ELEN E6884/COMS 86884 Speech Recognition Lecture 8

Michael Picheny, Ellen Eide, Stanley F. Chen
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
{picheny,eeide,stanchen}@us.ibm.com

27 October 2005

Administrivia

- main feedback from last lecture
  - a little too fast
  - FST's still unclear
- Lab 2 not graded yet, will be handed back next week
- Lab 3 out, due Sunday after next

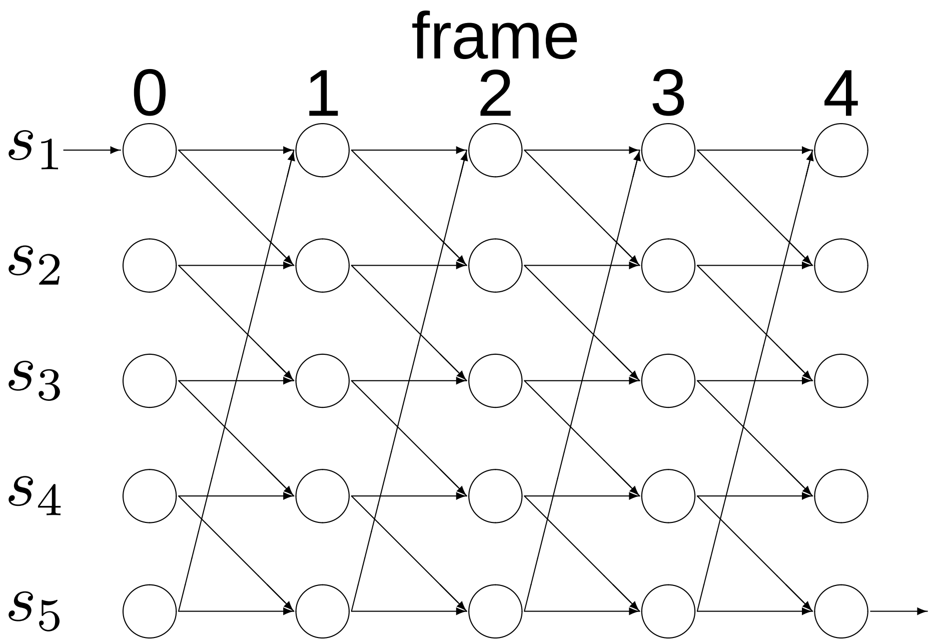
Lab 2 Review

- output distributions on states *vs.* arcs?
  - advantages of either representation?
- computing total likelihood for each word HMM separately *vs.* using Viterbi algorithm on one big HMM?
  - hint: what about computing *Viterbi* likelihood for each word HMM separately?

Lab 2 Review

Viterbi algorithm as shortest distance problem

- for arc a, frame t, distance from $(\textit{src}(a), t)$ to $(\textit{dst}(a), t+1)$ is ...
  - $-\log\left[P(a)P(x_t|a)\right]$



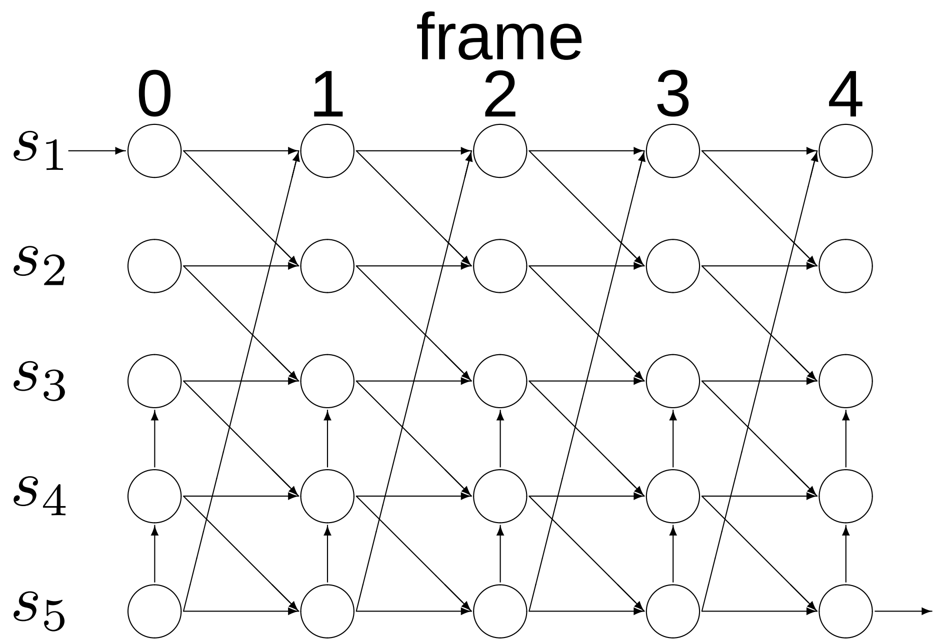
Viterbi As Shortest Distance Problem

- need to traverse chart in an order such that . . .
  - all chart arcs go from cell traversed earlier . . .
  - to cell traversed later
- loop first through frames, then through states

Viterbi As Shortest Distance Problem

What if we add skip arcs?

- for skip arc a, distance from $(\textit{src}(a), t)$ to $(\textit{dst}(a), t)$ is ...
  - $-\log[P(a)]$



Viterbi As Shortest Distance Problem

Handling skip arcs

- at a given frame, for all skip arcs a, must visit . . .
  - state *src*(a) before state *dst*(a)
- topologically sort states with respect to skip arcs only
  - then, natural ordering will work

  for t in $[0 \ldots (T-1)]$:
    for $s_{\texttt{src}}$ in $[1 \ldots S]$:

- in practice, may process skip arcs and emitting arcs in separate stages
- recap: beware of skip arcs

Lab 2 Review

- Q: if an HMM were a fruit, what type of fruit would it be?
  - A: a Hidden Markov Banana

Viterbi Algorithm

```
C[0...T, 1...S].vProb = 0
C[0, start].vProb = 1
for t in [0...(T−1)]:
   for s_src in [1...S]:
      for a in outArcs(s_src):
         s_dst = dest(a)
         curProb = C[t, s_src].vProb × arcProb(a, t)
         if curProb > C[t+1, s_dst].vProb:
            C[t+1, s_dst].vProb = curProb
            C[t+1, s_dst].trace = a
```

(do backtrace starting from $C[T, \mathit{final}]$ to find best path)

Forward Algorithm

$C[0 \ldots T, 1 \ldots S]$`.fProb = 0`

$C[0, \textit{start}]$`.fProb = 1`

`for` t `in` $[0 \ldots (T-1)]$`:`

  `for` $s_{\texttt{src}}$ `in` $[1 \ldots S]$`:`

    `for` a `in` *outArcs*$(s_{\texttt{src}})$`:`

      $s_{\texttt{dst}}$ `=` *dest*(a)

      curProb `=` $C[t, s_{\texttt{src}}]$`.fProb` $\times$ *arcProb*(a, t)

      $C[t+1, s_{\texttt{dst}}]$`.fProb +=` *curProb*

totProb `=` $C[T, \textit{final}]$`.fProb`

Backward Algorithm

```
C[0...T, 1...S].bProb = 0
C[T, final].bProb = 1
for t in [(T − 1)...0]:
  for s_src in [1...S]:
    for a in outArcs(s_src):
      s_dst = dest(a)
      curProb = C[t + 1, s_dst].bProb × arcProb(a, t)
      C[t, s_src].bProb += curProb
      fbCount = C[t, s_src].fProb × curProb / totProb
      addCount(a, t, fbCount)
```

Gaussian Update

- *occupancy count* $\gamma_{u,t}$ for given arc at frame t of utterance u
  - posterior prob of arc at that frame, *i.e.*, *fbCount*
- collect counts (for each dimension d)

$$
\begin{aligned}
S_0 &= \sum_{\text{utt } u} \sum_{\text{frame } t} \gamma_{u,t} \\
S_{1,d} &= \sum_{\text{utt } u} \sum_{\text{frame } t} \gamma_{u,t} \, x_{u,t,d} \\
S_{2,d} &= \sum_{\text{utt } u} \sum_{\text{frame } t} \gamma_{u,t} \, x_{u,t,d}^2
\end{aligned}
$$

Mean Update

$$S_0 = \sum_{\text{utt } u} \sum_{\text{frame } t} \gamma_{u,t}$$

$$S_{1,d} = \sum_{\text{utt } u} \sum_{\text{frame } t} \gamma_{u,t} \, x_{u,t,d}$$

$$S_{2,d} = \sum_{\text{utt } u} \sum_{\text{frame } t} \gamma_{u,t} \, x_{u,t,d}^2$$

$$\mu_d = \frac{\sum_u \sum_t \gamma_{u,t} \, x_{u,t,d}}{\sum_u \sum_t \gamma_{u,t}} = \frac{S_{1,d}}{S_0}$$

Variance Update

$$S_0 = \sum_{\text{utt } u} \sum_{\text{frame } t} \gamma_{u,t}$$

$$S_{1,d} = \sum_{\text{utt } u} \sum_{\text{frame } t} \gamma_{u,t}\, x_{u,t,d}$$

$$S_{2,d} = \sum_{\text{utt } u} \sum_{\text{frame } t} \gamma_{u,t}\, x_{u,t,d}^2$$

- update only diagonal terms $\Sigma_{d,d}$ in covariance matrix

$$\Sigma_{d,d} = \frac{\sum_{u,t} \gamma_{u,t}(x_{u,t,d} - \mu_d)^2}{\sum_{u,t} \gamma_{u,t}}$$

$$= \frac{1}{S_0}\left[\sum_{u,t} \gamma_{u,t} x_{u,t,d}^2 - 2\mu_d \sum_{u,t} \gamma_{u,t} x_{u,t,d} + \mu_d^2 \sum_{u,t} \gamma_{u,t}\right]$$

$$= \frac{S_{2,d} - 2\mu_d S_{1,d} + \mu_d^2 S_0}{S_0} = \frac{S_{2,d} - \mu_d^2 S_0}{S_0}$$

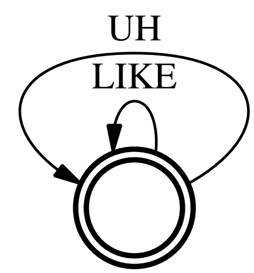
The Big Picture

- weeks 1–4: small vocabulary ASR
- weeks 5–8: large vocabulary ASR
  - week 5: language modeling
  - week 6: pronunciation modeling
  - week 7: training
  - week 8: FST's; search
- weeks 9–13: advanced topics

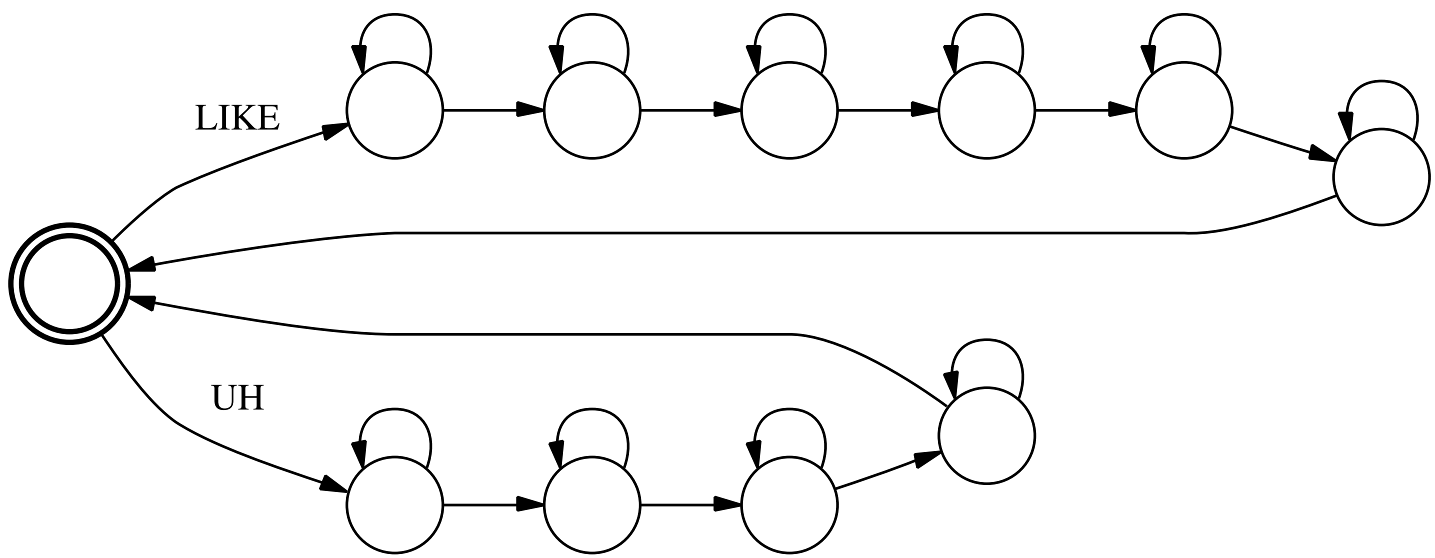
Where Were We? ⇒ LVCSR Decoding

What did we do for small vocabulary tasks?

- graph/FSA representing language model
  - *i.e.*, all allowed word sequences



- expand to underlying HMM

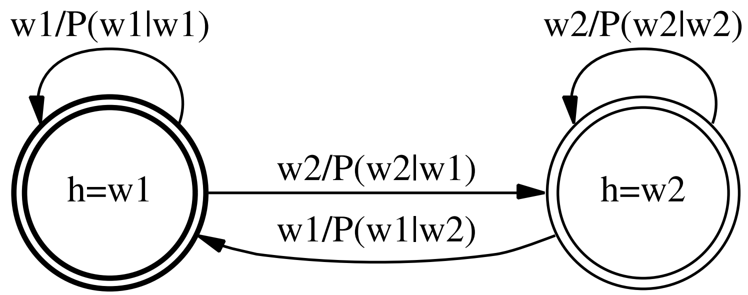


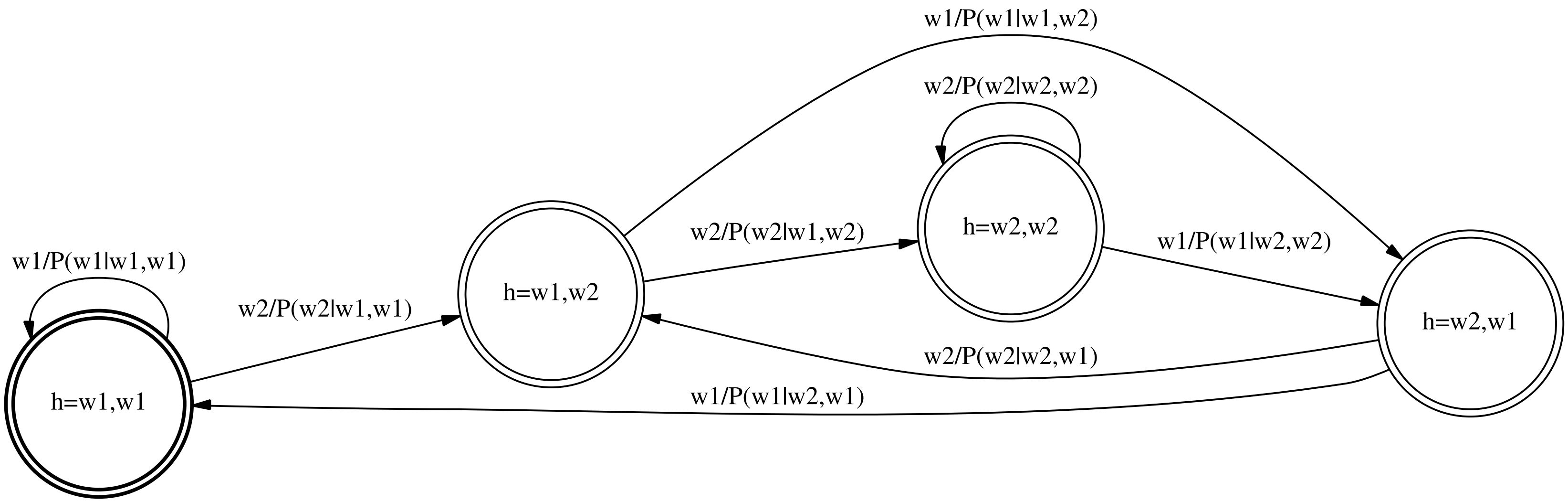
- run the Viterbi algorithm!

Decoding

Well, can we do the same thing for LVCSR?

- Issue 1: Can we express an n-gram model as an FSA?
  - yup

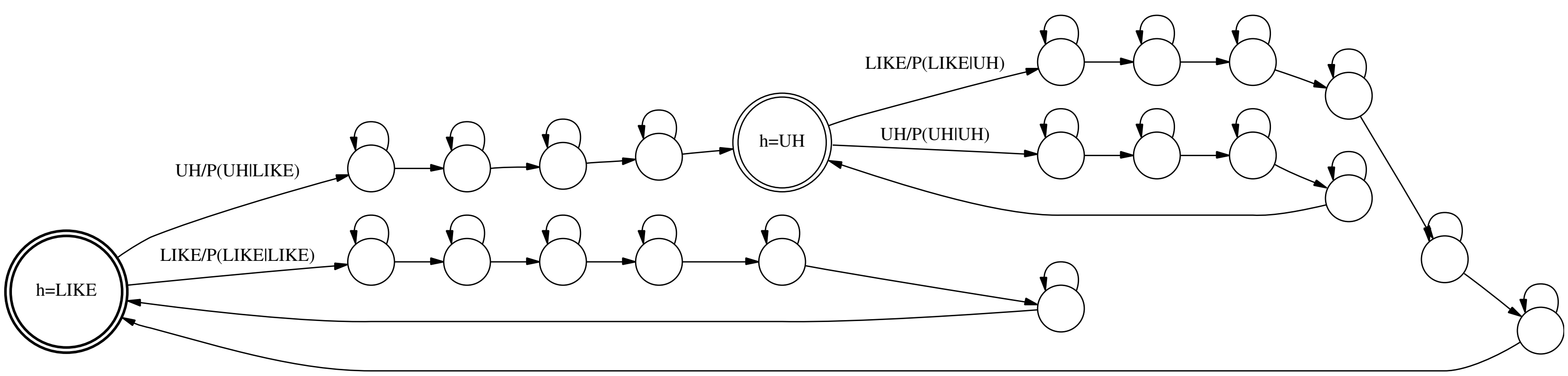




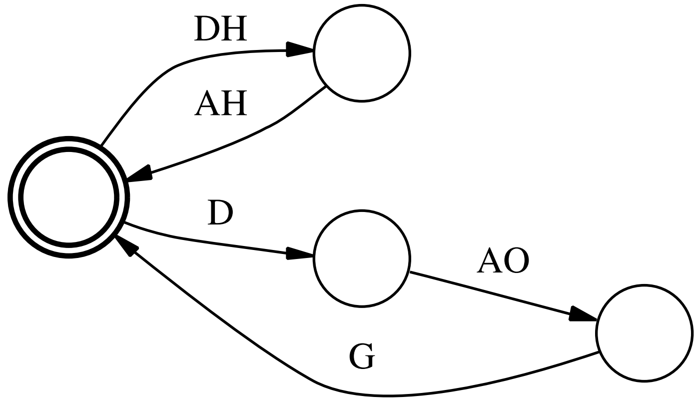
Decoding

Issue 2: How can we expand a word graph to its underlying HMM?

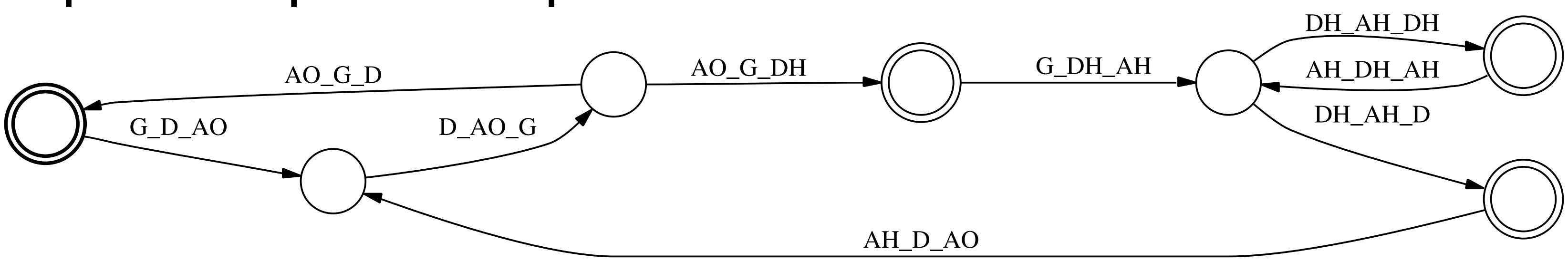
- word models
  - replace each word with its HMM
- CI phone models
  - replace each word with its phone sequence(s)
  - replace each phone with its HMM



Graph Expansion with Context-Dependent Models



- how can we do context-dependent expansion?
  - handling branch points is tricky
- example of triphone expansion



Graph Expansion with Context-Dependent Models

Is there a better way?

- is there some elegant theoretical framework . . .
- that makes it easy to do this type of expansion . . .
- and also makes it easy to do lots of other graph operations useful in ASR?
- $\Rightarrow$ finite-state transducers (FST's)!

Outline

- Unit I: finite-state transducers
  - how do we build decoding graphs for LVCSR?
- Unit II: introduction to search
- Unit III: making decoding graphs smaller
- Unit IV: efficient Viterbi decoding
- Unit V: other decoding paradigms

Remix: A Reintroduction to FSA's and FST's

The *semantics* of (unweighted) finite-state acceptors

- the meaning of an FSA is the set of strings (*i.e.*, token sequences) it accepts
  - set may be infinite
- two FSA's are equivalent if they accept the same set of strings
- things that *don't* affect semantics
  - how labels are distributed along a path
  - invalid paths (paths that don't connect initial and final states)
- see board

You Say Tom-ay-to; I Say Tom-ah-to

- a finite-state acceptor is . . .
  - a set of strings . . .
  - expressed (compactly) using a finite-state machine
- what is a finite-state transducer?
  - a one-to-many mapping from strings to strings
  - expressed (compactly) using a finite-state machine

The Semantics of Finite-State Transducers

- the meaning of an (unweighted) FST is the string mapping it represents
  - a set of strings (possibly infinite) it can accept
    - all other strings are mapped to the empty set
  - for each accepted string . . .
    - the set of strings (possibly infinite) mapped to
- two FST's are equivalent if they represent the same mapping
- things that *don't* affect semantics
  - how labels are distributed along a path
  - invalid paths (paths that don't connect initial and final states)
- see board

The Semantics of Composition

- for a set of strings A (FSA) ...
- for a mapping from strings to strings T (FST) ...
  - let $T(s)$ = the set of strings that s is mapped to
- the composition $A \circ T$ is the set of strings (FSA) ...

$$A \circ T = \bigcup_{s \in A} T(s)$$

- maps all strings in A simultaneously

Graph Expansion as Repeated Composition

- want to expand from set of strings (LM) to set of strings (underlying HMM)
  - how is an HMM a set of strings? (ignoring arc probs)
- can be decomposed into sequence of composition operations
  - words $\Rightarrow$ pronunciation variants
  - pronunciation variants $\Rightarrow$ CI phone sequences
  - CI phone sequences $\Rightarrow$ CD phone sequences
  - CD phone sequences $\Rightarrow$ GMM sequences
- to do graph expansion
  - design several FST's
  - implement *one* operation: composition!

FST Design and The Power of FST's

- figure out which strings to accept (*i.e.*, which strings should be mapped to non-empty sets)
  - (and what "state" we need to keep track of, *e.g.*, for CD expansion)
  - design corresponding FSA
- add in output tokens
  - creating additional states/arcs as necessary

FST Design and The Power of FST's

Context-independent examples (1-state)

- 1:0 mapping
  - removing swear words (two ways)
- 1:1 mapping
  - mapping pronunciation variants to phone sequences
  - one label per arc?
- 1:many mapping
  - mapping from words to pronunciation variants
- 1:infinite mapping
  - inserting optional silence

FST Design and The Power of FST's

- can do more than one "operation" in single FST
- can be applied just as easily to whole LM (infinite set of strings) as to single string

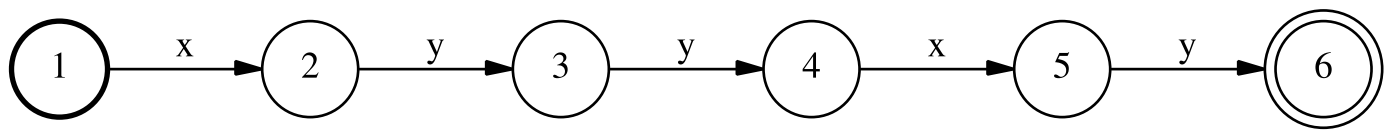
FST Design and The Power of FST's

How to express context-dependent phonetic expansion via FST's?

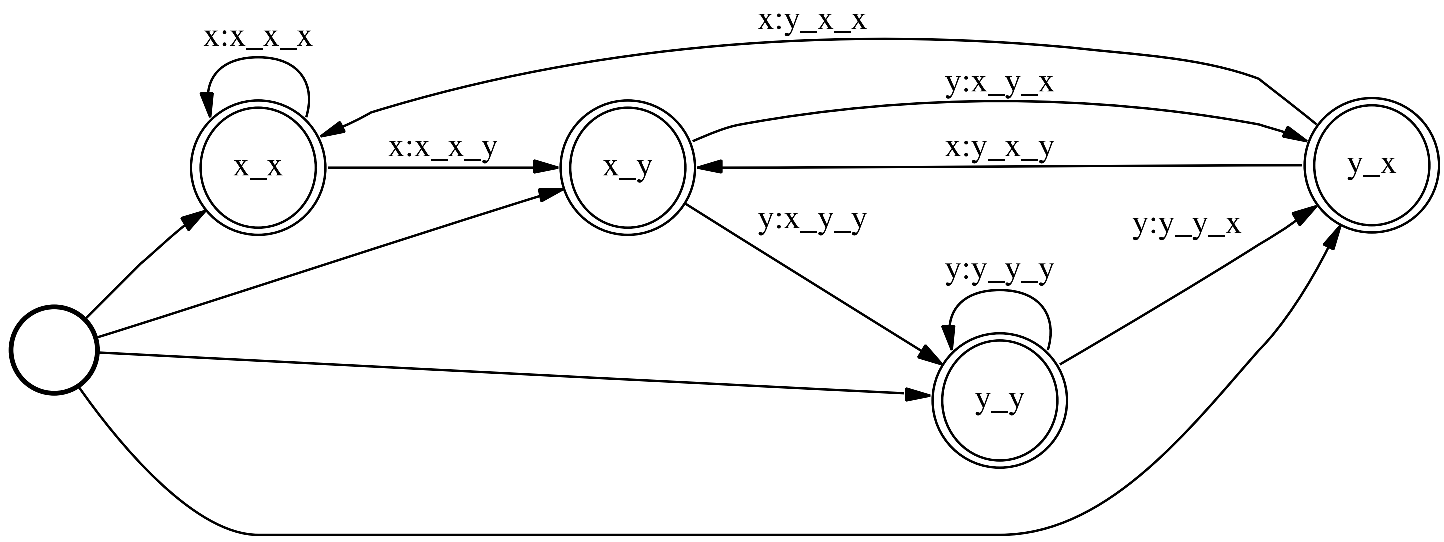
- step 1: rewrite each phone as a triphone
  - rewrite AX as DH_AX_R if DH to left, R to right
- what information do we need to store in each state of FST?
  - strategy: delay output of each phone by one arc

How to Express CD Expansion via FST's?

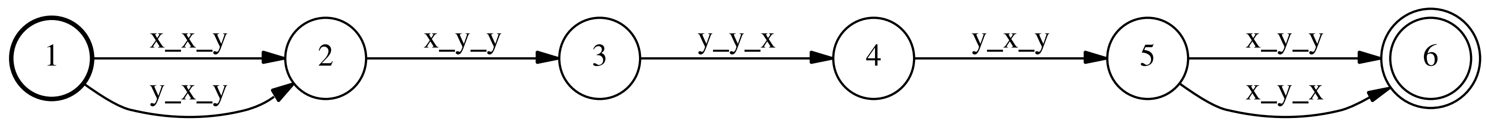
A



T

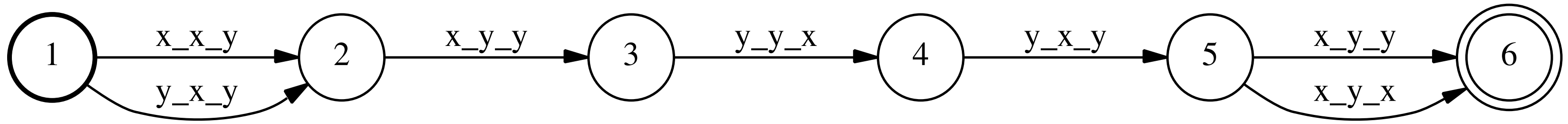


$A \circ T$



How to Express CD Expansion via FST's?

Example



- point: composition automatically expands FSA to correctly handle context!
  - makes multiple copies of states in original FSA . . .
  - that can exist in different triphone contexts
  - (and makes multiple copies of *only* these states)

How to Express CD Expansion via FST's?

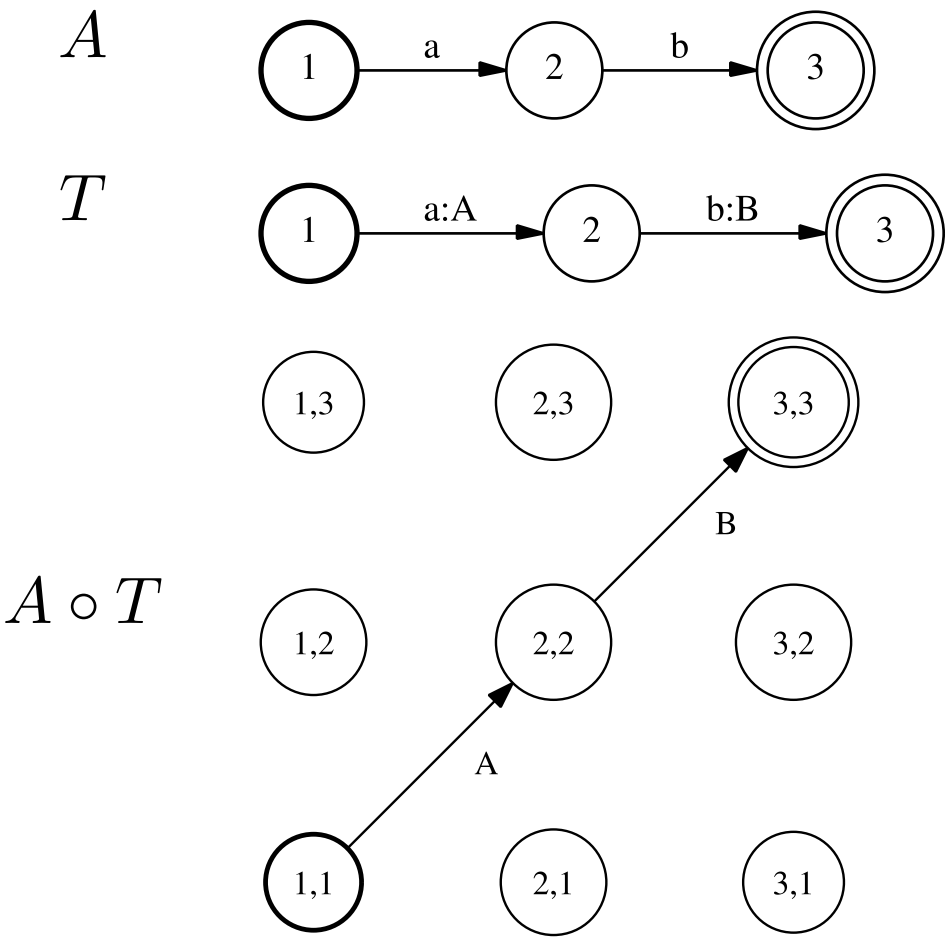
- step 1: rewrite each phone as a triphone
  - rewrite AX as DH_AX_R if DH to left, R to right
- step 2: rewrite each triphone with correct context-dependent HMM for center phone
  - how to do this?
    - note: OK if FST accepts more strings than it needs

Graph Expansion

- final decoding graph: $L \circ T_1 \circ T_2 \circ T_3 \circ T_4$
  - L = language model FSA
  - T_1 = FST mapping from words to pronunciation variants
  - T_2 = FST mapping from pronunciation variants to CI phone sequences
  - T_3 = FST mapping from CI phone sequences to CD phone sequences
  - T_4 = FST mapping from CD phone sequences to GMM sequences
- we know how to design each FST
- how do we implement composition?

Computing Composition

Example

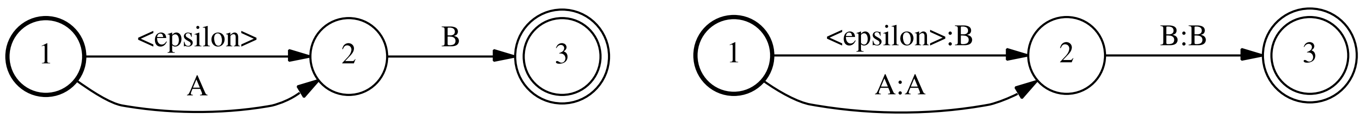


- optimization: start from initial state, build outward

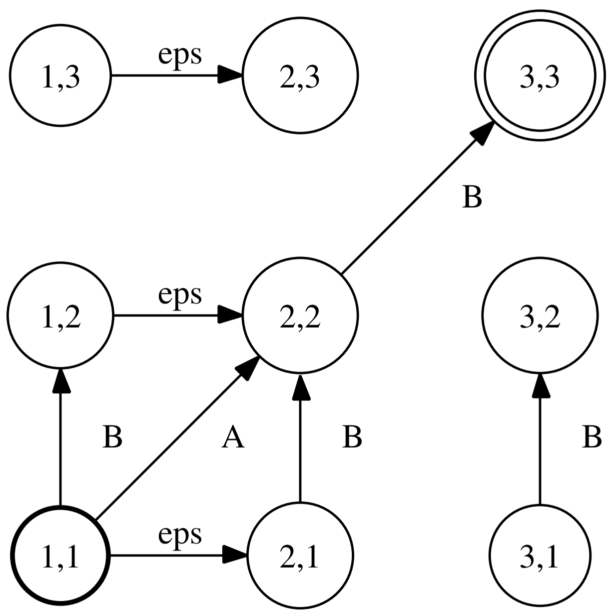
Composition and ϵ-Transitions

- basic idea: can take ϵ-transition in one FSM without moving in other FSM
  - a little tricky to do exactly right
  - do the readings if you care: (Pereira, Riley, 1997)

A, T

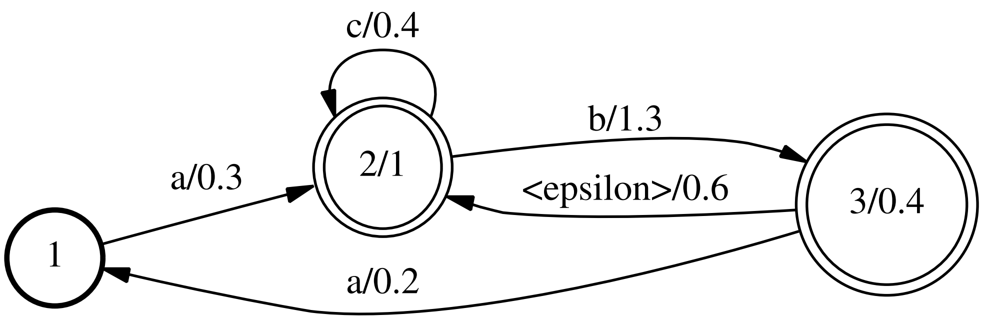


$A \circ T$



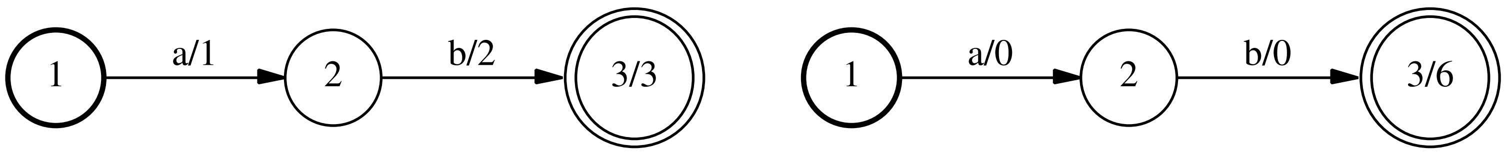
What About Those Probability Thingies?

- *e.g.*, to hold language model probs, transition probs, etc.
- FSM's ⇒ *weighted* FSM's
  - weighted acceptors (WFSA's), transducers (WFST's)
- each arc has a score or *cost*
  - so do final states



Semantics

- total cost of path is sum of its arc costs plus final cost



- typically, we take costs to be negative log probabilities
  - (total probability of path is product of arc probabilities)

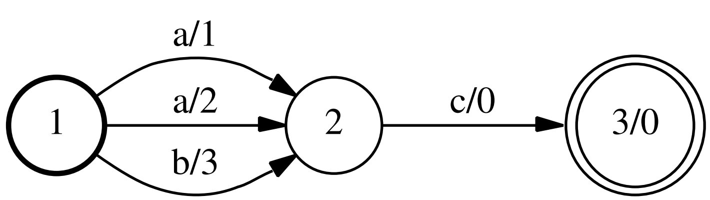
Semantics of Weighted FSA's

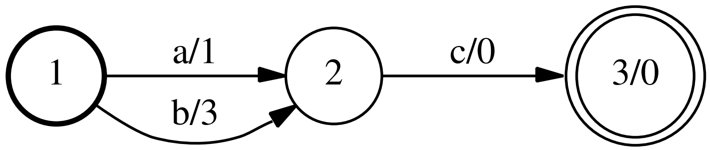
The semantics of weighted finite-state acceptors

- the meaning of an FSA is the set of strings (*i.e.*, token sequences) it accepts
  - each string additionally has a cost
- two FSA's are equivalent if they accept the same set of strings with same costs
- things that *don't* affect semantics
  - how costs or labels are distributed along a path
  - invalid paths (paths that don't connect initial and final states)
- see board

Semantics of Weighted FSA's

- each string has a single cost
- what happens if two paths in FSA labeled with same string?
  - how to compute cost for this string?
- usually, use $\min$ operator to compute combined cost (Viterbi)
  - can combine paths with same labels into one without changing semantics

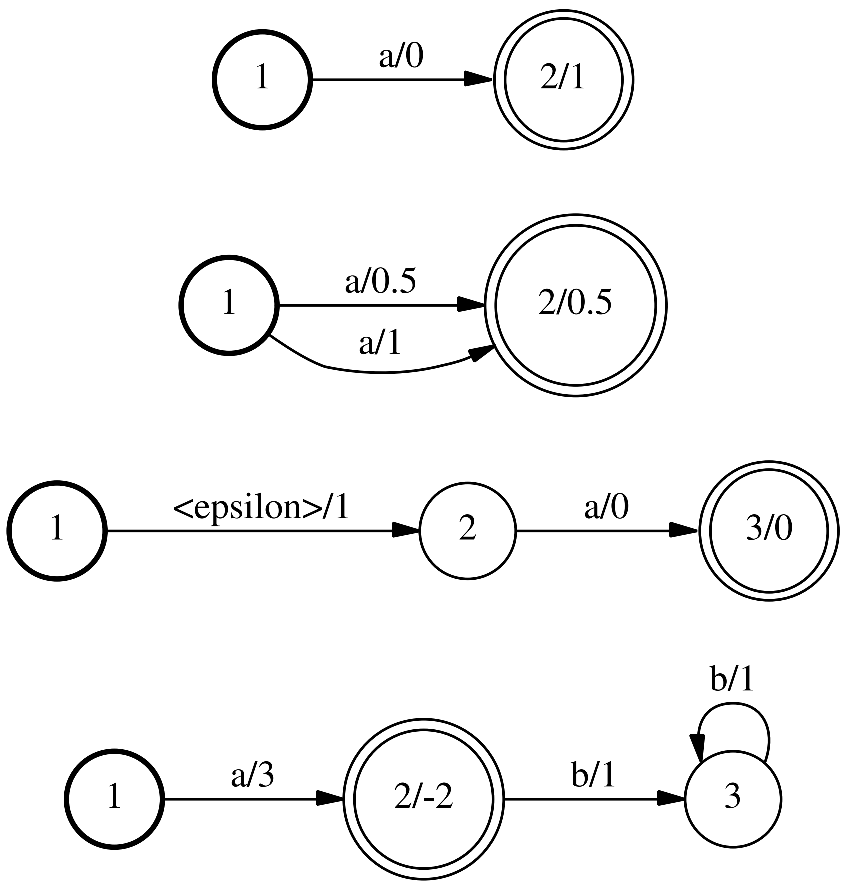




- operations $(+, \min)$ form a *semiring* (the *tropical* semiring)
  - other semirings are possible

Which Of These Is Different From the Others?

- FSM's are equivalent if same label sequences with same costs



The Semantics of Weighted FST's

- the meaning of an (unweighted) FST is the string mapping it represents
  - a set of strings (possibly infinite) it can accept
  - for each accepted string . . .
    - the set of strings (possibly infinite) mapped to . . .
    - and a cost for each string mapped to
- two FST's are equivalent if they represent the same mapping with the same costs
- things that *don't* affect semantics
  - how costs and labels are distributed along a path
  - invalid paths (paths that don't connect initial and final states)

The Semantics of Weighted Composition

- for a set of strings A (WFSA) ...
- for a mapping from strings to strings T (WFST) ...
  - let $T(s)$ = the set of strings that s is mapped to
- the composition $A \circ T$ is the set of strings (WFSA) ...

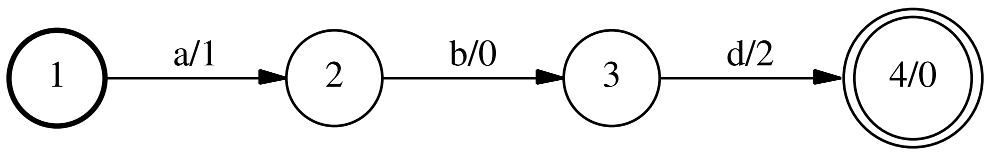
$$A \circ T = \bigcup_{s \in A} T(s)$$

  - cost associated with output string is "sum" of ...
    - cost of input string in A
    - cost of mapping in T

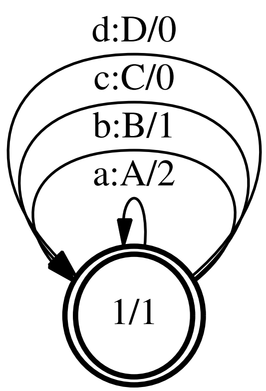
Computing Weighted Composition

Just add arc costs

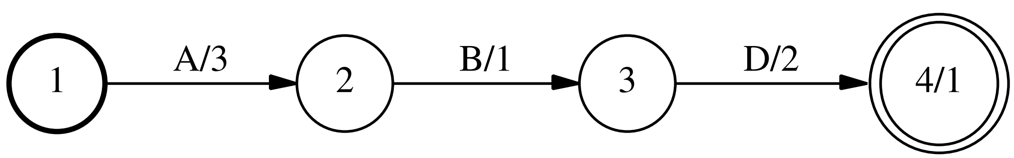
A



T



$A \circ T$



Why is Weighted Composition Useful?

- probability of a path is product of probabilities along path
  - LM probs; arc probs; pronunciation probs; etc.
- if costs are negative log probabilities . . .
  - and use addition to combine scores along paths and in composition . . .
  - probabilities will be combined correctly
- $\Rightarrow$ composition can be used to combine scores from different models

Weighted Graph Expansion

- final decoding graph: $L \circ T_1 \circ T_2 \circ T_3 \circ T_4$
  - L = language model FSA (w/ LM costs)
  - T_1 = FST mapping from words to pronunciation variants (w/ pronunciation costs)
  - T_2 = FST mapping from pronunciation variants to CI phone sequences
  - T_3 = FST mapping from CI phone sequences to CD phone sequences
  - T_4 = FST mapping from CD phone sequences to GMM sequences (w/ HMM transition costs)
- in final graph, each path has correct "total" cost

Recap

- WFSA's and WFST's can represent many important structures in ASR
- graph expansion can be expressed as series of composition operations
  - need to build FST to represent each expansion step, *e.g.*,

```
1 2 THE
2 3 DOG
3
```

  - with composition operation, we're done!
- composition is efficient
- context-dependent expansion can be handled effortlessly

Unit II: Introduction to Search

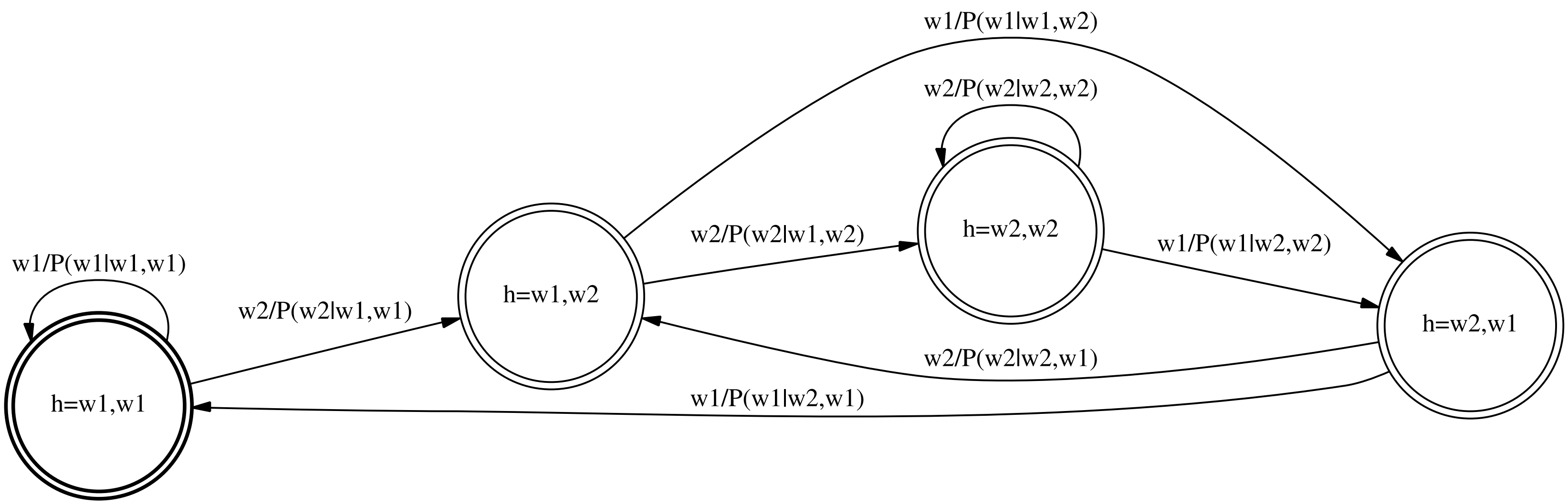
Where are we?

$$\begin{aligned}\text{class}(\mathbf{x}) &= \arg\max_{\omega} P(\omega|\mathbf{x}) \\ &= \arg\max_{\omega} \frac{P(\omega)P(\mathbf{x}|\omega)}{P(\mathbf{x})} \\ &= \arg\max_{\omega} P(\omega)P(\mathbf{x}|\omega)\end{aligned}$$

- can build the one big HMM we need for decoding
- use the Viterbi algorithm on this HMM
- how can we do this efficiently?

Just How Bad Is It?

- trigram model (*e.g.*, vocabulary size $|V| = 2$)



  - $|V|^3$ word arcs in FSA representation
  - each word expands to ∼4 phones ⇒ 4×3 = 12-state HMM
  - if $|V| = 50000$, $50000^3 \times 12 \approx 10^{15}$ states in graph
  - PC's have $\sim 10^9$ bytes of memory

Just How Bad Is It?

- decoding time for Viterbi algorithm
  - in each frame, loop through every state in graph
  - if 100 frames/sec, 10^{15} states . . .
    - how many cells to compute per second?
  - PC's can do $\sim 10^{10}$ floating-point ops per second
- point: cannot use small vocabulary techniques "as is"

Unit II: Introduction to Search

What can we do about the memory problem?

- Approach 1: don't store the whole graph in memory
  - pruning
    - at each frame, keep states with the highest Viterbi scores
    - < 100000 active states out of 10^{15} total states
  - only keep parts of the graph with active states in memory
- Approach 2: shrink the graph
  - use a simpler language model
  - graph-compaction techniques (w/o changing semantics!)
    - compact representation of n-gram models
    - graph *determinization* and *minimization*

Two Paradigms for Search

- Approach 1: dynamic graph expansion
  - since late 1980's
  - can handle more complex language models
  - decoders are incredibly complex beasts
    - *e.g.*, cross-word CD expansion without FST's
    - everyone knew the name of everyone else's decoder
- Approach 2: static graph expansion
  - pioneered by AT&T in late 1990's
  - enabled by minimization algorithms for WFSA's, WFST's
  - static graph expansion is complex
    - theory is clean; doing expansion in $<$2GB RAM is difficult
  - decoding is relatively simple

Static Graph Expansion

- in recent years, more commercial focus on limited-domain systems
  - telephony applications, *e.g.*, replacing directory assistance operators
  - no need for gigantic language models
- static graph decoders are faster
  - graph optimization is performed off-line
- static graph decoders are much simpler
  - not entirely unlike small vocabulary Viterbi decoder

Static Graph Expansion

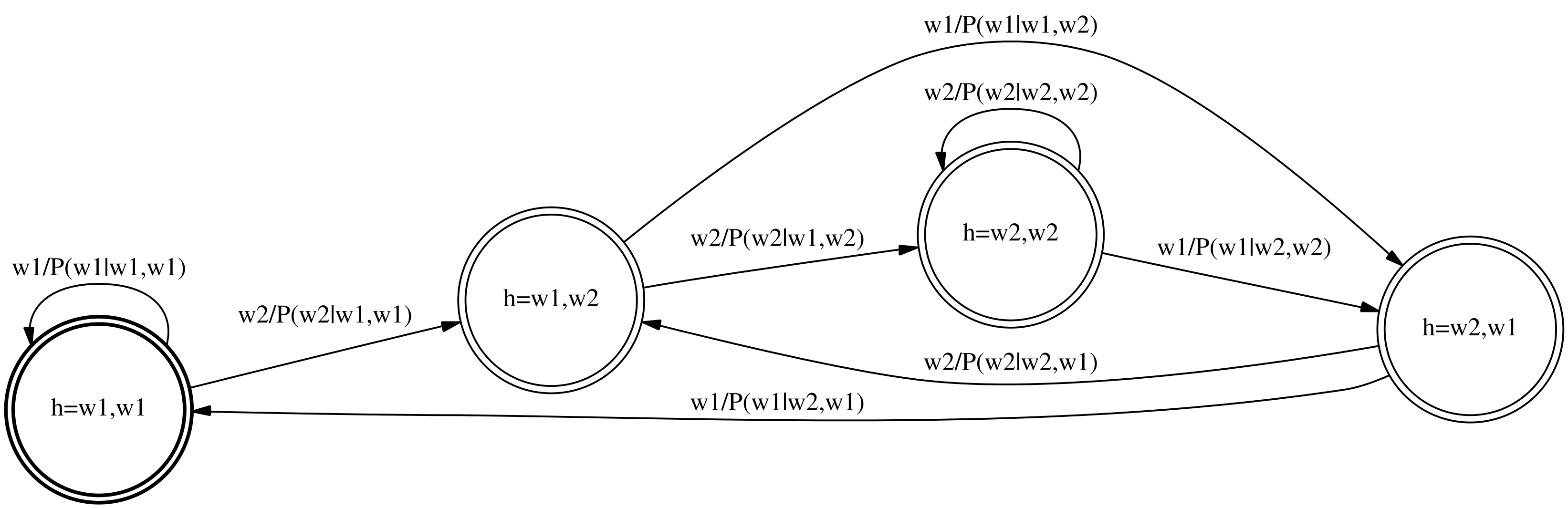
Outline

- Unit III: making decoding graphs smaller
  - shrinking n-gram models
  - graph optimization
- Unit IV: efficient Viterbi decoding
- Unit V: other decoding paradigms
  - dynamic graph expansion revisited
  - stack search (asynchronous search)
  - two-pass decoding

Unit III: Making Decoding Graphs Smaller

Compactly representing n-gram models

- for trigram model, $|V|^2$ states, $|V|^3$ arcs in naive representation



- only a small fraction of the possible $|V|^3$ trigrams will occur in the training data
  - is it possible to keep arcs only for occurring trigrams?

Compactly Representing N-Gram Models

- can express smoothed n-gram models via backoff distributions

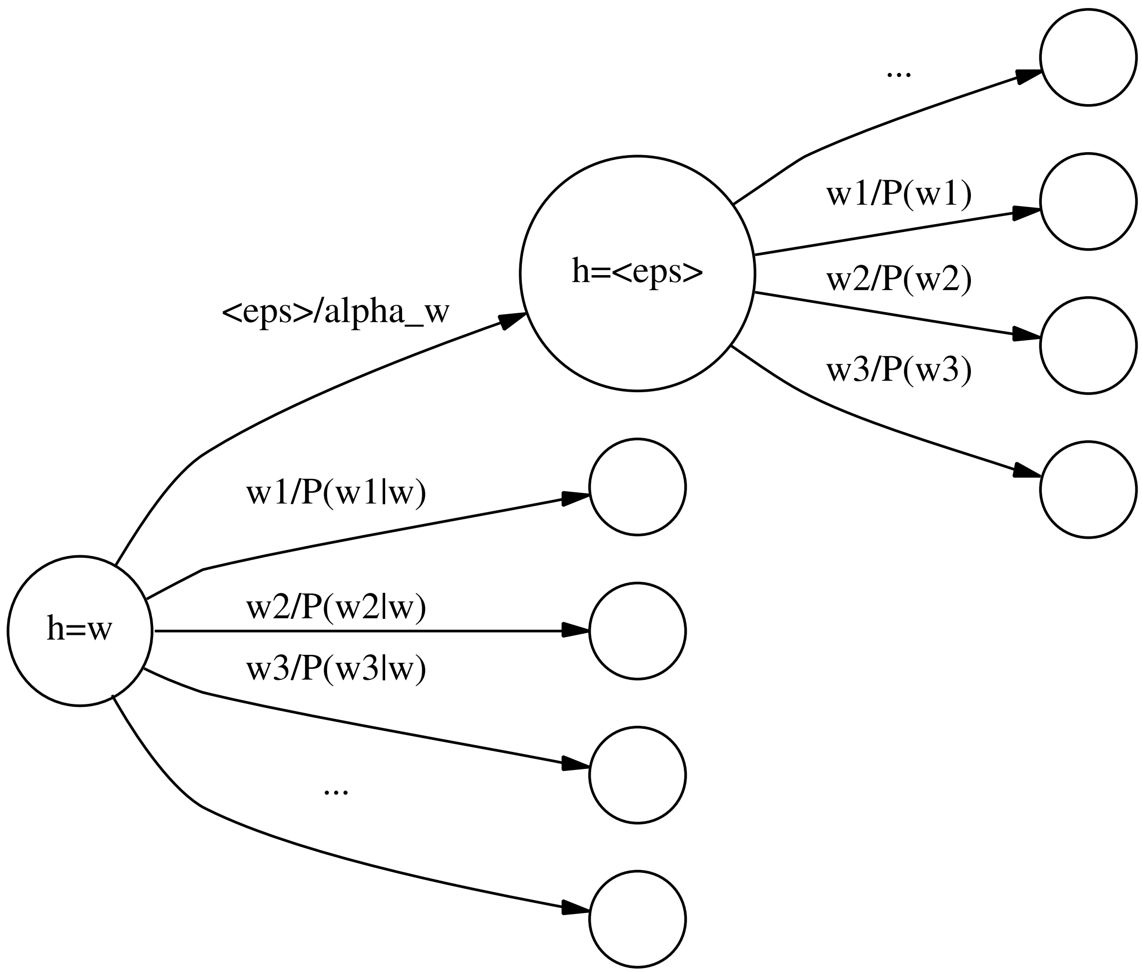
$$P_{\text{smooth}}(w_i|w_{i-1}) = \begin{cases} P_{\text{primary}}(w_i|w_{i-1}) & \text{if count}(w_{i-1}w_i) > 0 \\ \alpha_{w_{i-1}} P_{\text{smooth}}(w_i) & \text{otherwise} \end{cases}$$

- *e.g.*, Witten-Bell smoothing

$$\begin{aligned} P_{\text{WB}}(w_i|w_{i-1}) \quad = \quad & \frac{c_h(w_{i-1})}{c_h(w_{i-1}) + N_{1+}(w_{i-1})} P_{\text{MLE}}(w_i|w_{i-1}) + \\ & \frac{N_{1+}(w_{i-1})}{c_h(w_{i-1}) + N_{1+}(w_{i-1})} P_{\text{WB}}(w_i) \end{aligned}$$

Compactly Representing N-Gram Models

$$P_{\text{smooth}}(w_i|w_{i-1}) = \begin{cases} P_{\text{primary}}(w_i|w_{i-1}) & \text{if count}(w_{i-1}w_i) > 0 \\ \alpha_{w_{i-1}} P_{\text{smooth}}(w_i) & \text{otherwise} \end{cases}$$



Compactly Representing N-Gram Models

- by introducing backoff states
  - only need arcs for n-grams with nonzero count
  - compute probabilities for n-grams with zero count by traversing backoff arcs
- does this representation introduce any error?
  - hint: are there multiple paths with same label sequence?
  - hint: what is “total” cost of label sequence in this case?
- can we make the LM even smaller?

Pruning N-Gram Language Models

Can we make the LM even smaller?

- sure, just remove some more arcs
- which arcs to remove?
  - count cutoffs
    - *e.g.*, remove all arcs corresponding to bigrams $w_{i-1}w_i$ occurring fewer than 10 times in the training data
  - likelihood/entropy-based pruning
    - choose those arcs which when removed, change the likelihood of the training data the least
    - (Seymore and Rosenfeld, 1996), (Stolcke, 1998)

Pruning N-Gram Language Models

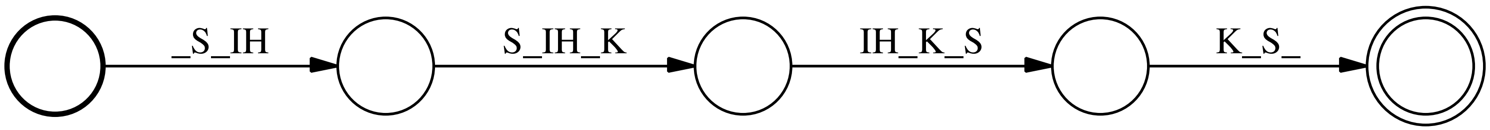
Language model graph sizes

- original: trigram model, $|V|^3 = 50000^3 \approx 10^{14}$ word arcs
- backoff: $>$100M unique trigrams $\Rightarrow$ $\sim$100M word arcs
- pruning: keep $<$5M n-grams $\Rightarrow$ $\sim$5M word arcs
  - 4 phones/word $\Rightarrow$ 12 states/word $\Rightarrow$ $\sim$60M states?
  - we're done?

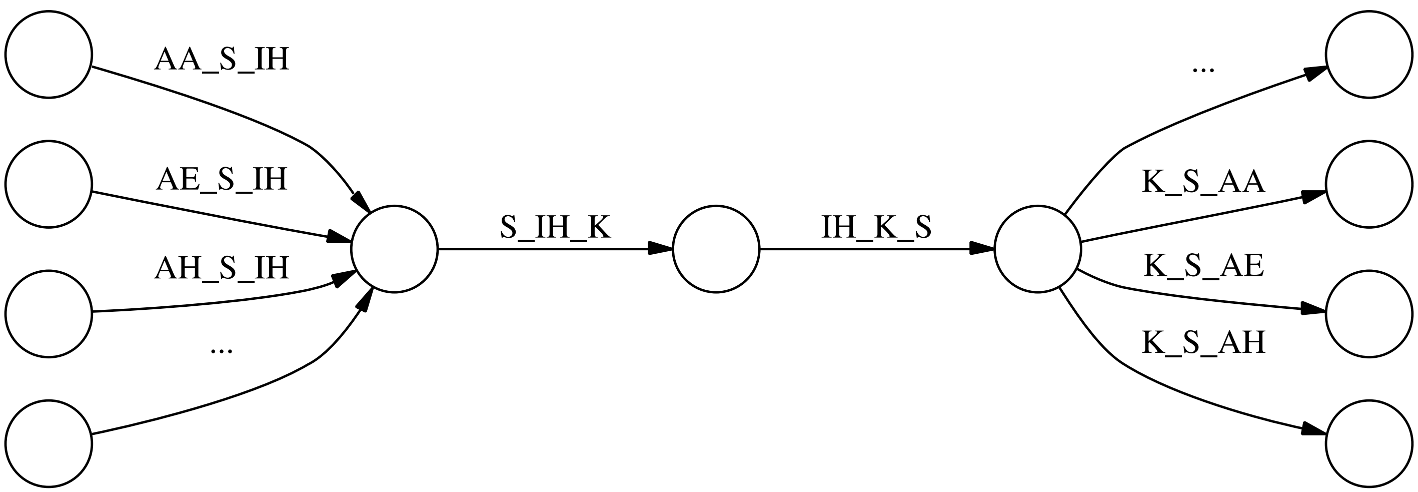
Pruning N-Gram Language Models

Wait, what about cross-word context-dependent expansion?

- with word-internal models, each word really is only ~12 states



- with cross-word models, each word is hundreds of states?
  - 50 CD variations of first three states, last three states



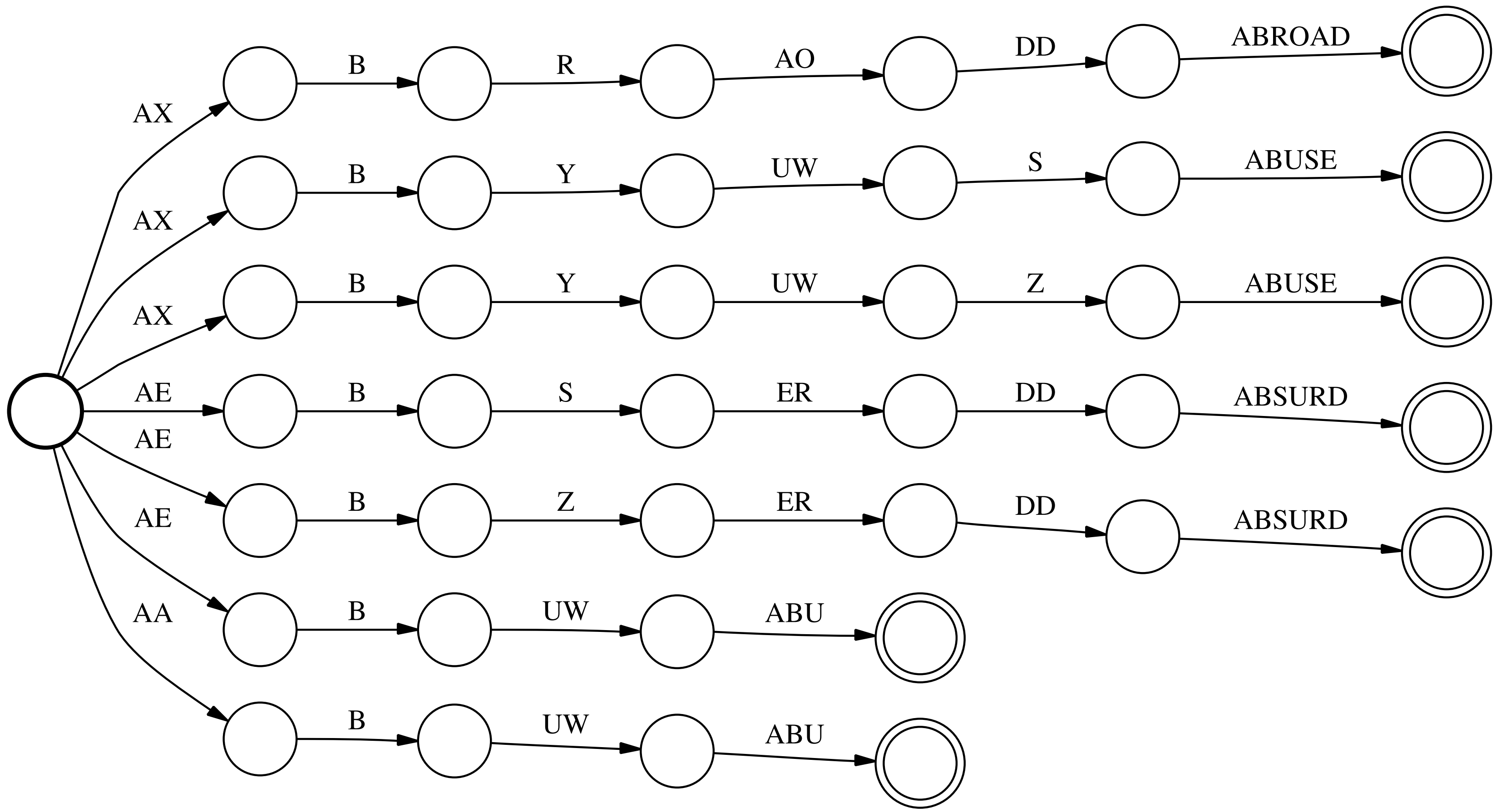
Unit III: Making Decoding Graphs Smaller

What can we do?

- prune the LM word graph even more?
  - will degrade performance
- can we shrink the graph further without changing its meaning?

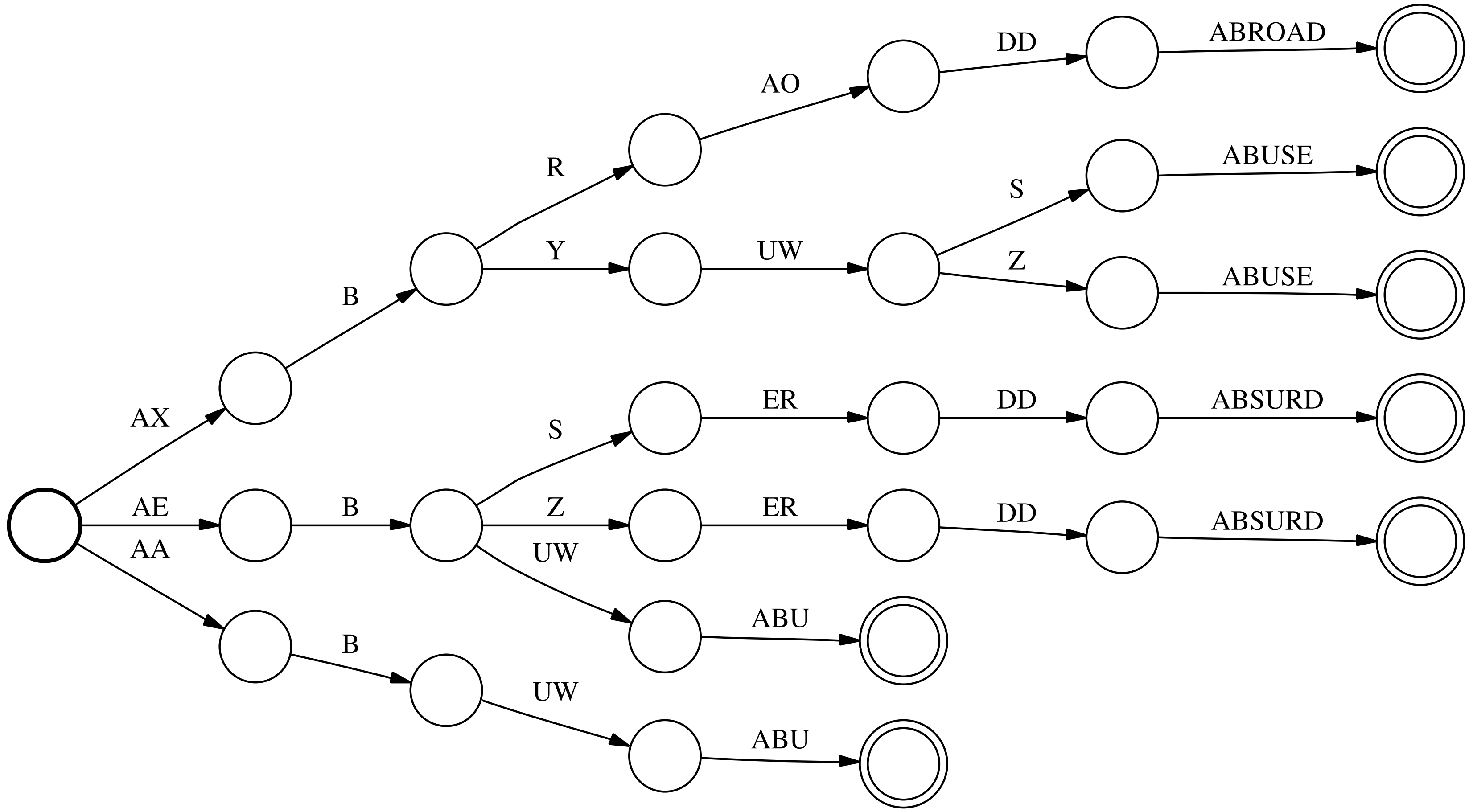
Graph Compaction

- consider word graph for isolated word recognition
  - expanded to phone level: 39 states, 38 arcs



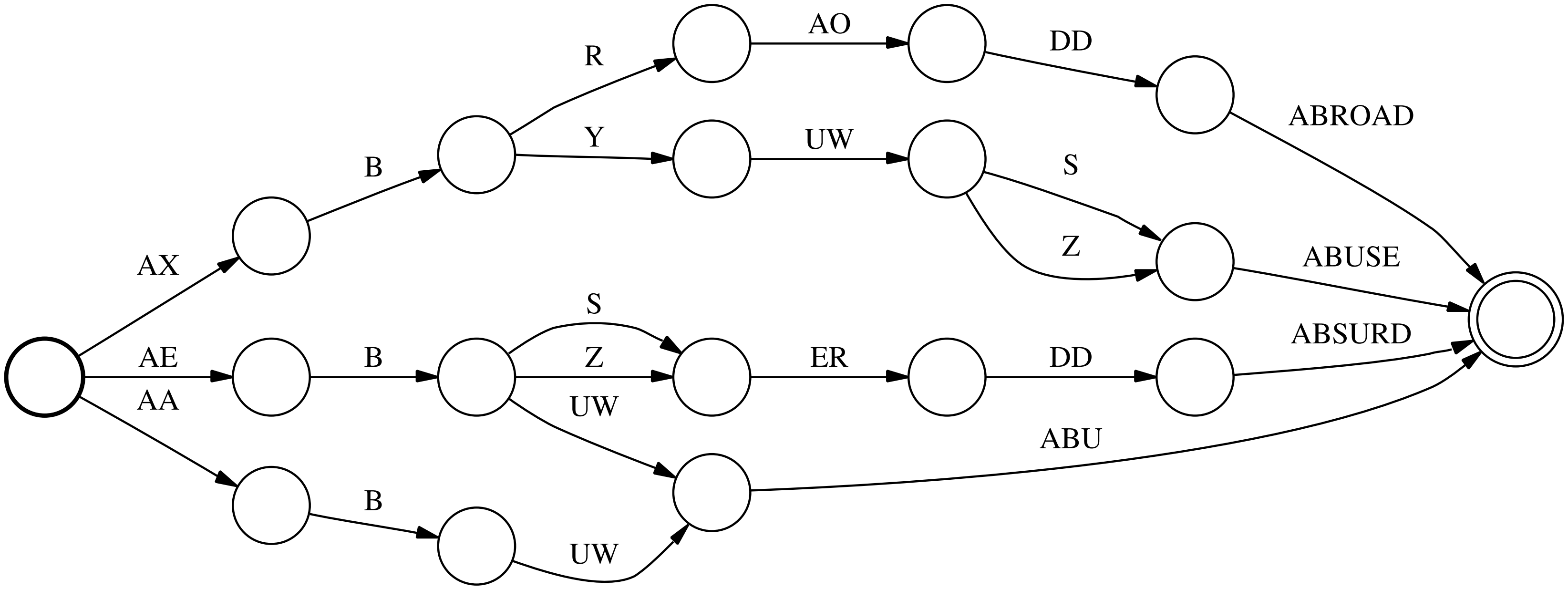
Determinization

- share common prefixes: 29 states, 28 arcs



Minimization

- share common suffixes: 18 states, 23 arcs

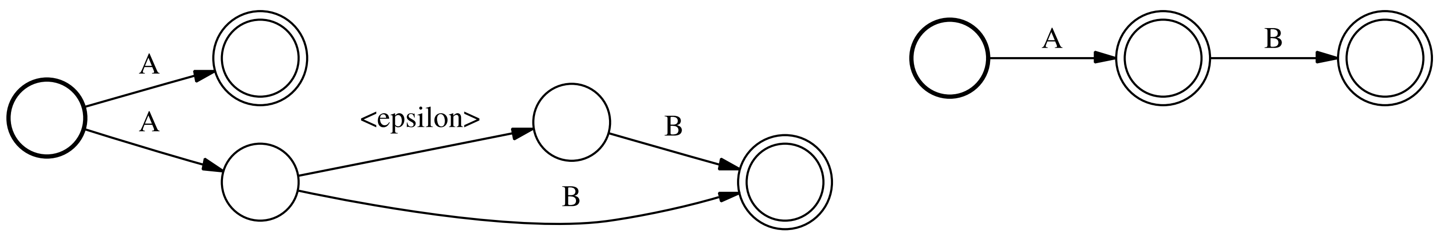


Determinization and Minimization

- by sharing arcs between paths . . .
  - we reduced size of graph by half . . .
  - without changing semantics of graph
  - speeds search (even more than size reduction implies)
- *determinization* — prefix sharing
  - produce *deterministic* version of an FSM
- *minimization* — suffix sharing
  - given a deterministic FSM, find equivalent FSM with minimal number of *states*
- can apply to weighted FSM's and transducers as well
  - *e.g.*, on fully-expanded decoding graphs

Determinization

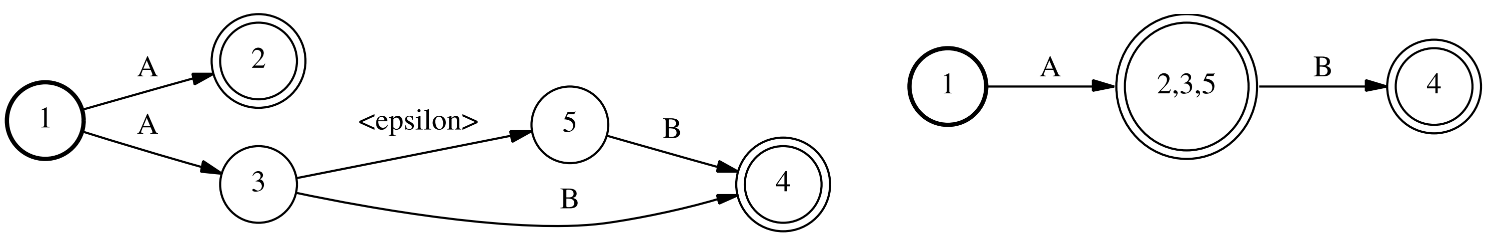
- what is a *deterministic* FSM?
  - no two arcs exiting the same state have the same input label
  - no ϵ arcs
  - *i.e.*, for any input label sequence . . .
    - at most one path from start state labeled with that sequence



- why determinize?
  - may reduce number of states, or may increase number (drastically)
  - speeds search
  - required for minimization algorithm to work as expected

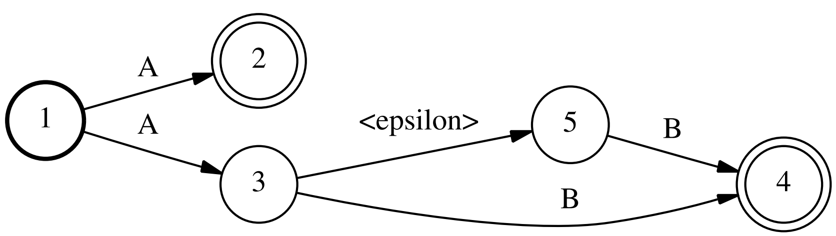
Determinization

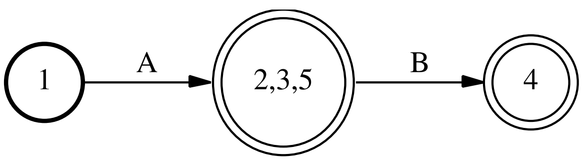
- basic idea
  - for an input label sequence, find set of all states you can reach from start state with that sequence in original FSM
  - collect all such state sets (over all input sequences)
  - map each unique state set into state in new FSM
  - by construction, each label sequence will reach single state in new FSM



Determinization

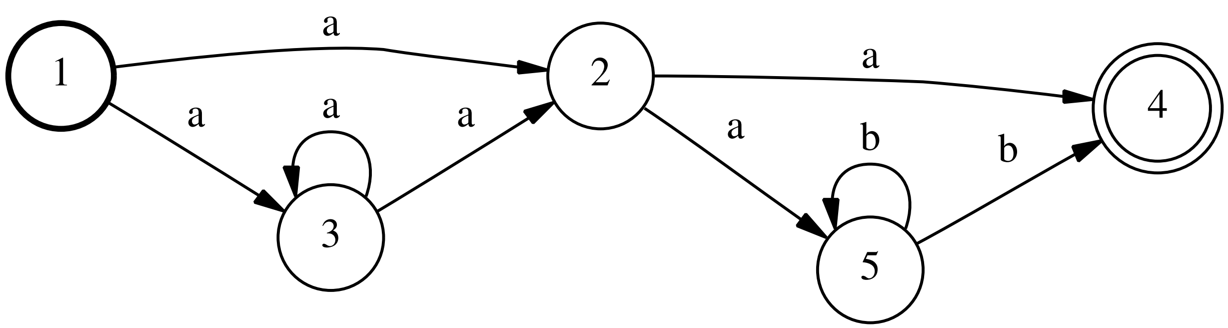
- start from start state
- keep list of state sets not yet expanded
  - for each, find outgoing arcs, creating new state sets as needed
- must follow ϵ arcs when computing state sets

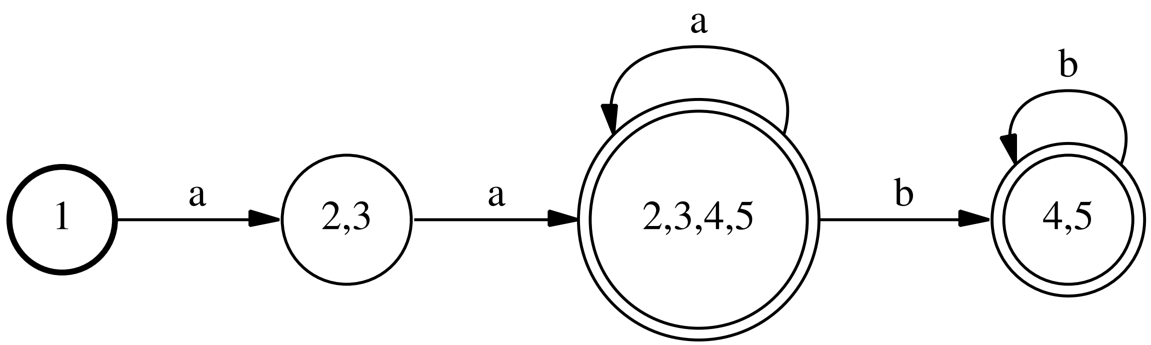




Determinization

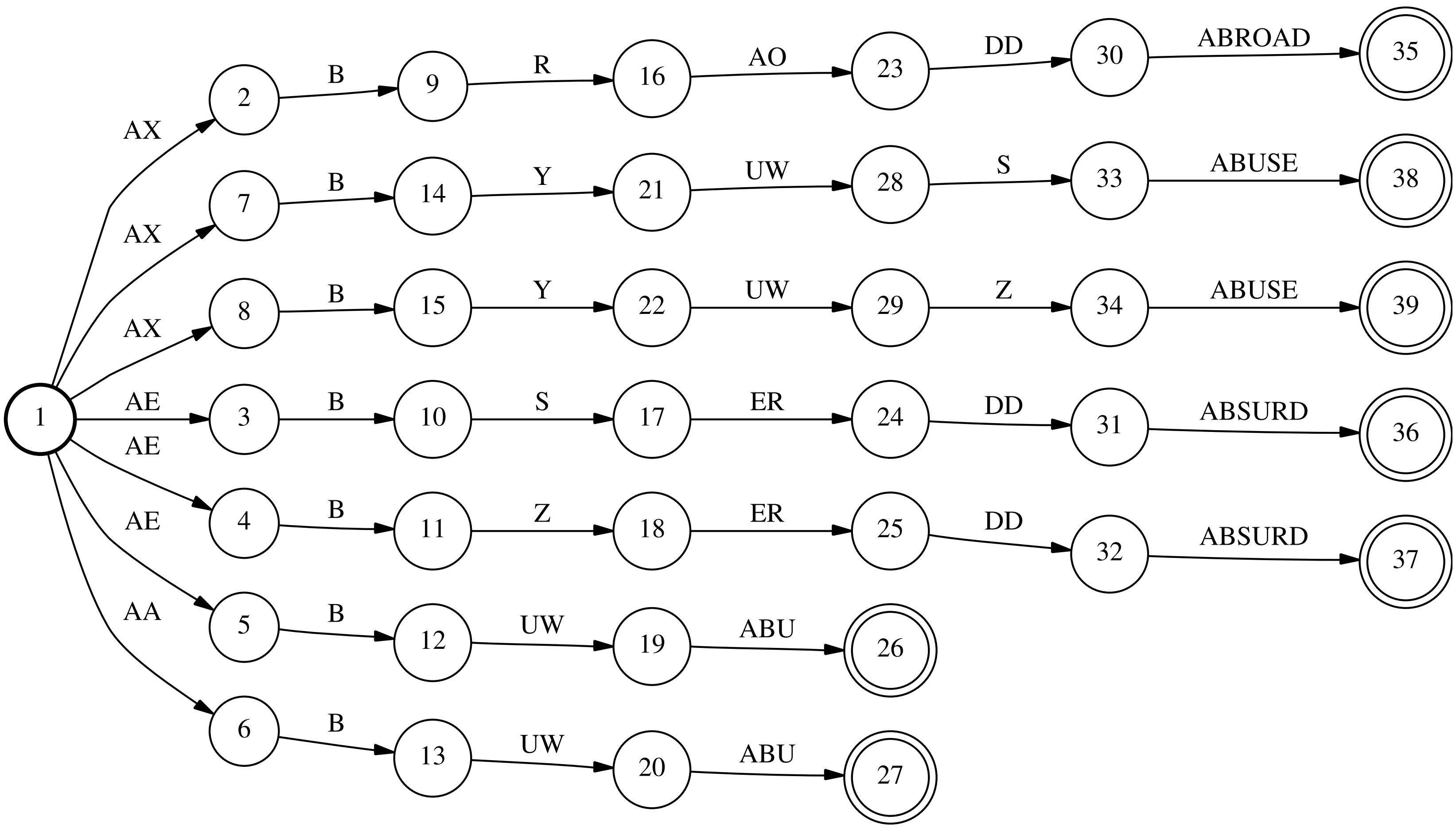
Example 2





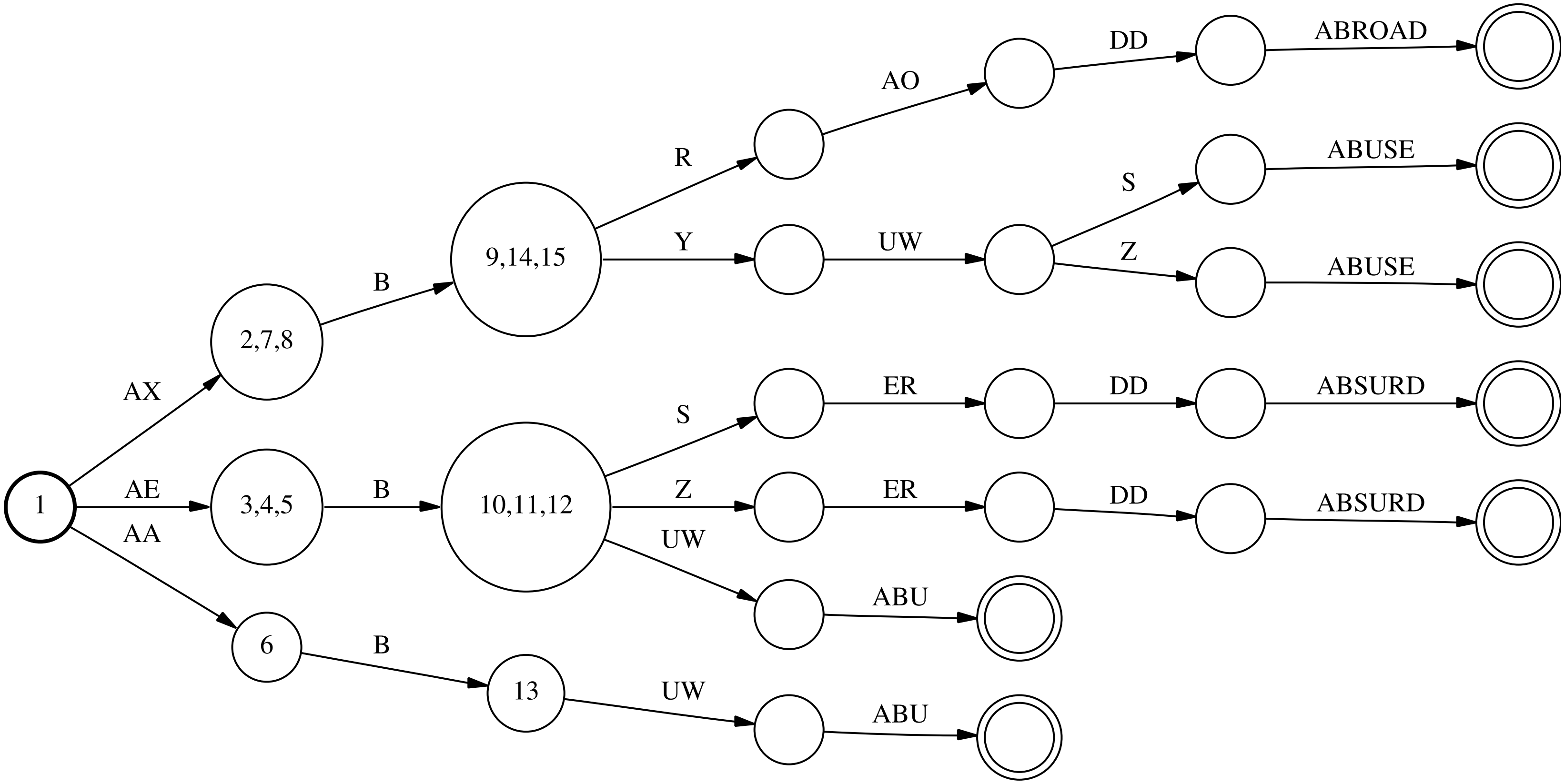
Determinization

Example 3



Determinization

Example 3, cont'd

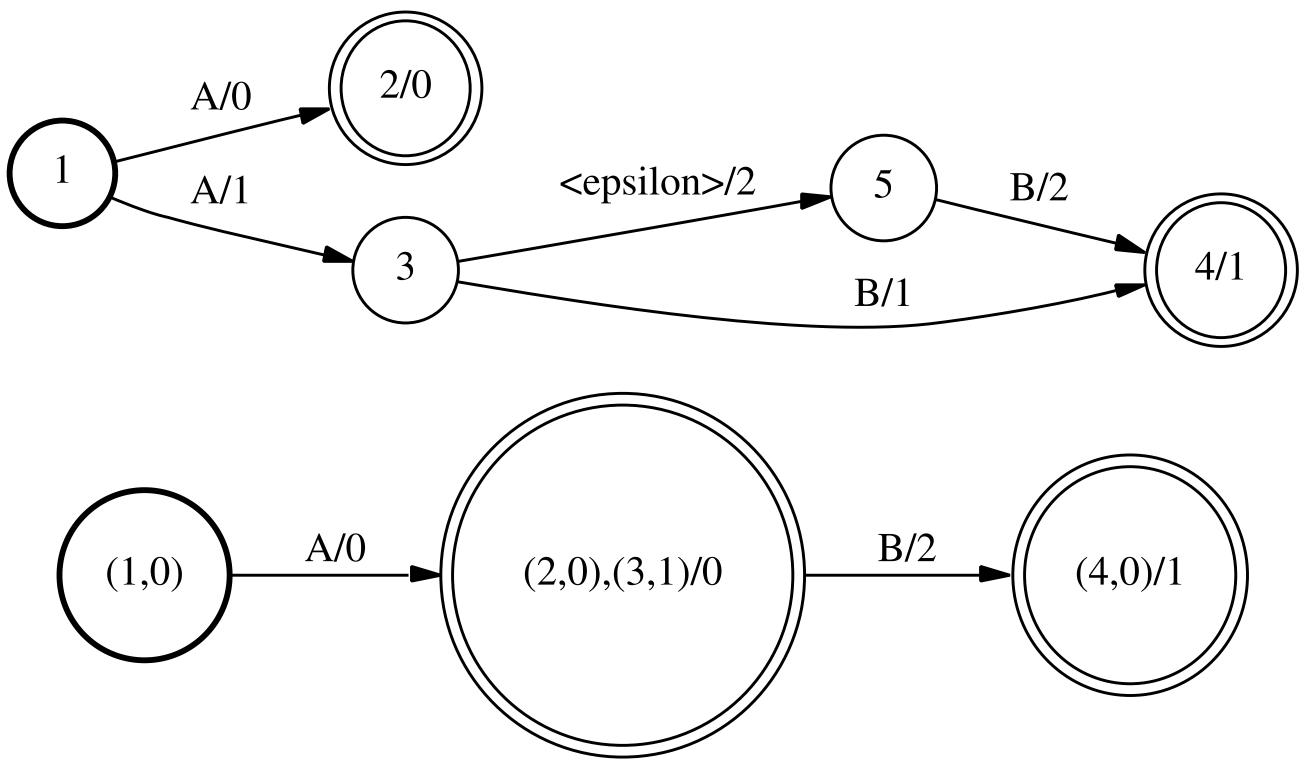


Determinization

- are all unweighted FSA's determinizable?
  - *i.e.*, will the determinization algorithm always terminate?
  - for an FSA with s states, what are the maximum number of states in its determinization?

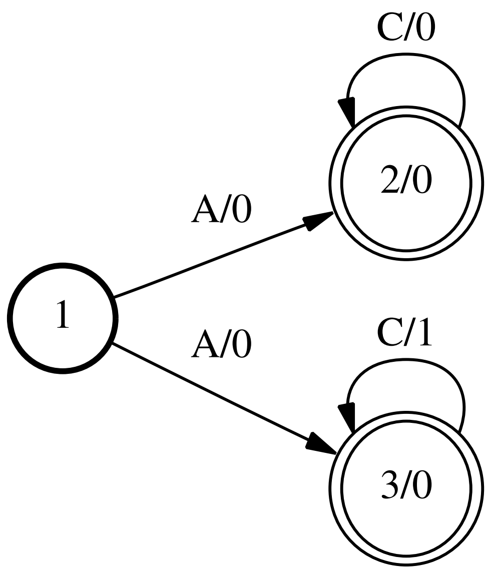
Weighted Determinization

- same idea, but need to keep track of costs
- instead of states in new FSM mapping to state sets $\{s_i\}$ ...
  - they map to sets of state/cost pairs $\{s_i, c_i\}$
  - need to track leftover costs



Weighted Determinization

- will the weighted determinization algorithm always terminate?



Weighted Determinization

What about determinizing finite-state transducers?

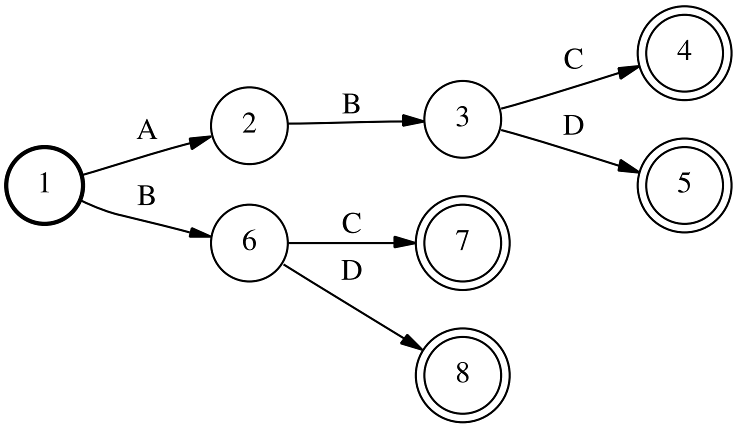
- why would we want to?
  - so we can minimize them; smaller $\Leftrightarrow$ faster?
  - composing a deterministic FSA with a deterministic FSM often produces a (near) deterministic FSA
- instead of states in new FSM mapping to state sets $\{s_i\}$ ...
  - they map to sets of state/output-sequence pairs $\{s_i, o_i\}$
  - need to track leftover output tokens

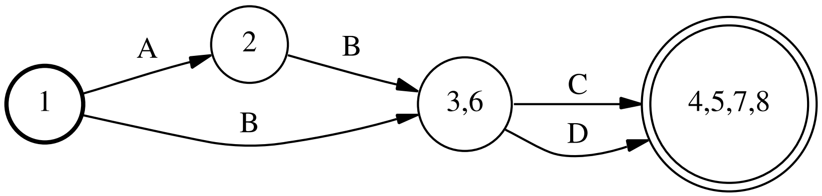
Minimization

- given a deterministic FSM . . .
  - find equivalent FSM with minimal number of *states*
  - number of arcs may be nowhere near minimal
    - minimizing number of arcs is NP-complete

Minimization

- merge states with same set of following strings (or *follow sets*)
  - with acyclic FSA's, can list all strings following each state





states	following strings
1	ABC, ABD, BC, BD
2	BC, BD
3, 6	C, D
4,5,7,8	ϵ

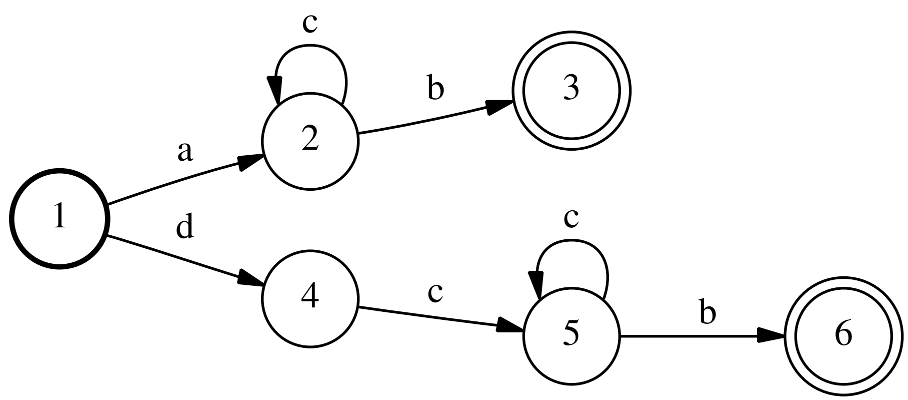
Minimization

- for cyclic FSA's, need a smarter algorithm
  - may be difficult to enumerate all strings following a state
- strategy
  - keep current partitioning of states into disjoint sets
    - each partition holds a set of states that may be mergeable
  - start with single partition
  - whenever find evidence that two states within a partition have different follow sets . . .
    - split the partition
  - at end, each partition contains states with identical follow sets

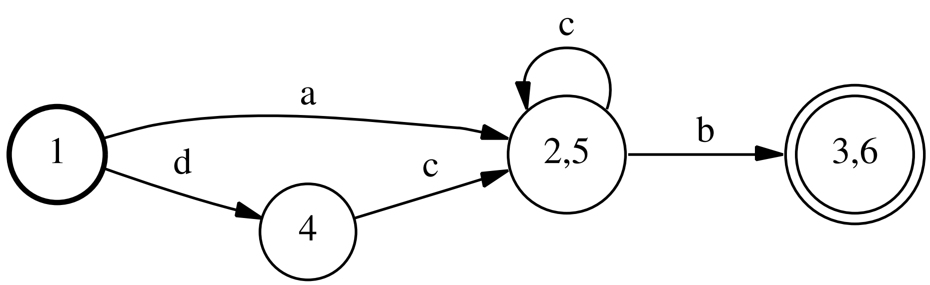
Minimization

- invariant: if two states are in different partitions . . .
  - they have different follow sets
  - converse does not hold
- first split: final and non-final states
  - final states have ϵ in their follow sets; non-final states do not
- if two states in same partition have . . .
  - different number of outgoing arcs, or different arc labels . . .
  - or arcs go to different partitions . . .
  - the two states have different follow sets

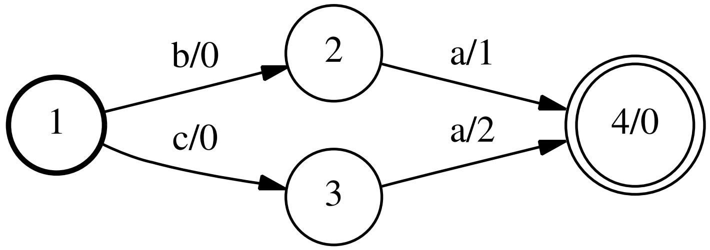
Minimization



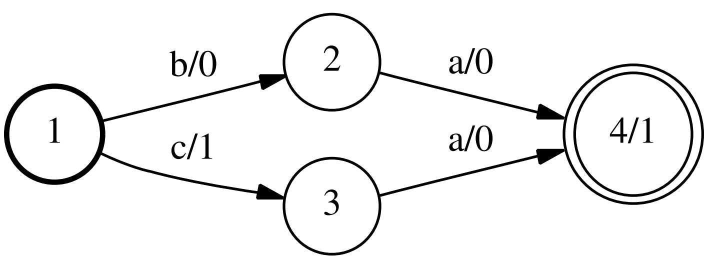
action	evidence	partitioning
		{1,2,3,4,5,6}
split 3,6	final	{1,2,4,5}, {3,6}
split 1	has a arc	{1}, {2,4,5}, {3,6}
split 4	no b arc	{1}, {4}, {2,5}, {3,6}

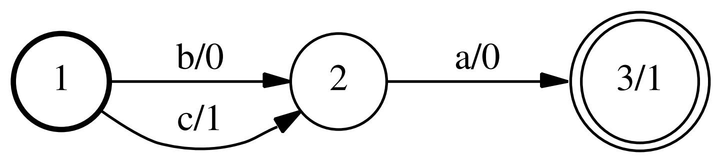


Weighted Minimization



- want to somehow normalize scores such that . . .
  - if two arcs can be merged, they will have the same cost
- then, apply regular minimization where cost is part of label
- *push* operation
  - move scores as far forward (backward) as possible





Weighted Minimization

What about minimization of FST's?

- yeah, it's possible
- use push operation, except on output labels rather than costs
  - move output labels as far forward as possible
- enough said

Pop quiz

- does minimization always terminate?

Unit III: Making Decoding Graphs Smaller

Recap

- backoff representation for n-gram LM's
- n-gram pruning
- use finite-state operations to further compact graph
  - determinization and minimization
- 10^{15} states $\Rightarrow$ 10–20M states/arcs
  - 2–4M n-grams kept in LM

Practical Considerations

- graph expansion
  - start with word graph expressing LM
  - compose with series of FST's to expand to underlying HMM
- strategy: build big graph, then minimize at the end?
  - problem: can't hold big graph in memory
- better strategy: minimize graph after each expansion step
  - never let the graph get too big
- it's an art
  - recipes for efficient graph expansion are still evolving

Where Are We?

- Unit I: finite-state transducers
- Unit II: introduction to search
- Unit III: making decoding graphs smaller
  - now know how to make decoding graphs that can fit in memory
- Unit IV: efficient Viterbi decoding
  - making decoding fast
  - saving memory during decoding
- Unit V: other decoding paradigms

Viterbi Algorithm

$C[0 \ldots T, 1 \ldots S]$`.vProb = 0`

$C[0, \textit{start}]$`.vProb = 1`

`for` t `in` $[0 \ldots (T-1)]$`:`

    `for` $s_{\texttt{src}}$ `in` $[1 \ldots S]$`:`

        `for` a `in` $\textit{outArcs}(s_{\texttt{src}})$`:`

            $s_{\texttt{dst}}$ `=` $\textit{dest}(a)$

            $\textit{curProb}$ `=` $C[t, s_{\texttt{src}}]$`.vProb` $\times$ $\textit{arcProb}(a, t)$

            `if` $\textit{curProb}$ `>` $C[t+1, s_{\texttt{dst}}]$`.vProb:`

                $C[t+1, s_{\texttt{dst}}]$`.vProb =` $\textit{curProb}$

                $C[t+1, s_{\texttt{dst}}]$`.trace =` a

(do backtrace starting from $C[T, \textit{final}]$ to find best path)

Real-Time Decoding

- real-time decoding
  - decoding k seconds of speech in k seconds (*e.g.*, 0.1$\times$ RT)
  - why is this desirable?
- decoding time for Viterbi algorithm, 10M states in graph
  - in each frame, loop through every state in graph
  - say 100 CPU cycles to process each state
  - for each second of audio, $100 \times 10M \times 100 = 10^{11}$ CPU cycles
  - PC's do $\sim 10^9$ cycles/second (*e.g.*, 3GHz P4)
- we cannot afford to evaluate each state at each frame
  - $\Rightarrow$ pruning!

Pruning

- at each frame, only evaluate states with best scores
  - at each frame, have a set of *active* states
  - loop only through active states at each frame
  - for states reachable at next frame, keep only those with best scores
    - these are active states at next frame

for t in $[0 \ldots (T-1)]$:
  for $s_{\texttt{src}}$ in $[1 \ldots S]$:
    for a in *outArcs*$(s_{\texttt{src}})$:
      $s_{\texttt{dst}}$ = *dest*(a)
      update $C[t+1, s_{\texttt{dst}}]$ from $C[t, s_{\texttt{src}}]$, *arcProb*(a, t)

Pruning

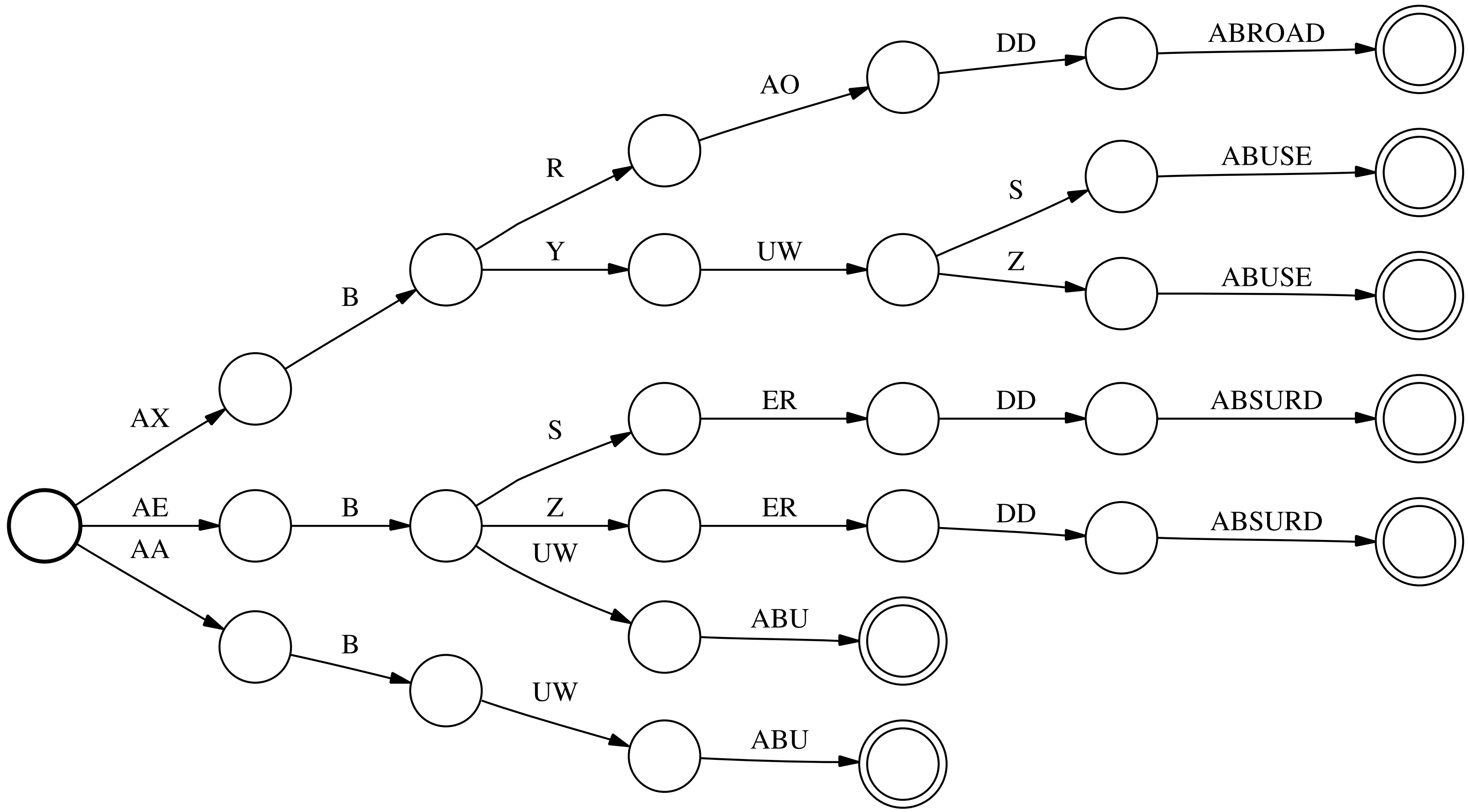
- when not considering every state at each frame . . .
  - we may make *search errors*
  - *i.e.*, we may not find the path with the highest likelihood
- tradeoff: the more states we evaluate . . .
  - the fewer the number of search errors
  - the more computation required
- the field of *search* in ASR
  - minimizing search errors while minimizing computation

Basic Pruning

- *beam* pruning
  - in a frame, keep only those states whose logprobs are within some distance of best logprob at that frame
  - intuition: if a path's score is much worse than current best, it will probably never become best path
  - weakness: if poor audio, overly many states within beam?
- *rank* or *histogram* pruning
  - in a frame, keep k highest scoring states for some k
  - intuition: if the correct path is ranked very poorly, the chance of picking it out later is very low
    - bounds computation per frame
  - weakness: if clean audio, keeps states with bad scores?
- do both

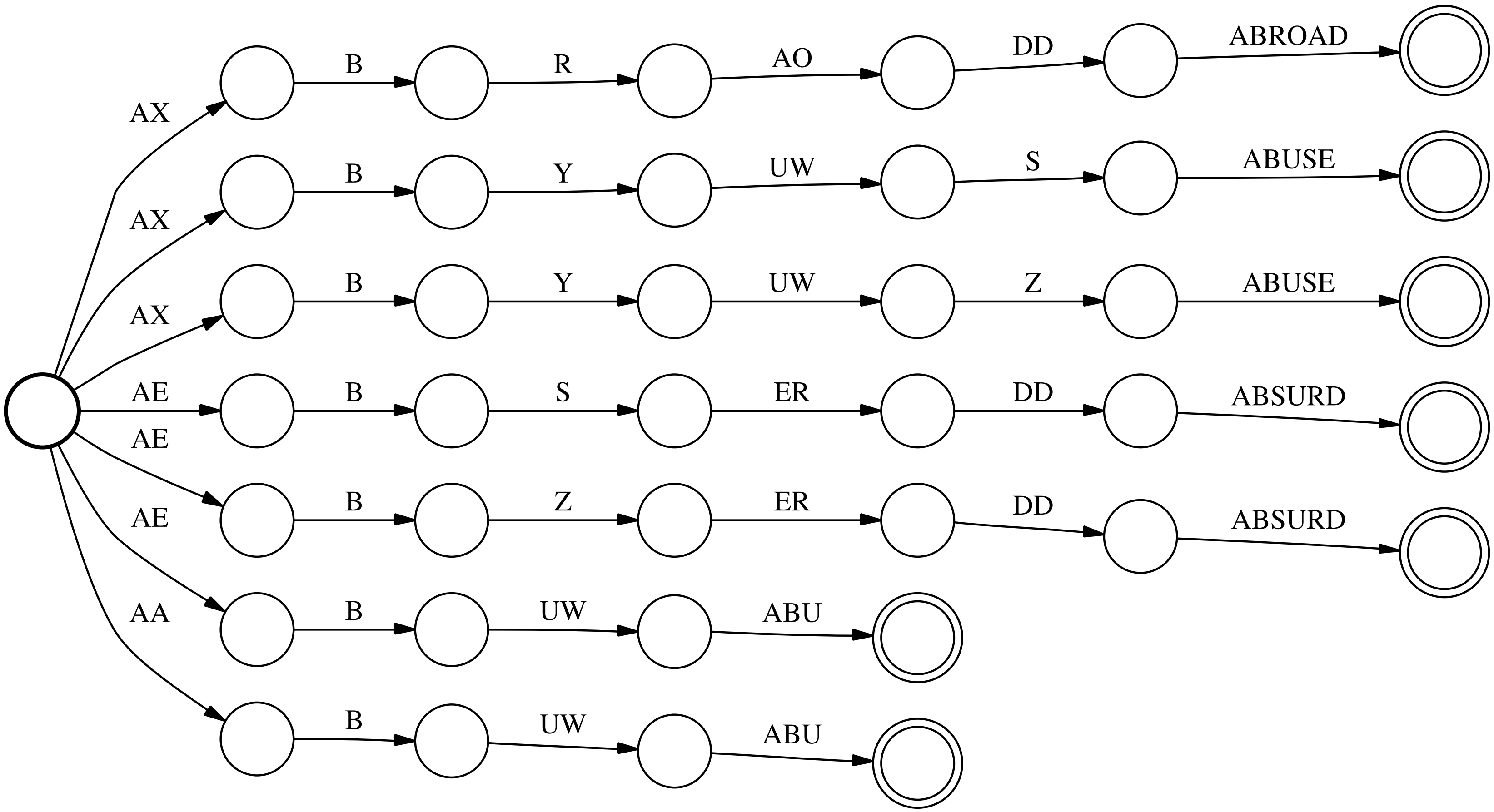
Pruning Visualized

- active states are small fraction of total states (<1%)
  - tend to be localized in small regions in graph



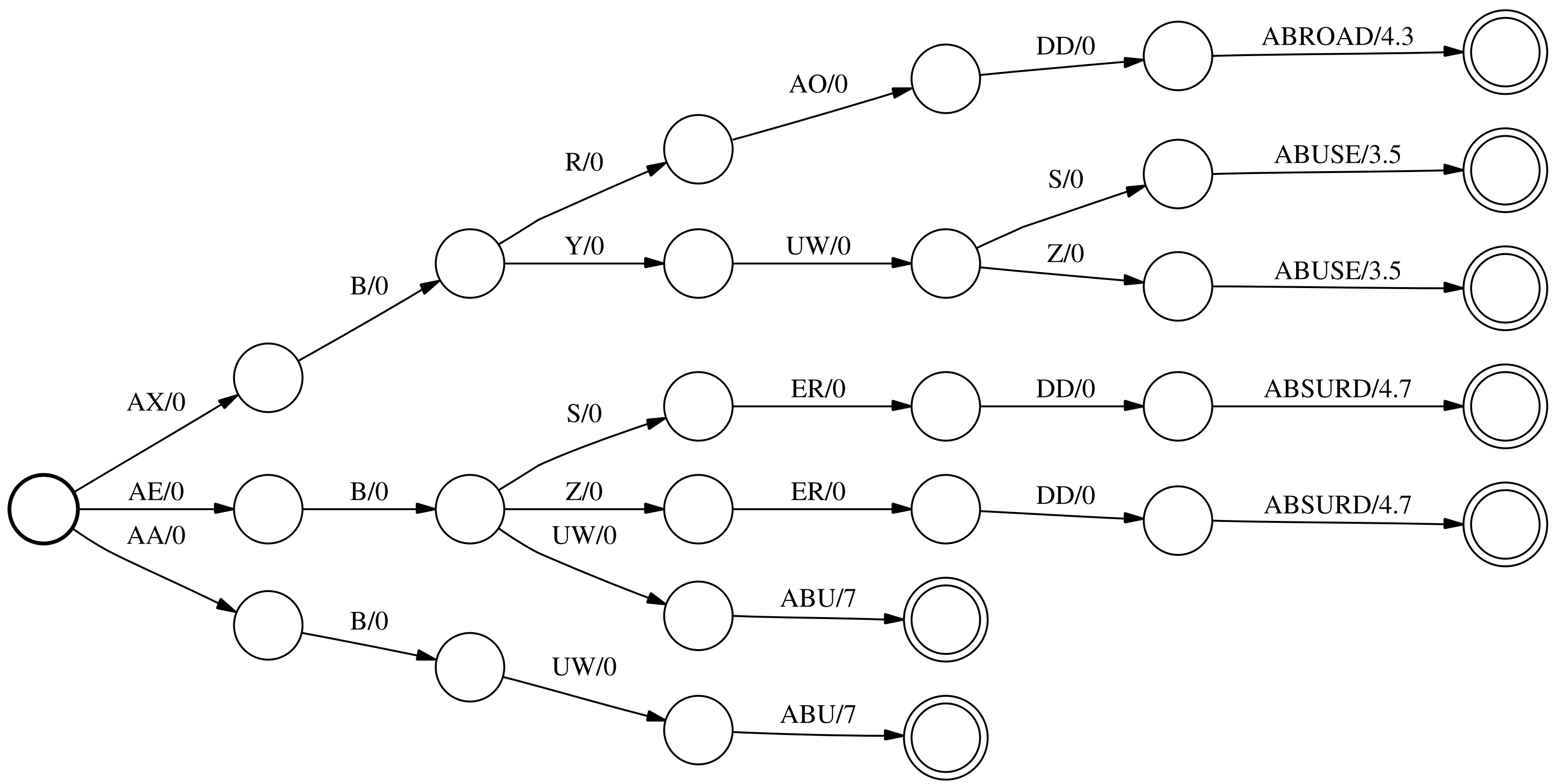
Pruning and Determinization

- most uncertainty occurs at word starts
  - determinization drastically reduces branching at word starts



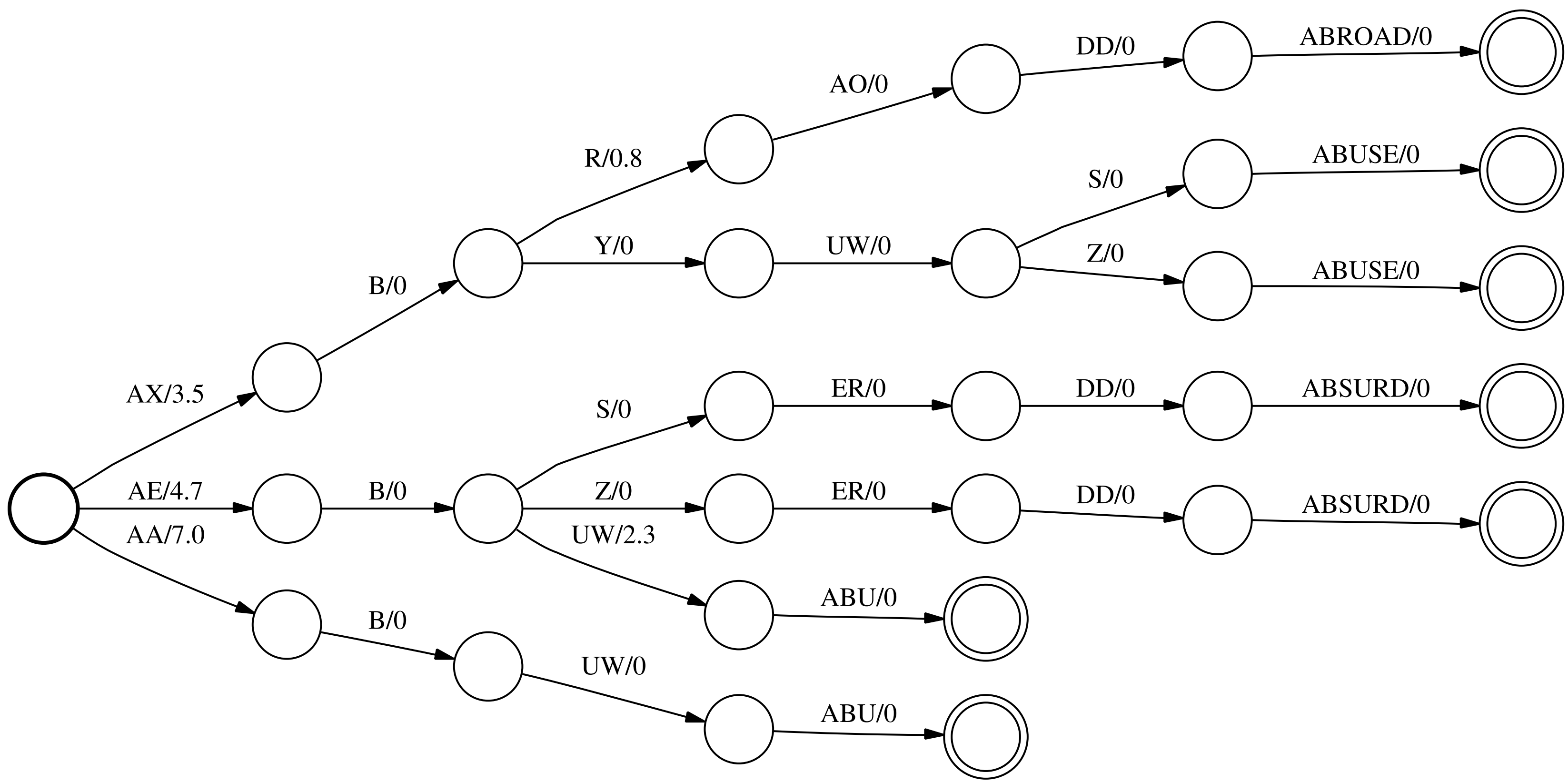
Language Model Lookahead

- in practice, word labels and LM scores at word ends
  - so determinization works
  - what's wrong with this picture? (hint: think beam pruning)



Language Model Lookahead

- move LM scores as far ahead as possible
  - at each point, total cost ⇔ min LM cost of following words
  - *push* operation does this



Historical Note

- in the old days (pre-AT&T-style decoding)
  - people determinized their decoding graphs
  - did the push operation for LM lookahead
  - . . . without calling it determinization or pushing
    - ASR-specific implementations
- nowadays (late 1990's–)
  - implement general finite-state operations
  - FSM toolkits
  - can apply finite-state operations in many contexts in ASR

Efficient Viterbi Decoding

- saving computation
  - pruning
  - determinization
  - LM lookahead
  - $\Rightarrow$ process $\sim$10000 states/frame in $<$ 1x RT on PC's
    - much faster with smaller LM's or allowing more search errors
- saving memory (*e.g.*, 10M state decoding graph)
  - 10 second utterance $\Rightarrow$ 1000 frames
  - 1000 frames $\times$ 10M states = 10 billion cells in DP chart

Saving Memory in Viterbi Decoding

- to compute Viterbi probability (ignoring backtrace) ...
  - do we need to remember whole chart throughout?
- do we need to keep cells for all states or just active states?
  - depends how hard you want to work

```
for t in [0 ... (T − 1)]:
  for s_src in [1 ... S]:
    for a in outArcs(s_src):
      s_dst = dest(a)
      update C[t + 1, s_dst] from C[t, s_src], arcProb(a, t)
```

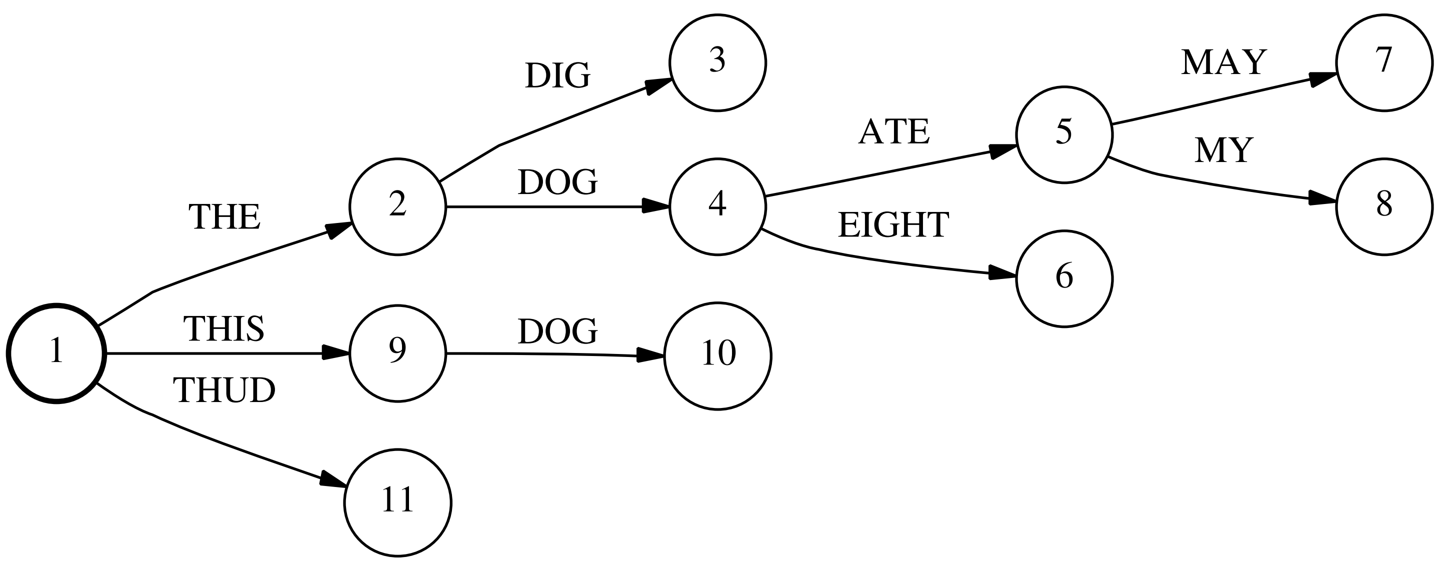
Saving Memory in Viterbi Decoding

What about backtrace information?

- need to remember whole chart?
- conventional Viterbi backtrace
  - remember arc at each frame in best path
  - really, all we want are the words
- instead of keeping pointer to best incoming arc
  - keep pointer to best incoming word sequence
  - can store word sequences compactly in tree

Token Passing

- maintain "word tree"; each node corresponds to word sequence
- backtrace pointer points to node in tree . . .
  - holding word sequence labeling best path to cell
- set backtrace to same node as at best last state . . .
  - unless cross word boundary



Saving Memory in Viterbi Decoding

Memory usage

- before
  - static decoding graph
  - (# states) $\times$ (# frames) cells
- after
  - static decoding graph (shared memory) $\Leftarrow$ the biggie
  - (# (active) states) $\times$ (2 frames) cells
  - backtrace word tree

Where Are We?

- Unit V: other decoding paradigms
  - dynamic graph expansion — saving memory
  - stack search — best-first search
  - two-pass decoding — enable complex models

Two Approaches to Decoding

- Approach 1: dynamic graph expansion
  - don't store the whole graph in memory
  - only keep parts of the graph with active states in memory
  - can use more complex LM's
- Approach 2: static graph expansion
  - just shrink the graph
  - use a simpler language model
  - faster

Dynamic Graph Expansion

- how can we store a really big graph such that . . .
  - it doesn't take that much memory, but . . .
  - easy to expand any part of it that we need
- observation: composition is associative

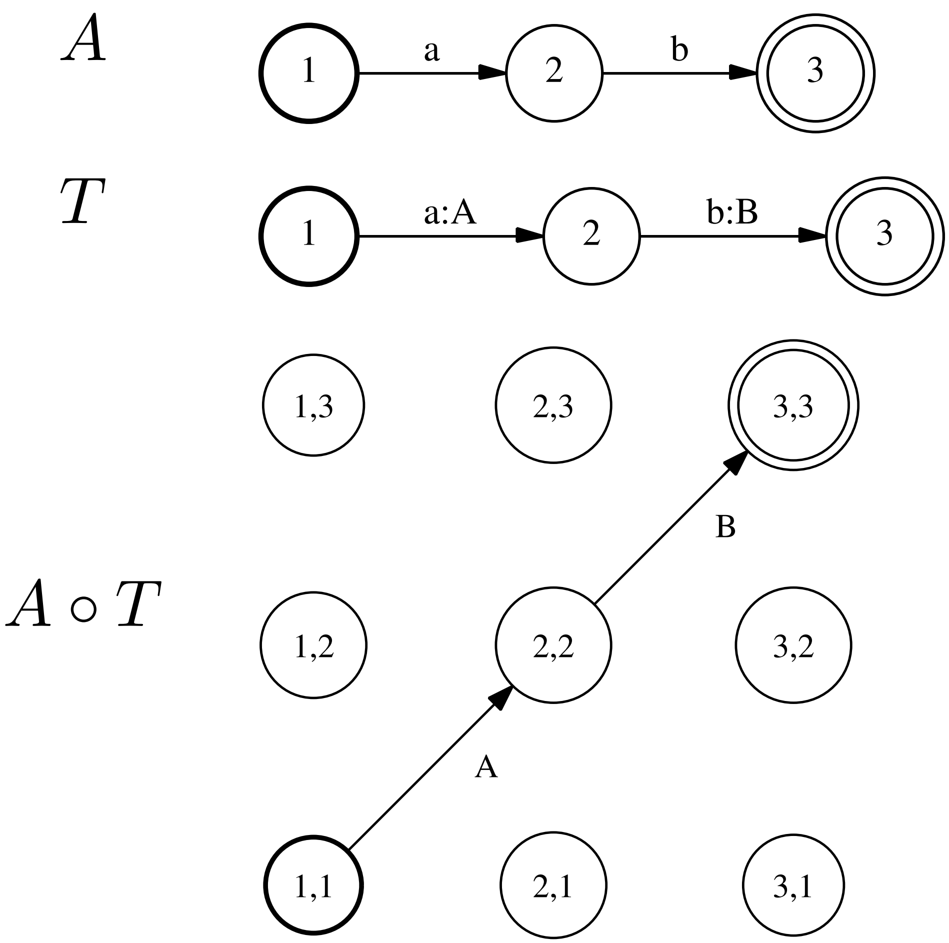
$$(A \circ T_1) \circ T_2 = A \circ (T_1 \circ T_2)$$

- observation: decoding graph is composition of LM with a bunch of FST's

$$\begin{aligned} G_{\text{decode}} &= A_{\text{LM}} \circ T_{\text{wd}\to\text{pn}} \circ T_{\text{CI}\to\text{CD}} \circ T_{\text{CD}\to\text{HMM}} \\ &= A_{\text{LM}} \circ (T_{\text{wd}\to\text{pn}} \circ T_{\text{CI}\to\text{CD}} \circ T_{\text{CD}\to\text{HMM}}) \end{aligned}$$

Dynamic Graph Expansion

Computing composition



Dynamic Graph Expansion

- for a graph $G = A \circ T \ldots$
  - easy to calculate outgoing arcs of a state $s_G = (s_A, s_T)$

$$G_{\text{decode}} = A_{\text{LM}} \circ (T_{\text{wd}\to\text{pn}} \circ T_{\text{CI}\to\text{CD}} \circ T_{\text{CD}\to\text{HMM}})$$

- idea: just store graphs A_{LM} and $T = T_{\text{wd}\to\text{pn}} \circ T_{\text{CI}\to\text{CD}} \circ T_{\text{CD}\to\text{HMM}}$
  - easy to calculate outgoing arcs of any state in G_{decode}
  - in active state list, each state is represented as pair of states (s_A, s_T)
- instead of storing one big graph, store two smaller graphs
  - minimize each of the smaller graphs
  - other decompositions are possible
  - dynamic graph expansion was really complicated before FSM perspective

Where Are We?

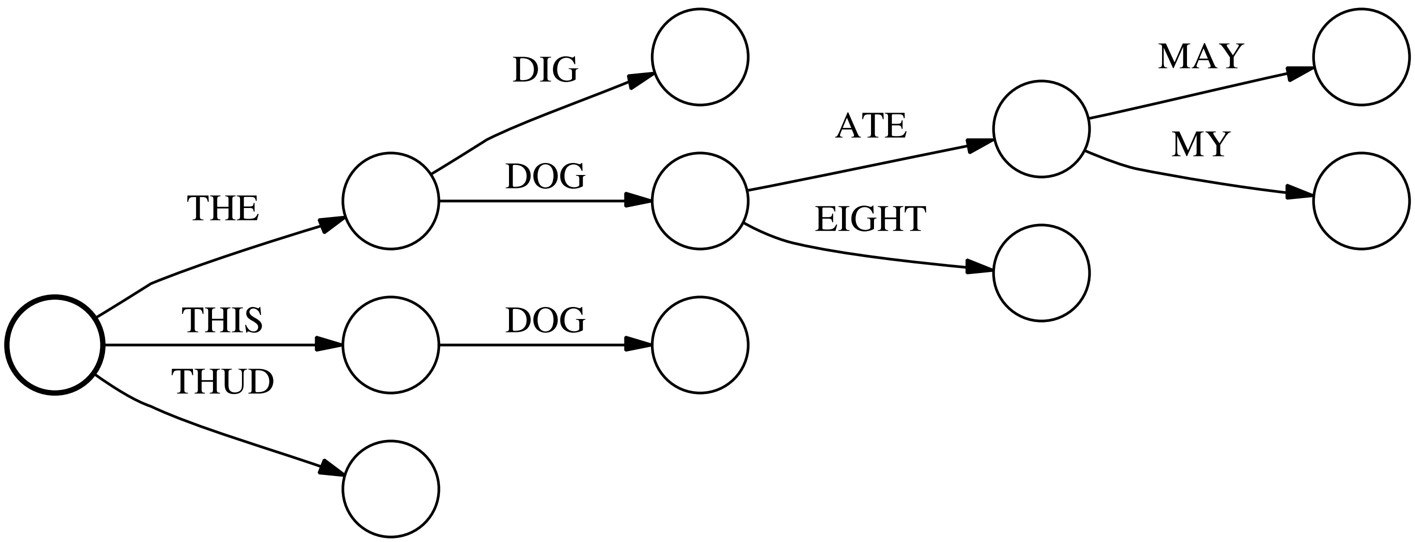
- Unit V: other decoding paradigms
  - dynamic graph expansion
  - stack search
  - two-pass decoding

Stack Search

- Viterbi search — synchronous search
  - extend all paths and calculate all scores synchronously
  - expand states with mediocre scores in case they improve later
- stack search — asynchronous search
  - pursue best-looking path first!
  - if lucky, expand very few states at each frame
- pioneered at IBM in mid-1980's; first real-time dictation system
- may be competitive at low-resource operating points
  - going out of fashion

Stack Search

- extend hypotheses word-by-word
- use *fast match* to decide which word to extend best path with
  - decode single word with simpler acoustic model



Stack Search

- advantages
  - if best path pans out, very little computation
- disadvantages
  - difficult to decide which path to extend
    - hypotheses are of different lengths in frames
    - in synchronous search, pruning is straightforward
  - may need to recompute the same values multiple times
    - in DP terminology, not evaluating cells in topological order
- point: in practice, have enough compute power for Viterbi
  - fewer search errors

Where Are We?

- Unit V: other decoding paradigms
  - dynamic graph expansion
  - stack search
  - two-pass decoding

What About My Fuzzy Logic 15-Phone Acoustic Model and 7-Gram Neural Net Language Model with SVM Boosting?

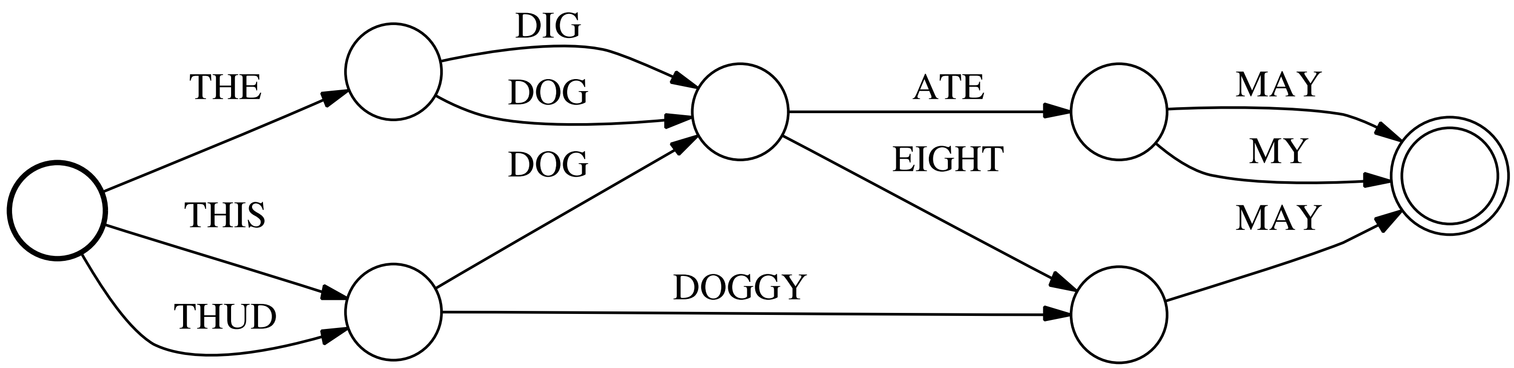
- some of the ASR models we develop in research are . . .
  - too expensive to implement in normal (first-pass) decoding
- first-pass decoding
  - find best word sequence from among "all" word sequences
- *rescoring*
  - find best word sequence from constrained search space
    - namely, best-scoring word sequences from first pass
  - large enough set to hopefully contain "correct" hypothesis
  - small enough set that not too expensive to rescore

Two-Pass Decoding

- for interactive applications, one-pass near-real-time decoding is ideal
  - start processing when audio signal starts, be done soon after audio signal ends
- two-pass decoding generally yields better accuracy
  - 1st pass: decode, but return many likely hypotheses rather than single most likely
  - 2nd pass: choose best of returned hypotheses using more complex models
    - *e.g.*, N-best list rescoring in Lab 3
  - can still be used for interactive apps if 2nd pass really fast

Lattice Rescoring

- first pass: return likely hypotheses as a graph or *lattice*
  - in Viterbi, store k-best tracebacks at each word-end cell



- can use models that are impractical with first-pass decoding
  - *e.g.*, 5-gram LM's, sesquiphone phonetic decision trees, etc.
- some techniques need lattices
  - *e.g.*, confidence estimation, consensus decoding, lattice MLLR, etc.

N-Best List Rescoring

- for exotic models, evaluating on lattices may be too slow
  - lattice encodes exponential number of paths (in length of utterance)
  - for some models, computation linear in number of hypotheses
- easy to generate N-best lists from lattices
  - A* algorithm
- harder to judge quality of model used for rescoring in this paradigm
  - first-pass model biases results

Two-Pass Decoding

Recap

- great for doing research
  - generate lattices once
  - lattice/N-best rescoring is cheap
  - reasonable indicator of value of model
- in real-world apps, value less clear
  - performance gain from 2nd pass usually not perceptible by users
  - increases latency

The Road Ahead

- weeks 1–4: small vocabulary ASR
- weeks 5–8: large vocabulary ASR
- weeks 9–12: advanced topics
  - adaptation; robustness
  - discriminative training; ROVER; consensus
  - advanced language modeling
  - audiovisual speech recognition
- week 13: final presentations

Course Feedback

1. Was this lecture mostly clear or unclear? What was the muddiest topic?
2. Comments on lab 2?
3. Other feedback (pace, content, atmosphere)?