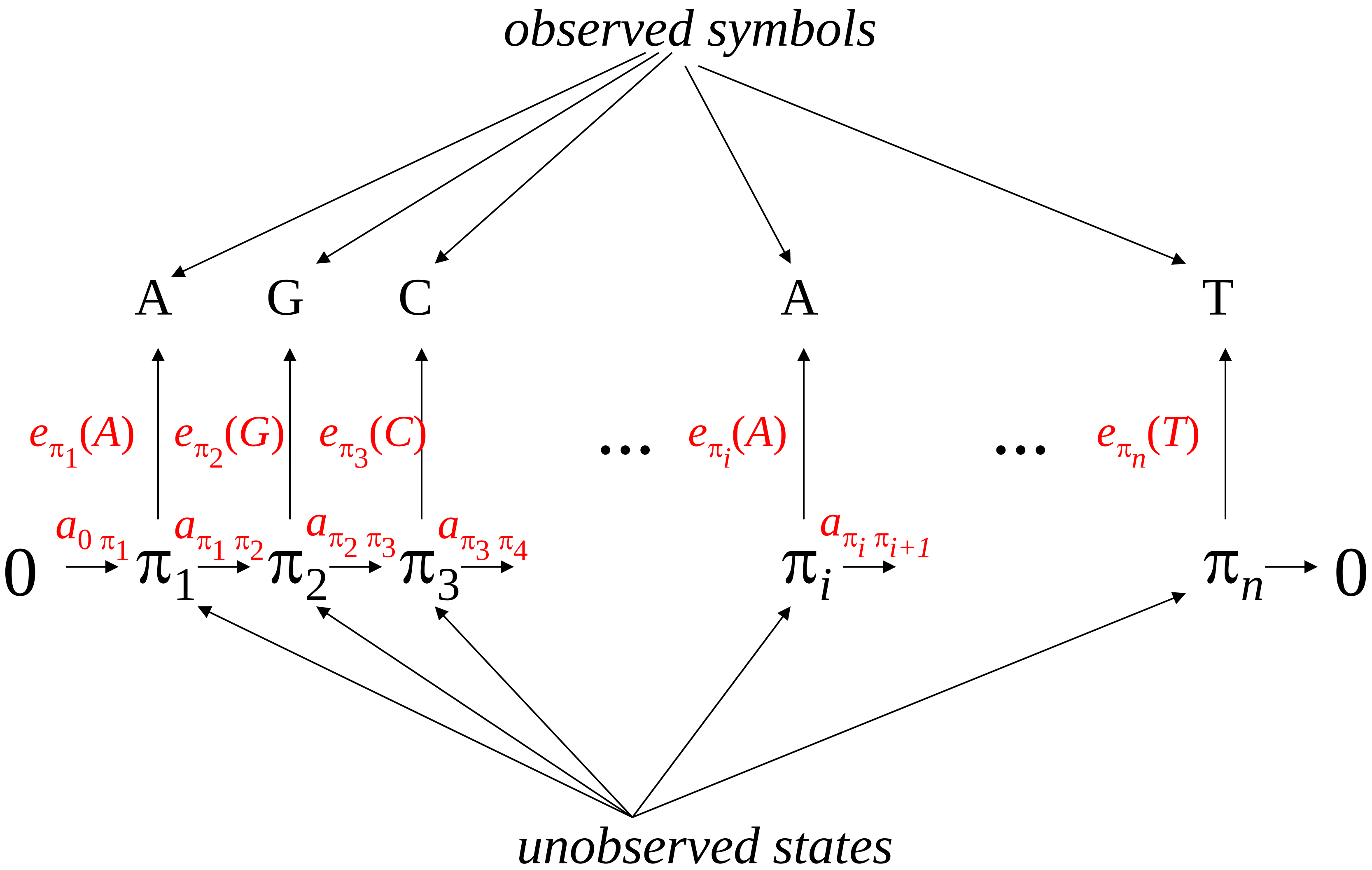
Today's Lecture

- Review of
  - HMM probability calculations
  - Viterbi training
- Forward & forward/backward algorithms
- Baum-Welch training

Hidden Markov Model



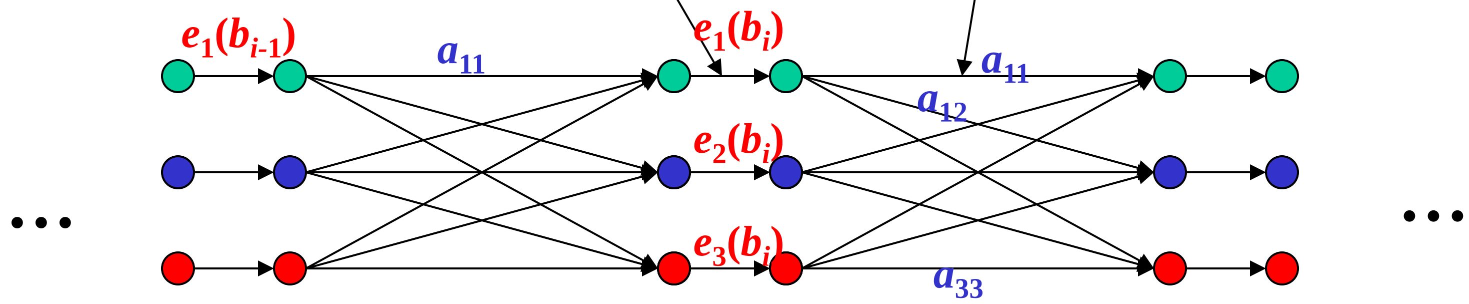
HMM Probabilities of Sequences

- Prob of sequence of states $\pi_1\pi_2\pi_3 \ldots \pi_n$ is $a_{0\pi_1} a_{\pi_1\pi_2} a_{\pi_2\pi_3} a_{\pi_3\pi_4} \ldots a_{\pi_{n-1}\pi_n}$.
- Prob of seq of observed symbols $b_1b_2b_3 \ldots b_n$, conditional on state sequence is $e_{\pi_1}(b_1) e_{\pi_2}(b_2)\, e_{\pi_3}(b_3) \ldots e_{\pi_n}(b_n)$
- Joint probability = $a_{0\pi_1} \prod^{n}_{i=1} a_{\pi_i\pi_{i+1}}\, e_{\pi_i}(b_i)$ (define $a_{\pi_n\pi_{n+1}}$ to be 1)
- (Unconditional) prob of observed sequence = sum (of joint probs) over all possible state paths
  - not practical to compute directly, by 'brute force'! We will use dynamic programming.

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-1} position i-1

b_i position i

b_{i+1} position i+1

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

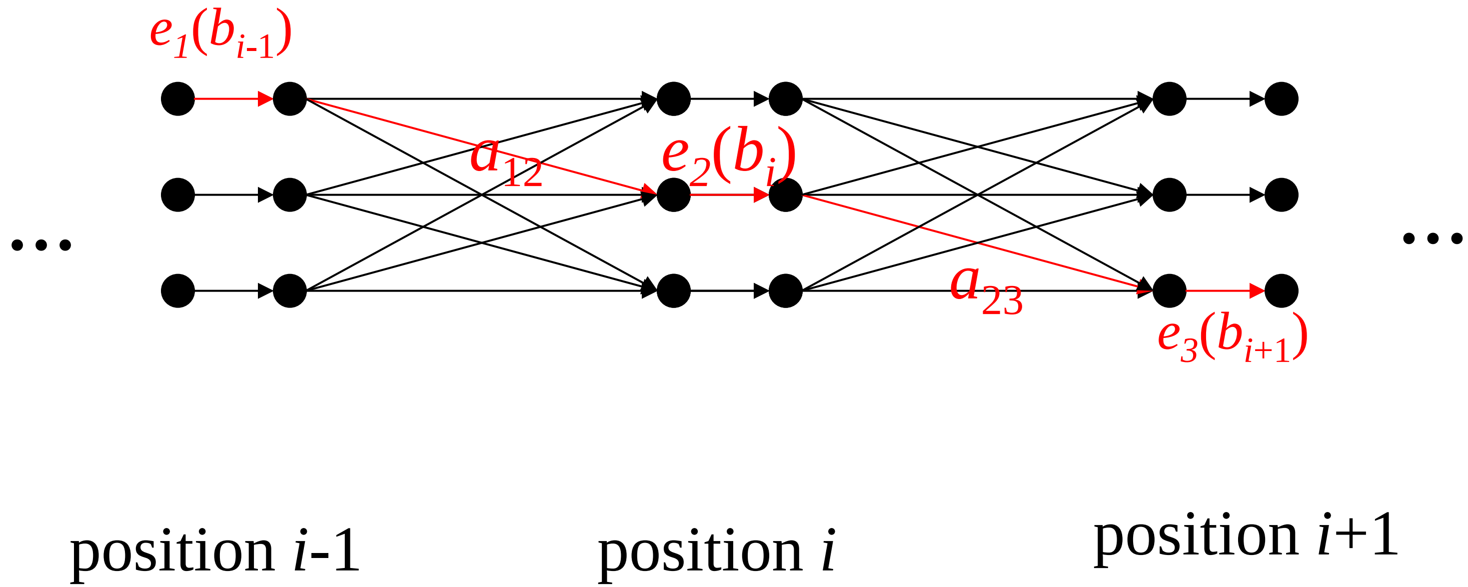
  = *joint probability* of state sequence & observed symbol sequence
- Sum of (product) path weights, over all paths,

  = *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
- ***Highest-weight path*** = *highest probability state sequence*

Path Weights



- By general results on WDAGs, can use dynamic programming to find highest weight path:
  - = "Viterbi algorithm" to find highest probability path (most probable "parse")
  - in this case can use log probabilities & sum weights

HMM Parameter Estimation

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

Parameter estimation when state sequence is known

- When underlying state sequence for each training sequence is *known*,
  - e.g.: weight matrix; Markov 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)

Parameter estimation when state sequences *unknown*

- *Viterbi training*
  1. choose starting parameter values
  2. find highest weight paths (Viterbi) for each sequence
  3. estimate new emission and transition probs as above, *assuming* Viterbi state sequence is true
  4. iterate steps 2 and 3 until convergence
     - not guaranteed to occur – but nearly always does
  5. does *not* necessarily give ML estimates, but often are reasonably good

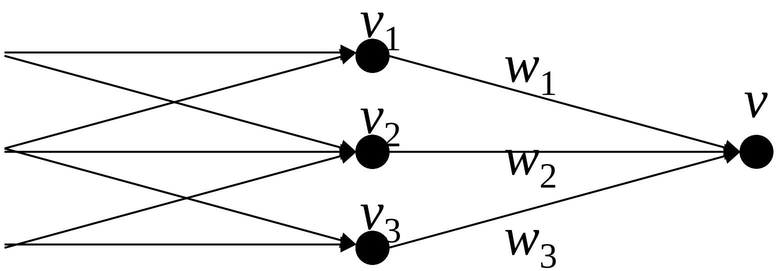
More algorithms!!

- Can also 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!
- In forward/backward algorithm,
  - work through all nodes a second time, in opposite direction
    - i.e. in reverse graph – constraining paths to start in rightmost column of nodes

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.

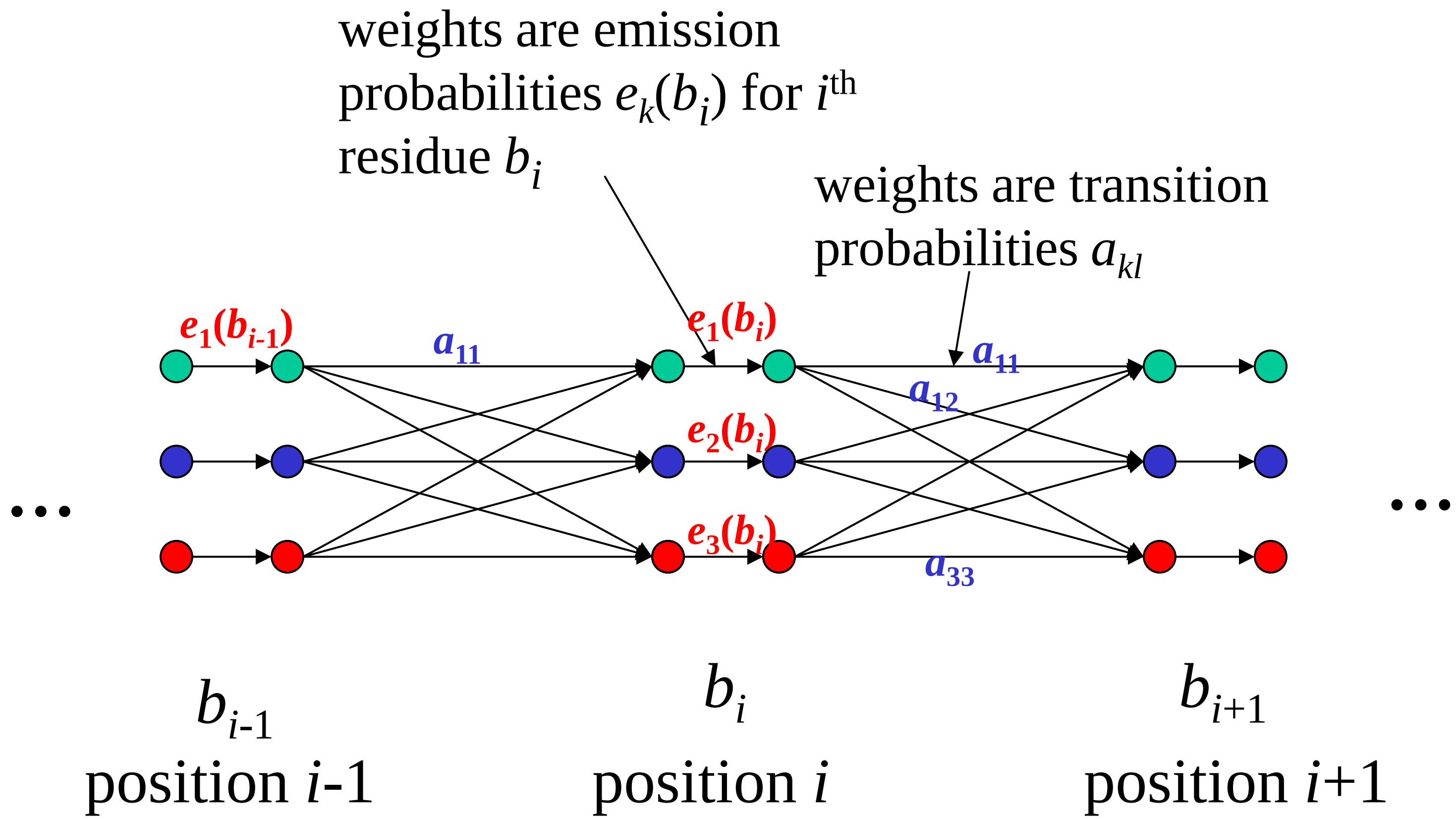
Compute $f(v)$ by dynam. prog: $f(v) = \sum_i w_i\, f(v_i)$, where v_i ranges over the parents of v, and w_i = weight of the edge from v_i to v.

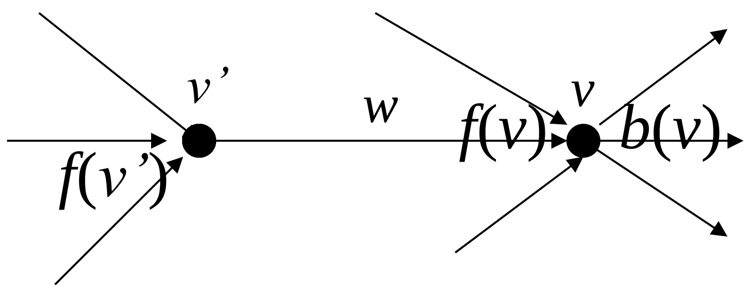


Similarly for $b(v) = \sum_{p \text{ beginning at } v} \text{weight}(p)$

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

WDAG for 3-state HMM, length n sequence





$f(v)b(v)$ = 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
    - & will discuss further in discussion section
- Complexity: $O(|V|+|E|)$, total # vertices and edges.
  - # nodes = $2ns + 2$ where n = sequence length, s = # states.
  - # edges = $(n-1)s^2 + ns + 2s$
  - So overall complexity is $O(ns^2)$
    - (actually s^2 can be reduced to # ‘allowed’ transitions between states – depends on model topology).

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