CS 224S/LING 281 Speech Recognition, Synthesis, and Dialogue

Dan Jurafsky

Lecture 15:

ASR: Search (Lattices, N-best lists, A*, etc) and Scoring (sclite)

Evaluation

- How to evaluate the word string output by a speech recognizer?

Word Error Rate

- Word Error Rate =

100 (Insertions+Substitutions + Deletions)

Total Word in Correct Transcript

Aligment example:

REF: portable **** PHONE UPSTAIRS last night so

HYP: portable FORM OF STORES last night so

Eval I S S

WER = 100 (1+2+0)/6 = 50%

NIST sctk-1.3 scoring software: Computing WER with sclite

- http://www.nist.gov/speech/tools/
- Sclite aligns a hypothesized text (HYP) (from the recognizer) with a correct or reference text (REF) (human transcribed)

```
id: (2347-b-013)
Scores: (#C #S #D #I) 9 3 1 2
REF:  was an engineer SO I   i was always with **** **** MEN UM
   and they
HYP:  was an engineer ** AND i was always with THEM THEY ALL THAT
   and they
Eval:                 D  S                     I    I    S   S
```

Sclite output for error analysis

```
CONFUSION PAIRS                  Total                 (972)
                                 With >=  1 occurances (972)

   1:    6  ->  (%hesitation) ==> on
   2:    6  ->  the ==> that
   3:    5  ->  but ==> that
   4:    4  ->  a ==> the
   5:    4  ->  four ==> for
   6:    4  ->  in ==> and
   7:    4  ->  there ==> that
   8:    3  ->  (%hesitation) ==> and
   9:    3  ->  (%hesitation) ==> the
  10:    3  ->  (a-) ==> i
  11:    3  ->  and ==> i
  12:    3  ->  and ==> in
  13:    3  ->  are ==> there
  14:    3  ->  as ==> is
  15:    3  ->  have ==> that
  16:    3  ->  is ==> this
```

Sclite output for error analysis

```
 17:    3  ->  it ==> that
18:    3  ->  mouse ==> most
19:    3  ->  was ==> is
20:    3  ->  was ==> this
21:    3  ->  you ==> we
22:    2  ->  (%hesitation) ==> it
23:    2  ->  (%hesitation) ==> that
24:    2  ->  (%hesitation) ==> to
25:    2  ->  (%hesitation) ==> yeah
26:    2  ->  a ==> all
27:    2  ->  a ==> know
28:    2  ->  a ==> you
29:    2  ->  along ==> well
30:    2  ->  and ==> it
31:    2  ->  and ==> we
32:    2  ->  and ==> you
33:    2  ->  are ==> i
34:    2  ->  are ==> were
```

Better metrics than WER?

- WER has been useful
- But should we be more concerned with meaning ("semantic error rate")?
  - Good idea, but hard to agree on
  - Has been applied in dialogue systems, where desired semantic output is more clear

Part II: Search (= "Decoding")

- Speeding things up: Viterbi beam decoding
- Problems with Viterbi decoding
- Multipass decoding
  - N-best lists
  - Lattices
  - Word graphs
  - Meshes/confusion networks
- A* search

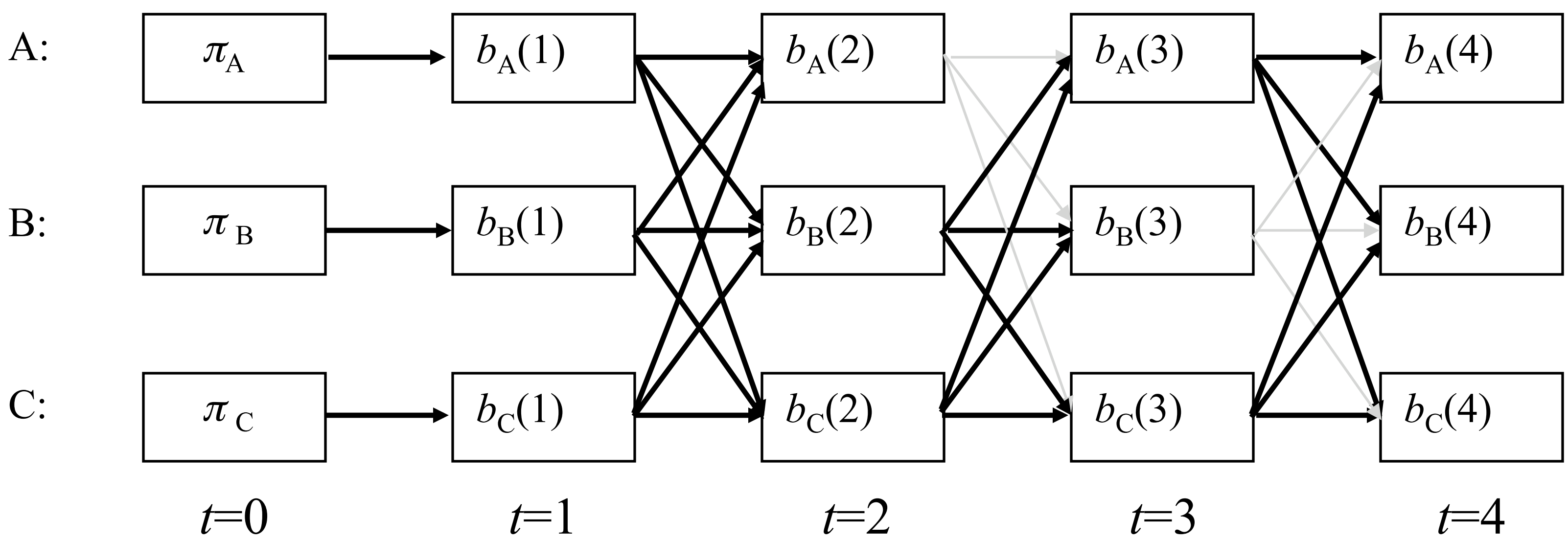
Speeding things up

- Viterbi is $O(N^2T)$, where N is total number of HMM states, and T is length
- This is too large for real-time search
- A ton of work in ASR search is just to make search faster:
  - Beam search (pruning)
  - Fast match
  - Tree-based lexicons

Beam search

- Instead of retaining all candidates (cells) at every time frame
- Use a threshold T to keep subset:
  - At each time t
  - Identify state with lowest cost D_{min}
  - Each state with cost > D_{min}+ T is discarded ("pruned") before moving on to time t+1
  - Unpruned states are called the **active** states

Viterbi Beam Search



Slide from John-Paul Hosom

Viterbi Beam search

- Is the most common and powerful search algorithm for LVCSR
- Note:
  - What makes this possible is time-synchronous
  - We are comparing paths of equal length
  - For two different word sequences W_1 and W_2:
    - We are comparing $P(W_1|O_0^t)$ and $P(W_2|O_0^t)$
    - Based on same partial observation sequence O_0^t
    - So denominator is same, can be ignored
  - Time-asynchronous search (A*) is harder

Viterbi Beam Search

- Empirically, beam size of 5-10% of search space
- Thus 90-95% of HMM states don't have to be considered at each time t
- Vast savings in time.

On-line processing

- Problem with Viterbi search
  - Doesn't return best sequence til final frame

- This delay is unreasonable for many applications.

- **On-line processing**
  - *usually smaller* delay in determining answer
  - at cost of *always increased* processing time.

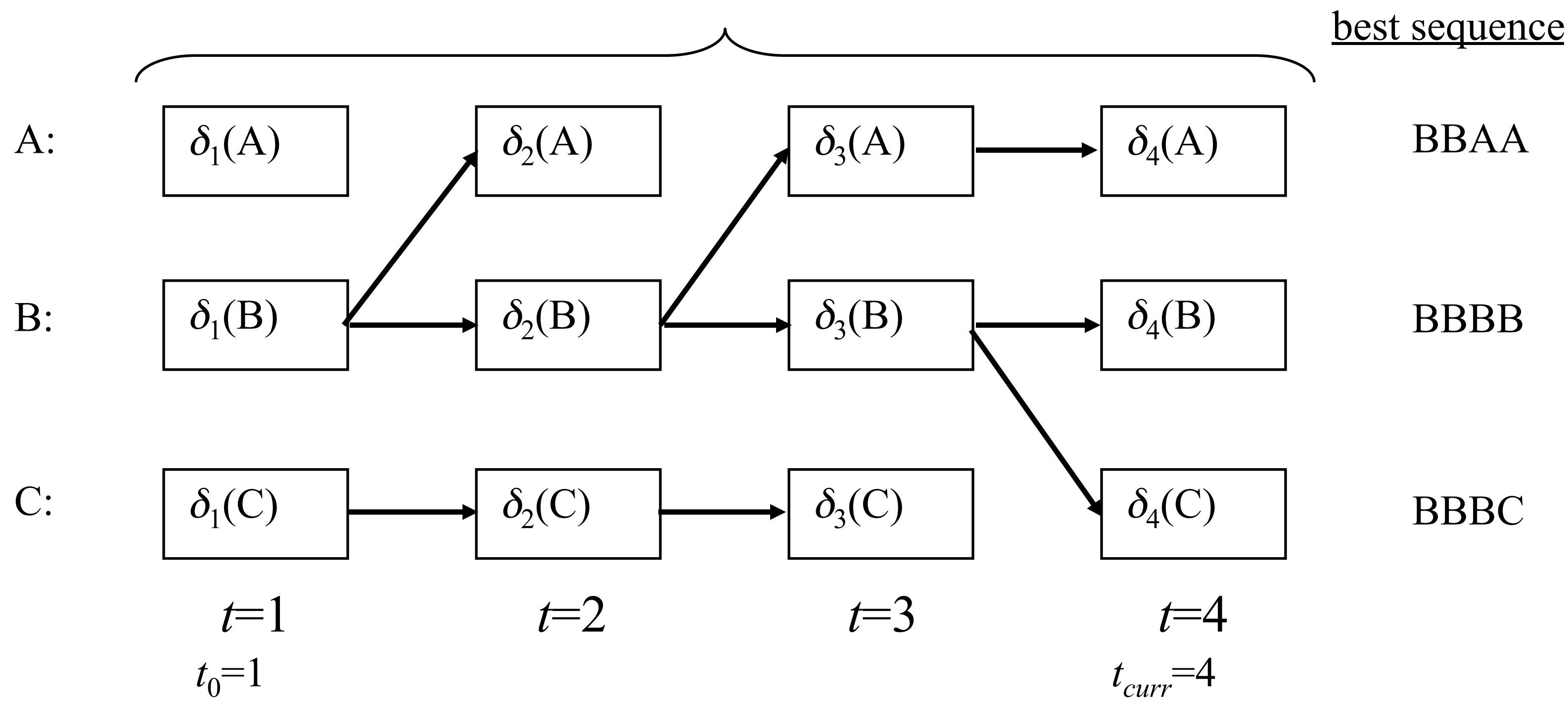
On-line processing

- At every time interval I (e.g. 1000 msec or 100 frames):
  - At current time t_{curr}, for each active state q_{tcurr}, find best path $P(q_{tcurr})$ that goes from from t_0 to t_{curr} (using backtrace (ψ))
  - Compare set of best paths P and find last time t_{match} at which all paths P have the same state value at that time
  - If t_{match} exists {
    Output result from t_0 to t_{match}
    Reset/Remove ψ values until t_{match}
    Set t_0 to $t_{match}+1$
    }
- Efficiency depends on interval I, beam threshold, and how well the observations match the HMM.

Slide from John-Paul Hosom

On-line processing

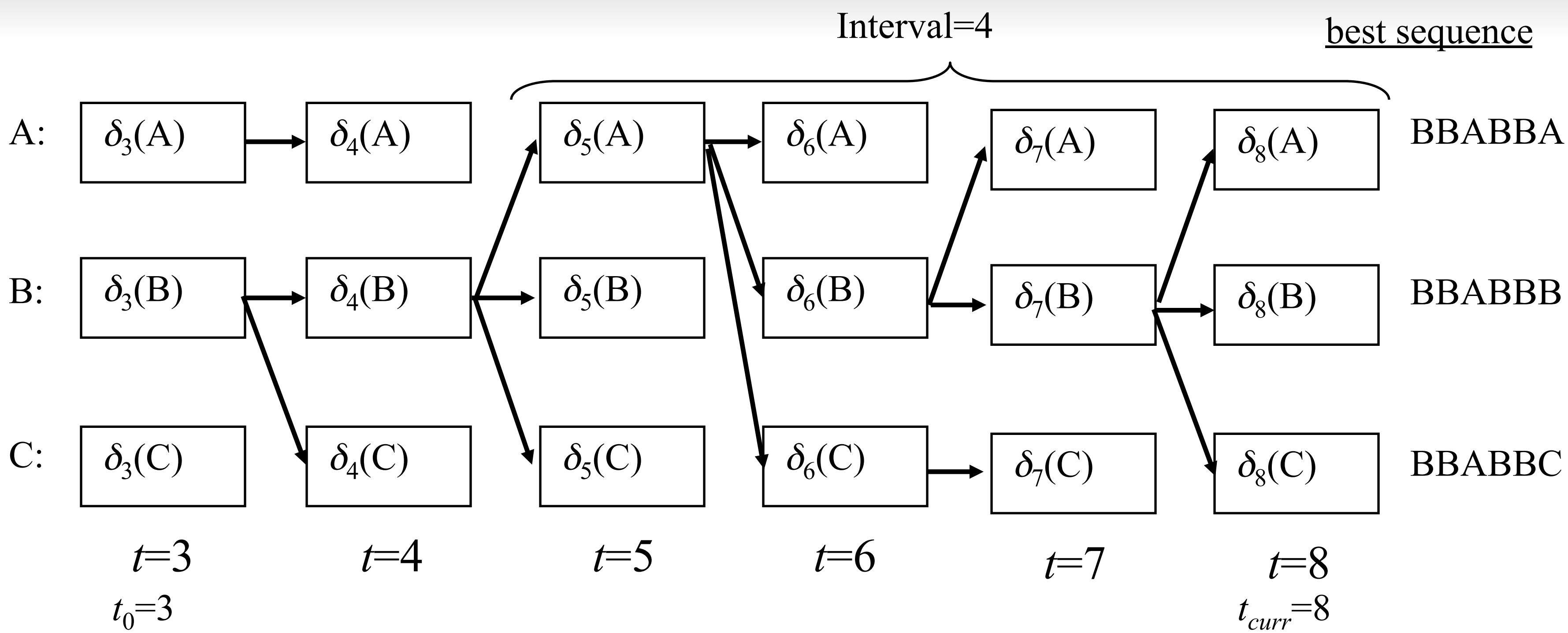
- Example (Interval = 4 frames):



- In this case, at time 4, all best paths for all states A, B, and C have state B in common at time 2. So, t_{match} = 2.
- Now output states BB for times 1 and 2, because no matter what happens in the future, this will not change. Set t_0 to 3

Slide from John-Paul Hosom

On-line processing

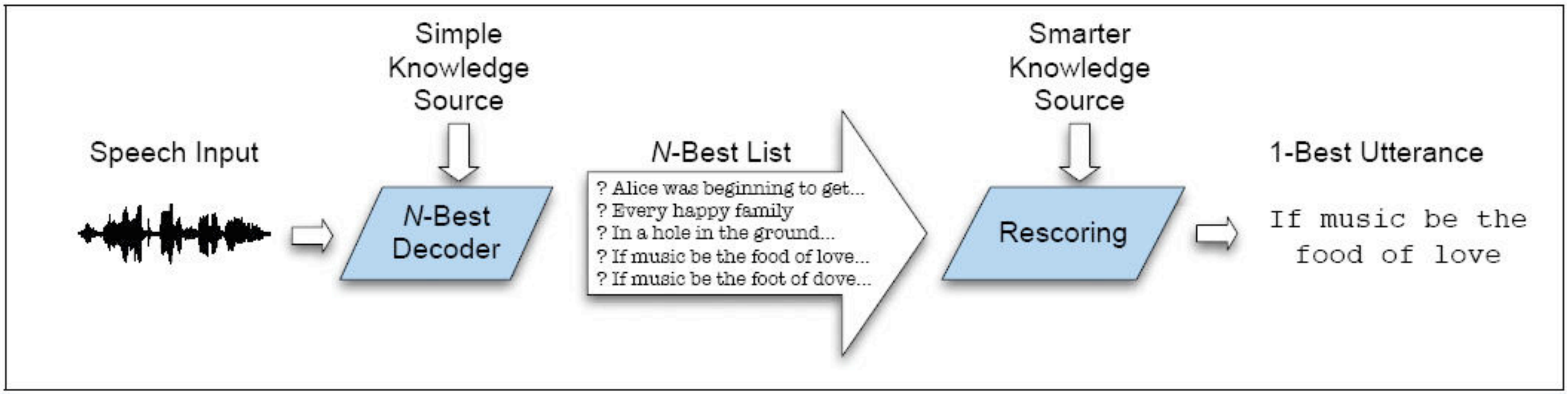


- Now t_{match} = 7, so output from t=3 to t=7: BBABB, then set t_0 to 8.
- If T=8, then output state with best δ_8, for example C. Final result (obtained piece-by-piece) is then BBBBABBC

Problems with Viterbi

- It’s hard to integrate sophisticated knowledge sources
  - Trigram grammars
  - Parser-based LM
    - long-distance dependencies that violate dynamic programming assumptions
  - Knowledge that isn’t left-to-right
    - Following words can help predict preceding words
- Solutions
  1. Return **multiple hypotheses** and use smart knowledge to rescore them
  2. Use a different search algorithm, A* Decoding (=Stack decoding)

Multipass Search



Ways to represent multiple hypotheses

- N-best list
  - Instead of single best sentence (word string), return ordered list of N sentence hypotheses
- Word lattice
  - Compact representation of word hypotheses and their times and scores
- Word graph
  - FSA representation of lattice in which times are represented by topology

Another Problem with Viterbi

- The forward probability of observation given word string

$$P(O|W) = \sum_{S \in S_1^T} P(O, S|W)$$

- The Viterbi algorithm makes the "Viterbi Approximation"

$$P(O|W) \approx \max_{S \in S_1^T} P(O, S|W)$$

- Approximates probability of observation given word, with prob of observation given only best state sequence.

Solving the best-path-not-best-words problem

- Viterbi returns best path (state sequence) not best word sequence
  - Best path can be very different than best word string if words have many possible pronunciations
- Two solutions
  1) Modify Viterbi to sum over different paths that share the same word string.
     - Do this as part of N-best computation
       - Compute **N-best word strings**, **not N-best phone paths**
  2) Use a different decoding algorithm (A*) that computes true Forward probability.

Sample N-best list

Rank	Path	AM logprob	LM logprob
1.	it's an area that's naturally sort of mysterious	-7193.53	-20.25
2.	that's an area that's naturally sort of mysterious	-7192.28	-21.11
3.	it's an area that's not really sort of mysterious	-7221.68	-18.91
4.	that scenario that's naturally sort of mysterious	-7189.19	-22.08
5.	there's an area that's naturally sort of mysterious	-7198.35	-21.34
6.	that's an area that's not really sort of mysterious	-7220.44	-19.77
7.	the scenario that's naturally sort of mysterious	-7205.42	-21.50
8.	so it's an area that's naturally sort of mysterious	-7195.92	-21.71
9.	that scenario that's not really sort of mysterious	-7217.34	-20.70
10.	there's an area that's not really sort of mysterious	-7226.51	-20.01

N-best lists

- Again, we don’t want the N-best paths
- That would be trivial
  - Store N values in each state cell in Viterbi trellis instead of 1 value
- But:
  - Most of the N-best paths will have the same word string
    - Useless!!!
  - It turns out that a factor of N is too much to pay

Computing N-best lists

- In the worst case, an admissible algorithm for finding the N most likely hypotheses is exponential in the length of the utterance.
  - S. Young. 1984. "Generating Multiple Solutions from Connected Word DP Recognition Algorithms". Proc. of the Institute of Acoustics, 6:4, 351-354.
- For example, if AM and LM score were nearly identical for all word sequences, we must consider all permutations of word sequences for whole sentence (all with the same scores).
- But of course if this is true, can't do ASR at all!

Computing N-best lists

- Instead, various non-admissible algorithms:
  - (Viterbi) Exact N-best
  - (Viterbi) Word Dependent N-best
- And one admissible
  - A* N-best

Exact N-best for time-synchronous Viterbi

- Due to Schwartz and Chow; also called "sentence-dependent N-best"
- Idea: each state stores multiple paths
- Idea: maintain separate records for paths with distinct **word** histories
  - History: whole word sequence up to current time t and word w
  - When 2 or more paths come to the same state at the same time, merge paths w/same history and sum their probabilities.
    - i.e. compute the forward probability within words
  - Otherwise, retain only N-best paths for each state

Exact N-best for time-synchronous Viterbi

- Efficiency:
  - Typical HMM state has 2 or 3 predecessor states within word HMM
  - So for each time frame and state, need to compare/merge 2 or 3 sets of N paths into N new paths.
  - At end of search, N paths in final state of trellis give N-best word sequences
  - Complexity is O(N)
    - Still too slow for practical systems
      - N is 100 to 1000
    - More efficient versions: word-dependent N-best

Word-dependent (‘bigram’) N-best

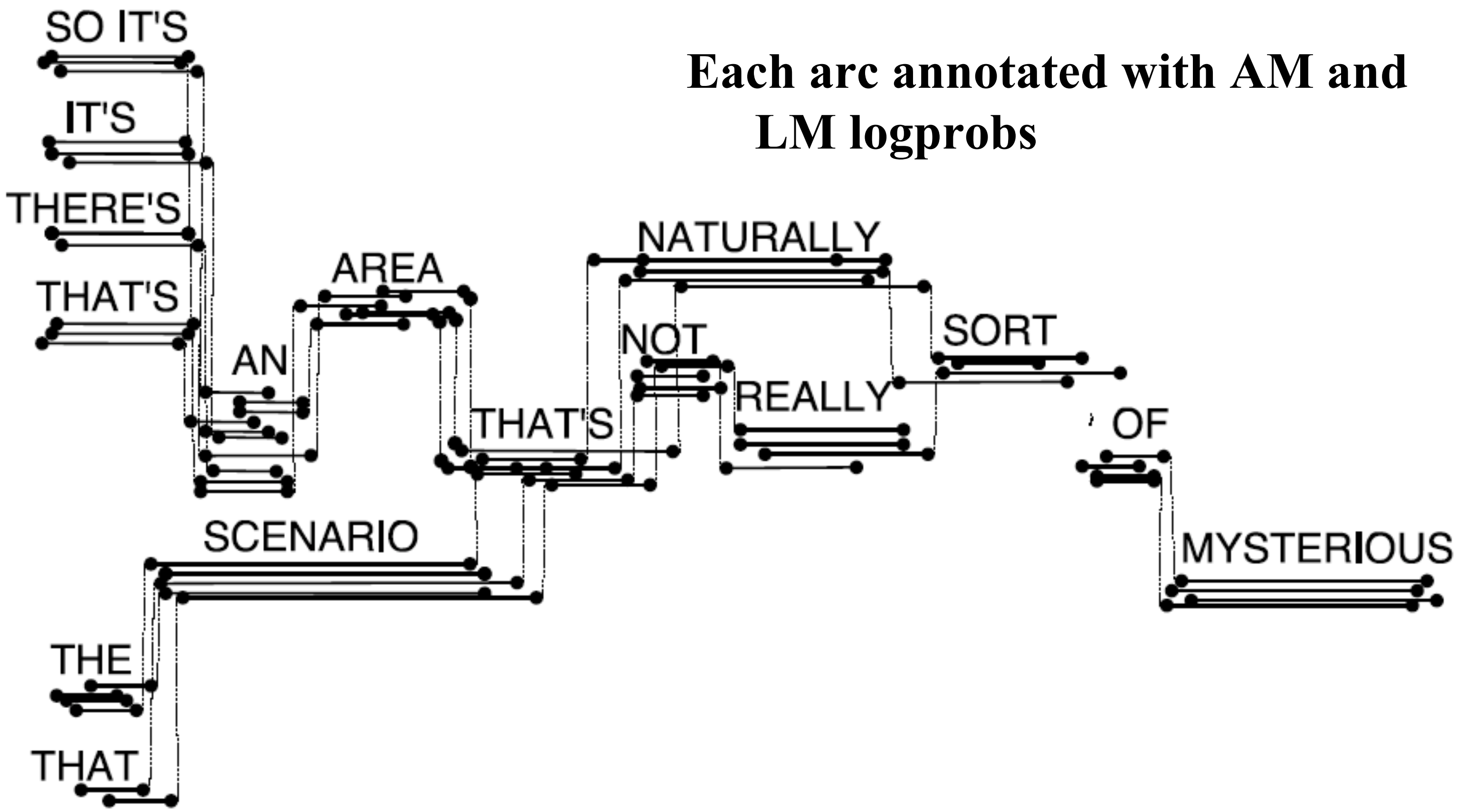
- Intuition:
  - Instead of each state merging all paths from start of sentence
  - We merge all paths that share the same previous word
- Details:
  - This will require us to do a more complex traceback at the end of sentence to generate the N-best list

Word-dependent ('bigram') N-best

- At each state preserve total probability for each of k << N previous words
  - K is 3 to 6; N is 100 to 1000
- At end of each word, record score for each previous word hypothesis and name of previous word
  - So each word ending we store "alternatives"
- But, like normal Viterbi, pass on just the best hypothesis
- At end of sentence, do a traceback
  - Follow backpointers to get 1-best
  - But as we follow pointers, put on a queue the alternate words ending at same point
  - On next iteration, pop next best

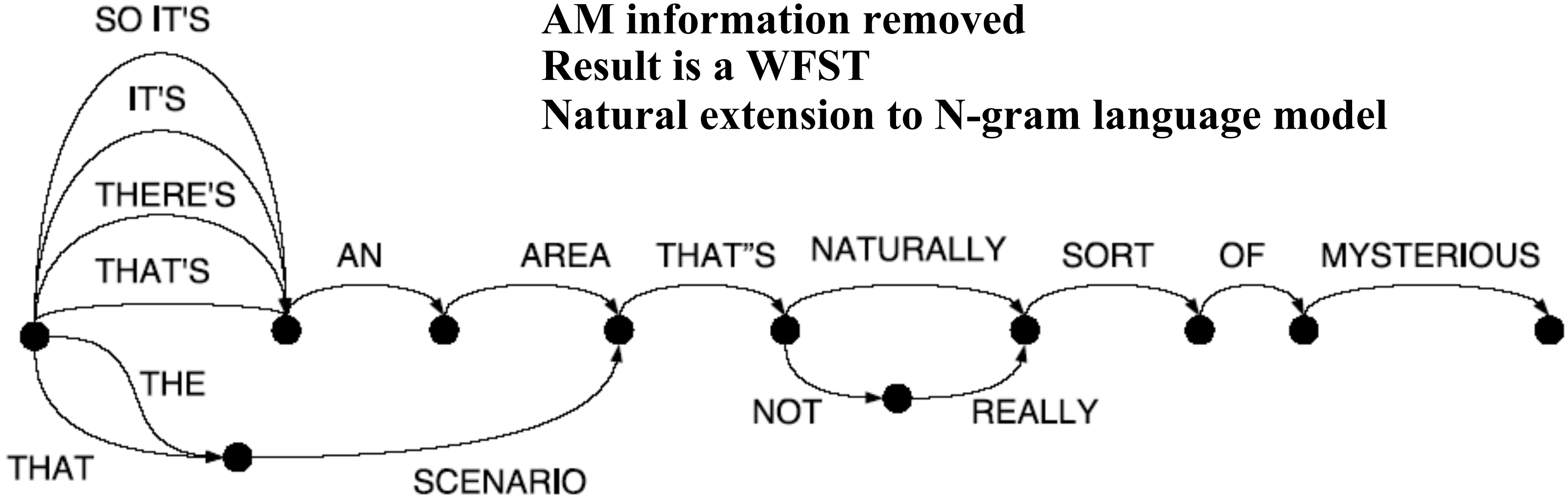
Word Lattice

Each arc annotated with AM and LM logprobs



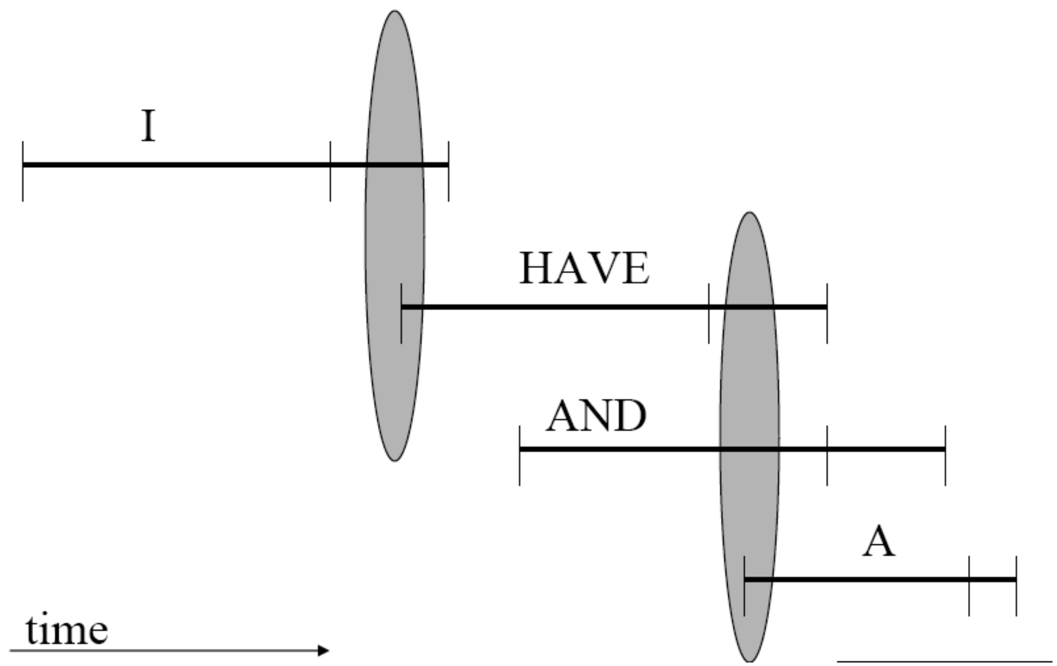
Word Graph

- Timing information removed
- Overlapping copies of words merged
- AM information removed
- Result is a WFST
- Natural extension to N-gram language model



Converting word lattice to word graph

- Word lattice can have range of possible end frames for word
- Create an edge from (w_i,t_i) to (w_j,t_j) if t_{j-1} is one of the end-times of w_i



Bryan Pellom's algorithm and figure, from his slides

Lattices

- Some researchers are careful to distinguish between word graphs and word lattices
- But we'll follow convention in using "lattice" to mean both word graphs and word lattices.
- Two facts about lattices:
  - Density: the number of word hypotheses or word arcs per uttered word
  - Lattice error rate (also called "lower bound error rate"): the lowest word error rate for any word sequence in lattice
    - Lattice error rate is the "oracle" error rate, the best possible error rate you could get from rescoring the lattice.
    - We can use this as an upper bound

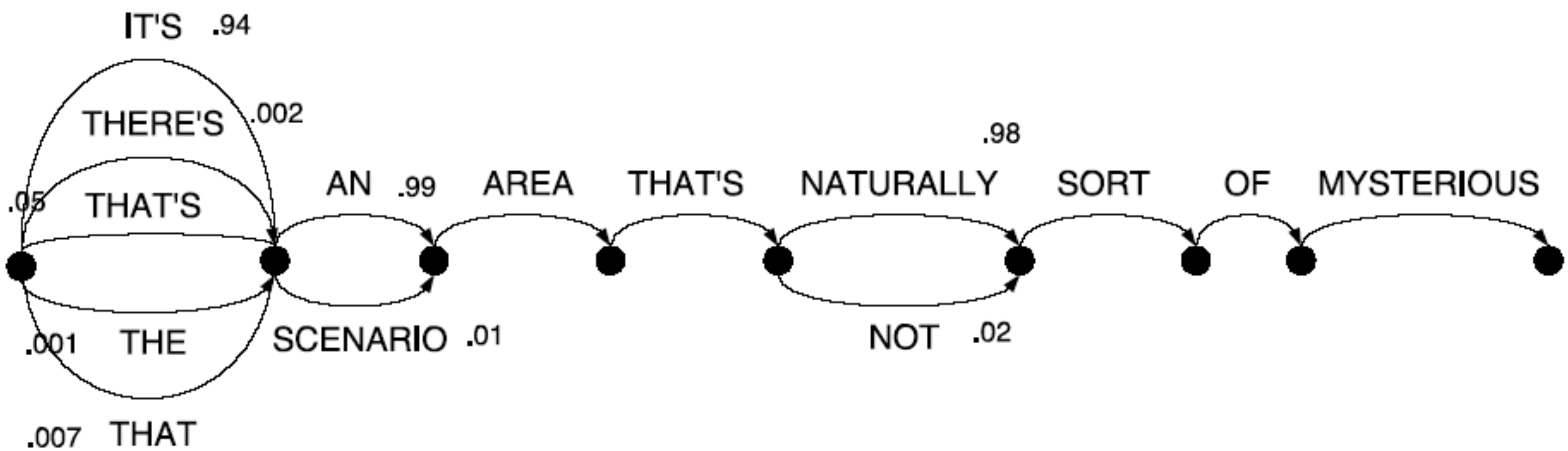
Posterior lattices

- We don't actually compute posteriors:

$$\hat{W} = \underset{W \in \mathcal{L}}{\operatorname{argmax}} \frac{P(O|W)P(W)}{P(O)} = \underset{W \in \mathcal{L}}{\operatorname{argmax}} P(O|W)P(W)$$

- Why do we want posteriors?
  - Without a posterior, we can choose best hypothesis, but we can't know how good it is!
  - In order to compute posterior, need to
    - Normalize over all different word hypothesis at a time
  - Align all the hypotheses, sum over all paths passing through word

Mesh = Sausage = pinched lattice



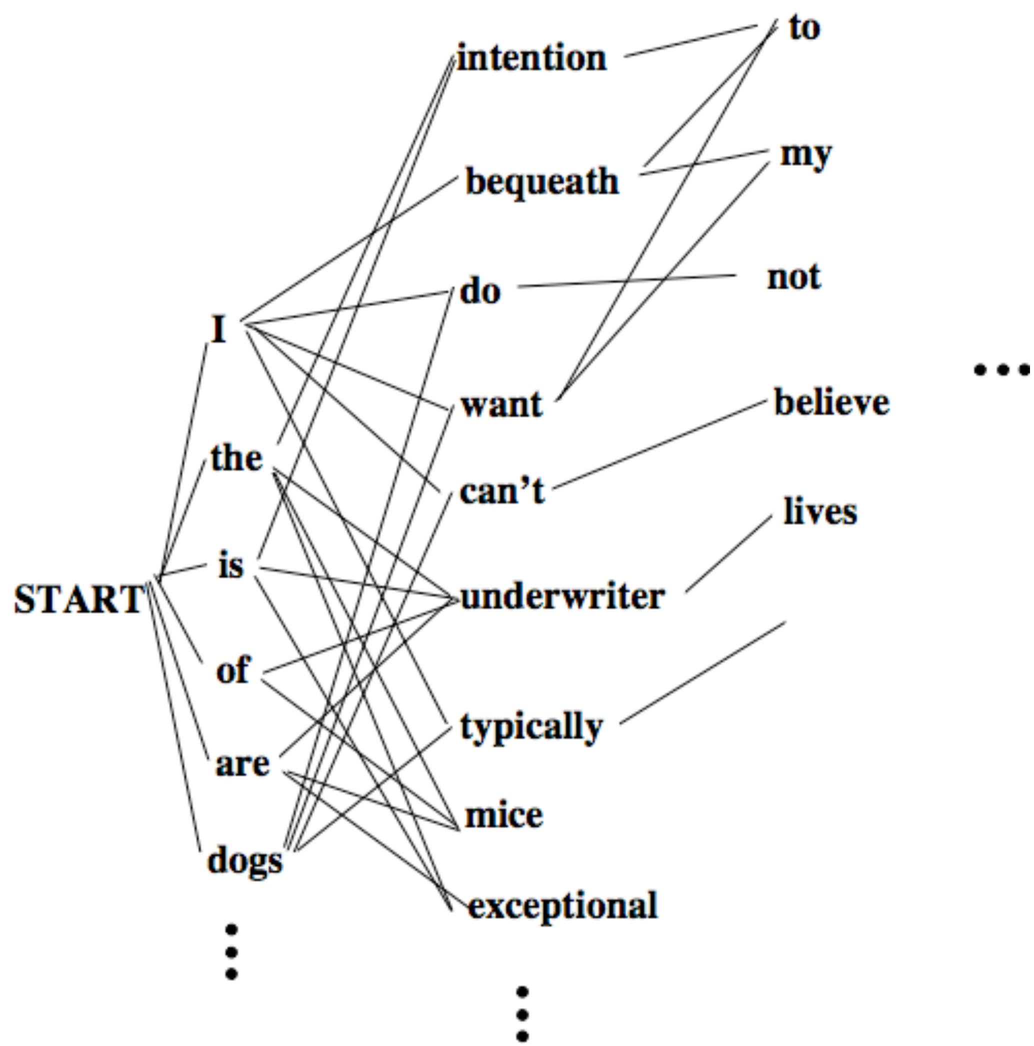
One-pass vs. multipass

- Potential problems with multipass
  - Can't use for real-time (need end of sentence)
    - (But can keep successive passes really fast)
  - Each pass can introduce inadmissible pruning
    - (But one-pass does the same w/beam pruning and fastmatch)
- Why multipass
  - Very expensive KSs. (NL parsing,higher-order n-gram, etc)
  - Spoken language understanding: N-best perfect interface
  - Research: N-best list very powerful offline tools for algorithm development
  - N-best lists needed for discriminant training (MMIE, MCE) to get rival hypotheses

A* Decoding = Stack decoding

- Intuition:
  - Viterbi wastes a lot of time on breadth-first search
    - Computing lots of paths will never need
  - If we had good heuristics for guiding decoding
  - We could do depth-first (best-first) search and not waste all our time on computing all those paths at every time step as Viterbi does.
- A* decoding, also called stack decoding, is an attempt to do that.
- A* also does not make the Viterbi assumption
- Uses the actual forward probability, rather than the Viterbi approximation

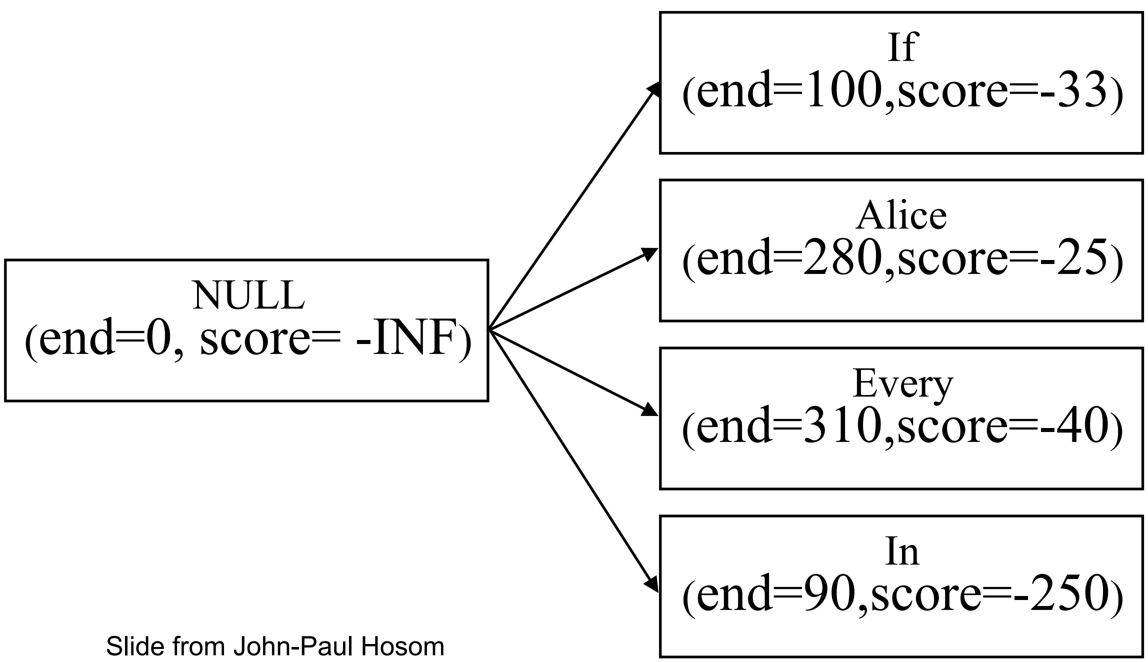
The search space for A* is the set of possible sentences



A* Search
"If music be the food of love"

(1) Start with NULL as root of sentence tree, set *n* to 0

(2) Determine *every possible* word starting at time *t*=1, adding to the stack the words (now a partial sentence), their end times, and scores (e.g. log probabilities, so closer to zero is better), with a link to the NULL partial sentence.



Slide from John-Paul Hosom

A* search

(3) Pop partial sentence with highest score, *P*, off the stack (keeping the word, time, score, and link information for future use)

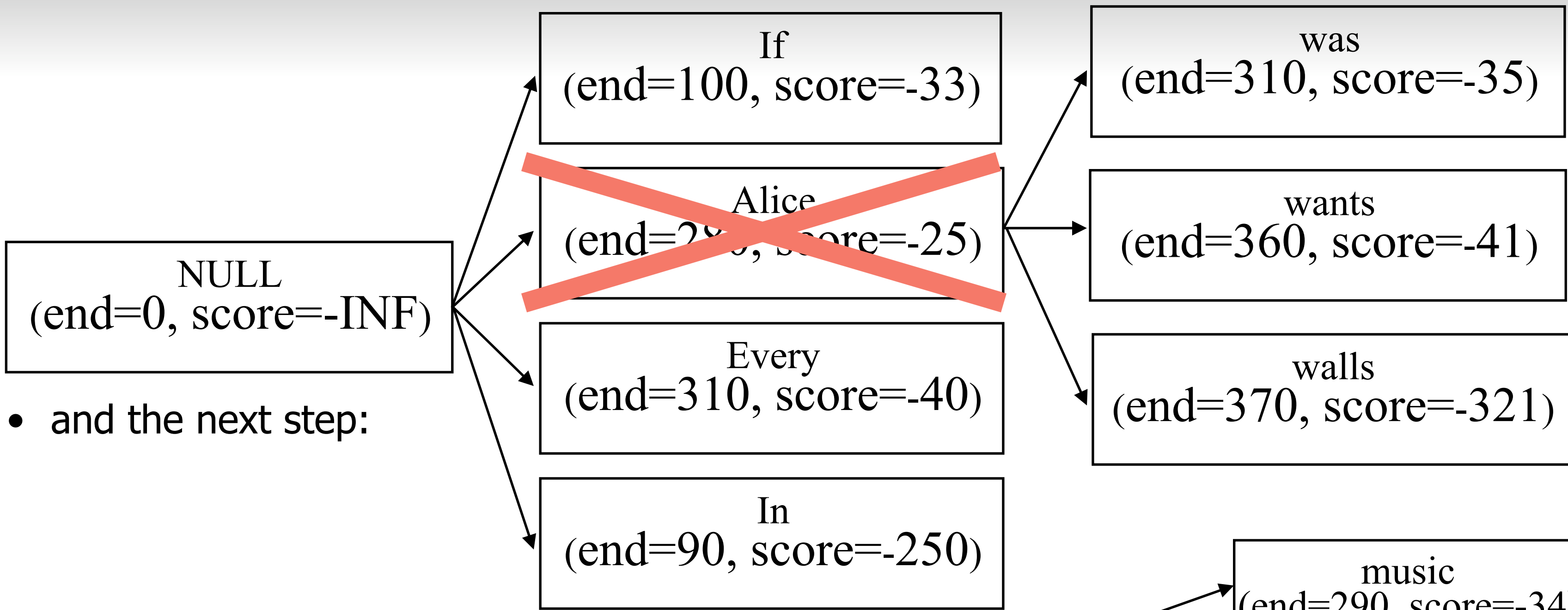
(4) If *P* is a complete sentence (end time of last word in partial sentence = *T*), then (a) output sentence by following links to all previous words in sentence, (b) increment *n.*
(c) If *n* == *N*, then stop; otherwise, go to (3)

(5) Determine *every possible* word starting at time *t*=(end time of last word in *P*), adding to the stack the new words, their end times, and scores, with a link to the last word in *P*.

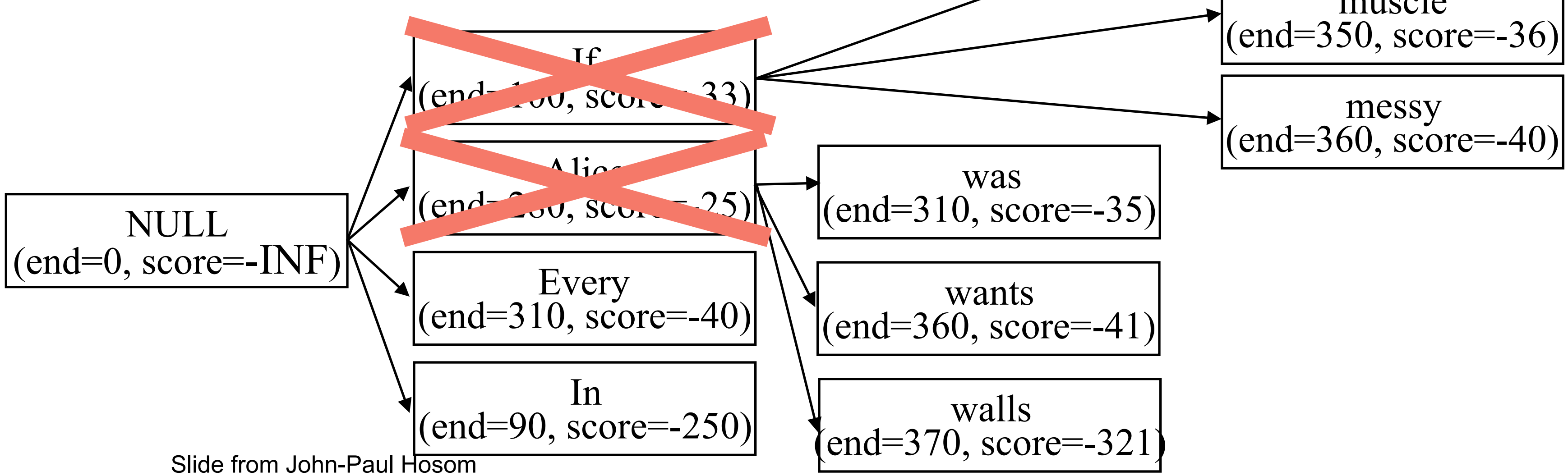
(6) Go to step (3)

Slide from John-Paul Hosom

A* Search



- and the next step:



Slide from John-Paul Hosom

Reminder: A* search

- A search algorithm is "admissible" if it can guarantee to find an optimal solution if one exists.
- Heuristic search functions rank nodes in search space by f(N), the goodness of each node N in a search tree, computed as:
- f(N) = g(N) + h(N)
  where
  - g(N) = The distance of the partial path already traveled from root S to node N
  - h(N) = Heuristic estimate of the remaining distance from node N to goal node G.

Reminder: A* search

- If the heuristic function h(N) of estimating the remaining distance from N to goal node G is an underestimate of the true distance, best-first search is admissible, and is called A* search.

A* search

- A* search is "time-asynchronous" search
- The score is an evaluation of how good a *partial* sentence is
- <u>Possible</u> formula for score:
  - $$score = p(\mathbf{o}_1\ \mathbf{o}_2\ \ldots\ \mathbf{o}_t \mid w_1\ w_2\ \ldots\ w_n) \cdot p(w_1\ w_2\ \ldots\ w_n)$$
  - t is the end time of the partial sentence
  - n is the number of words currently in the partial sentence.
- How to compute this?
- Forward Algorithm! But we can't do this!!! Why not???
- This results in lower scores for longer utterances.
- Also, the score should reflect how good we think the final sentence *will be* when we get to the end.

Slide from John-Paul Hosom

A* search for speech

- The forward algorithm can tell us the cost of the current path so far g(.)
- We need an estimate of the cost from the current node to the end h(.)

Making A* work: h(.)

- If h(.) is zero, breadth first search
- Stupid estimates of h(.):
  - Amount of time left in utterance
- Slightly smarter:
  - Estimate expected cost-per-frame for remaining path
  - Multiply that by remaining time
  - This can be computed from the training set (how much was the average acoustic cost for a frame in the training set)
- Even better: two-pass decoding
  - First pass, do Viterbi forward
  - Now do A* **backwards**, using Viterbi best path as estimate h* for any hypothesis!

A* N-best

- A* (stack-decoding) is best-first search
- So we can just keep generating results until it finds N complete paths
- This is the N-best list

A*: When to extend new words

- Stack decoding is asynchronous
- Need to detect when a phone/word ends, so search can extend to next phone/word
- If we had a cost measure: how well input matches HMM state sequence so far
- We could look for this cost measure slowly going down, and then sharply going up as we start to see the start of the next word.
- Can't use forward algorithm because can't compare hypotheses of different lengths
- Can do various length normalizations to get a normalized cost

Fast match

- Efficiency: don’t want to expand to every single next word to see if it’s good.
- Need a quick heuristic for deciding which sets of words are good expansions
- “Fast match” is the name for this class of heuristics.
- Can do some simple approximation to words whose initial phones seem to match the upcoming input

Summary

- Search
  - Defining the goal for ASR decoding
  - Speeding things up: Viterbi beam decoding
  - Problems with Viterbi decoding
  - Multipass decoding
    - N-best lists
    - Lattices
    - Word graphs
    - Meshes/confusion networks
  - A* search