Bigram	Probability
Chicago is	
is cold	
is hot	

Example: Laplace Smoothing

Corpus Statistics:

Unigram	Frequency
Chicago	4+1
is	8+1
cold	6+1
hot	0+1

Bigram	Frequency
Chicago is	2+1
is cold	4+1
is hot	0+1
…	0+1

$$P(w_i) = \frac{c_i}{N} \rightarrow P_{\text{Laplace}}(w_i) = \frac{c_i+1}{N+V}$$

Unigram	Probability
Chicago	
is	
cold	
hot	

Bigram	Probability
Chicago is	
is cold	
is hot	

Example: Laplace Smoothing

Corpus Statistics:

Unigram	Frequency
Chicago	4+1
is	8+1
cold	6+1
hot	0+1

Bigram	Frequency
Chicago is	2+1
is cold	4+1
is hot	0+1
…	0+1

$$P(w_i) = \frac{c_i}{N} \rightarrow P_{\text{Laplace}}(w_i) = \frac{c_i+1}{N+V}$$

Unigram	Probability
Chicago	$\frac{5}{22} = 0.23$
is	$\frac{9}{22} = 0.41$
cold	$\frac{7}{22} = 0.32$
hot	$\frac{1}{22} = 0.05$

Bigram	Probability
Chicago is	
is cold	
is hot	

Example: Laplace Smoothing

Corpus Statistics:

Bigram	Frequency
Chicago Chicago	0+1
Chicago is	2+1
Chicago cold	0+1
Chicago hot	0+1

$$P(w_i) = \frac{c_i}{N} \rightarrow P_{\text{Laplace}}(w_i) = \frac{c_i+1}{N+V}$$

Unigram	Frequency
Chicago	4
is	8
cold	6
hot	0

Bigram	Frequency
Chicago is	2+1
is cold	4+1
is hot	0+1
…	0+1

Unigram	Probability
Chicago	$\frac{5}{22} = 0.23$
is	$\frac{9}{22} = 0.41$
cold	$\frac{7}{22} = 0.32$
hot	$\frac{1}{22} = 0.05$

Bigram	Probability
Chicago is	$\frac{3}{4+4} = \frac{3}{8} = 0.38$
is cold	$\frac{5}{8+4} = \frac{5}{12} = 0.42$
is hot	$\frac{1}{8+4} = \frac{1}{12} = 0.08$

Probabilities: Before and After

Bigram	Probability
Chicago is	$\frac{2}{4} = 0.50$
is cold	$\frac{4}{8} = 0.50$
is hot	$\frac{0}{8} = 0.00$

Bigram	Probability
Chicago is	$\frac{3}{8} = 0.38$
is cold	$\frac{5}{12} = 0.42$
is hot	$\frac{1}{12} = 0.08$

Add-K Smoothing

- Moves a bit less of the probability mass from seen to unseen events
- Rather than adding one to each count, add a fractional count
  - 0.5
  - 0.05
  - 0.01
- The value *k* can be optimized on a validation set
- $P(w_i) = \frac{c_i}{N} \rightarrow P_{\text{Add-K}}(w_i) = \frac{c_i+k}{N+kV}$
- $P(w_n|w_{n-1}) = \frac{c(w_{n-1}w_n)}{c(w_{n-1})} \rightarrow P_{\text{Add-K}}(w_n|w_{n-1}) = \frac{c(w_{n-1}w_n)+k}{c(w_{n-1})+kV}$

Add-K smoothing is useful for some tasks, but still tends to be suboptimal for language modeling.

- Other smoothing techniques?
  - **Backoff:** Use the specified n-gram size to estimate probability if its count is greater than 0; otherwise, *backoff* to a lower-order n-gram
  - **Interpolation:** Mix the probability estimates from multiple n-gram sizes, weighing and combining the n-gram counts

Interpolation

N	Weight
3	0.5
2	0.4
1	0.1

N-Gram	Probability	Value
I ❤️ 421	P(421 | I ❤️)	0.7
❤️ 421	P(421 | ❤️)	0.5
421	P(421)	0.2

$\boxed{0.5 * 0.7 + 0.4 * 0.5 + 0.1 * 0.2 = 0.57}$

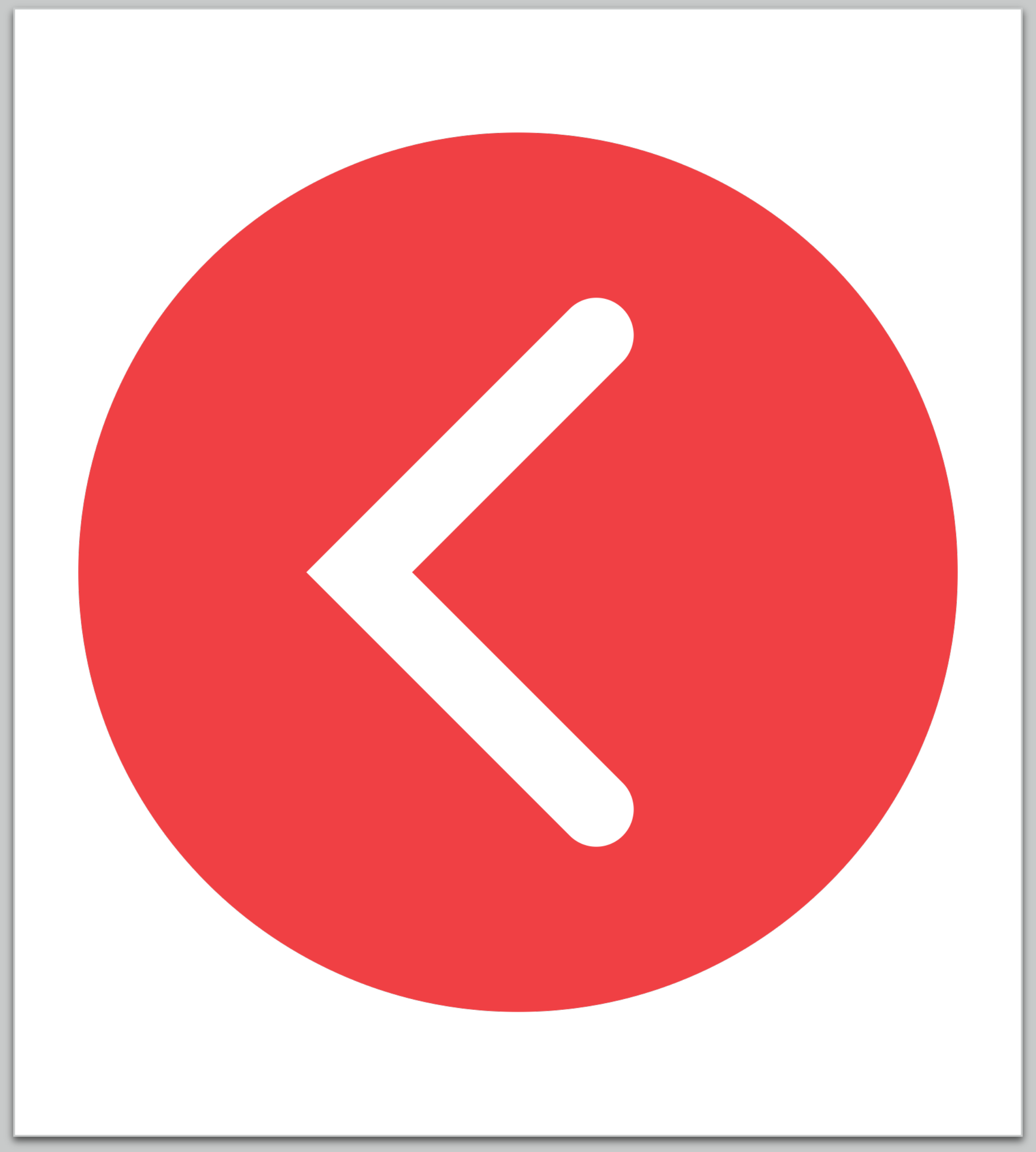
- **Linear interpolation**
  - $P'(w_n|w_{n-2}w_{n-1}) = \lambda_1 P(w_n|w_{n-2}w_{n-1}) + \lambda_2 P(w_n|w_{n-1}) + \lambda_3 P(w_n)$
    - Where $\sum_i \lambda_i = 1$
- **Conditional interpolation**
  - $P'(w_n|w_{n-2}w_{n-1}) = \lambda_1(w_{n-2}^{n-1}) P(w_n|w_{n-2}w_{n-1}) + \lambda_2(w_{n-2}^{n-1}) P(w_n|w_{n-1}) + \lambda_3(w_{n-2}^{n-1}) P(w_n)$

Context-conditioned weights

N-Gram	Probability	Value	Weight
I ❤️ 421	P(421 | I ❤️)	0.7	0.5
I 🚕 421	P(421 | I 🚕)	0.7	0.1

Backoff

- If the n-gram we need has zero counts, approximate it by backing off to the (n-1)-gram
- Continue backing off until we reach a size that has non-zero counts
- Just like with smoothing, some probability mass from higher-order n-grams needs to be redistributed to lower-order n-grams



Katz Backoff

- Incorporate a function α to distribute probability mass to lower-order n-grams
- Rely on a discounted probability P* if the n-gram has non-zero counts
- Otherwise, recursively back off to the Katz probability for the (n-1)-gram
- $P_{BO}(w_n|w_{n-N+1}^{n-1}) = \begin{cases} P^*(w_n|w_{n-N+1}^{n-1}), & \text{if } c(w_{n-N+1}^{n}) > 0 \\ \alpha(w_{n-N+1}^{n-1}) P_{BO}(w_n|w_{n-N+2}^{n-1}), & \text{otherwise} \end{cases}$

Kneser-Ney Smoothing

- One of the most commonly used and best-performing n-gram smoothing methods
- Incorporates absolute discounting
  - $P_{\text{AbsoluteDiscounting}}(w_i|w_{i-1}) = \underbrace{\frac{C(w_{i-1}w_i)-d}{\sum_v C(w_{i-1}v)}}_{\text{Discounted Bigram}} + \underbrace{\lambda(w_{i-1})P(w_i)}_{\text{Unigram with interpolation weight}}$

Kneser-Ney Smoothing

- Objective: Capture the intuition that although some lower-order n-grams are frequent, they are mainly only frequent in specific contexts
  - tall nonfat decaf peppermint ________
    - "york" is a more frequent unigram than "mocha" (7.4 billion results vs. 135 million results on Google), but it's mainly frequent when it follows the word "new"
- Creates a unigram model that estimates the probability of seeing the word *w* as a novel continuation, in a new unseen context
  - Based on the number of different contexts in which *w* has already appeared
  - $P_{\text{Continuation}}(w) = \frac{|\{v : C(vw) > 0\}|}{|\{(u', w') : C(u'w') > 0\}|}$

Kneser-Ney Smoothing

$$P_{\mathrm{KN}}(w_i|w_{i-n+1}^{i-1}) = \frac{\max\left(c_{KN}\left(w_{i-n+1}^{i}\right) - d,\ \ 0\right)}{\sum_v c_{KN}\left(w_{i-n+1}^{i-1}v\right)} + \lambda(w_{i-n+1}^{i-1})P_{\mathrm{KN}}(w_i|w_{i-n+2}^{i-1})$$

Kneser-Ney Smoothing

$$P_{\mathrm{KN}}(w_i|w_{i-n+1}^{i-1}) = \frac{\max\left(c_{KN}\left(w_{i-n+1}^{i}\right) - d,\ \ 0\right)}{\sum_v c_{KN}\left(w_{i-n+1}^{i-1} v\right)} + \lambda(w_{i-n+1}^{i-1}) P_{\mathrm{KN}}(w_i|w_{i-n+2}^{i-1})$$

Normalizing constant to distribute the probability mass that's been discounted

$$\lambda(w_{i-1}) = \frac{d}{\sum_v C(w_{i-1}v)} |\{w : c(w_{i-1}w) > 0\}|$$

Kneser-Ney Smoothing

$$P_{\text{KN}}(w_i|w_{i-n+1}^{i-1}) = \frac{\max\left(c_{KN}\left(w_{i-n+1}^{i}\right) - d,\ \ 0\right)}{\sum_v c_{KN}\left(w_{i-n+1}^{i-1}v\right)} + \lambda(w_{i-n+1}^{i-1})P_{\text{KN}}(w_i|w_{i-n+2}^{i-1})$$

Normalizing constant to distribute the probability mass that's been discounted

$$\lambda(w_{i-1}) = \frac{d}{\sum_v C(w_{i-1}v)}|\{w : c(w_{i-1}w) > 0\}|$$

Normalized discount

Number of word types that can follow w_{i-1}

Kneser-Ney Smoothing

$$P_{\mathrm{KN}}(w_i|w_{i-n+1}^{i-1}) = \frac{\max(c_{KN}(w_{i-n+1}^{i}) - d,\ \ 0)}{\sum_v c_{KN}(w_{i-n+1}^{i-1}v)} + \lambda(w_{i-n+1}^{i-1})P_{\mathrm{KN}}(w_i|w_{i-n+2}^{i-1})$$

Normalizing constant to distribute the probability mass that's been discounted

Regular count for the highest-order n-gram, or the number of unique single word contexts for lower-order n-grams

Kneser-Ney Smoothing

$$P_{\mathrm{KN}}(w_i|w_{i-n+1}^{i-1}) = \frac{\max(c_{KN}(w_{i-n+1}^{i}) - d,\ \ 0)}{\sum_v c_{KN}(w_{i-n+1}^{i-1}v)} + \lambda(w_{i-n+1}^{i-1}) P_{\mathrm{KN}}(w_i|w_{i-n+2}^{i-1})$$

Normalizing constant to distribute the probability mass that's been discounted

Regular count for the highest-order n-gram, or the number of unique single word contexts for lower-order n-grams

Discounted n-gram probability …when the recursion terminates, unigrams are interpolated with the uniform distribution (ε = empty string)

$$P_{KN}(w) = \frac{\max(c_{KN}(w) - d, 0)}{\sum_{w'} c_{KN}(w')} + \lambda(\varepsilon)\frac{1}{V}$$

Stupid Backoff

- Gives up the idea of trying to make the language model a true probability distribution 😌
- No discounting of higher-order probabilities
- If a higher-order n-gram has a zero count, simply backoff to a lower-order n-gram, weighted by a fixed weight
- $S\left(w_i \middle| w_{i-k+1}^{i-1}\right) = \begin{cases} \frac{c(w_{i-k+1}^{i})}{c(w_{i-k+1}^{i-1})} & \text{if } c\left(w_{i-k+1}^{i}\right) > 0 \\ \lambda S\left(w_i \middle| w_{i-k+2}^{i-1}\right) & \text{otherwise} \end{cases}$
  - Terminates in the unigram, which has the probability:
    - $S(w) = \frac{c(w)}{N}$

Generally, 0.4 works well (Brants et al., 2007)

Summary: Language Modeling with N-Grams

- **N-grams:** Sequences of *n* letters
- **Language models:** Statistical models of language based on observed word or character co-occurrences
- N-gram probabilities can be computed using **maximum likelihood estimation**
- Language models can be **intrinsically evaluated** using **perplexity**
- Unknown words can be handled using **<UNK>** tokens
- Known words in unseen contexts can be handled using **smoothing**