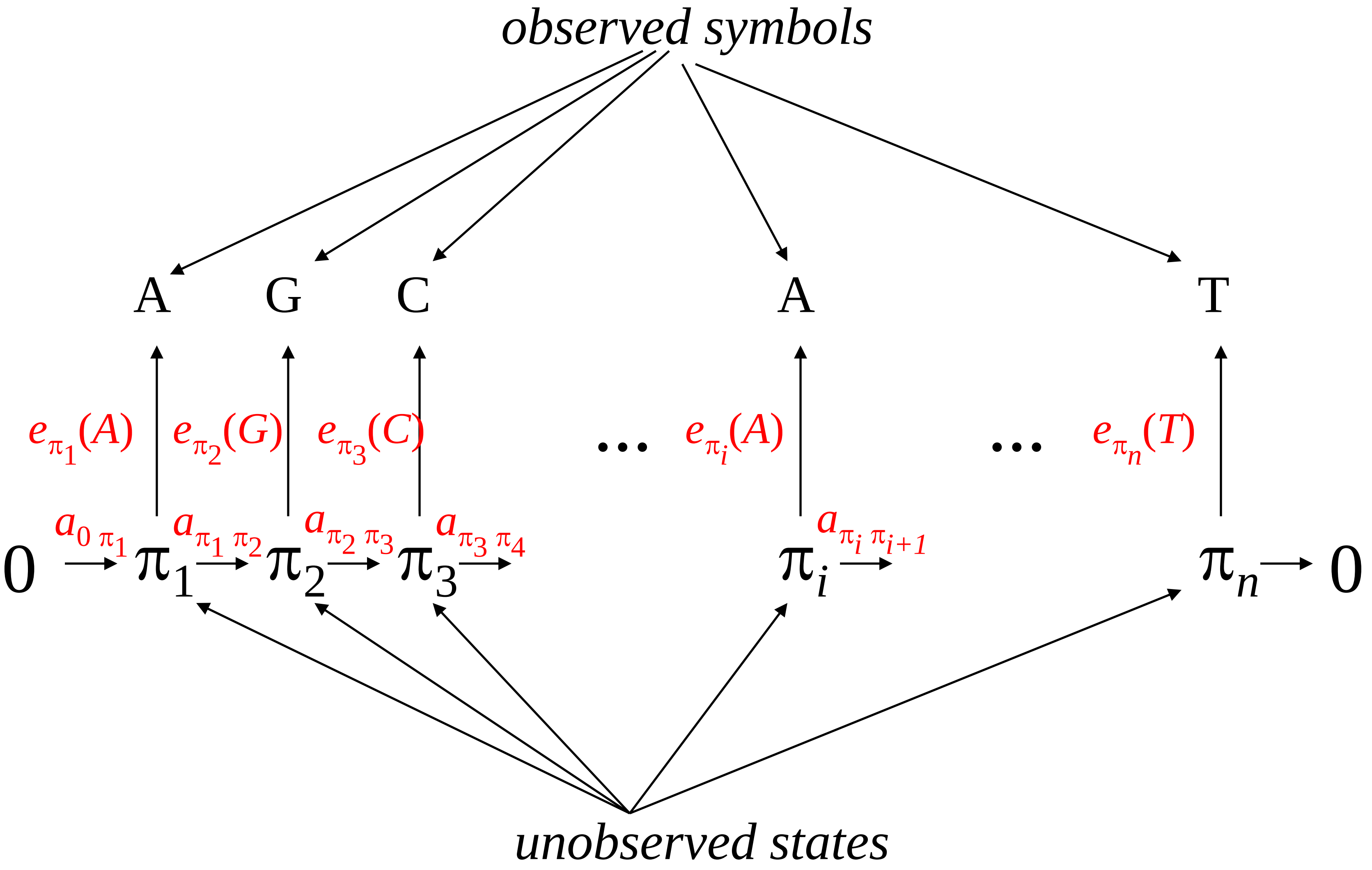
Lecture 15

- Forward & forward/backward algorithms

- HMM parameter estimation
  - Viterbi training
  - Baum-Welch training

Hidden Markov Model



observed symbols

A G C A T

$e_{\pi_1}(A)$ $e_{\pi_2}(G)$ $e_{\pi_3}(C)$ … $e_{\pi_i}(A)$ … $e_{\pi_n}(T)$

$0 \xrightarrow{a_{0\,\pi_1}} \pi_1 \xrightarrow{a_{\pi_1\,\pi_2}} \pi_2 \xrightarrow{a_{\pi_2\,\pi_3}} \pi_3 \xrightarrow{a_{\pi_3\,\pi_4}}$

$\pi_i \xrightarrow{a_{\pi_i\,\pi_{i+1}}}$

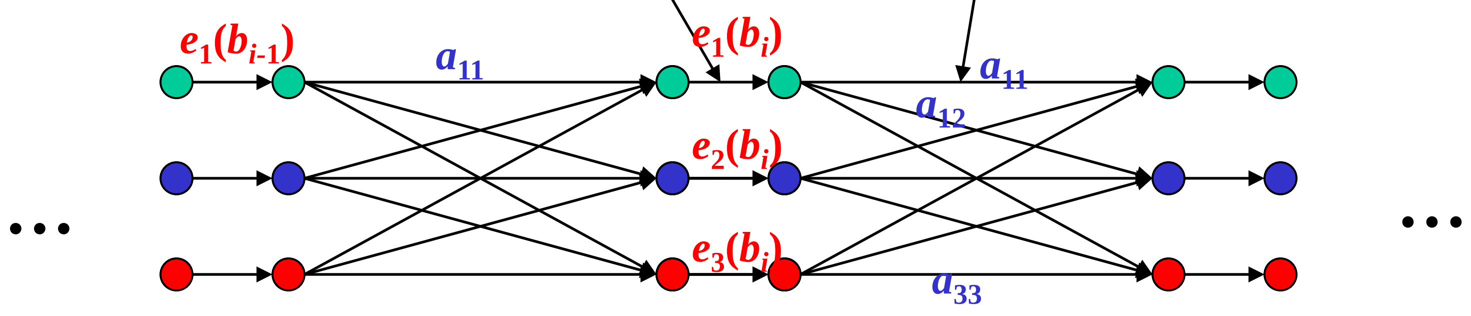
$\pi_n \rightarrow 0$

unobserved states

WDAG for 3-state HMM, length *n* sequence

weights are emission probabilities $e_k(b_i)$ for i^{th} residue b_i

weights are transition probabilities a_{kl}

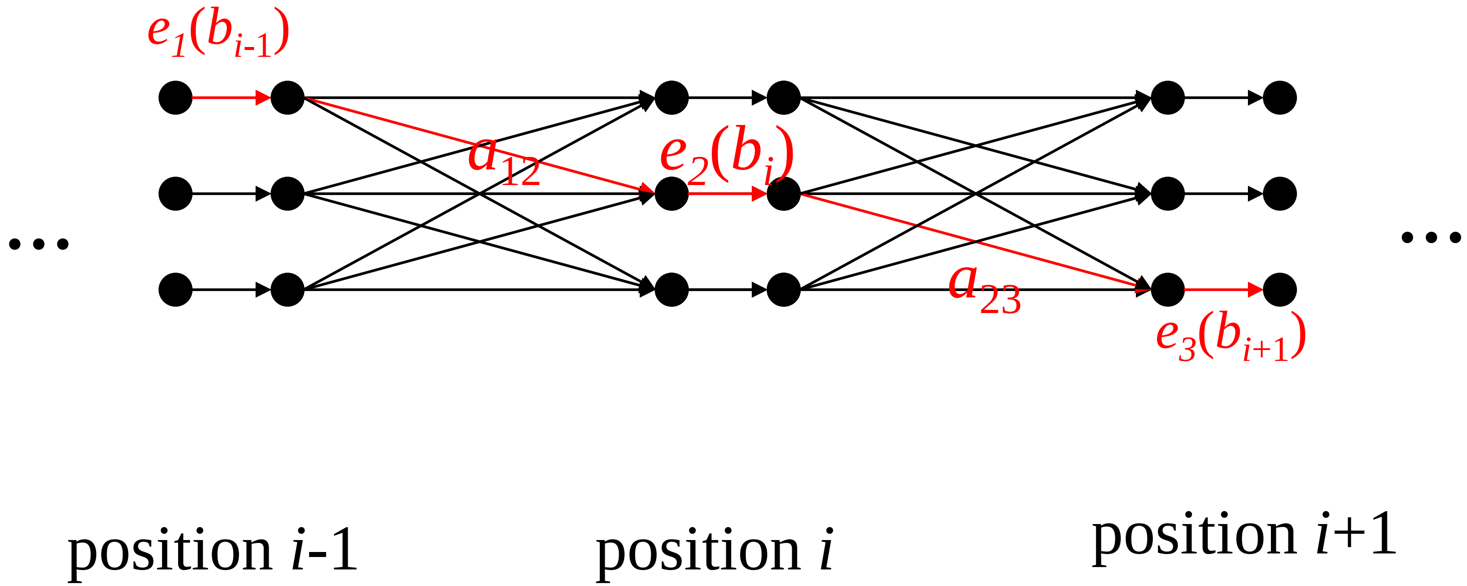


$b_{i\text{-}1}$
position *i*-1

b_i
position *i*

b_{i+1}
position *i*+1

Path Weights



- ***Paths*** through graph from begin node to end node correspond to ***sequences of states***
- ***Product weight*** along path

  = ***joint probability*** of state sequence & observed symbol sequence
- ***Highest-weight path*** = ***highest probability state sequence***
- ***Sum*** *of (product) path weights,* ***over all paths***,

  = ***probability of observed sequence***
- ***Sum*** *of (product) path weights* ***over***
  - all paths *going through a particular node*, or
  - all paths *that include a particular edge*,

  divided by prob of observed sequence,

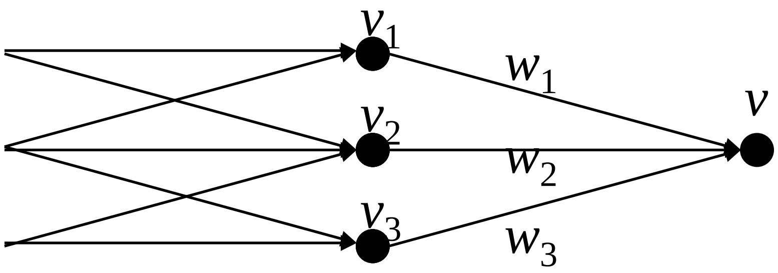
  = ***posterior probability*** of that edge or node

- use dynamic programming to find
  - – sum of all product path weights
    - = “forward algorithm” for probability of observed sequence
  - – sum of all product path weights through particular node or particular edge
    - = “forward/backward algorithm” to find posterior probabilities
- Now must use product weights and non-log-transformed probabilities
  - – because need to *add* probabilities

- In each case, compute successively for each node (by increasing depth: left to right)
    - the sum of the weights of all paths ending at that node
    - N.B. paths are constrained to begin at the begin node, end at end node!
- In forward/backward algorithm,
    - work through all nodes a second time, in opposite direction
        - i.e. in reverse graph – constraining paths to start at end node

For each vertex v, let $f(v) = \sum_{\text{paths } p \text{ ending at } v} \text{weight}(p)$, where weight($p$) = *product* of edge weights in p. Only consider paths starting at ‘begin’ node.

Compute $f(v)$ by dynam. prog: $f(v) = \sum_i w_i\, f(v_i)$, where v_i ranges over the parents of v, and w_i = weight of the edge from v_i to v.

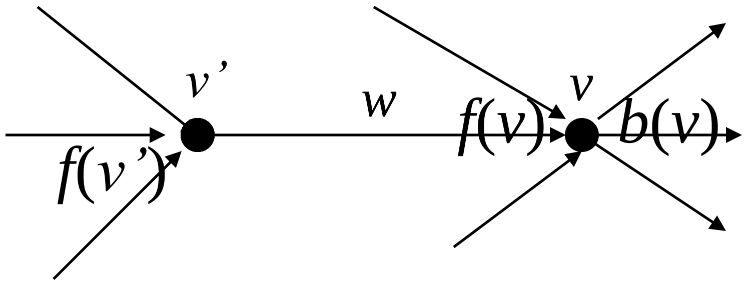


Similarly for $b(v) = \sum_{p \text{ beginning at } v} \text{weight}(p)$

The paths *beginning* at v are the ones *ending* at v in the *reverse* (*or inverted*) *graph*

from lecture 12 :

- Can “invert” any WDAG: create graph with
  - same vertices & edge weights
  - direction of each edge reversed
  - is still acyclic!
- inverted WDAG has same paths (& path weights), but in reverse direction
  - “*forward*” path in inverted WDAG = “*backward*” path in original WDAG (& vice versa)



f(*v*)*b*(*v*) = sum of the path weights of all paths *through v*.

f(*v'*)*wb*(*v*) = sum of the path weights of all paths *through the edge (v',v)*

Forward/backward algorithm

- Work through graph in forward direction:
  - compute and store $f(v)$
- Then work through graph in backward direction:
  - compute $b(v)$
  - compute $f(v)\ b(v)$ and $f(v)wb(v)$ as above, store in appropriate cumulative sums
  - only need to store $b(v)$ until have computed *b*'s at next position
- Posterior probability of being in state *s* at position *i* is $f(v)\ b(v)$ / *total sequence prob*
  - where *v* is the corresponding graph node

- Numerical issues: multiplying many small values can cause underflow. Remedies:
  - *Scale* weights to be close to 1 (affects all paths by same constant factor – which can be multiplied back later); or
  - (where possible) use log weights, so can add instead of multiplying.
  - see Rabiner & Tobias Mann links on web page

HMM Parameter Estimation

- *Parameters* = transition & emission probs
  - *parameter values ↔ probability model*
- If unknown, estimate from set of training sequences
- *Maximum likelihood* (ML) estimation (= choice of param vals to maximize prob of training data) is preferred
  - optimality properties of ML estimates discussed in Ewens & Grant

↔ finding maximum value on a multi-dimensional surface

- Hard problem! Can be many local maxima

Parameter estimation when state sequence is *known*

- When underlying state sequence for each training sequence is *known*,
  - e.g.: site model

then ML estimates are given by:

- emission probabilities:

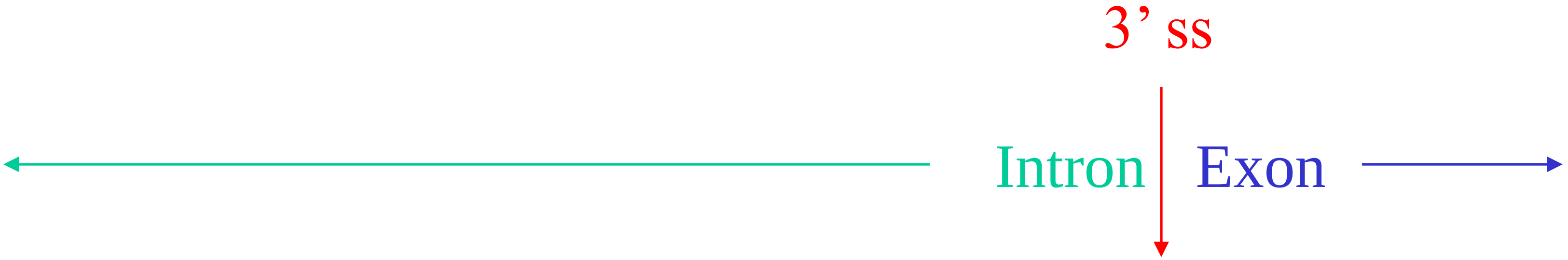
  $e_k(b)$^ = (# times symbol b emitted by state k) / (# times state k occurs) .

- transition probabilities:

  a_{kl} ^ = (# times state k followed by state l) / (# times state k occurs)

- in denominator above, *omit occurrence at last position of sequence* (for transition probabilities)
  - But include it for emission probs
- can include pseudocounts, to incorporate prior expectations/avoid small sample overfitting (Bayesian justification)

HMM for *C. elegans* 3’ Splice Sites



A	3276	3516	2313	476	67	757	240	8192	0	3359	2401	2514
C	970	648	664	236	129	1109	6830	0	0	1277	1533	1847
G	593	575	516	144	39	595	12	0	8192	2539	1301	1567
T	3353	3453	4699	7336	7957	5731	1110	0	0	1017	2957	2264
CONSENSUS	W	W	W	T	T	t	C	A	G	r	w	w

Emission probabilities

A	0.400	0.429	0.282	0.058	0.008	0.092	0.029	1.000	0.000	0.410	0.293	0.307
C	0.118	0.079	0.081	0.029	0.016	0.135	0.834	0.000	0.000	0.156	0.187	0.225
G	0.072	0.070	0.063	0.018	0.005	0.073	0.001	0.000	1.000	0.310	0.159	0.191
T	0.409	0.422	0.574	0.896	0.971	0.700	0.135	0.000	0.000	0.124	0.361	0.276

$$0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12$$

‘hidden’ states

Parameter estimation when state sequence *unknown*

- *Viterbi training*

1. choose starting parameter values
   - must be valid probabilities; avoid 0 unless topology dictates
   - make them *biologically plausible* given state interpretation
2. find Viterbi highest weight paths for each sequence
3. estimate new emission and transition probs as above, *assuming* the Viterbi state sequence
4. iterate steps 2 and 3 until convergence
   - not guaranteed to occur – but nearly always does
5. repeat steps 1 – 4 with other starting values
   - choose values with highest total path score

- Viterbi training does ***not*** necessarily give ML estimates, but often are reasonably good

Baum-Welch training

- Special case of EM (‘expectation-maximization’) algorithm
- like Viterbi training, but
  - uses *all* paths, each weighted by its probability

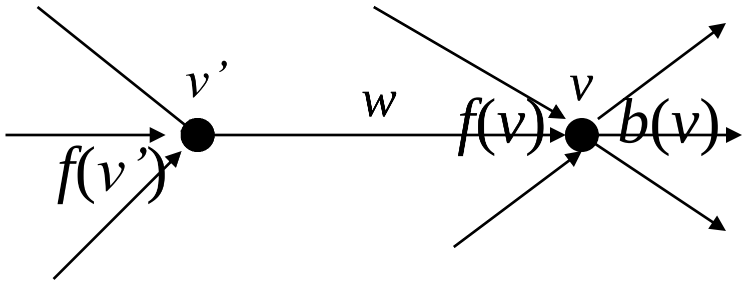
    rather than just highest probability path.
- sometimes give significantly better results than Viterbi
  - e.g. for PFAM

Implementing Baum-Welch

– An edge in the WDAG contributes *fractional* (or *weighted*) *counts* given by its posterior probability:

– (*): $(\sum_{\text{all paths } p \text{ through edge } e} \text{weight}(p)) / (\sum_{\text{all paths } p} \text{weight}(p))$

(Fractional counts are computed using forward-backward algorithm)



f(*v*)*b*(*v*) = sum of the path weights of all paths *through v*.

f(*v'*)*wb*(*v*) = sum of the path weights of all paths *through the edge (v',v)*

- Compute new param estimates
  - $e_k(b)$^ = (frac. # times symbol b emitted by state k) / (frac. # times state k occurs)
  - a_{kl} ^ = (frac. # times state k followed by state l) / (frac. # times state k occurs)
    - (In denom,, omit frac counts at last position of sequence)

where "frac. # times" is given by (*) for appropriate edge type (emission or transition)

- New Baum-Welch parameter estimates have higher likelihood
  - general property of EM algorithm
  - not true in general for Viterbi training

- Iterate: get series of estimates converging to a *local* maximum on likelihood surface

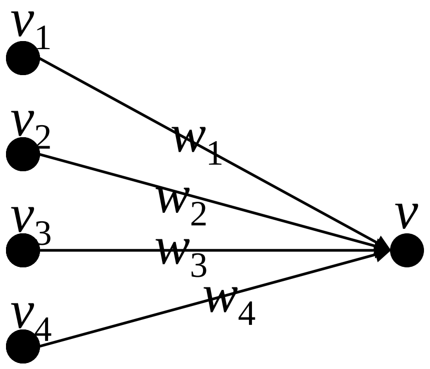
Search of parameter space

- ML estimates correspond by definition to *global* maximum;
- but there may be many *local* maxima, and EM or Viterbi search can get "trapped" in one
- remedies:
  - Consider multiple starts (multiple choices for starting parameters)
  - use "reasonable values" to start search (e.g. unlikely transitions should be given small initial probabilities)

- Allow search to “jump” out of local maxima:
  - Add “noise” to counts at each iteration; gradually reduce the amount of noise
  - Use Viterbi-like training, but
    - sample paths probabilistically
      - (in retracing Viterbi path, choose edge at random according to its prob, rather than taking highest prob parent);
    - use “temperature” T to adjust probabilities;
      - initially with large T making all probs approximately equal;
      - then gradually reduce T

Probabilistic Viterbi Backtracking

reset all weights w to $w^{1/T}$. For large T (>> 1), this makes distinct w's relatively close; for small T (<< 1), relatively far apart



choose parent v_i with probability $w_i\, f(v_i) / f(v)$. For large T, all parents almost equally likely to be chosen; for small T, strongly favor largest (max) $w_i\, f(v_i)$

given choice of paths, re-estimate weights; iterate