Introduction to
Natural Language Processing (600.465)

Markov Models

Dr. Jan Hajič
CS Dept., Johns Hopkins Univ.
hajic@cs.jhu.edu
www.cs.jhu.edu/~hajic

Review: Markov Process

- Bayes formula (chain rule):

$$P(W) = P(w_1,w_2,\ldots,w_T) = \prod_{i=1..T} p(w_i|w_1,w_2,\ldots,w_{i-n+1},\ldots,w_{i-1})$$

approximation

- n-gram language models:
  - Markov process (chain) of the order n-1:

$$P(W) = P(w_1,w_2,\ldots,w_T) = \prod_{i=1..T} p(w_i|w_{i-n+1},w_{i-n+2},\ldots,w_{i-1})$$

Using just <u>one</u> distribution (Ex.: trigram model: $p(w_i|w_{i-2},w_{i-1})$):

Positions: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Words: My car **broke down ,** and within hours Bob 's car **broke down ,** too .

$$p(,|\text{broke down}) = p(w_5|w_3,w_4)) = p(w_{14}|w_{12},w_{13})$$

Markov Properties

- Generalize to any process (not just words/LM):
  - Sequence of random variables: $X = (X_1, X_2, \ldots, X_T)$
  - Sample space S (*states*), size N: $S = \{s_0, s_1, s_2, \ldots, s_N\}$

1. Limited History (Context, Horizon):

$$\forall i \in 1..T;\ P(X_i|X_1,\ldots,X_{i-1}) = P(X_i|X_{i-1})$$

1 7 3 7 9 0 6 7 3 4 5... 1 7 3 7 9 0 6 7 3 4 5...

2. Time invariance (M.C. is stationary, homogeneous)

$$\forall i \in 1..T,\ \forall y,x \in S;\ P(X_i=y|X_{i-1}=x) = p(y|x)$$

1 7 3 7 9 0 6 7 3 4 5...

? ok...same ***distribution***

Long History Possible

- What if we want trigrams:

1 7 3 7 9 0 6 7 3 4 5...

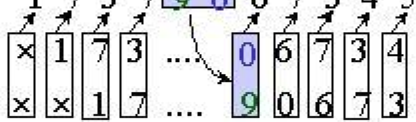
- Formally, use transformation:

Define new variables Q_i, such that $X_i = \{Q_{i-1}, Q_i\}$:

Then

$$P(X_i|X_{i-1}) = P(Q_{i-1}, Q_i|Q_{i-2}, Q_{i-1}) = P(Q_i|Q_{i-2}, Q_{i-1})$$

Predicting (X_i): 1 7 3 7 9 0 6 7 3 4 5...

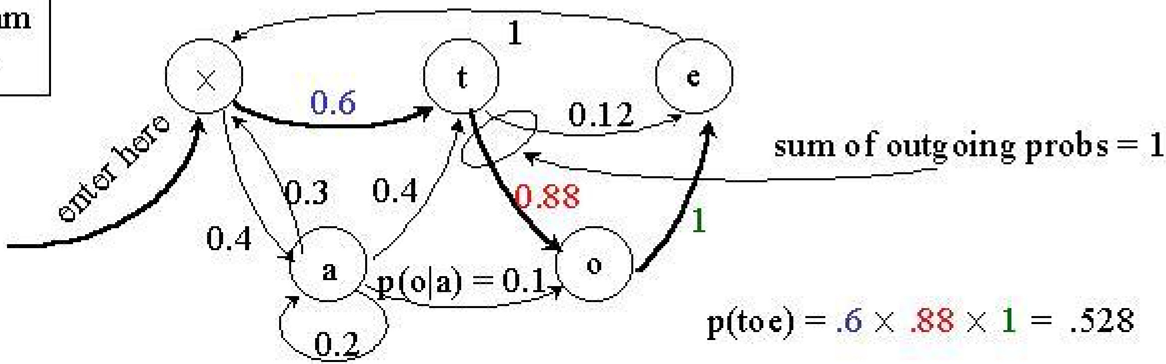


History ($X_{i-1} = \{Q_{i-2}, Q_{i-1}\}$):

Graph Representation: State Diagram

- $S = \{s_0, s_1, s_2, \ldots, s_N\}$: states
- Distribution $P(X_i|X_{i-1})$:
  - transitions (as arcs) with probabilities attached to them:

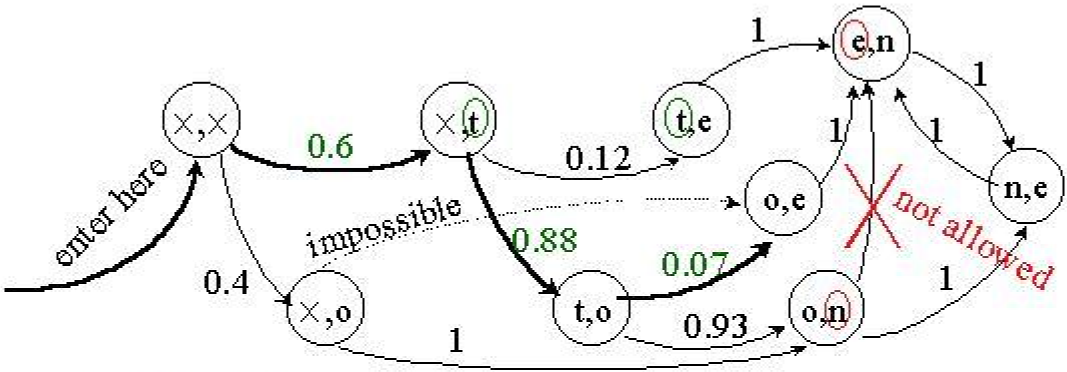
Bigram case:



$p(toe) = .6 \times .88 \times 1 = .528$

The Trigram Case

- $S = \{s_0, s_1, s_2, \ldots, s_N\}$: states: pairs $s_i = (x,y)$
- Distribution $P(X_i|X_{i-1})$: (r.v. X: generates pairs s_i)



p(toe) = $.6 \times .88 \times .07 \cong .037$

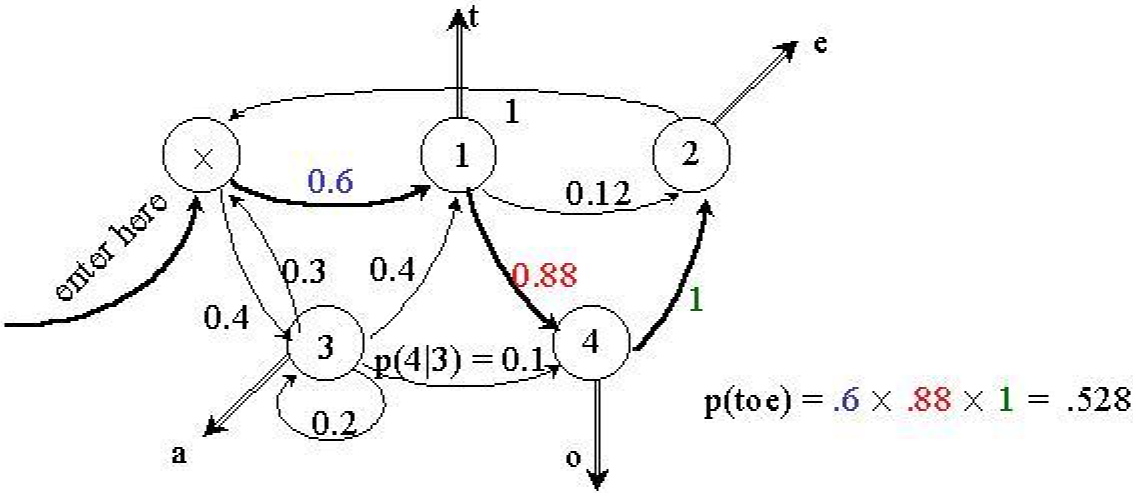
p(one) = ?

Finite State Automaton

- States ~ symbols of the [input/output] alphabet
  – pairs (or more): last element of the n-tuple
- Arcs ~ transitions (sequence of states)
- [Classical FSA: alphabet symbols on arcs:
  – transformation: arcs ↔ nodes]
- Possible thanks to the "limited history" M'ov Property
- So far: <u>***Visible***</u> Markov Models (VMM)

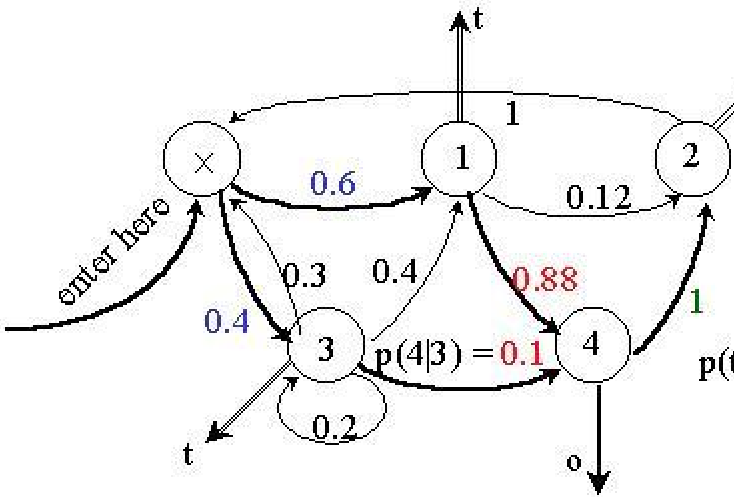
Hidden Markov Models

- The simplest HMM: states generate [observable] output (using the "data" alphabet) but remain "invisible":



Added Flexibility

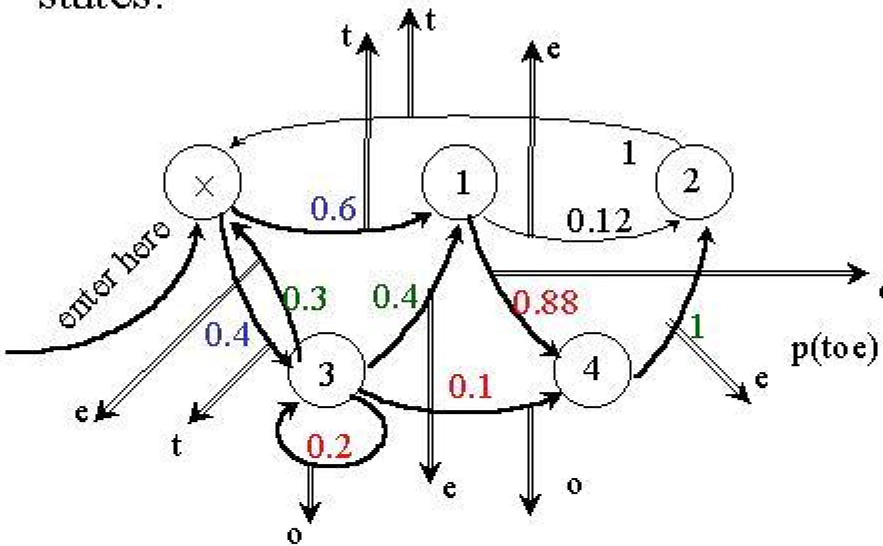
- So far, no change; but different states may generate the same output (why not?):



p(toe) = .6 × .88 × 1 +
.4 × .1 × 1 = .568

Output from Arcs...

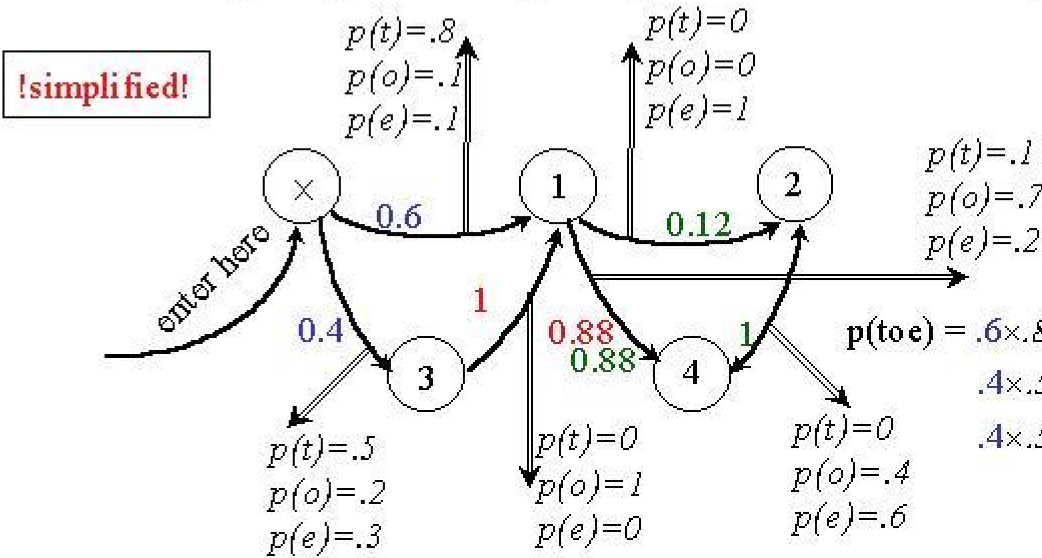
- Added flexibility: Generate output from arcs, not states:



$$p(toe) = .6 \times .88 \times 1 + .4 \times .1 \times 1 + .4 \times .2 \times .3 + .4 \times .2 \times .4 = .624$$

... and Finally, Add Output Probabilities

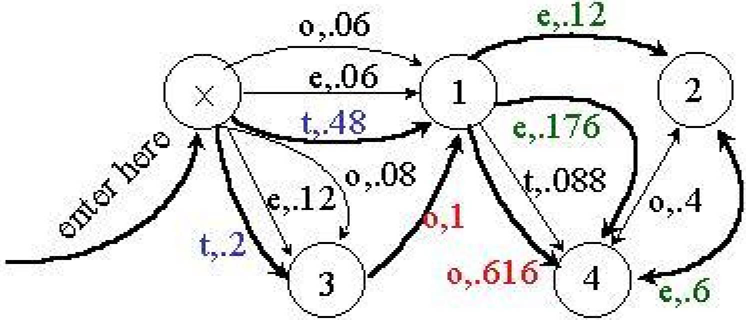
- Maximum flexibility: [Unigram] distribution (sample space: output alphabet) at each output arc:



p(toe) = .6×.8×.88×.7×1×.6 +
.4×.5×1×1×.88×.2 +
.4×.5×1×1×.12×1
≅ .237

Slightly Different View

- Allow for multiple arcs from $s_i \rightarrow s_j$, mark them by output symbols, get rid of output distributions:



p(toe) = .48×.616×.6+ .2×1×.176 + .2×1×.12 ≅ .237

In the future, we will use the view more convenient for the problem at hand.

Formalization

- HMM (the most general case):
  - five-tuple (S, s_0, Y, P_S, P_Y), where:
    - $S = \{s_0, s_1, s_2, \ldots, s_T\}$ is the set of states, s_0 is the initial state,
    - $Y = \{y_1, y_2, \ldots, y_V\}$ is the output alphabet,
    - $P_S(s_j|s_i)$ is the set of prob. distributions of transitions,
      - size of P_S : $|S|^2$.
    - $P_Y(y_k|s_i,s_j)$ is the set of output (emission) probability distributions.
      - size of P_Y: $|S|^2 \times |Y|$
- Example:
  - $S = \{x, 1, 2, 3, 4\}$, $s_0 = x$
  - $Y = \{ t, o, e \}$

Formalization - Example

- Example (for graph, see foils 11,12):
  - $S = \{x, 1, 2, 3, 4\}$, $s_0 = x$
  - $Y = \{ e, o, t \}$
  - P_S:

	x	1	2	3	4
x	0	.6	0	.4	0
1	0	0	.12	0	.88
2	0	0	0	0	1
3	0	1	0	0	0
4	0	0	1	0	0

$\Sigma = 1$

P_Y:

e	x	1	2	3	4
x					.2

o	x	1	2	3	4
x					.7

t	x	1	2	3	4
x		.8		.5	
1			0		.1
2					0
3		0			
4			0		

$\Sigma = 1$

Using the HMM

- The generation algorithm (of limited value :-)):
  1. Start in $s = s_0$.
  2. Move from s to s' with probability $P_S(s'|s)$.
  3. Output (emit) symbol y_k with probability $P_S(y_k|s,s')$.
  4. Repeat from step 2 (until somebody says enough).
- More interesting usage:
  - Given an output sequence $Y = \{y_1,y_2,\ldots,y_k\}$, compute its probability.
  - Given an output sequence $Y = \{y_1,y_2,\ldots,y_k\}$, compute the most likely sequence of states which has generated it.
  - ...plus variations: e.g., n best state sequences

Introduction to
Natural Language Processing (600.465)

HMM Algorithms: Trellis and Viterbi

Dr. Jan Hajič
CS Dept., Johns Hopkins Univ.
hajic@cs.jhu.edu
www.cs.jhu.edu/~hajic

HMM: The Two Tasks

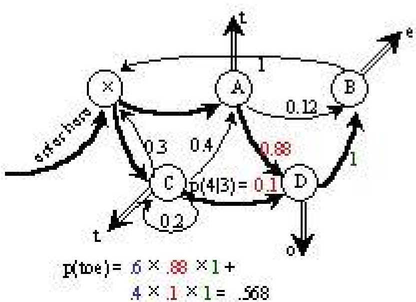
- HMM (the general case):
  - five-tuple (S, S_0, Y, P_S, P_Y), where:
    - $S = \{s_1, s_2, \ldots, s_T\}$ is the set of states, S_0 is the initial state,
    - $Y = \{y_1, y_2, \ldots, y_V\}$ is the output alphabet,
    - $P_S(s_j|s_i)$ is the set of prob. distributions of transitions,
    - $P_Y(y_k|s_i,s_j)$ is the set of output (emission) probability distributions.
- Given an HMM & an output sequence $Y = \{y_1, y_2, \ldots, y_k\}$:

  (Task 1) compute the probability of Y;

  (Task 2) compute the most likely sequence of states which has generated Y.

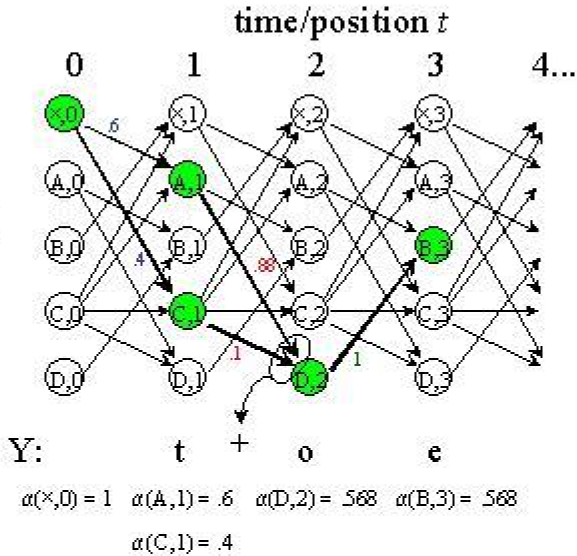
Trellis - Deterministic Output

HMM:



"rollout"

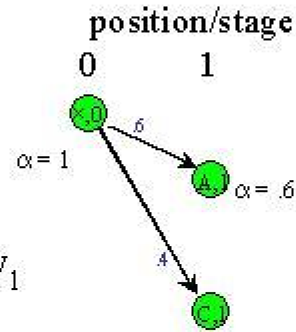
Trellis:



- trellis state: (HMM state, position)
- each state: holds ***one*** number (prob): α
- probability or Y: $\Sigma\alpha$ in the last state

Creating the Trellis: The Start

- Start in the start state (×),
  - set its $\alpha(\times, 0)$ to 1.
- Create the first stage:
  - get the first "output" symbol y_1
  - create the first stage (column)
  - but only those trellis states which generate y_1
  - set their $\alpha(state, 1)$ to the $P_S(state|\times)\ \underbrace{\alpha(\times, 0)}_{1}$
- …and forget about the *0*-th stage



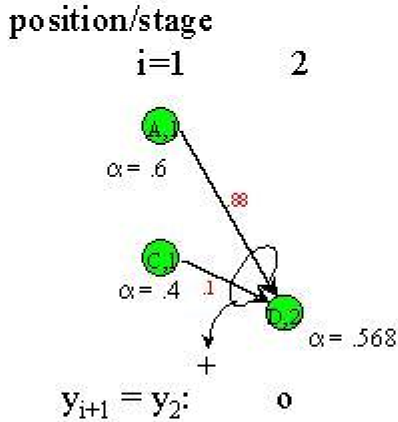
y_1: t

Trellis: The Next Step

- Suppose we are in stage *i*
- Creating the next stage:
  - create all trellis states in the next stage which generate y_{i+1}, but only those reachable from any of the stage-*i* states
  - set their $\alpha(state, i+1)$ to:

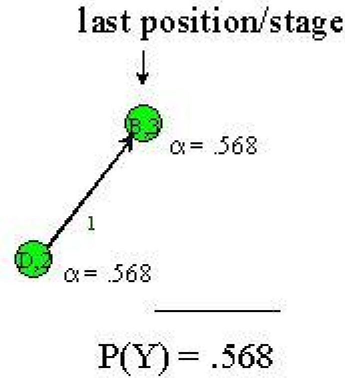
    $P_S(state|prev.state) \times \alpha(prev.state, i)$

    (add up all such numbers on arcs going to a common trellis state)
  - ...and forget about stage *i*



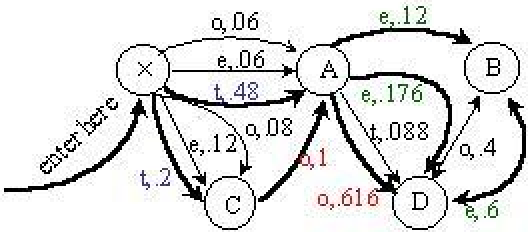
Trellis: The Last Step

- Continue until "output" exhausted
  - $|Y| = 3$: until stage 3
- Add together all the $\alpha(state, |Y|)$
- That's the P(Y).
- Observation (pleasant):
  - memory usage max: $2|S|$
  - multiplications max: $|S|^2|Y|$



Trellis: The General Case (still, bigrams)

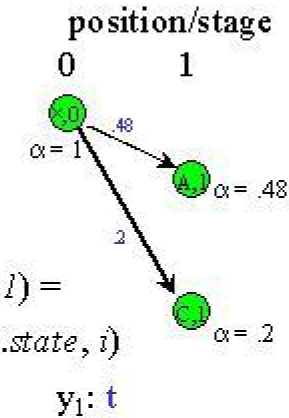
- Start as usual:
  - start state (×), set its $\alpha(\times,0)$ to 1.

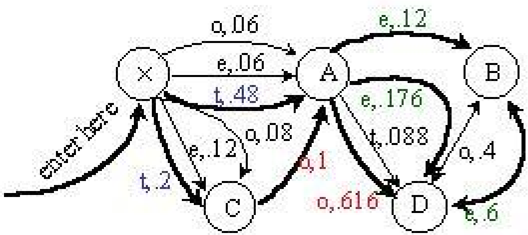


$$p(\text{toe}) = .48 \times .616 \times .6 + .2 \times 1 \times .176 + .2 \times 1 \times .12 \cong .237$$

General Trellis: The Next Step

- We are in stage *i* :
  - Generate the next stage *i+1* as before (except now arcs generate output, thus use only those arcs marked by the output symbol y_{i+1})
  - For each generated *state*, compute $\alpha(state, i+1) =$
    $= \Sigma_{\text{incoming arcs}} P_Y(y_{i+1}|state, prev.state) \times \alpha(prev.state, i)$

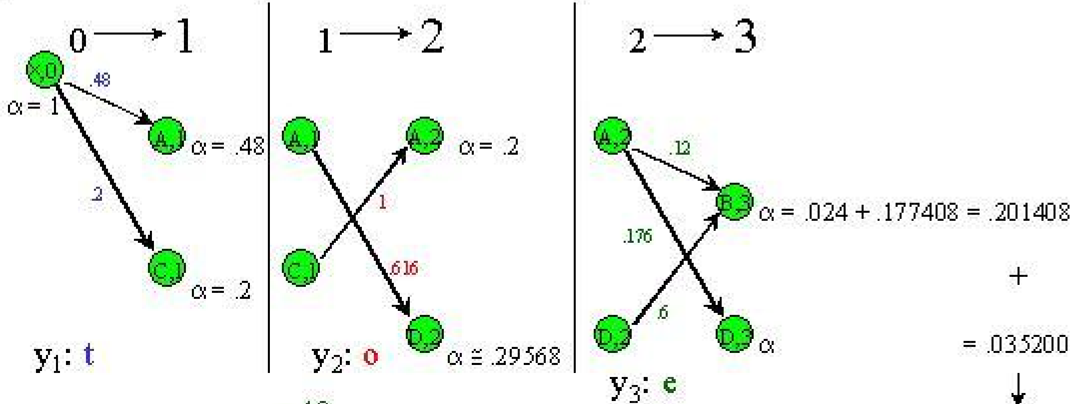


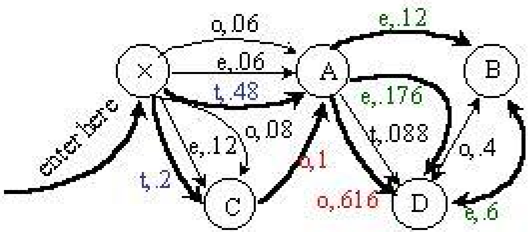


...and forget about stage *i* as usual.

Trellis: The Complete Example

Stage:

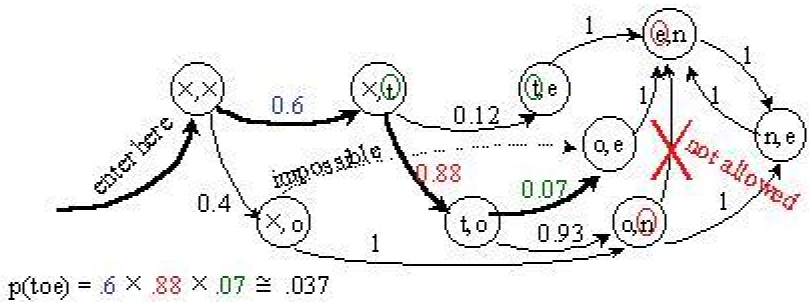


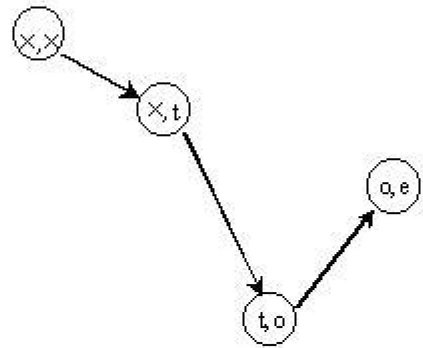


P(Y) = P(toe) = .236608

The Case of Trigrams

- Like before, but:
  - states correspond to bigrams,
  - output function always emits the second output symbol of the pair (state) to which the arc goes:





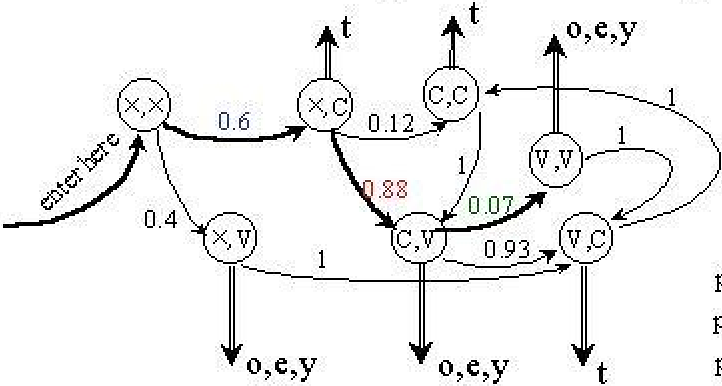
Multiple paths not possible → trellis not really needed

Trigrams with Classes

- More interesting:
  - n-gram class LM: $p(w_i|w_{i-2},w_{i-1}) = p(w_i|c_i)\, p(c_i|c_{i-2},c_{i-1})$

    → states are pairs of classes (c_{i-1},c_i), and emit "words":

(letters in our example)



p(t|C) = 1 **usual,**
p(o|V) = .3 **non-**
p(e|V) = .6 **overlapping**
p(y|V) = .1 **classes**

p(toe) = .6 × 1 × .88 × .3 × .07 × .6 ≅ .00665

p(teo) = .6 × 1 × .88 × .6 × .07 × .3 ≅ .00665

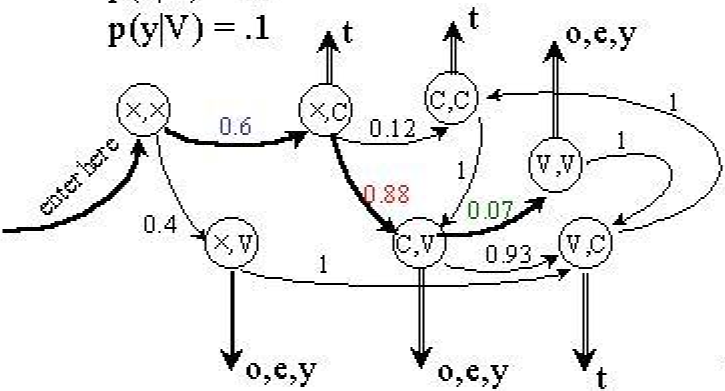
p(toy) = .6 × 1 × .88 × .3 × .07 × .1 ≅ .00111

p(tty) = .6 × 1 × .12 × 1 × 1 × .1 ≅ .0072

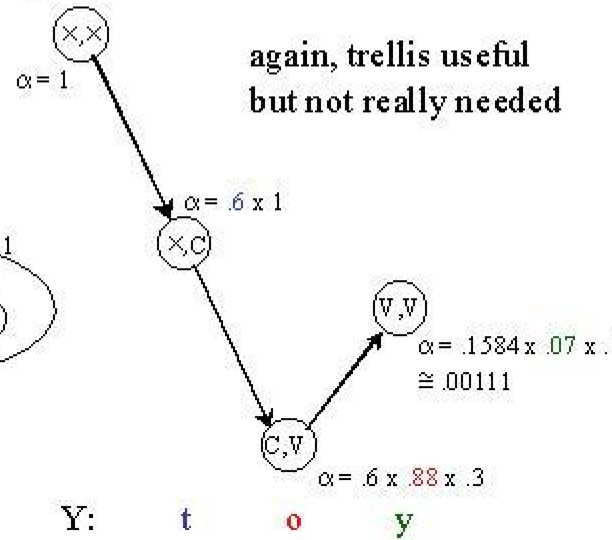
Class Trigrams: the Trellis

- Trellis generation (Y = "toy"):

p(t|C) = 1
p(o|V) = .3
p(e|V) = .6
p(y|V) = .1

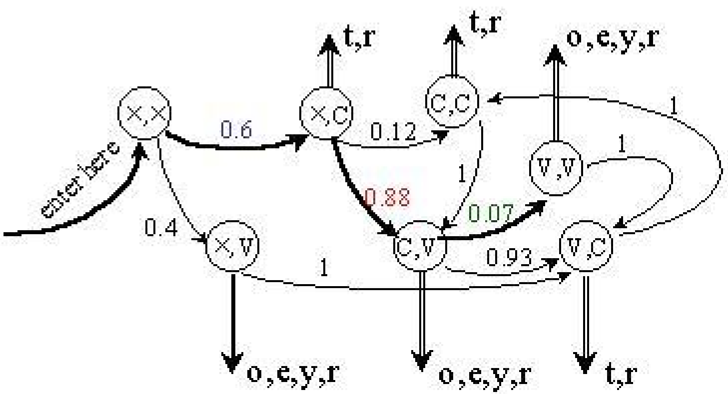


again, trellis useful but not really needed



Overlapping Classes

- Imagine that classes may overlap
  - e.g. ‘r’ is sometimes vowel sometimes consonant, belongs to V as well as C:

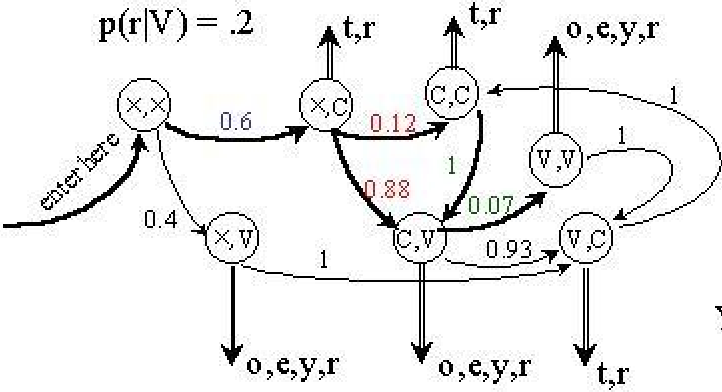


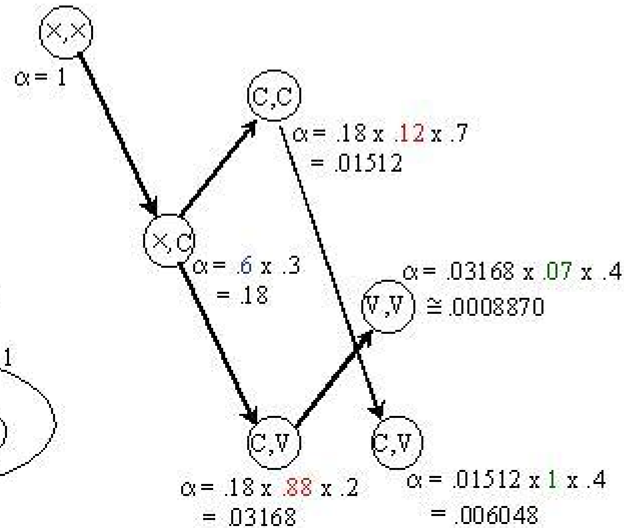
p(t|C) = .3
p(r|C) = .7
p(o|V) = .1
p(e|V) = .3
p(y|V) = .4
p(r|V) = .2

p(try) = ?

Overlapping Classes: Trellis Example

p(t|C) = .3
p(r|C) = .7
p(o|V) = .1
p(e|V) = .3
p(y|V) = .4
p(r|V) = .2





Y: **t** **r** **y** p(Y) = .006935

Trellis: Remarks

- So far, we went left to right (computing α)
- Same result: going right to left (computing β)
  - supposed we know where to start (finite data)
- In fact, we might start in the middle going left and right
- Important for parameter estimation

  (Forward-Backward Algortihm alias Baum-Welch)
- Implementation issues:
  - scaling/normalizing probabilities, to avoid too small numbers & addition problems with many transitions

The Viterbi Algorithm

- Solving the task of finding the most likely sequence of states which generated the observed data
- i.e., finding

$$S_{best} = \text{argmax}_S P(S|Y)$$

which is equal to (Y is constant and thus P(Y) is fixed):

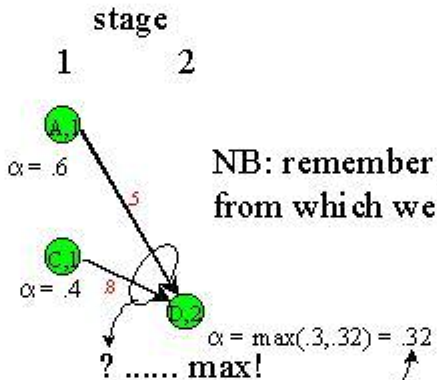
$$S_{best} = \text{argmax}_S P(S,Y) =$$

$$= \text{argmax}_S P(s_0, s_1, s_2, \ldots, s_k, y_1, y_2, \ldots, y_k) =$$

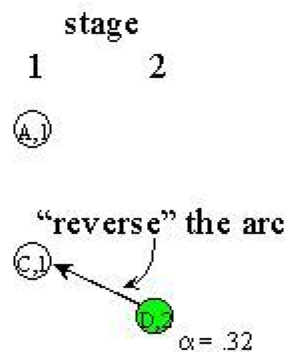
$$= \text{argmax}_S \Pi_{i=1..k}\, p(y_i|s_i, s_{i-1}) p(s_i|s_{i-1})$$

The Crucial Observation

- Imagine the trellis build as before (but do not compute the αs yet; assume they are o.k.); stage *i*:



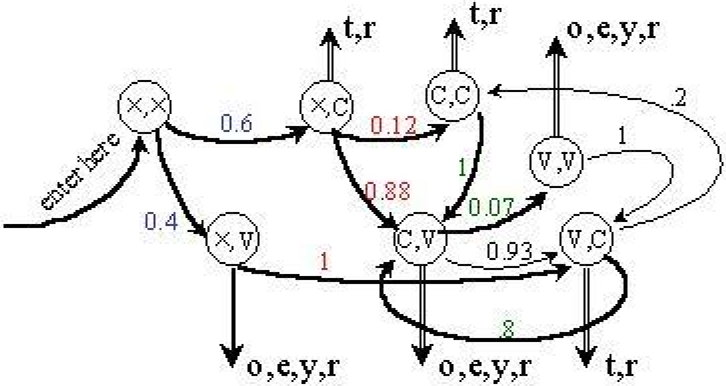
NB: remember previous state from which we got the maximum:



this is certainly the "backwards" maximum to (D,2)... but
it <u>cannot change even whenever we go forward</u> (M. Property: Limited History)

Viterbi Example

- 'r' classification (C or V?, sequence?):



p(t|C) = .3
p(r|C) = .7
p(o|V) = .1
p(e|V) = .3
p(y|V) = .4
p(r|V) = .2

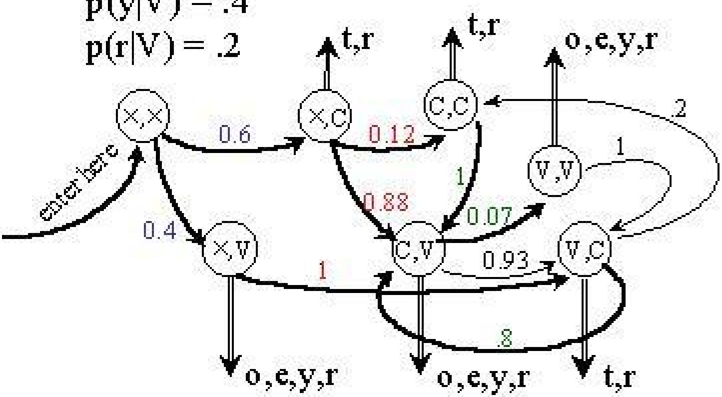
$\text{argmax}_{XYZ}\ p(rry|XYZ) = ?$

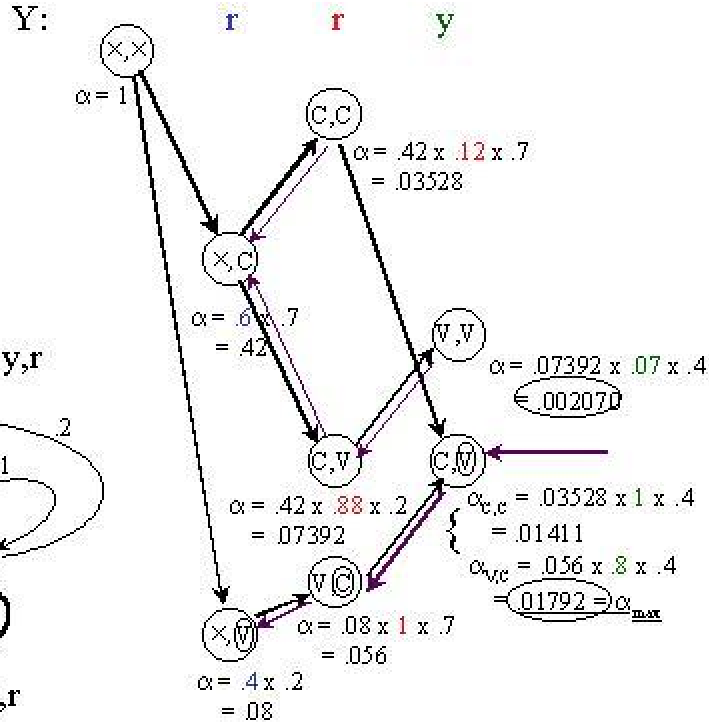
Possible state seq.: (×,v)(v,c)(c,v)[VCV], (×,c)(c,c)(c,v)[CCV], (×,c)(c,v)(v,v) [CVV]

Viterbi Computation

$p(t|C) = .3$
$p(r|C) = .7$
$p(o|V) = .1$
$p(e|V) = .3$
$p(y|V) = .4$
$p(r|V) = .2$

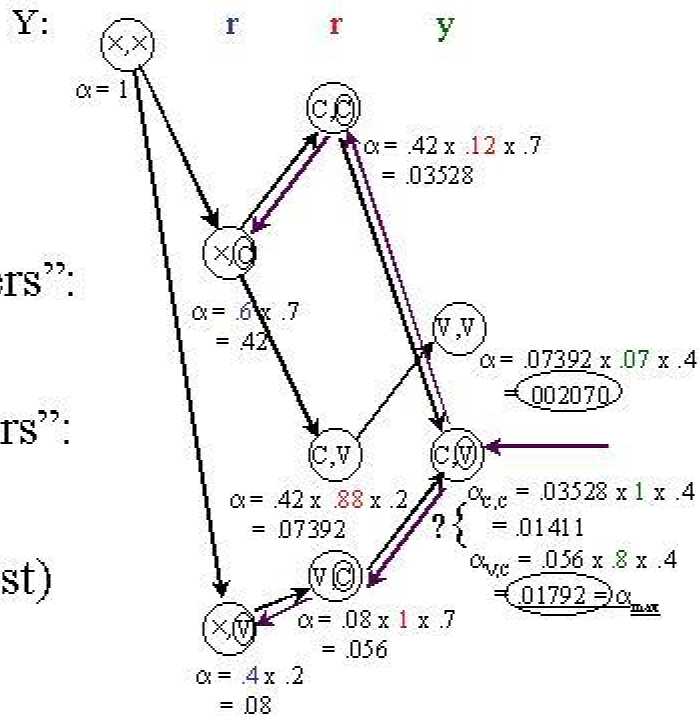
α in trellis state: best prob from start to here





n-best State Sequences

- Keep track of n best "back pointers":
- Ex.: n= 2:
  Two "winners":
  VCV (best)
  CCV (2nd best)

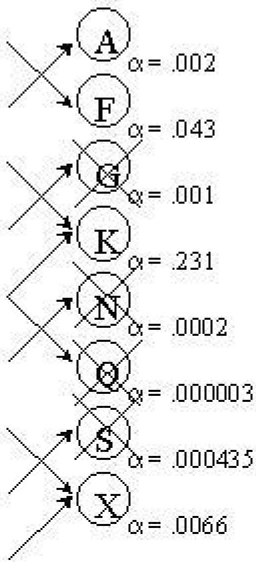


Tracking Back the n-best paths

- Backtracking-style algorithm:
  - Start at the end, in the best of the n states (s_{best})
  - Put the other n-1 best nodes/back pointer pairs on stack, except those leading from s_{best} to the same best-back state.
- Follow the back "beam" towards the start of the data, spitting out nodes on the way (backwards of course) using always only the best back pointer.
- At every beam split, push the diverging node/back pointer pairs onto the stack (node/beam width is sufficient!).
- When you reach the start of data, close the path, and pop the top-most node/back pointer(width) pair from the stack.
- Repeat until the stack is empty; expand the result tree if necessary.

Pruning

- Sometimes, too many trellis states in a stage:



criteria: (a) α < threshold
(b) $\Sigma\pi$ < threshold
(c) # of states > threshold
(get rid of smallest α)