Decoding Part II

Bhiksha Raj and Rita Singh

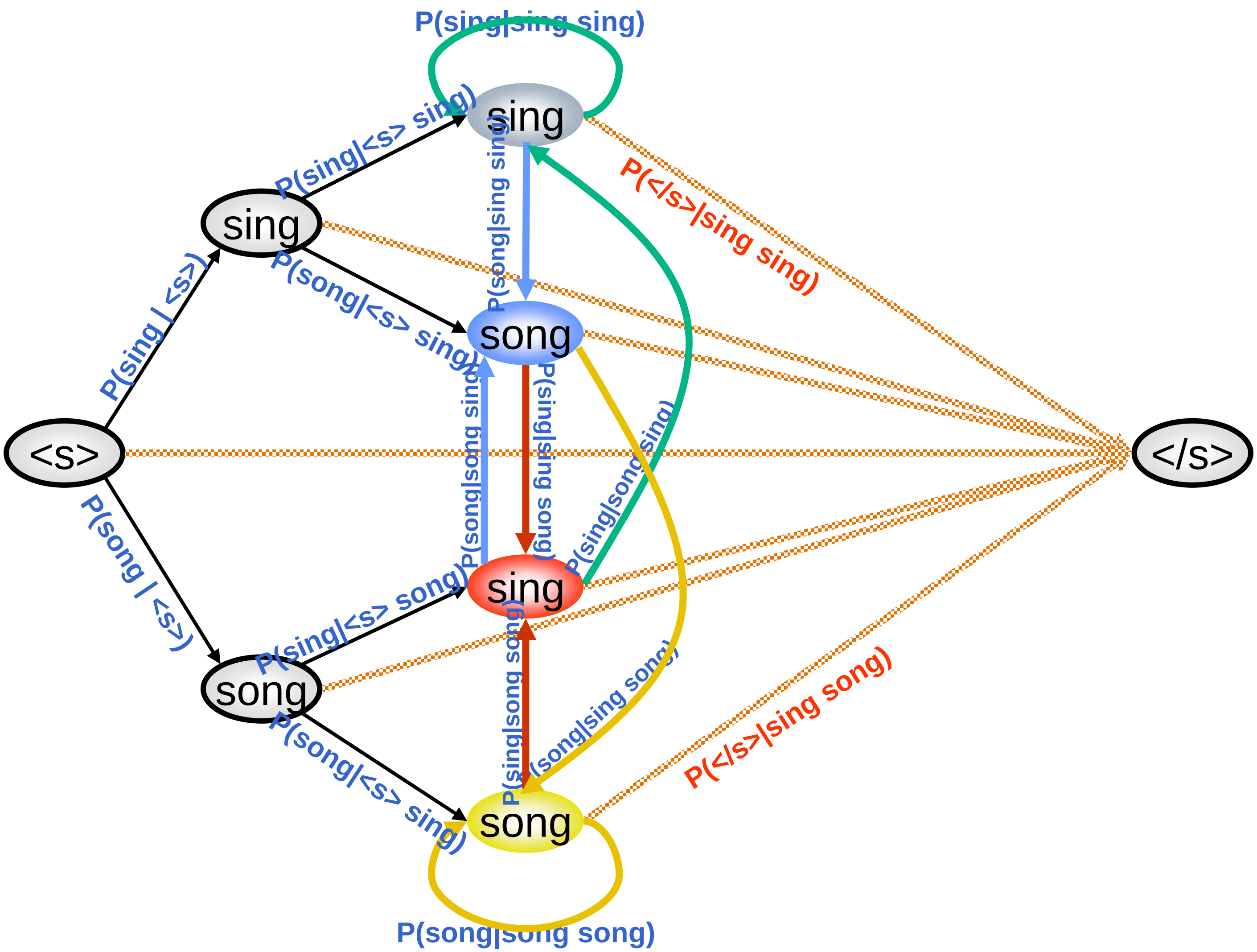
Recap and Lookahead

- Covered so far:
  - String Matching based Recognition
  - Introduction to HMMs
  - Recognizing Isolated Words
  - Learning word models from continuous recordings
  - Building word models from phoneme models
  - Context-independent and context-dependent models
  - Building decision trees
  - Tied-state models
  - Decoding: Concepts

  - Exercise: Training phoneme models
  - Exercise: Training context-dependent models
  - Exercise: Building decision trees
  - Exercise: Training tied-state models

- Decoding: Practical issues and other topics

A Full N-gram Lanugage Model Graph



- An N-gram language model can be represented as a graph for speech recognition

Generic N-gram representations

- A full N-gram graph can get *very very very* large
- A trigram decoding structure for a vocabulary of D words needs D word instances at the first level and D^2 word instances at the second level
  - Total of D(D+1) word models must be instantiated
- An N-gram decoding structure will need
  - $D + D^2 + D^3 \ldots D^{N-1}$ word instances
- A simple trigram LM for a vocabulary of 100,000 words would have…
  - 100,000 words is a reasonable vocabulary for a large-vocabulary speech recognition system
- … an indecent number of nodes in the graph and an obscene number of edges

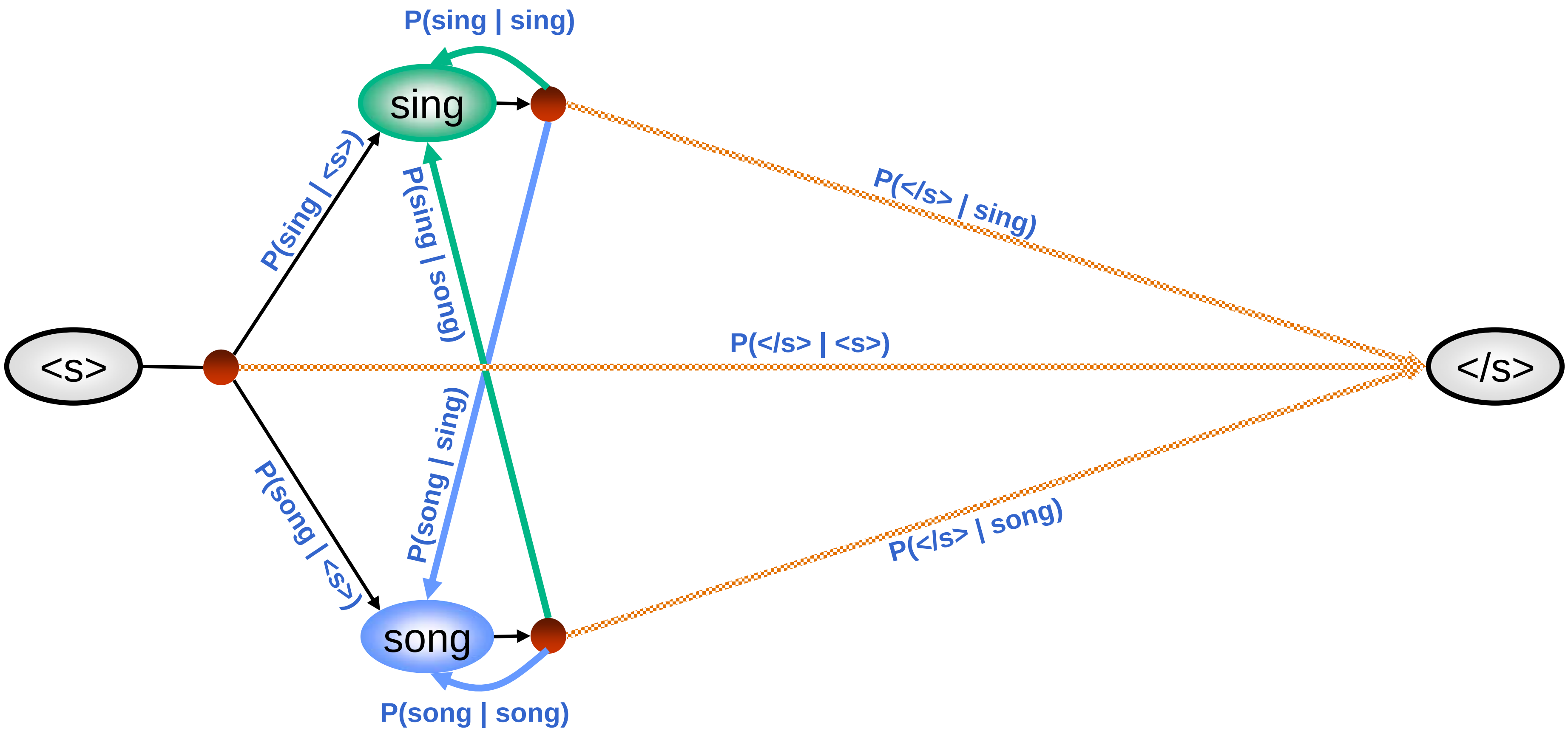
Lack of Data to the Rescue!

- We never have enough data to learn all D^3 trigram probabilities
- We learn a very small fraction of these probabilities
  - Broadcast news: Vocabulary size 64000, training text 200 million words
    - 10 million trigrams, 3 million bigrams!
- All other probabilities are obtained through backoff

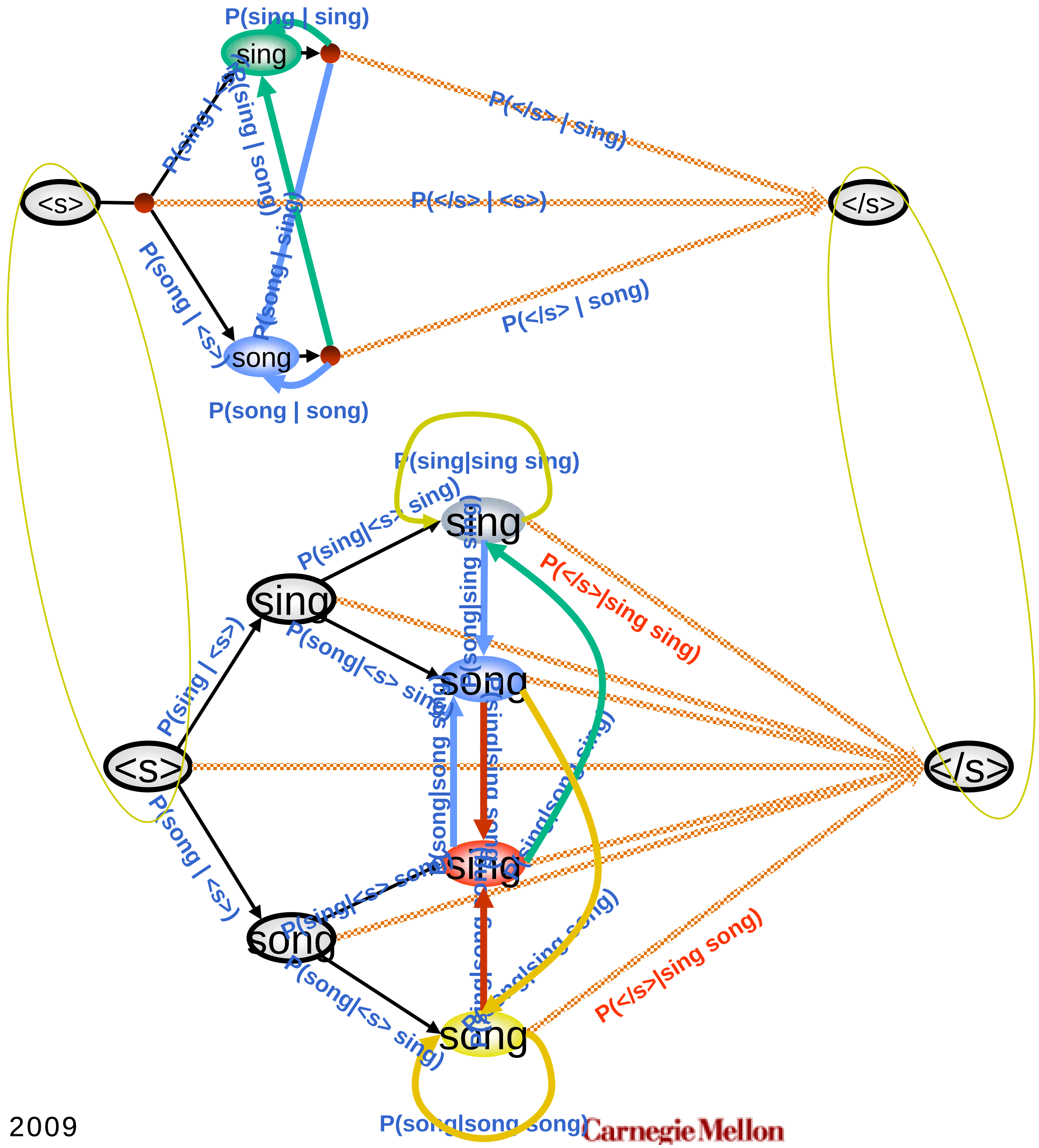
- This can be used to reduce graph size
  - If a trigram probability is obtained by backing off to a bigram, we can simply reuse bigram portions of the graph

- Thank you Mr. Zipf !!

The corresponding bigram graph



Put the two together



Using Backoffs

- The complete trigram LM for the two word language has the following set of probabilities:
  - P(sing |<s> song)
  - P(sing | <s> sing)
  - P(sing | sing sing)
  - P(sing | sing song)
  - P(sing | song sing)
  - P(sing | song song)

  - P(song |<s> song)
  - P(song | <s> sing)
  - P(song | sing sing)
  - P(song | sing song)
  - P(song | song sing)
  - P(song | song song)

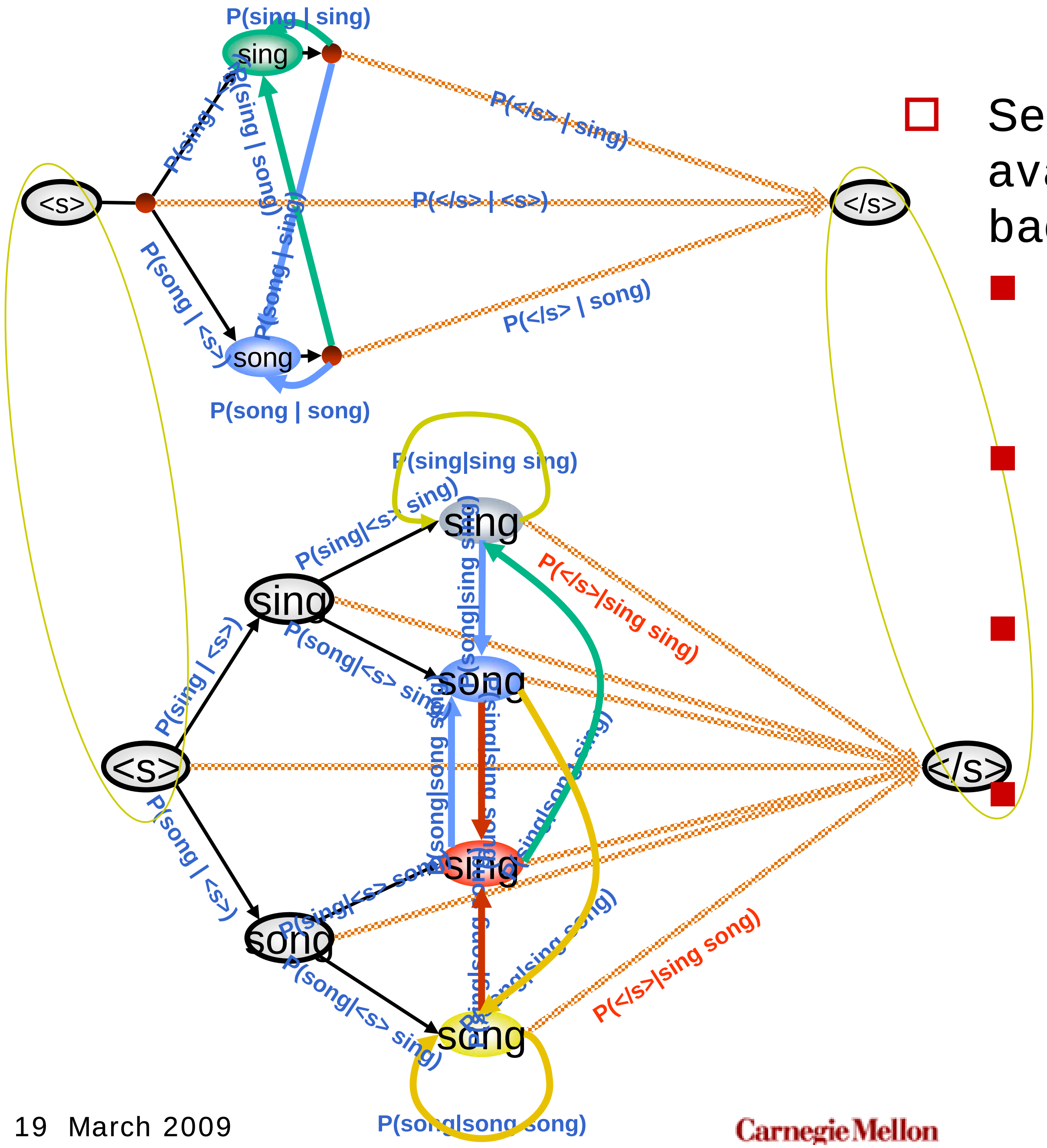
Using Backoffs

- The complete trigram LM for the two word language has the following set of probabilities:
  - P(sing |<s> song)
  - P(sing | <s> sing)
  - P(sing | sing sing)
  - P(sing | sing song)
  - P(sing | song sing)
  - P(sing | song song)

  - P(song |<s> song)
  - P(song | <s> sing)
  - P(song | sing sing)
  - P(song | sing song)
  - P(song | song sing)
  - P(song | song song)

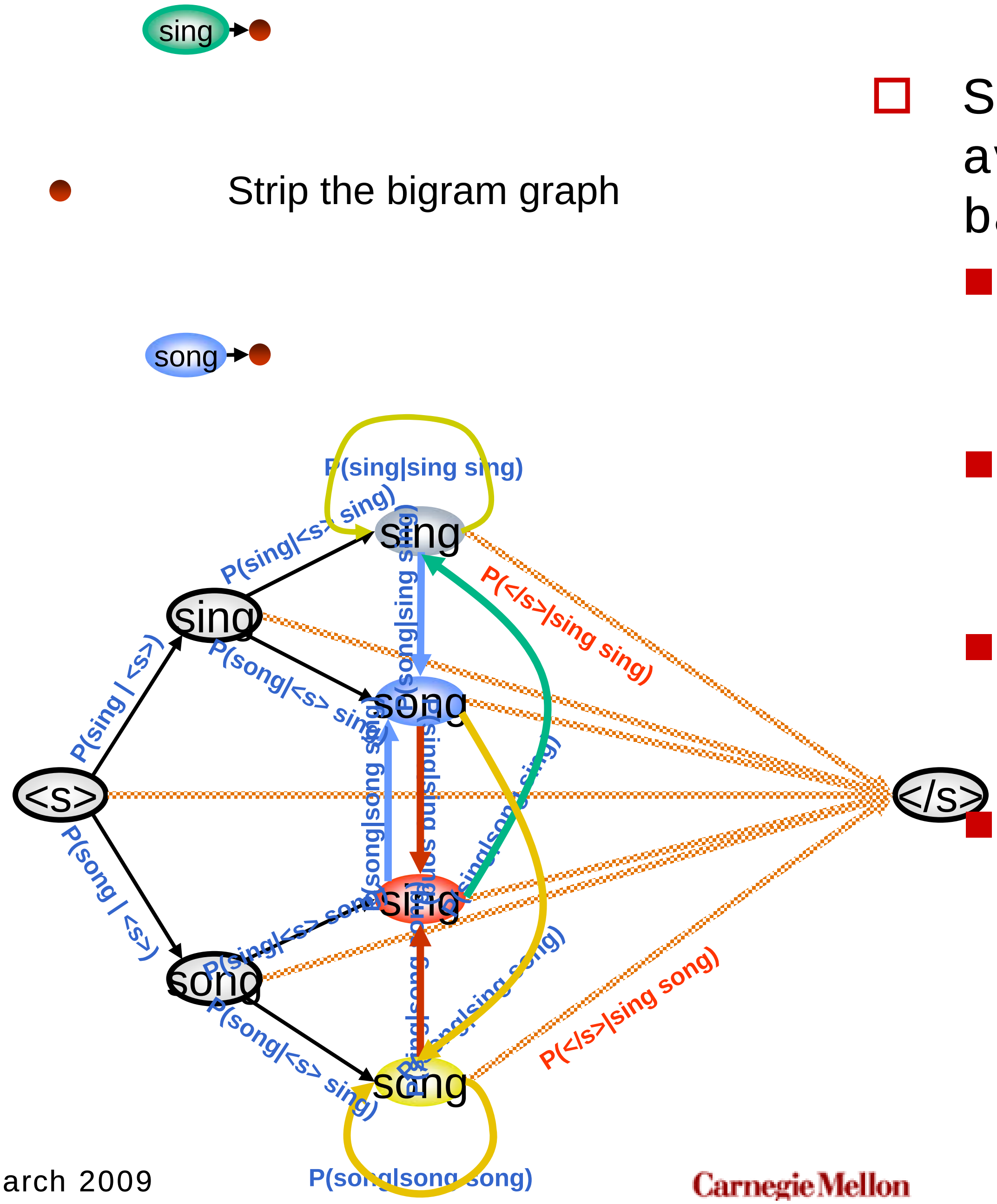
- Several of these are not available and obtained by backoff
  - P(sing | sing sing) = b(sing sing) P(sing|sing)
  - P(sing | song sing)= b(song sing) P(sing|sing)
  - P(song | song song) = b(song song)P(song|song)
  - P(song | song sing) = b(song sing)P(song|sing)

Several Trigrams are Backed off



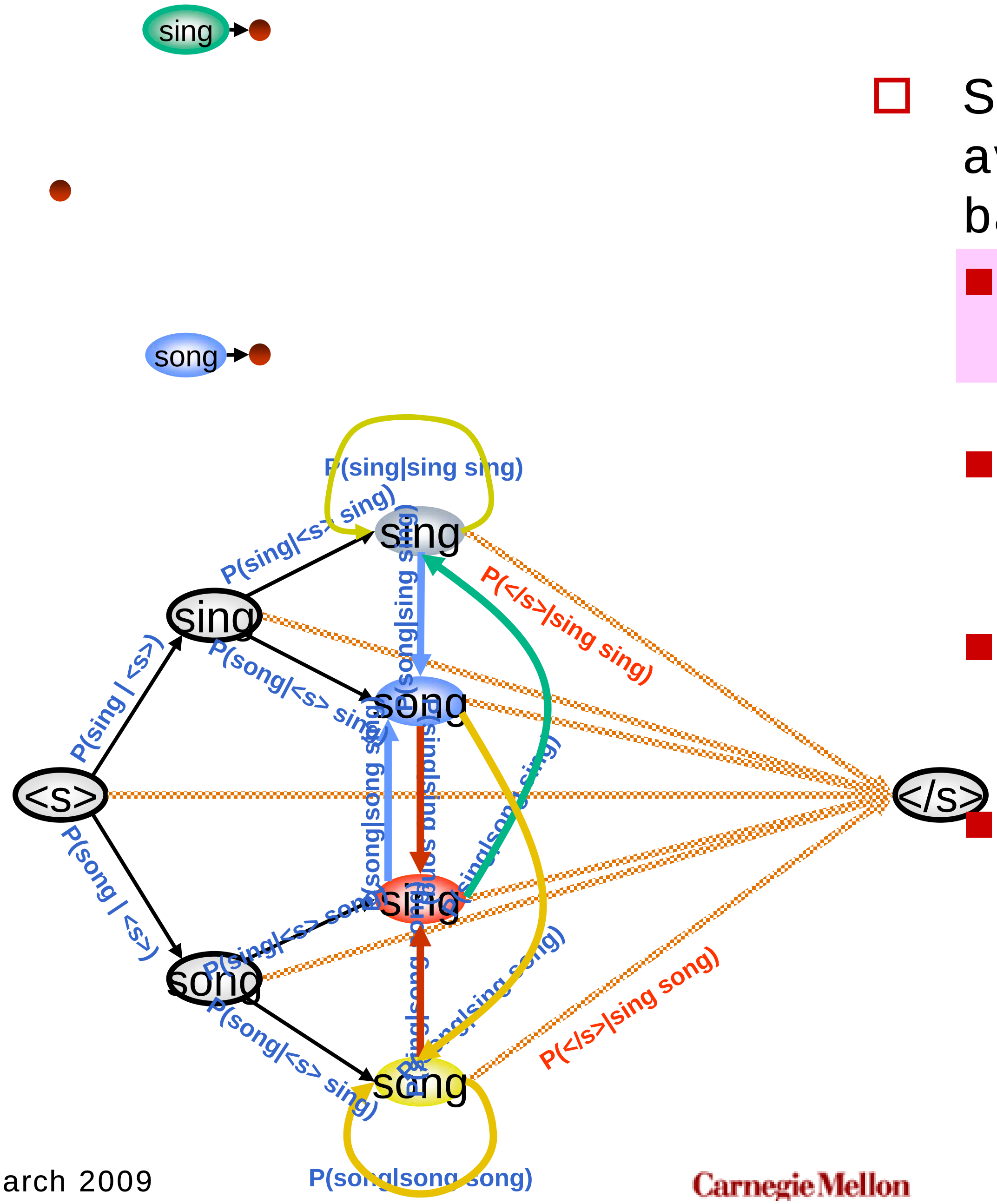
- Several of these are not available and obtained by backoff
  - P(sing | sing sing) = b(sing sing) P(sing|sing)
  - P(sing | song sing)= b(song sing) P(sing|sing)
  - P(song | song song) = b(song song)P(song|song)
  - P(song | song sing) = b(song sing)P(song|sing)

Several Trigrams are Backed off



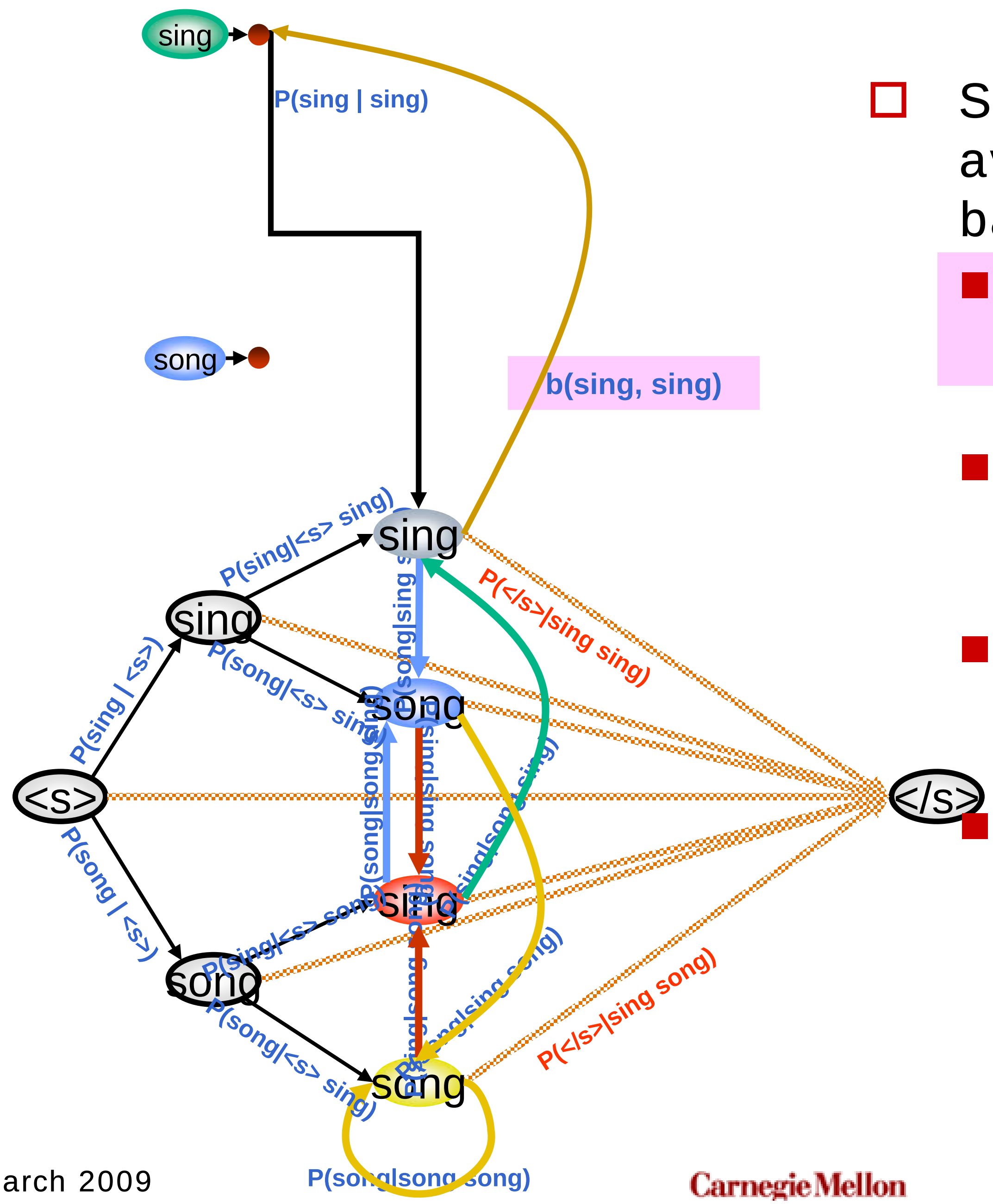
- Several of these are not available and obtained by backoff
  - P(sing | sing sing) = b(sing sing) P(sing|sing)
  - P(sing | song sing)= b(song sing) P(sing|sing)
  - P(song | song song) = b(song song)P(song|song)
  - P(song | song sing) = b(song sing)P(song|sing)

Backed off Trigram



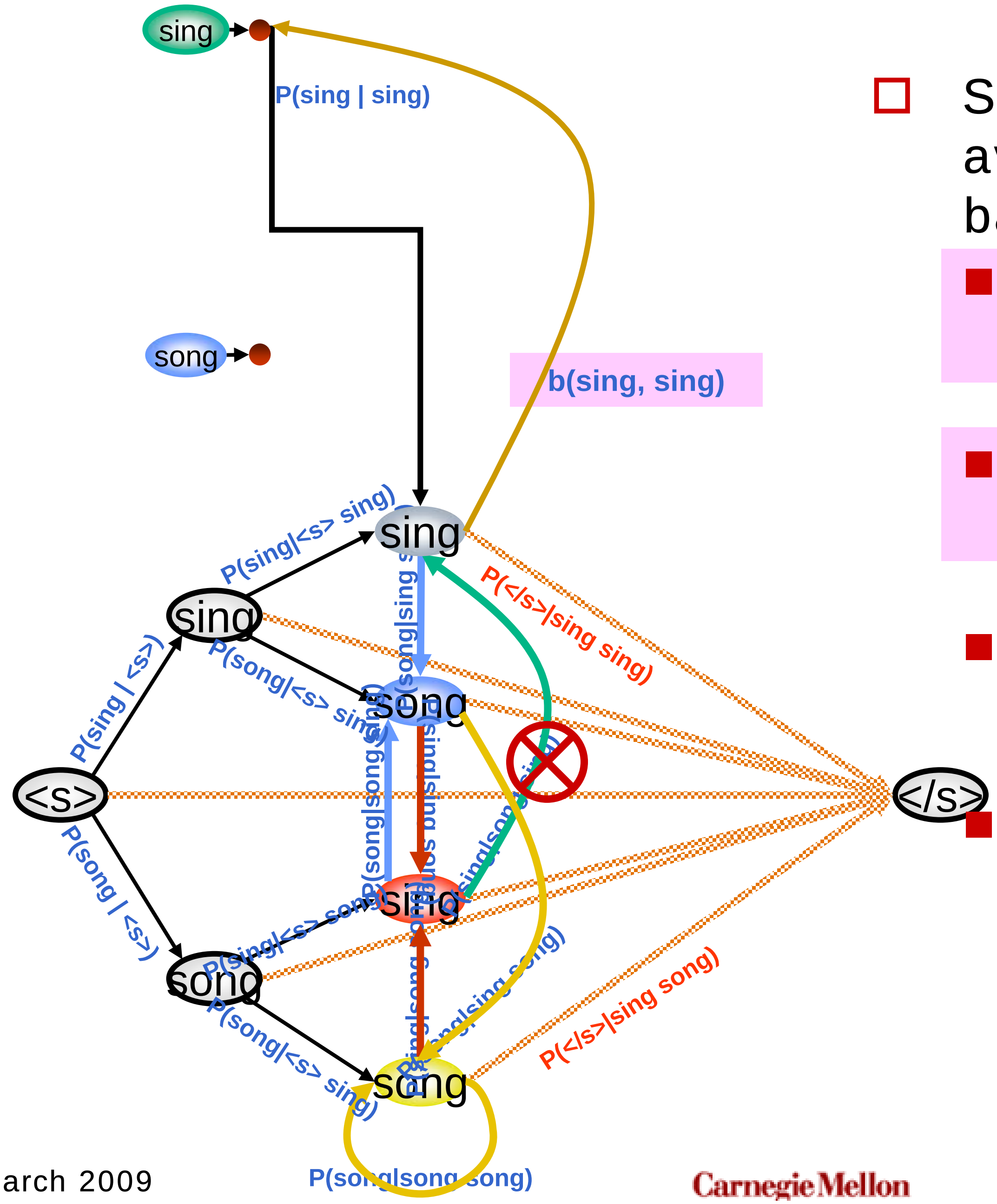
- Several of these are not available and obtained by backoff
  - P(sing | sing sing) = b(sing sing) P(sing|sing)
  - P(sing | song sing)= b(song sing) P(sing|sing)
  - P(song | song song) = b(song song)P(song|song)
  - P(song | song sing) = b(song sing)P(song|sing)

Hook backed off trigram to the bigram graph



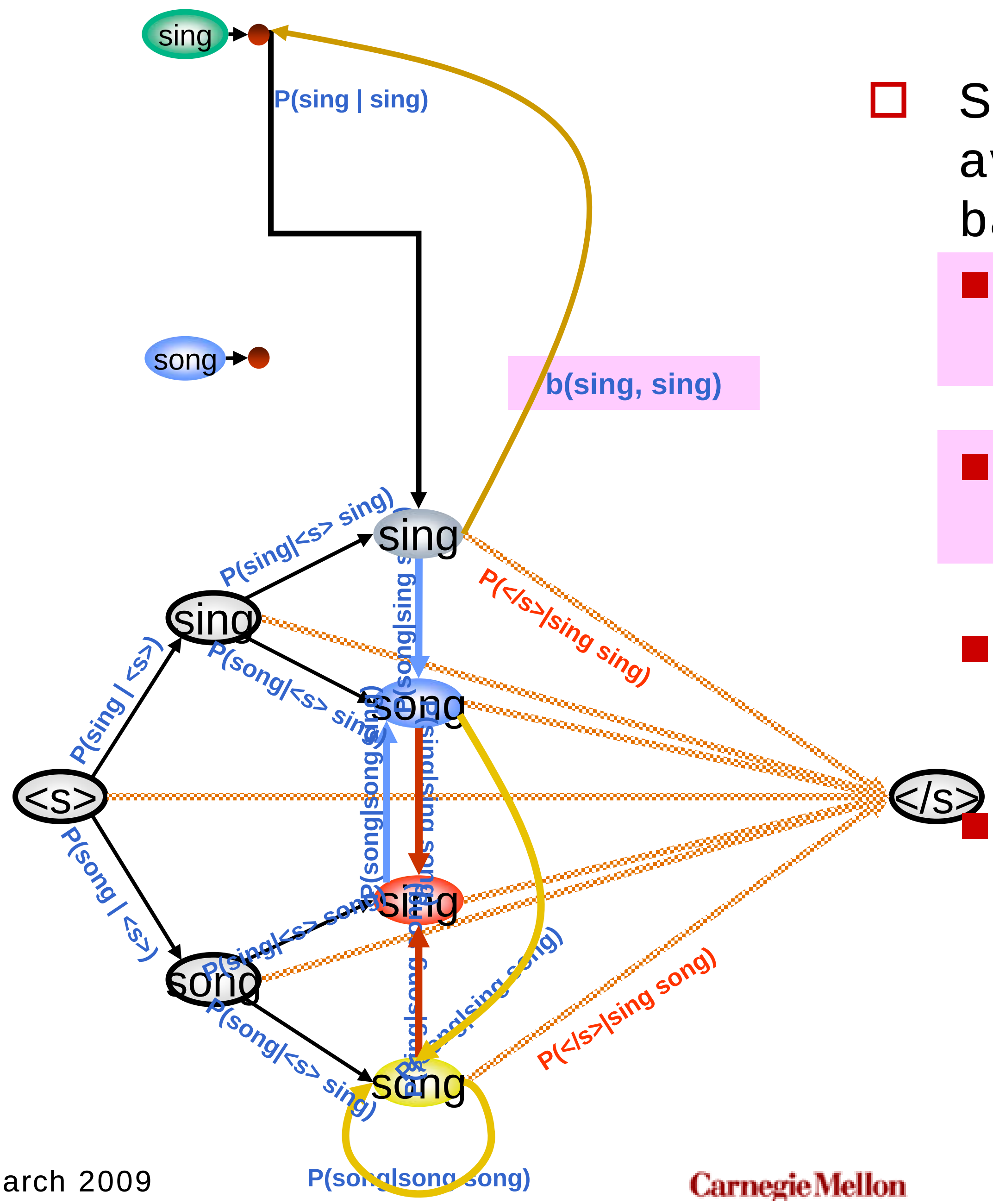
- Several of these are not available and obtained by backoff
  - P(sing | sing sing) = b(sing sing) P(sing|sing)
  - P(sing | song sing)= b(song sing) P(sing|sing)
  - P(song | song song) = b(song song)P(song|song)
  - P(song | song sing) = b(song sing)P(song|sing)

Hook backed off trigram to the bigram graph



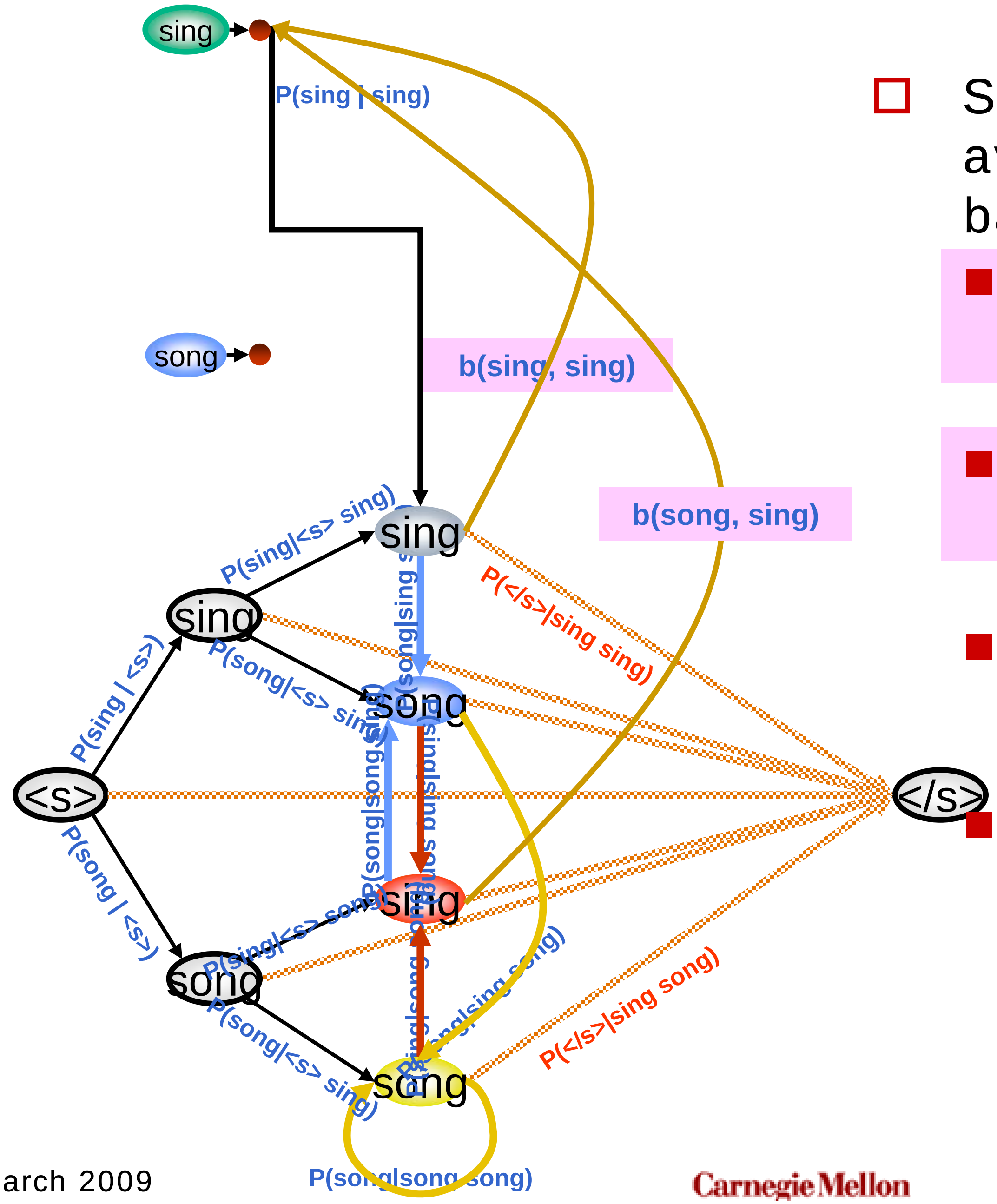
- Several of these are not available and obtained by backoff
  - P(sing | sing sing) = b(sing sing) P(sing|sing)
  - P(sing | song sing)= b(song sing) P(sing|sing)
  - P(song | song song) = b(song song)P(song|song)
  - P(song | song sing) = b(song sing)P(song|sing)

Hook backed off trigram to the bigram graph



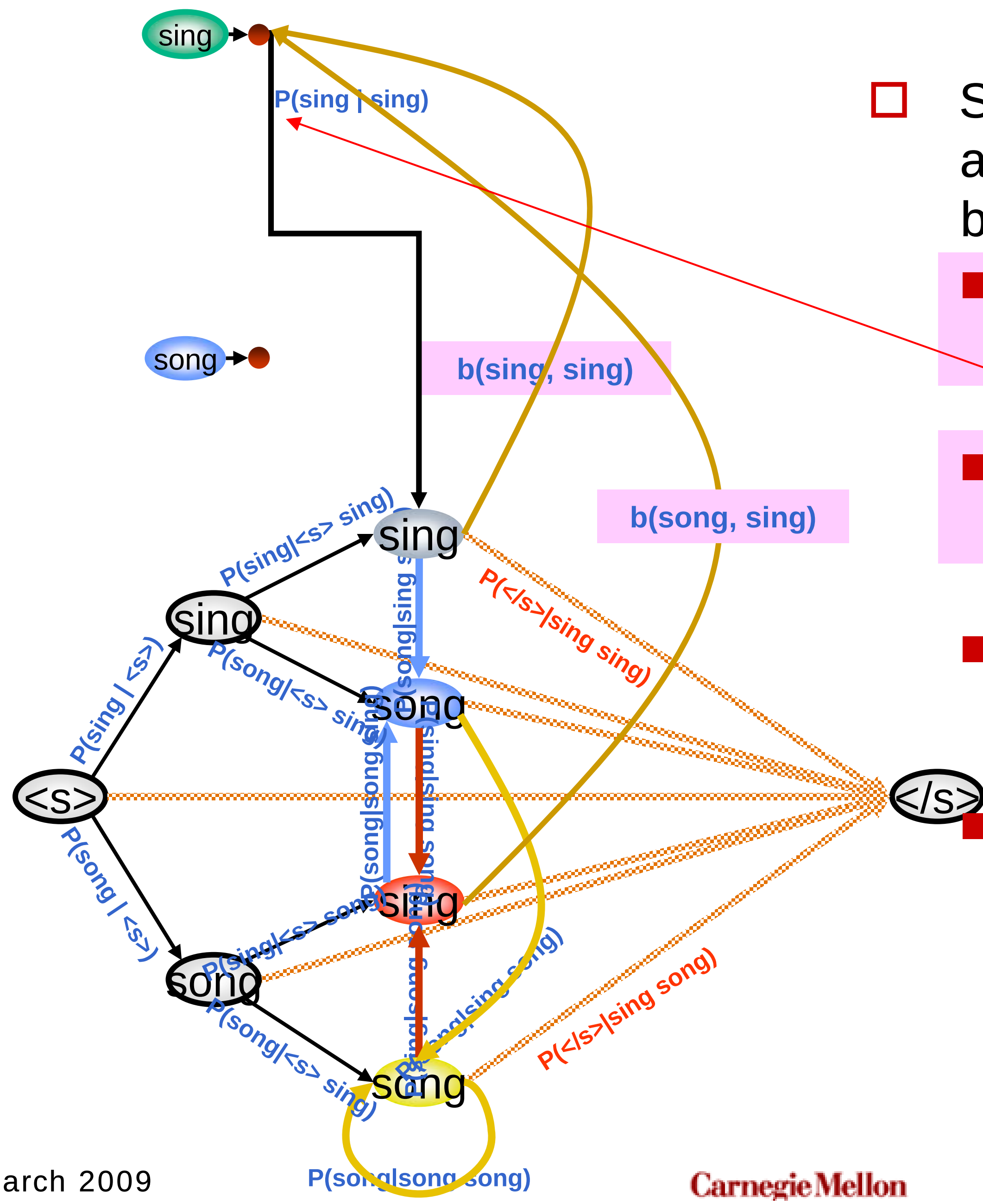
- Several of these are not available and obtained by backoff
  - P(sing | sing sing) = b(sing sing) P(sing|sing)
  - P(sing | song sing)= b(song sing) P(sing|sing)
  - P(song | song song) = b(song song)P(song|song)
  - P(song | song sing) = b(song sing)P(song|sing)

Hook backed off trigram to the bigram graph



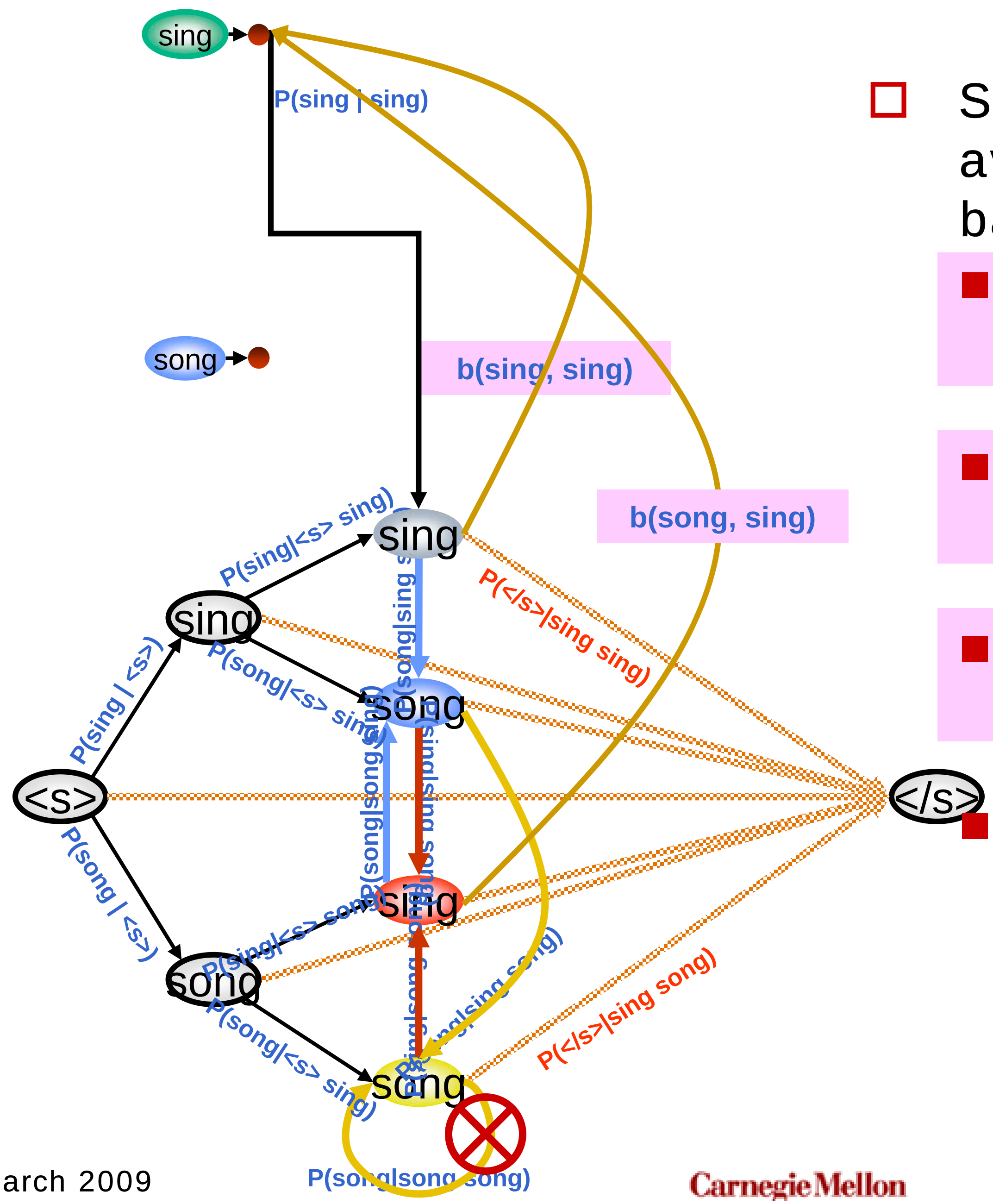
- Several of these are not available and obtained by backoff
  - P(sing | sing sing) = b(sing sing) P(sing|sing)
  - P(sing | song sing)= b(song sing) P(sing|sing)
  - P(song | song song) = b(song song)P(song|song)
  - P(song | song sing) = b(song sing)P(song|sing)

Hook backed off trigram to the bigram graph



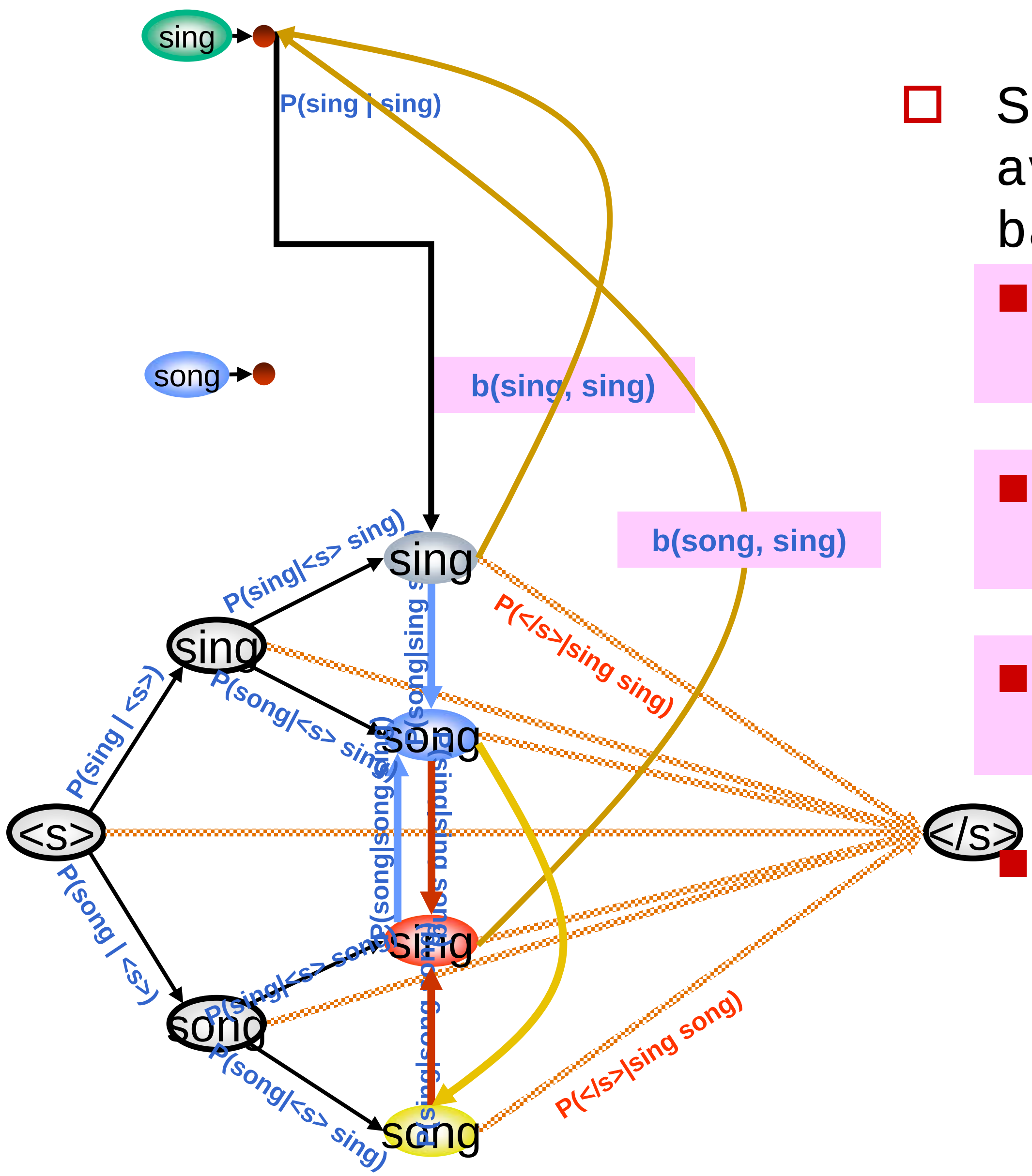
- Several of these are not available and obtained by backoff
  - P(sing | sing sing) = b(sing sing) P(sing|sing)
  - P(sing | song sing) = b(song sing) P(sing|sing)
  - P(song | song song) = b(song song)P(song|song)
  - P(song | song sing) = b(song sing)P(song|sing)

Hook backed off trigram to the bigram graph



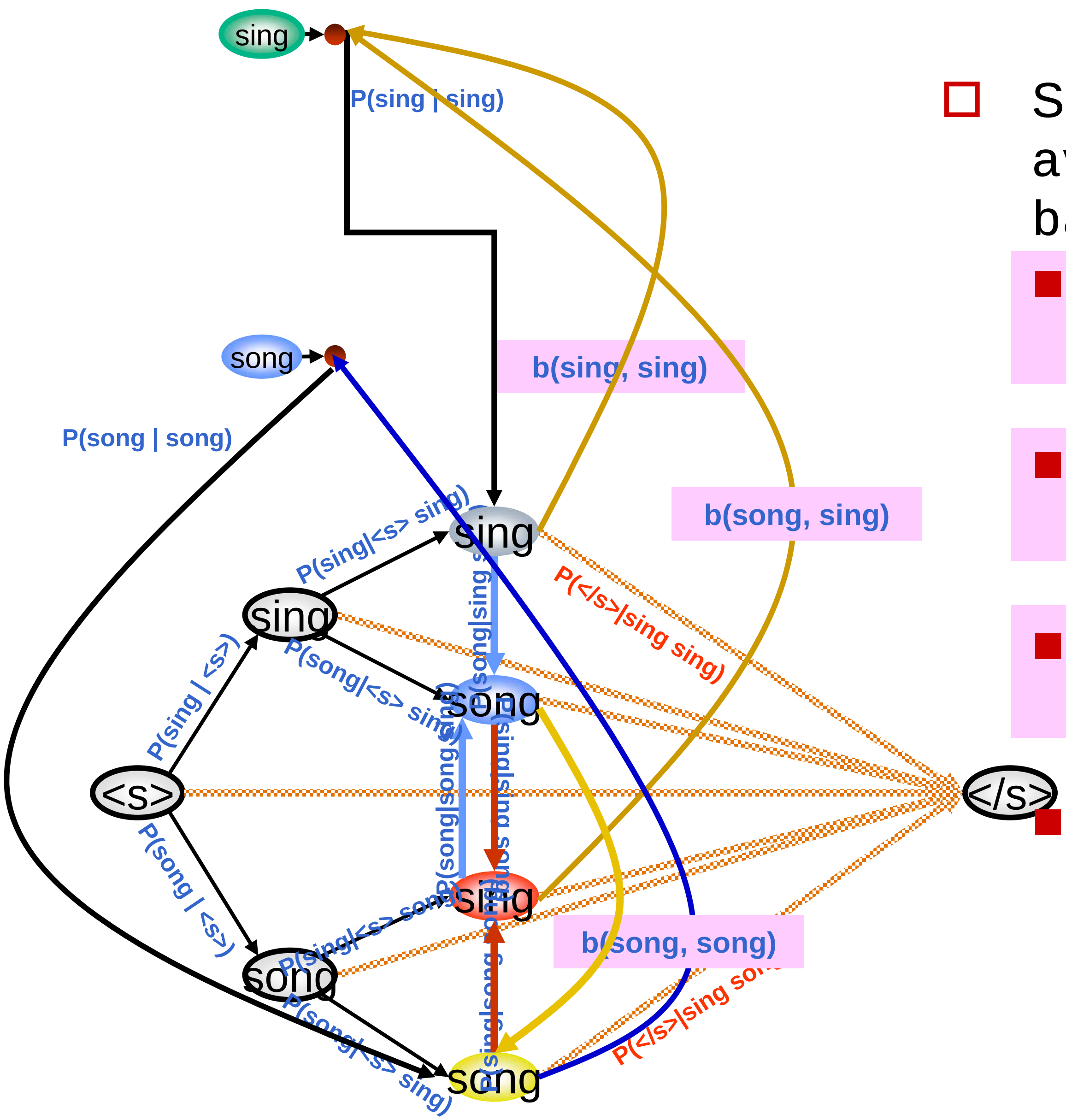
- Several of these are not available and obtained by backoff
  - P(sing | sing sing) = b(sing sing) P(sing|sing)
  - P(sing | song sing)= b(song sing) P(sing|sing)
  - P(song | song song) = b(song song)P(song|song)
  - P(song | song sing) = b(song sing)P(song|sing)

Hook backed off trigram to the bigram graph



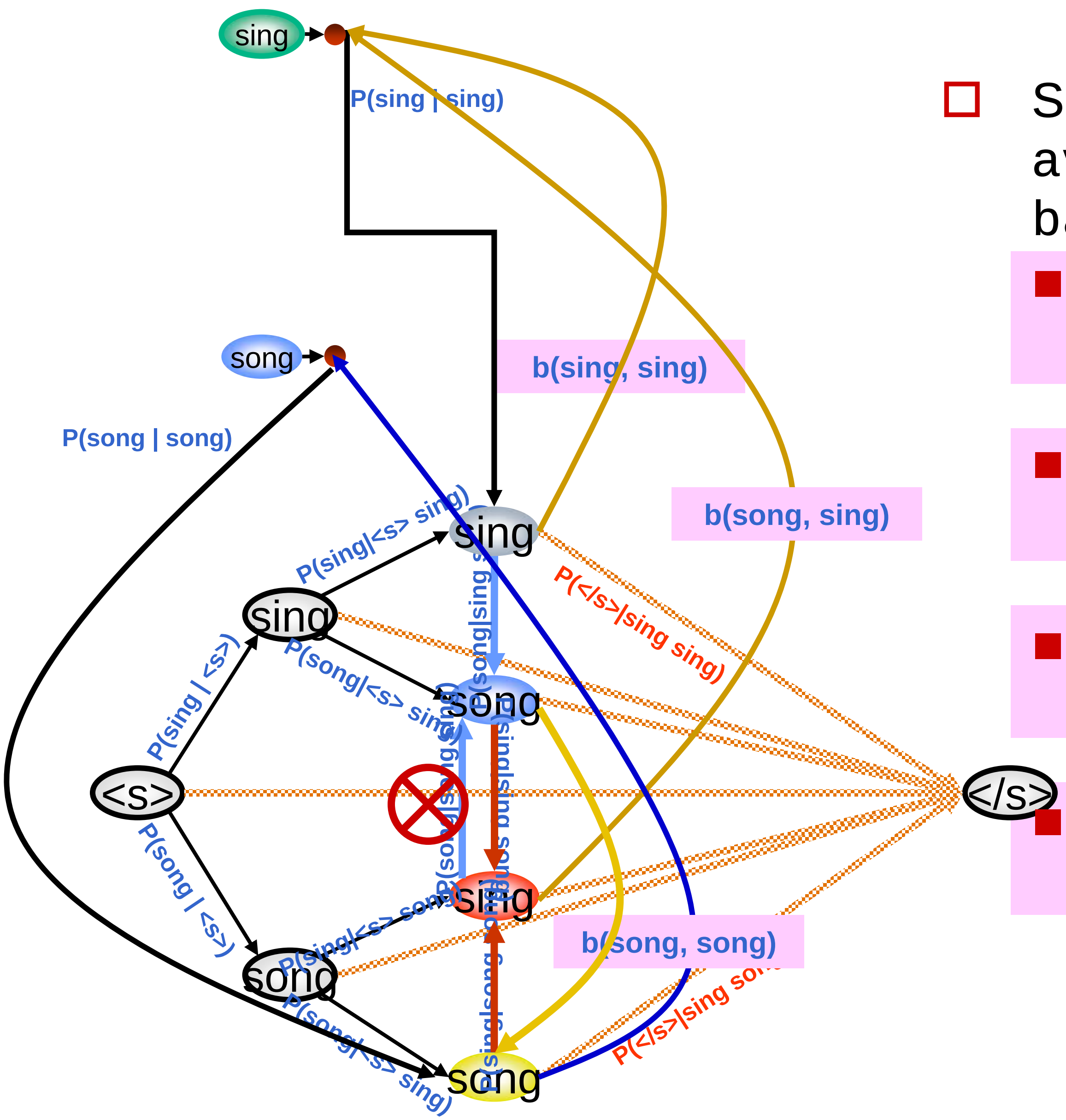
- Several of these are not available and obtained by backoff
  - P(sing | sing sing) = b(sing sing) P(sing|sing)
  - P(sing | song sing)= b(song sing) P(sing|sing)
  - P(song | song song) = b(song song)P(song|song)
  - P(song | song sing) = b(song sing)P(song|sing)

Hook backed off trigram to the bigram graph



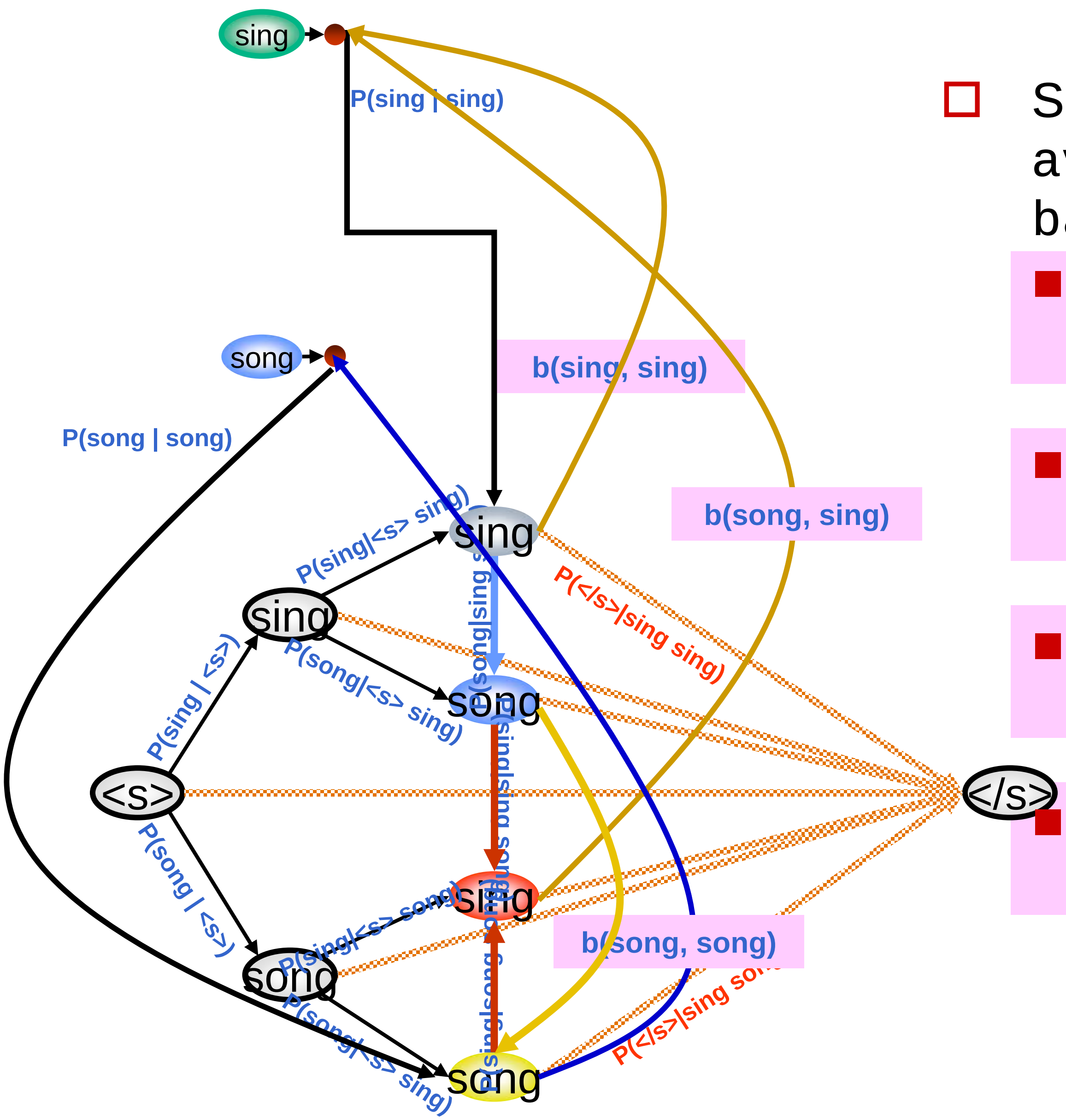
- Several of these are not available and obtained by backoff
  - P(sing | sing sing) = b(sing sing) P(sing|sing)
  - P(sing | song sing)= b(song sing) P(sing|sing)
  - P(song | song song) = b(song song)P(song|song)
  - P(song | song sing) = b(song sing)P(song|sing)

Hook backed off trigram to the bigram graph



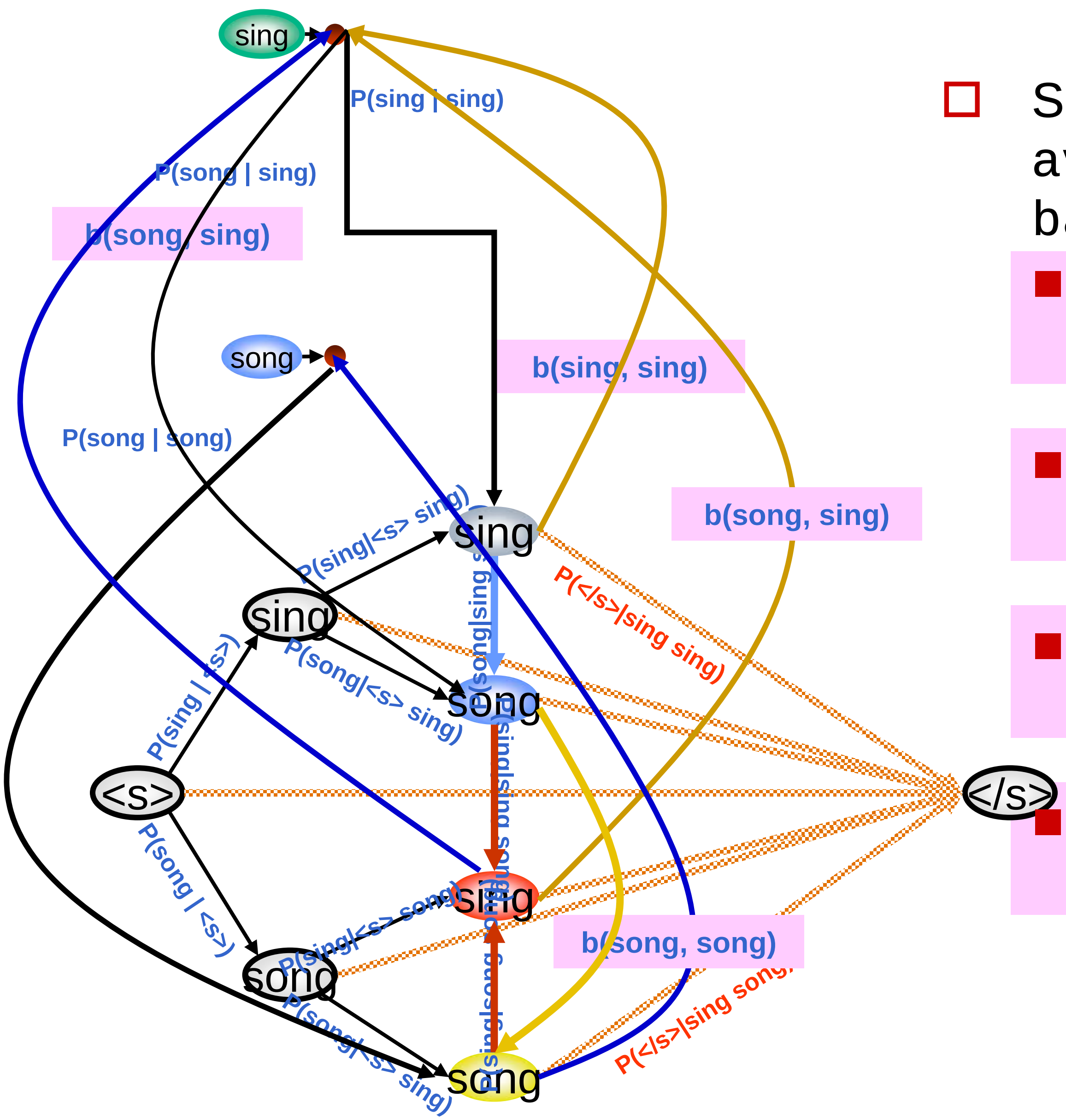
- Several of these are not available and obtained by backoff
  - P(sing | sing sing) = b(sing sing) P(sing|sing)
  - P(sing | song sing)= b(song sing) P(sing|sing)
  - P(song | song song) = b(song song)P(song|song)
  - P(song | song sing) = b(song sing)P(song|sing)

Hook backed off trigram to the bigram graph



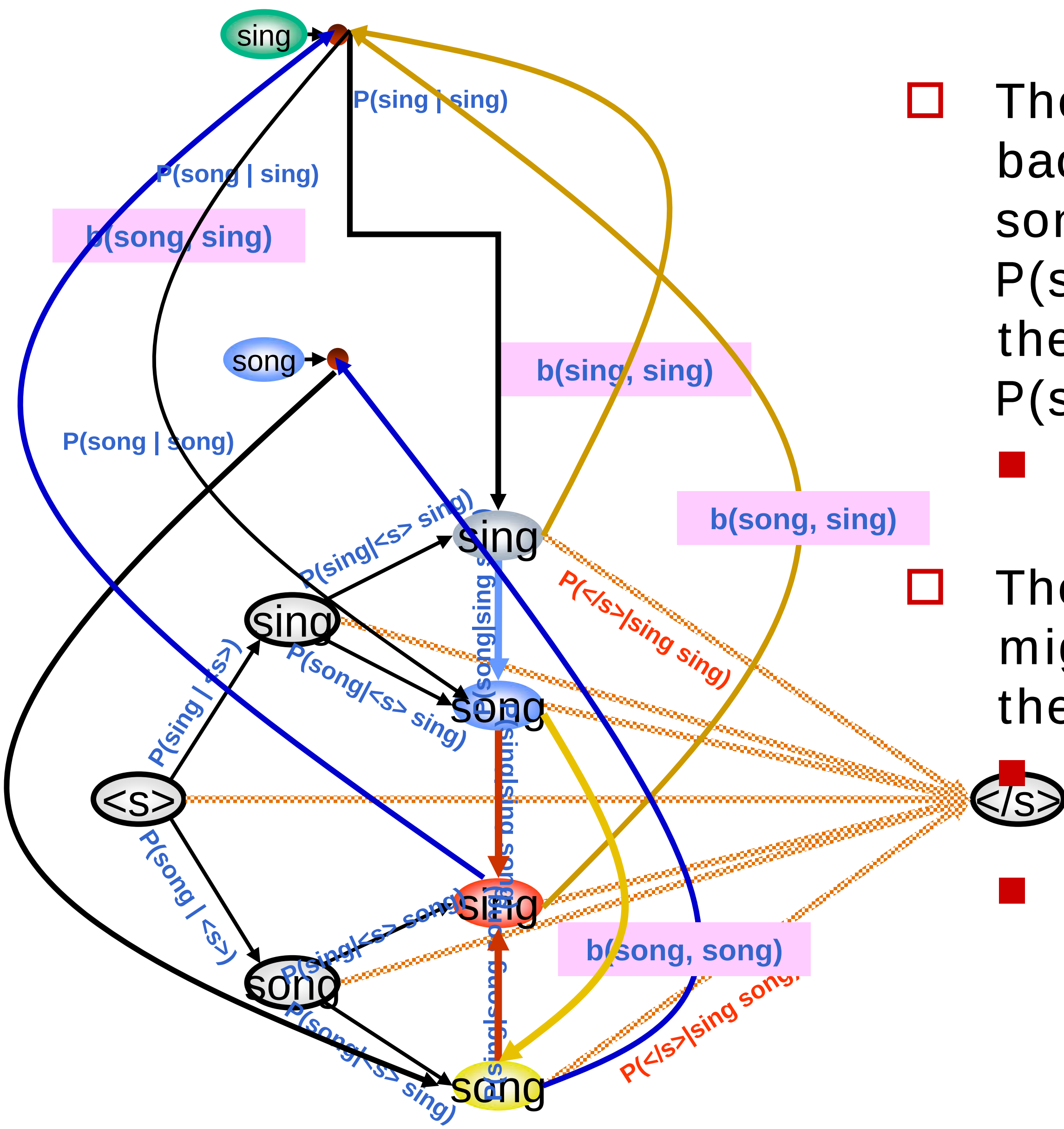
- Several of these are not available and obtained by backoff
  - P(sing | sing sing) = b(sing sing) P(sing|sing)
  - P(sing | song sing)= b(song sing) P(sing|sing)
  - P(song | song song) = b(song song)P(song|song)
  - P(song | song sing) = b(song sing)P(song|sing)

Hook backed off trigram to the bigram graph



- Several of these are not available and obtained by backoff
  - P(sing | sing sing) = b(sing sing) P(sing|sing)
  - P(sing | song sing)= b(song sing) P(sing|sing)
  - P(song | song song) = b(song song)P(song|song)
  - P(song | song sing) = b(song sing)P(song|sing)

Hook backed off trigram to the bigram graph



- The good: By adding a backoff arc to “sing” from song to compose P(song|song sing), we got the backed off probability for P(sing|song sing) for free
  - This can result in an enormous reduction of size
- The bad: P(sing|song sing) might already validly exist in the graph!!
  - Some N-gram arcs have two different variants
  - This introduces spurious *multiple* definitions of some trigrams

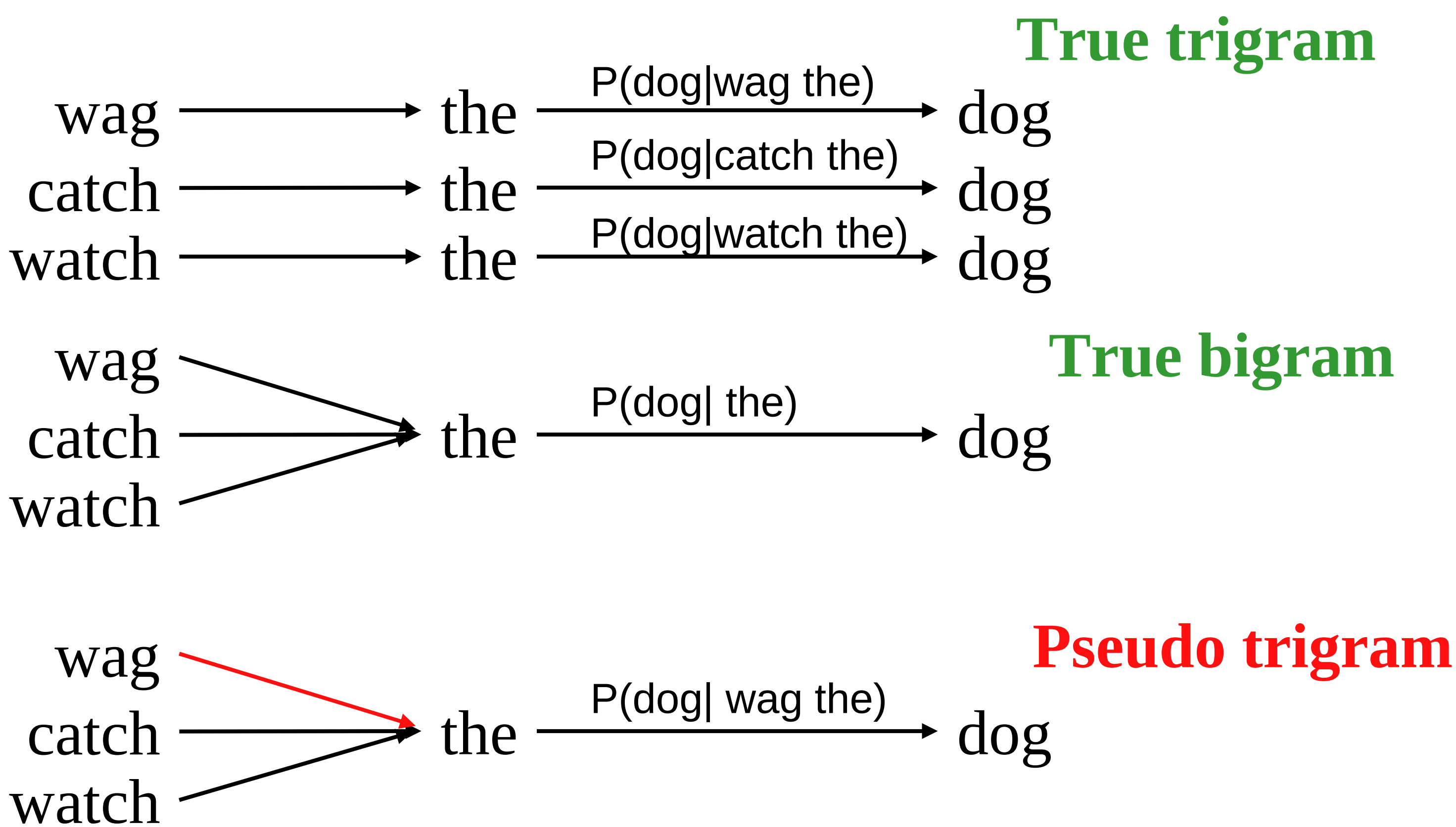
Even compressed graphs are large

- Even a compressed N-gram word graph can get very large
  - Explicit arcs for at least every bigram and every trigram in the LM
  - This can get to tens or hundreds of millions

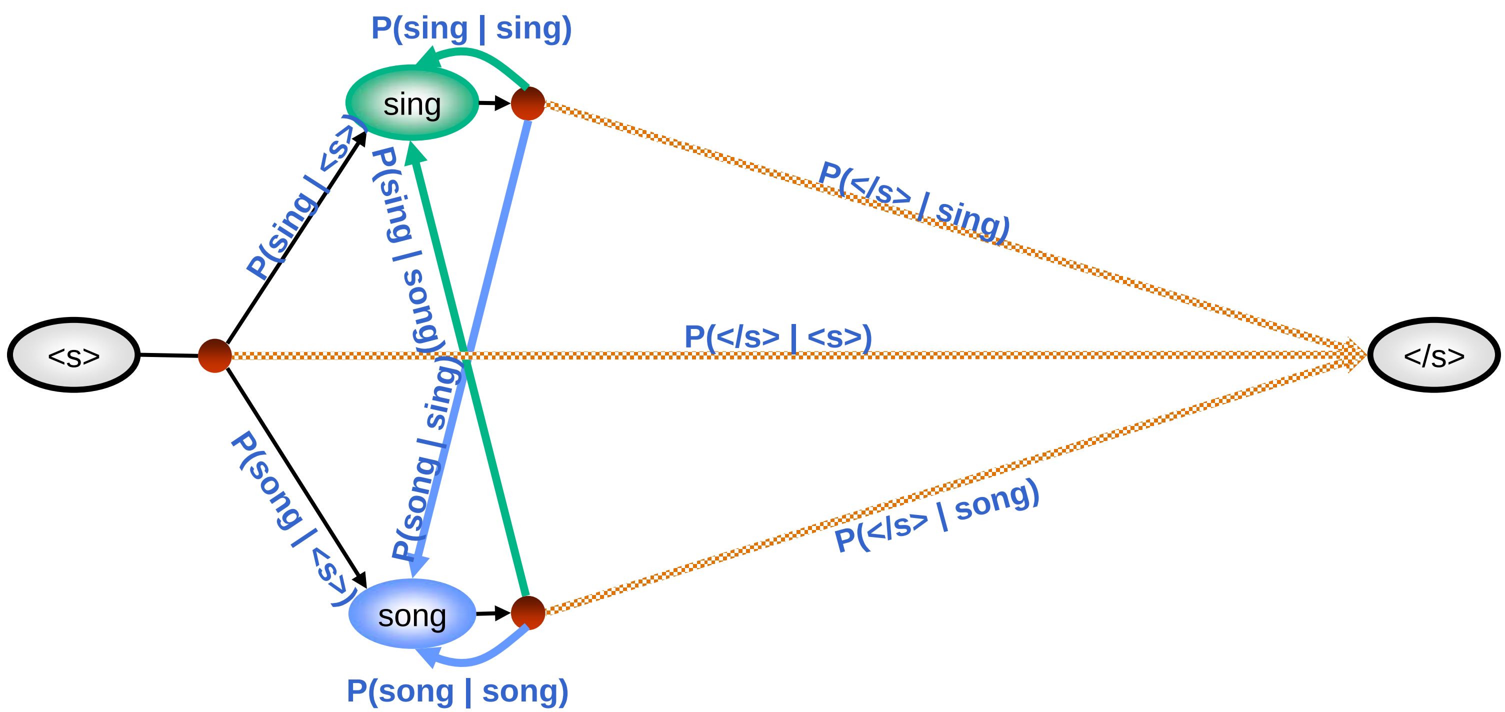
- Approximate structures required
  - The approximate structure is, well, approximate
  - It reduces the graph size
    - This breaks the requirement that every node in the graph represents a unique word history
    - We compensate by using additional external structures to track word history

The pseudo trigram approach

- Each word has its own HMM
  - Computation and memory intensive
- Only a "pseudo-trigram" search:

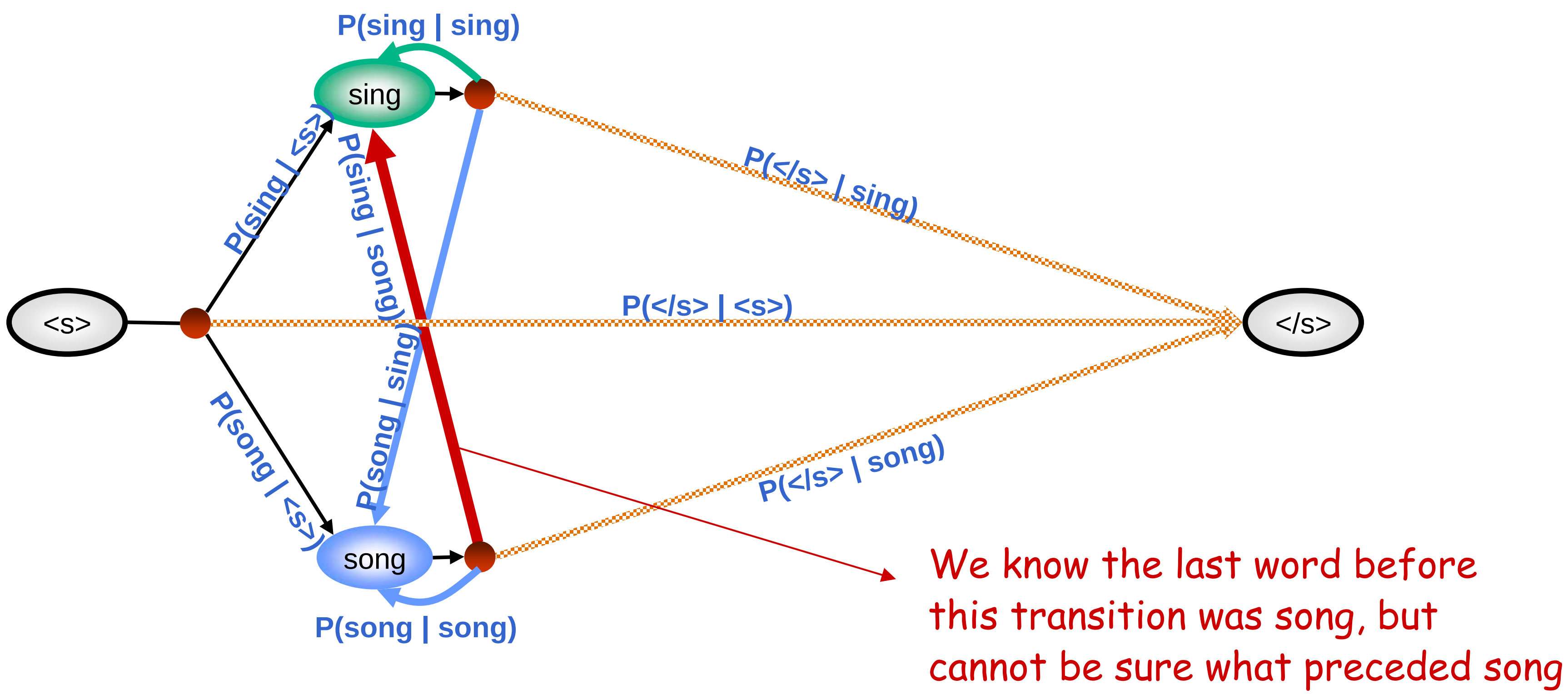


Pseudo Trigram



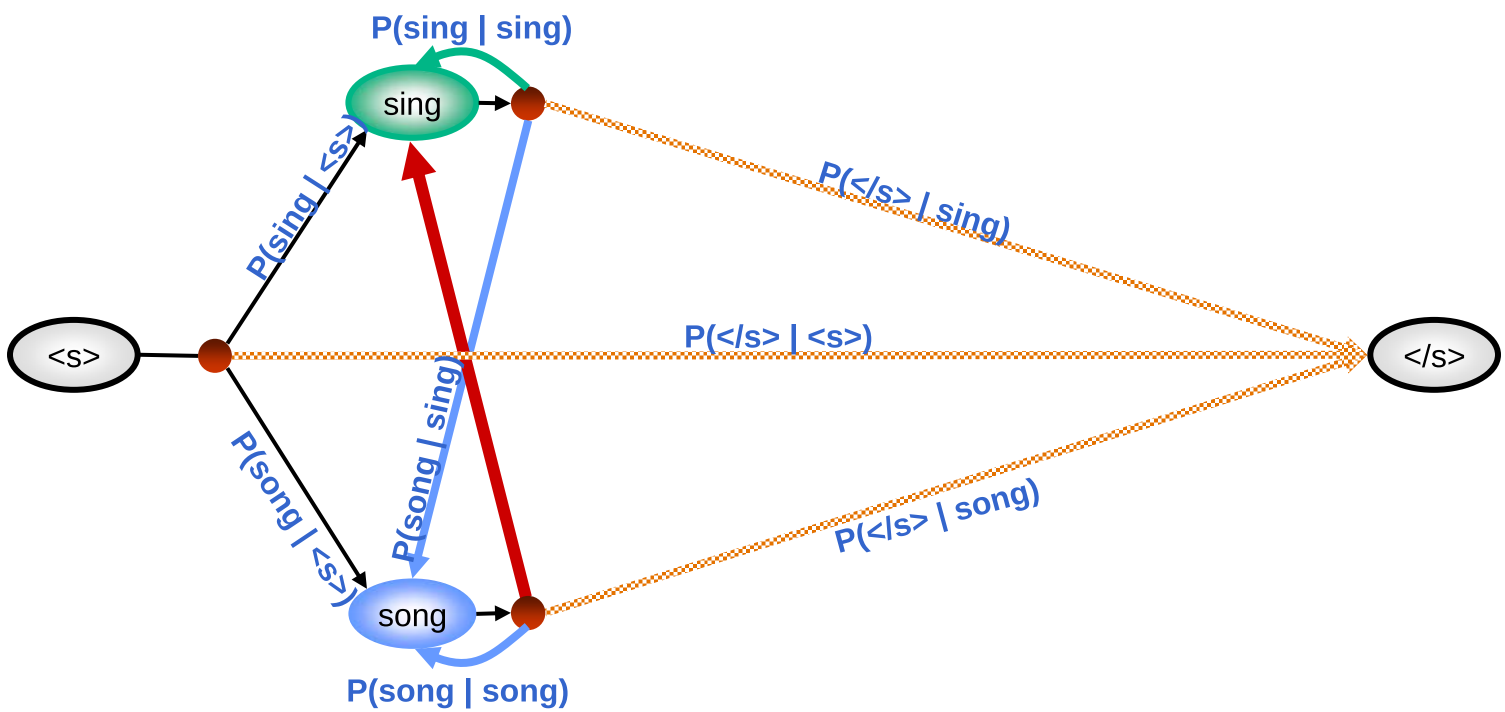
- Use a simple bigram graph
  - Each word only represents a *single word* history
  - At the outgoing edges from any word we can only be certain of the last word

Pseudo Trigram



- Use a simple bigram graph
  - Each word only represents a *single word* history
  - At the outgoing edges from any word we can only be certain of the last word
    - As a result we cannot apply *trigram* probabilities, since these require knowledge of two-word histories

Pseudo Trigram



- Solution: Obtain information about the word that preceded "song" on the path from the backpointer table
- Use that word along with "song" as a two-word history
  - Can now apply a trigram probability

Pseudo Trigram

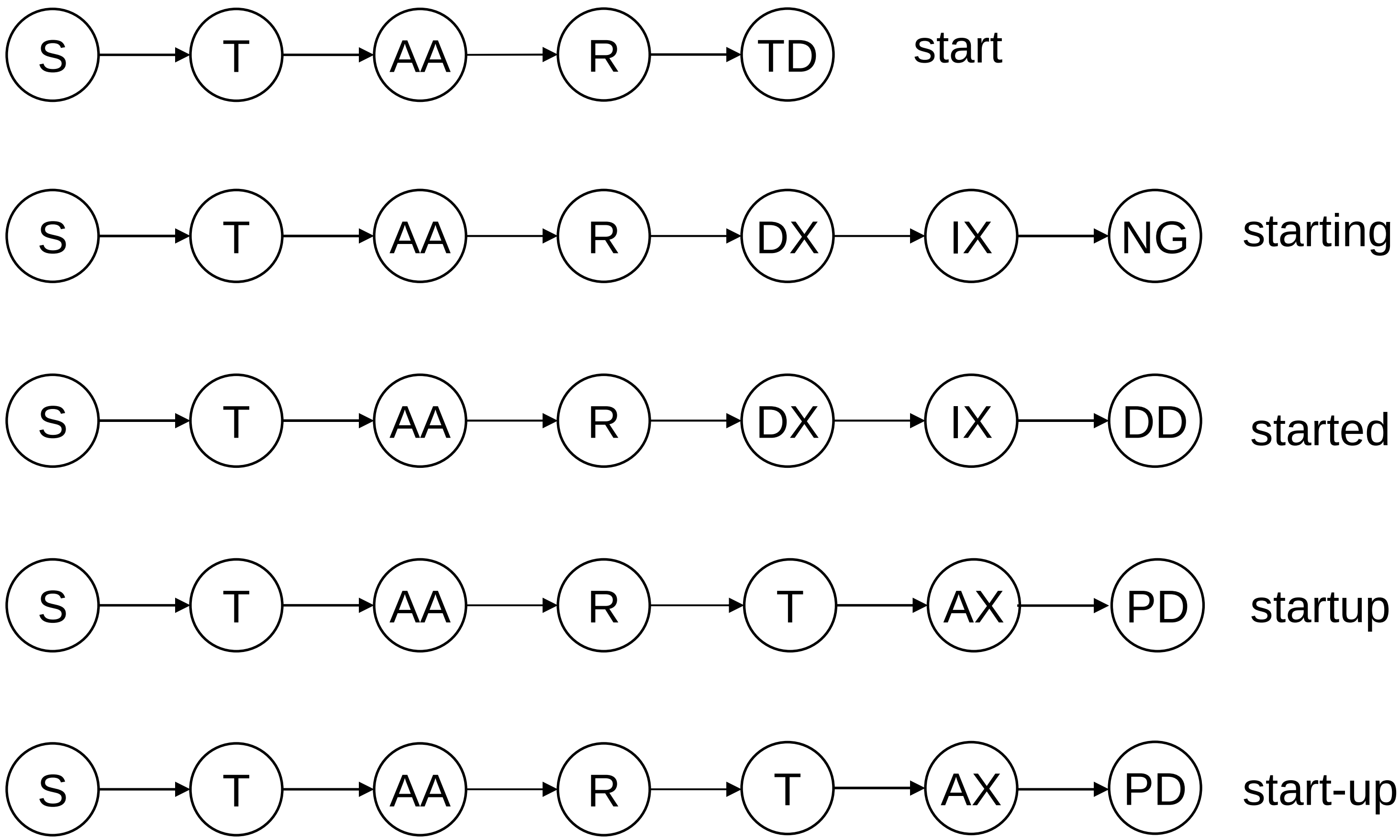
- The problem with the pseudo-trigram approach is that the LM probabilities to be applied can no longer be stored on the graph edge
  - The actual probability to be applied will differ according to the best previous word obtained from the backpointer table
  - As a result, the recognition output obtained from the structure is no longer guaranteed optimal in a Bayesian sense!
- Nevertheless the results are fairly close to optimal
  - The loss in optimality due to the reduced dynamic structure is acceptable, given the reduction in graph size
- This form of decoding is performed in the “fwdflat” mode of the sphinx3 decoder

Pseudo Trigram: Still not efficient

- Even a bigram structure can be inefficient to search
  - Large number of models
  - Many edges
  - Not taking advantage of shared portions of the graph

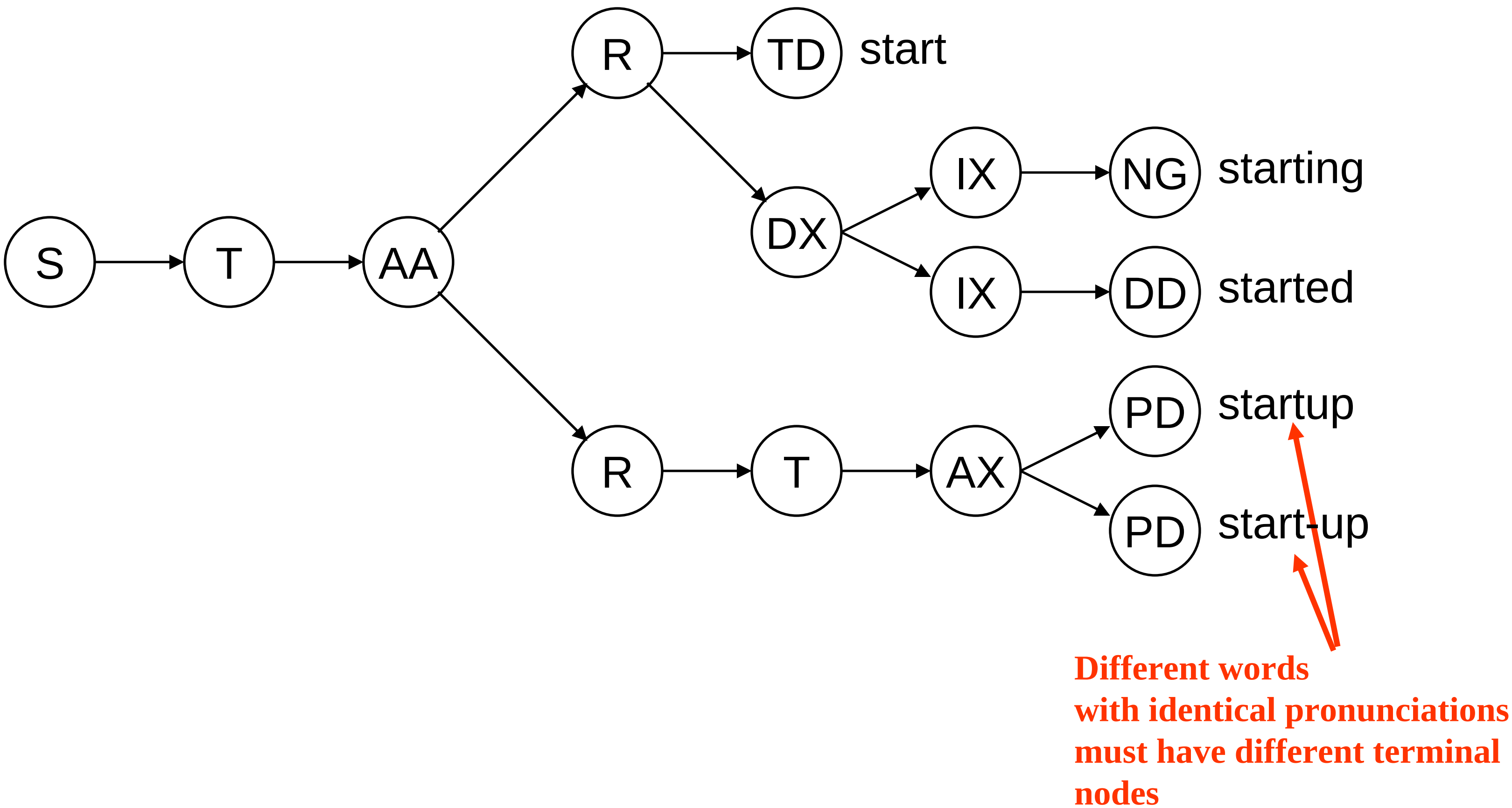
A Vocabulary of Five Words

"Flat" approach: a different model for every word



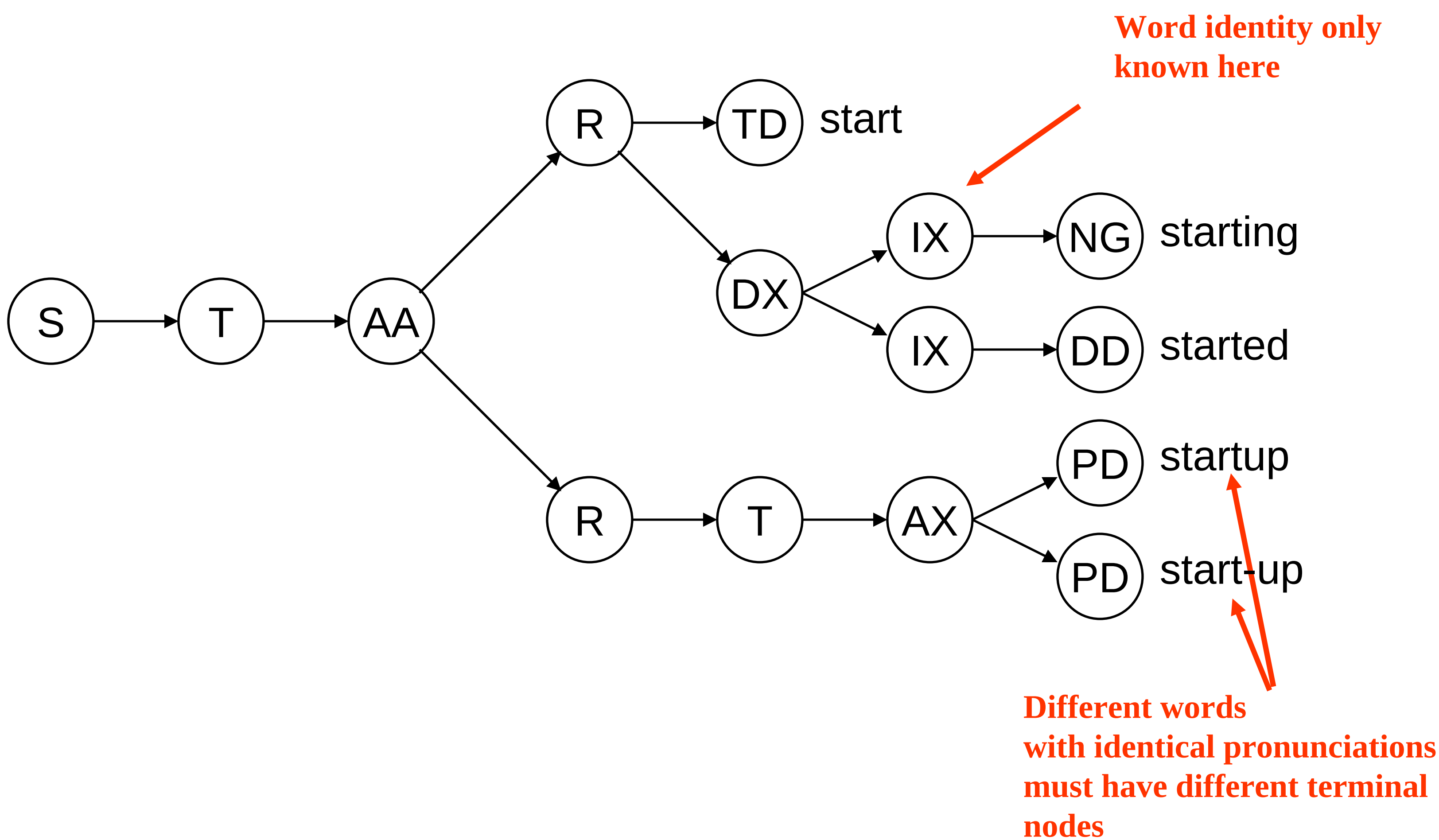
Lextree

- o Common portions of the words are shared
  - Example assumes triphone models

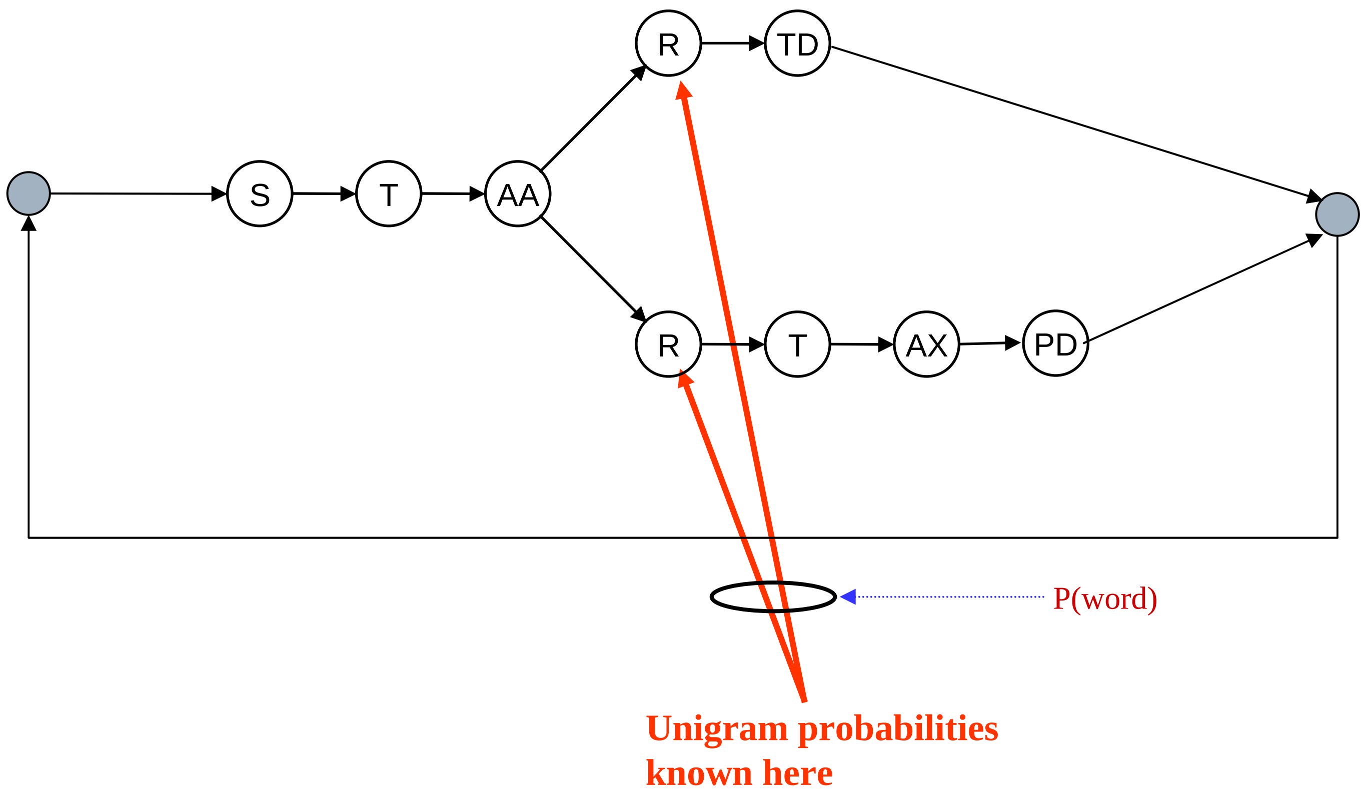


Lextree

- The probability of a word is obtained deep in the tree
    - Example assumes triphone models



Unigram Lextree Decoding



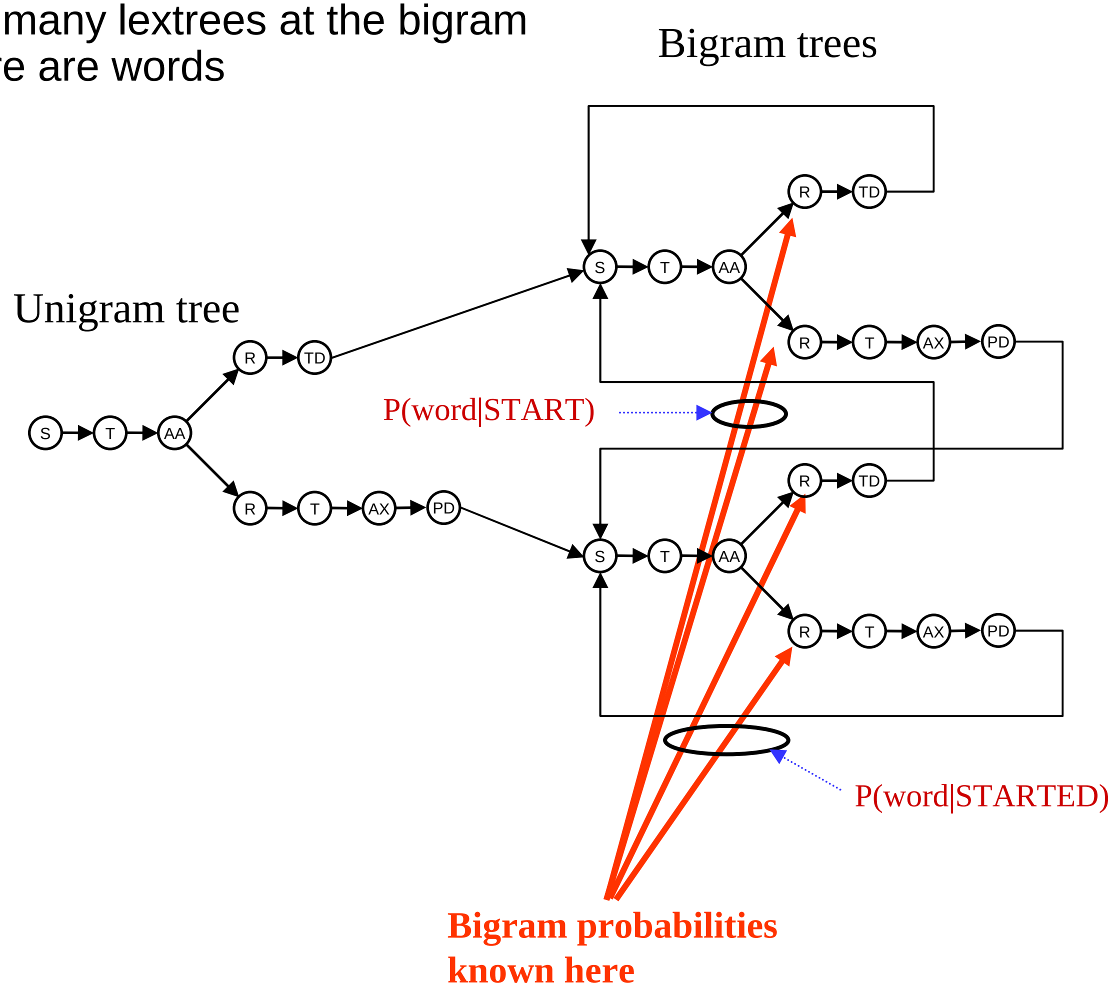
Lextrees

- Superficially, lextrees appear to be highly efficient structures
  - A lextree representation of a dictionary of 100000 words typically reduces the overall structure by a factor of 10, as compared to a “flat” representation

- However, all is not hunky dory..

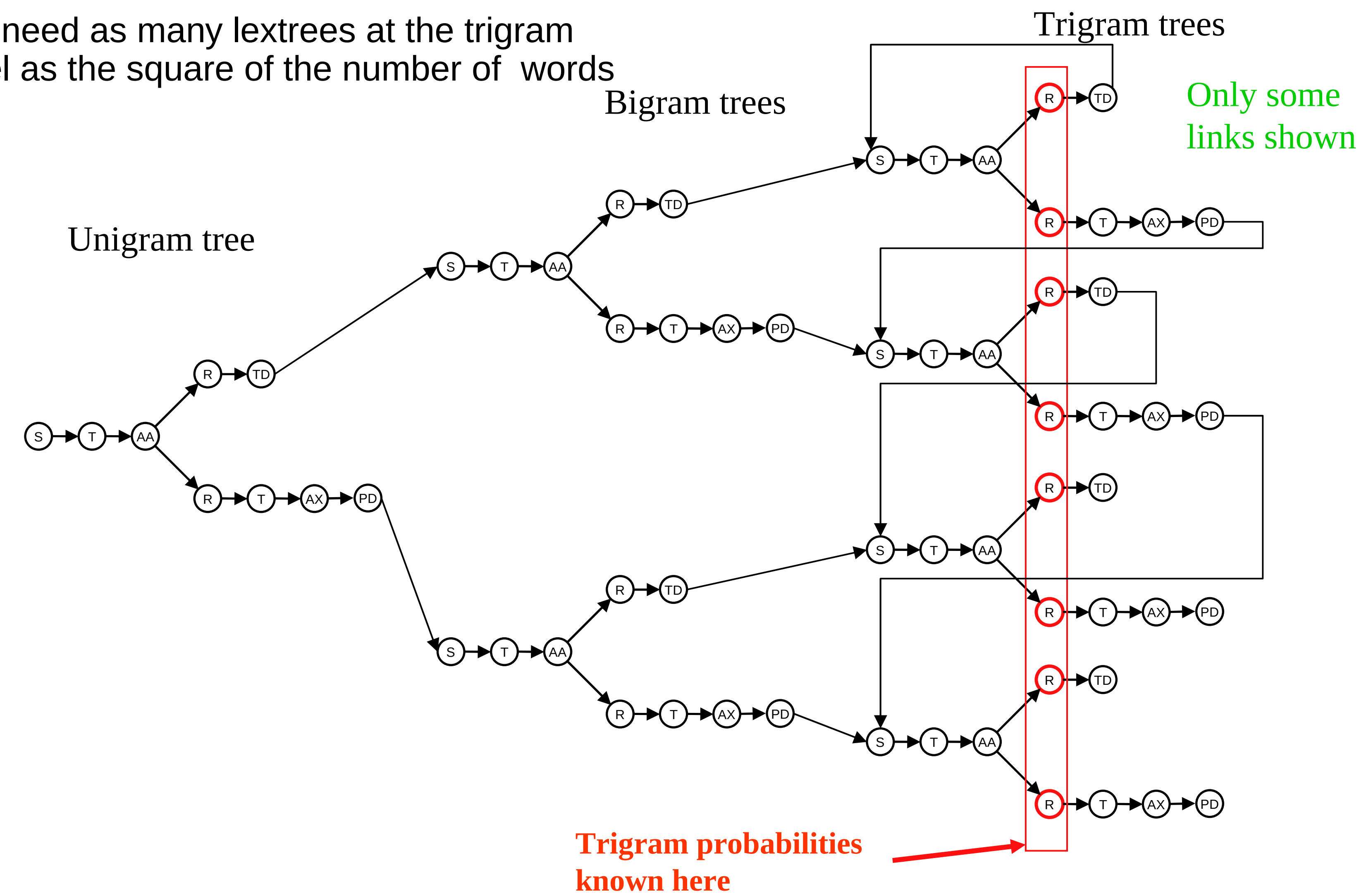
Bigram Lextree Decoding

- Since word identities are not known at entry we need as many lextrees at the bigram level as there are words



Trigram Lextree Decoding

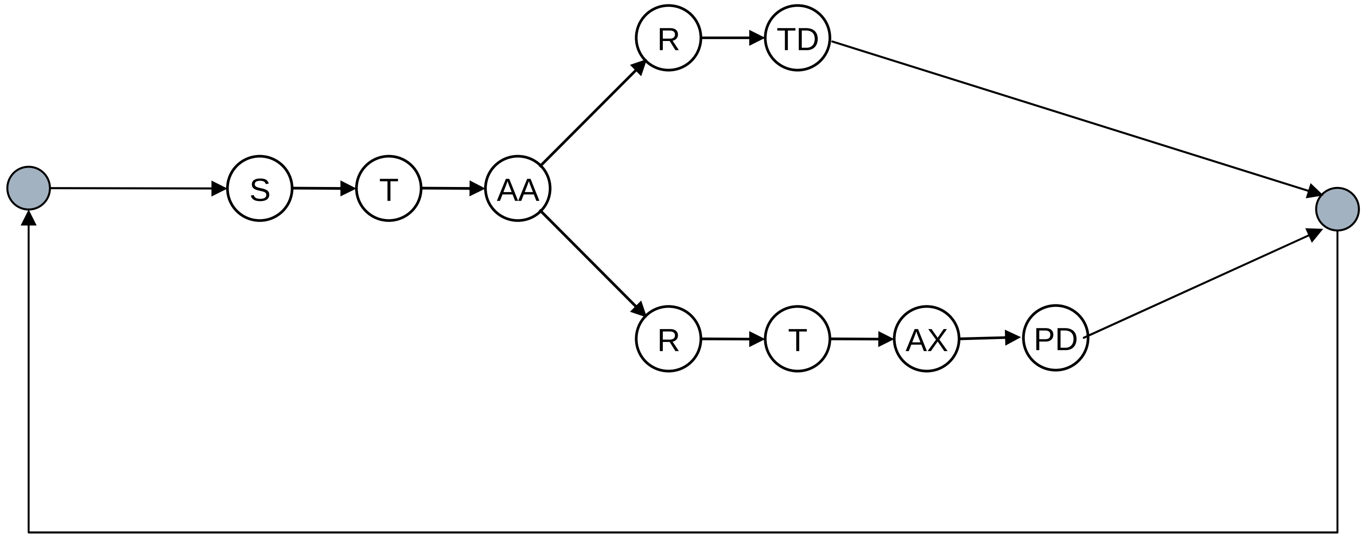
- We need as many lextrees at the trigram level as the square of the number of words



Lextrees

- The “ideal” lextree structure is MUCH larger than an ideal “flat” structure
- As in the case of flat structures, the size of the ideal structure can be greatly reduced by accounting for the fact that most Ngram probabilities are obtained by backing off
- Even so the structure can get very large.
- Approximate structures are needed.

Approximate Lextree Decoding



- Use a unigram Lextree structure
- Use the BP table of the paths entering the lextree to identify the two-word history
- Apply the corresponding trigram probability where the word is identity is known

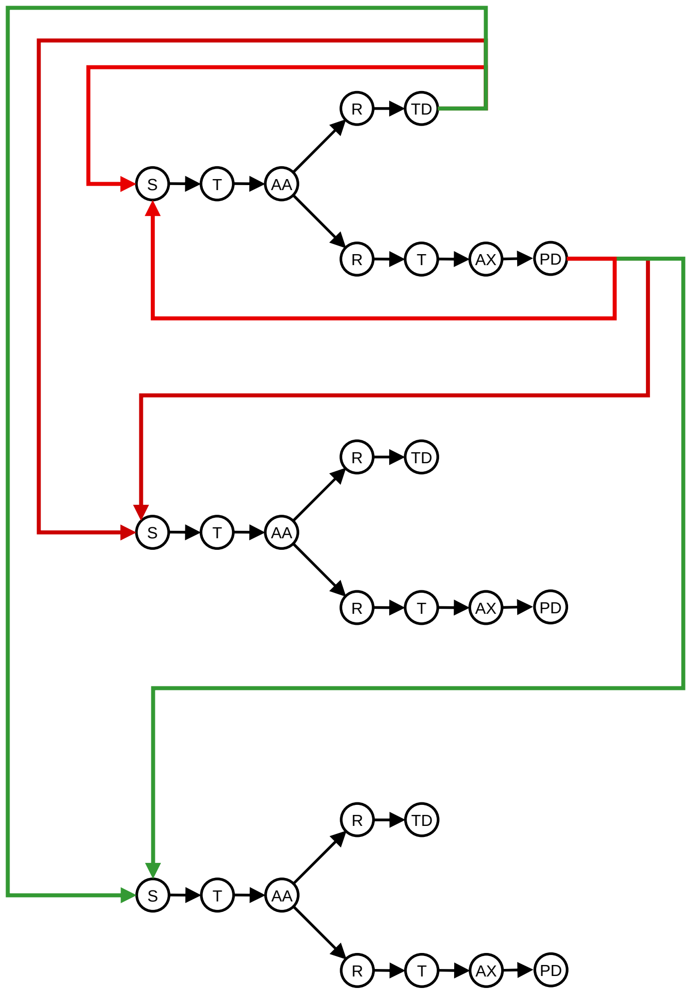
- **This is the approach taken by Sphinx 2 and Pocketsphinx**

Approximate Lextree Decoding

- Approximation is far worse than the pseudo-trigram approximation
  - The basic graph is a unigram graph
    - Pseudo-trigram uses a bigram graph!
- Far more efficient than any structure seen so far
  - Used for real-time large vocabulary recognition in '95!
- How do we retain the efficiency, and yet improve accuracy?
- Ans: Use *multiple* lextrees
  - Still a small number, e.g. 3.

Static 3-Lextree Decoding

- Multiple lextrees
- Lextrees differ in the times in which they may be entered
  - E.g. lextree 1 can be entered if (t%3 ==0), lextree 2 if (t%3==1) and lextree3 if (t%3==2).
- Trigram probability for any word uses the best bigram history for entire lextree (history obtained from backpointer table)
- **This is the strategy used by Sphinx3 in the "fwdtree" mode**
- Better than a single lextree, but still not even as accurate as a pseudo-trigram flat search



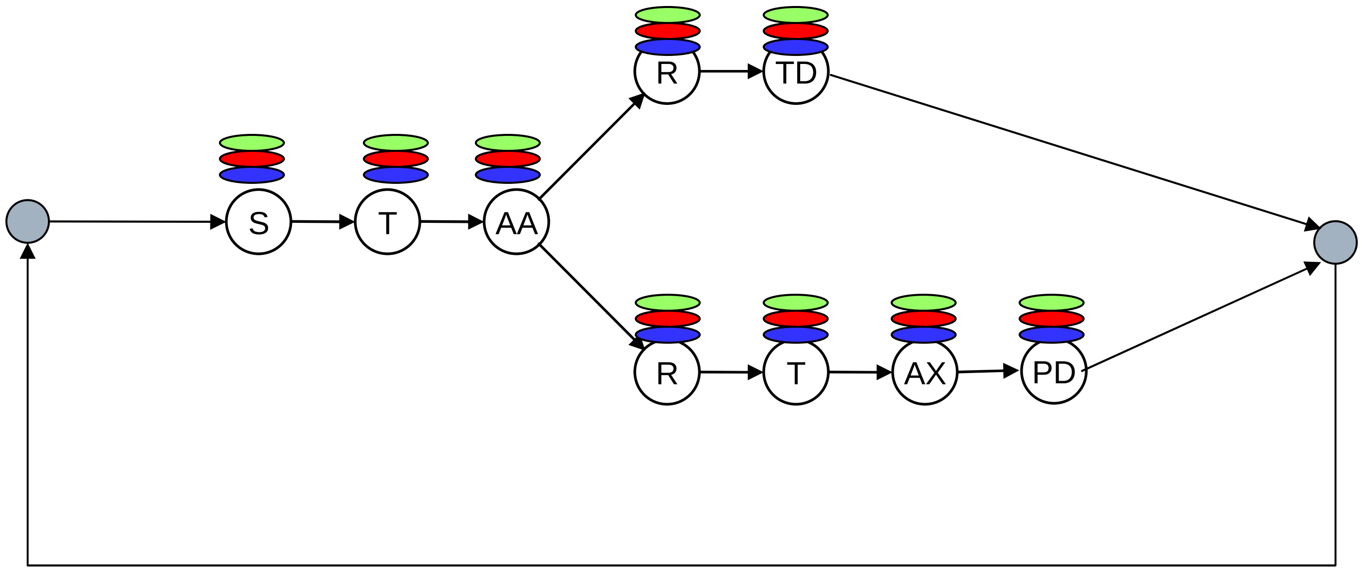
Dynamic Tree Composition

- Build a "theoretically" correct N-gram lextree
- However, only build the portions of the lextree that are requried
- Prune heavily to eliminate unpromising portions of the graphs
  - To reduce composition and freeing

- In practice, explicit composition of the lextree dynamically can be very expensive
  - Since portions of the large graph are being continuously constructed and abandoned

- Need a way to do this *virtually* -- get the same effect without actually constructing the tree

The Token Stack



- Maintain a single lextree structure
- However multiple paths can exist at any HMM state
  - This is not simple Viterbi decoding anymore
- Paths are represented by "tokens" that carry *only* the relevant information required to obtain Ngram probabilities
  - Very light
- Each state now bears a *stack* of tokens

Token Stack

- The token stack emulates full lextree graphs
- Efficiency is obtained by restricting the number of active tokens at any state
  - If we allow N tokens max at any state, we effectively only need the physical resources equivalent to N lextrees
  - But the tokens themselves represent components of many different N-gram level lextrees
- Most optimal of all described approaches
- **Sphinx4 takes this approach**
- Problems: Improper management of token stacks can lead to large portions of the graph representing different variants of the same word sequence hypothesis
  - No net benefit over multiple (N) fixed lextrees

Which to choose

- Depends on the task and your patience
- Options
  - Pocket sphinx/ sphinx2 : Single lextree
    - Very fast
    - Little tuning
  - Sphinx3 fwdflat: Bigram graph with backpointer histories
    - Slow
    - Somewhat suboptimal
    - Little tuning
  - Sphinx3 fwdtree: Multiple lextrees with backpointer histories
    - Fast
    - Suboptimal
    - Needs tuning
  - Sphinx4: Token-stack lextree
    - Speed > fwdflat, Speed < fwdtree
    - Potentially optimal
    - But only if very carefully tuned

Language weights

- The Bayesian classification equation for speech recognition is

Speech recognition system solves

$$word_1, word_2, \ldots, word_N =$$
$$\arg\max_{wd_1, wd_2, \ldots, wd_N} \{ P(signal | wd_1, wd_2, \ldots, wd_N) P(wd_1, wd_2, \ldots, wd_N) \}$$

Acoustic model

For HMM-based systems this is an HMM

Lanugage model

Language weights

- The standard Bayesian classification equation attempts to recognize speech for best *average sentence recognition error*
  - *NOT* word recognition error
  - Its defined over sentences

- But hidden in it is an assumption:
  - The infinity of possible word sequences is the same size as the infinity of possible acoustic realizations of them
  - They are not
  - The two probabilities are not comparable – the acoustic evidence will overwhelm the language evidence

- Compensating for it: The language weight
  - To compensate for it, we apply a *language* weight to the language probabilities
    - Raise them to a power
    - This increases the relative differences in the probabilities of words

Language weights

- The Bayesian classification equation for speech recognition is modified to

$$word_1, word_2, \ldots, word_N = \arg\max_{wd_1, wd_2, \ldots, wd_N} \{P(signal|wd_1, wd_2, \ldots, wd_N)P(wd_1, wd_2, \ldots, wd_N)^{lwt}\}$$

- Which is equivalent to

$$\arg\max_{wd_1, wd_2, \ldots,} \{\log(P(signal \mid wd_1, wd_2, \ldots)) + lwt * \log(P(wd_1, wd_2, \ldots))\}$$

- *Lwt* is the language weight

Language Weights

- They can be incrementally applied

$$\arg\max_{wd_1, wd_2, ...,} \{\log(P(signal \mid wd_1, wd_2, ...)) + lwt * \log(P(wd_1, wd_2, ...))\}$$

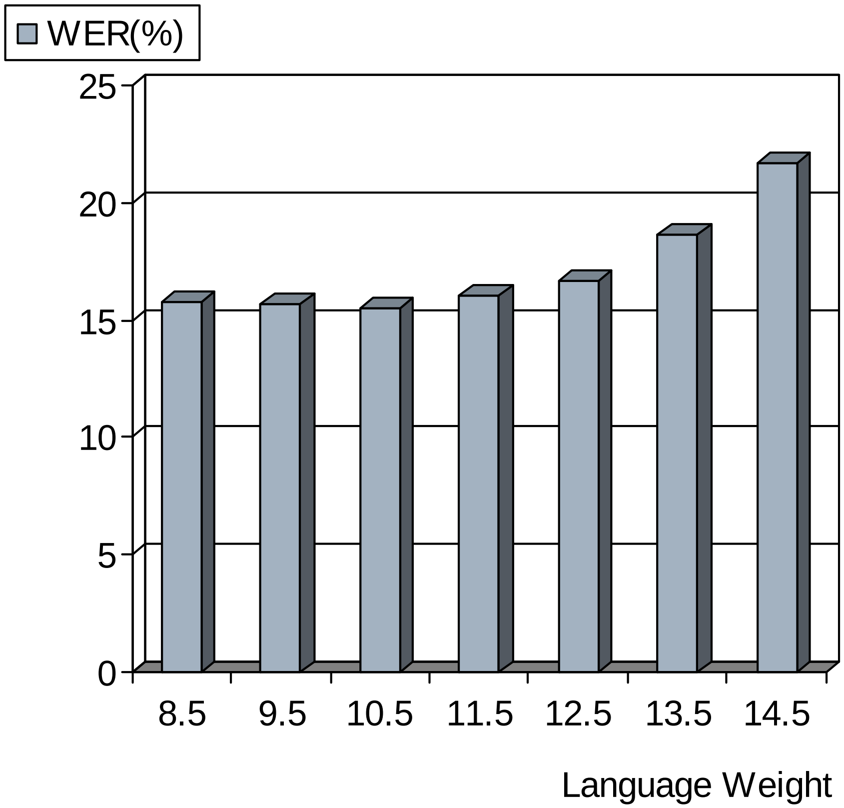
- Which is the same as

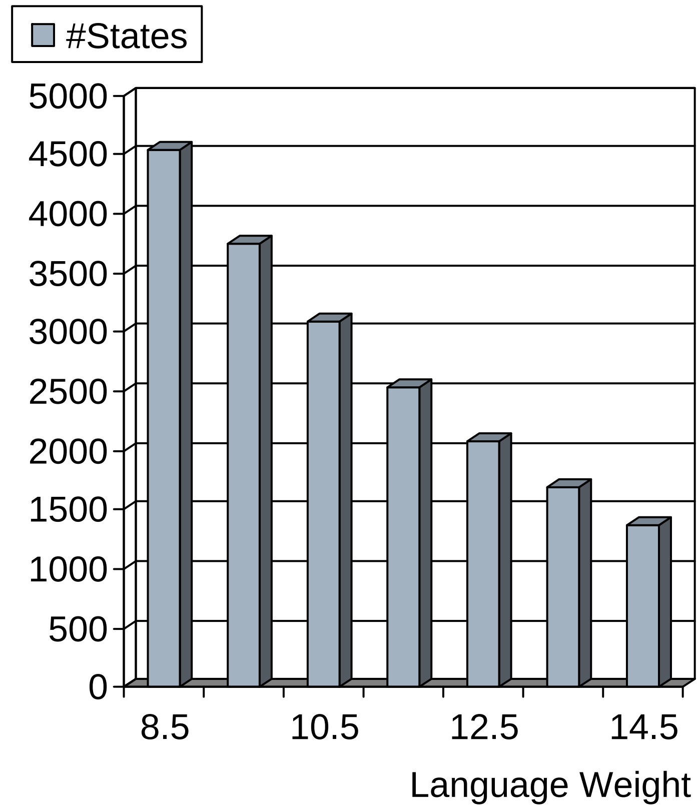
$$\arg\max_{wd_1, wd_2, ...,} \{\log P(signal \mid wd_1, wd_2, ...) + lwt * \log P(wd_1)\}$$
$$lwt * \log P(wd_2 \mid wd_1) + lwt * \log(P(wd_3 \mid wd_1, wd_2)...\}$$

- The language weight is applied to each N-gram probability that gets factored in!

Optimizing Language Weight: Example

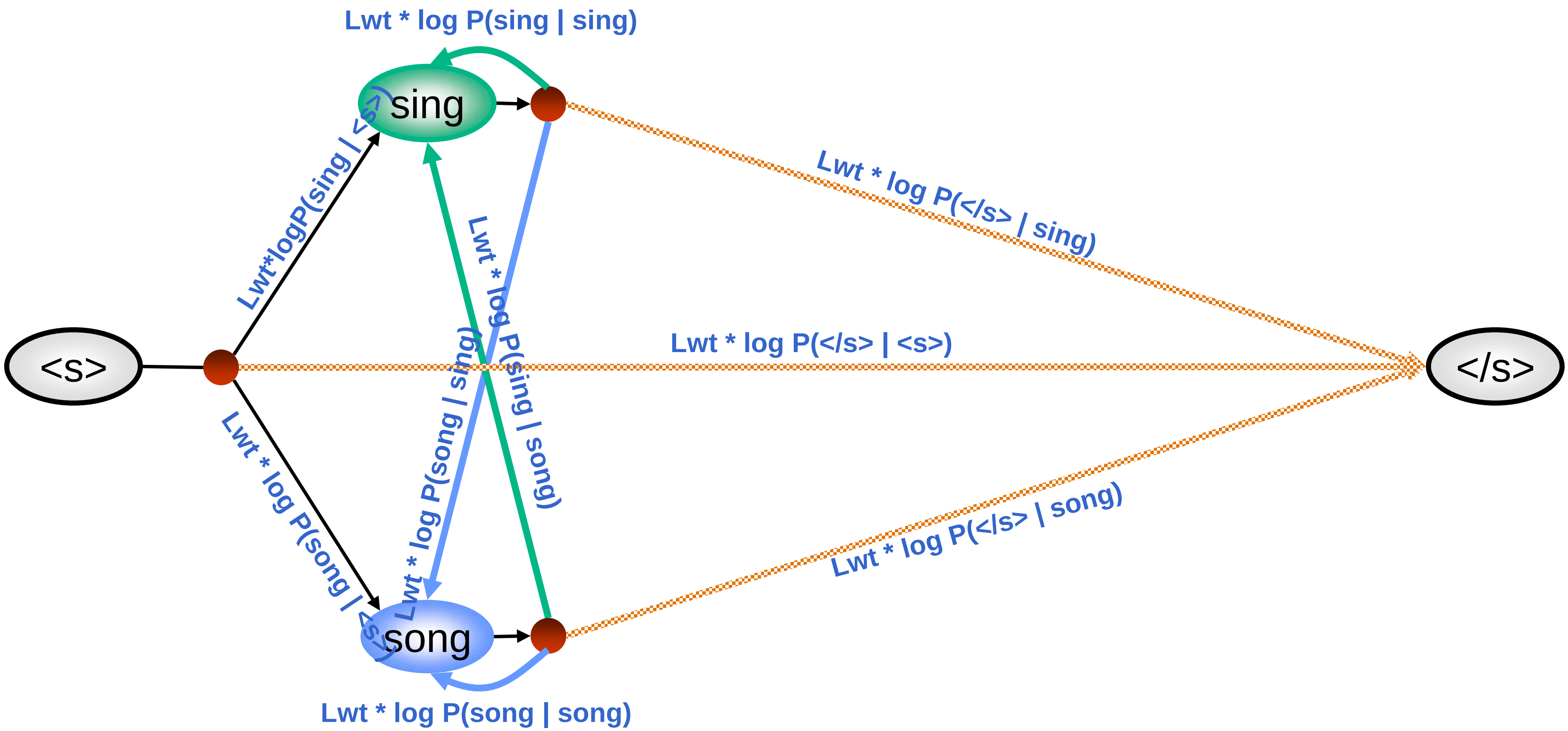
- No. of active states, and word error rate variation with language weight (20k word task)





- Relaxing pruning improves WER at LW=14.5 to 14.8%

The corresponding bigram graph



- The language weight simply gets applied to every edge in the language graph
  - Any language graph!

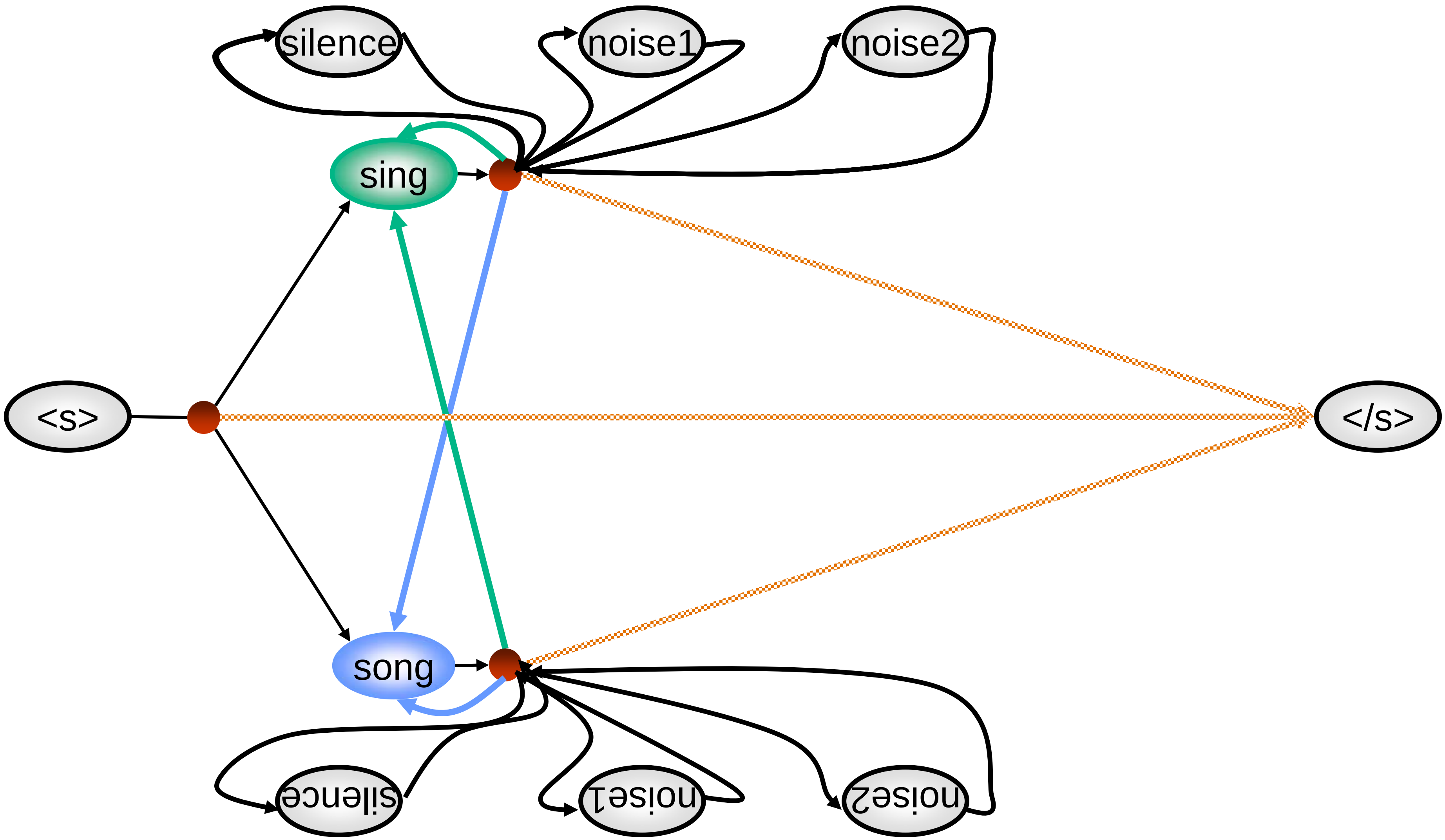
Language Weights

- Language weights are strange beasts
  - Increasing them *decreases* the a priori probability of any word sequences
  - But it *increases* the relative differences between the probabilities of word sequences

- The effect of language weights is not understood
  - Some claim increasing the language weight increases the contribution of the LM to recognition
    - This would be true if *only* the second point above were true

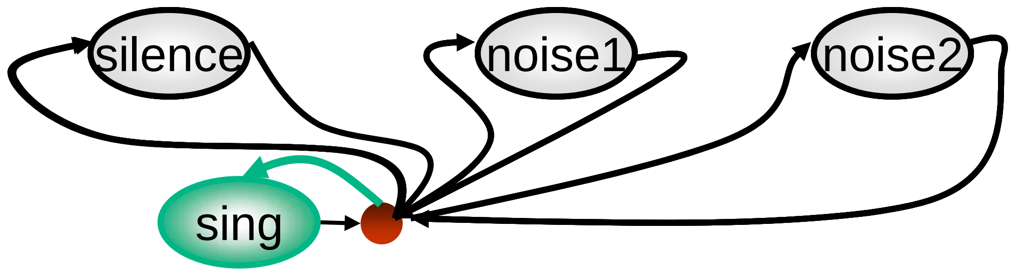
- How to set them
  - Try a bunch of different settings
  - Whatever works!
  - The optimal setting is recognizer dependent

Silences, Noises



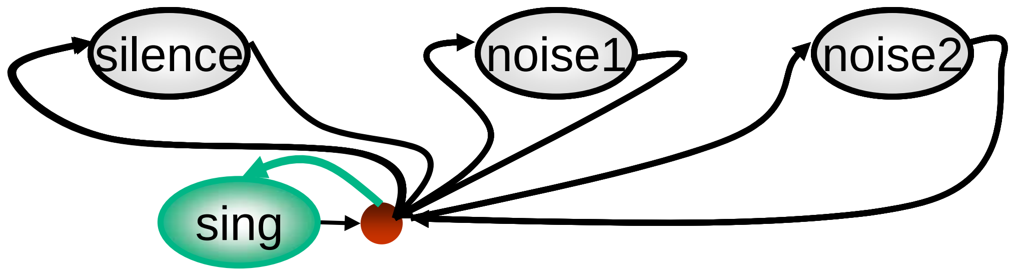
- How silences and noises are handled

Silences and Noises



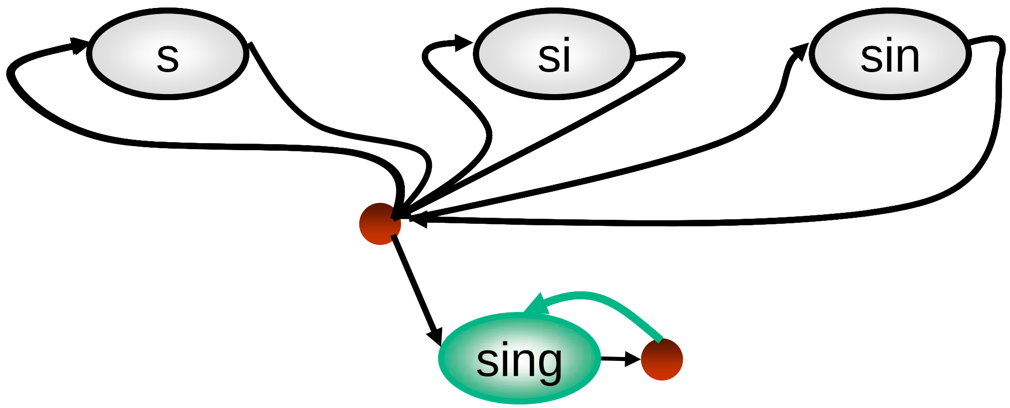
- Silences are given a special probability
  - Called silence penalty
  - Determines the probability that the speaker pauses between words
- Noises are given a “noise” probability
  - The probability of the noise occurring between words
  - Each noise may have a different probability

Silences and Noises



- Silences are given a special probability
  - Called silence penalty
  - Determines the probability that the speaker pauses between words
- Noises are given a “noise” probability
  - The probability of the noise occurring between words
  - Each noise may have a different probability

Stuttering



- Add loopy variants of the word before each word
  - Computationally very expensive
  - But used for reading tutors etc. when the number of possibilities is very small

Rescoring and N-best Hypotheses

- The tree of words in the backpointer table is often collapsed to a graph called a lattice
- The lattice is a much smaller graph than the original language graph
  - Not loopy for one

- Common technique:
  - Compute a lattice using a small, crude language model
  - Modify lattice so that the edges on the graph have probabilities derived from a high-accuracy LM
  - Decode using this new graph
  - Called Rescoring

- An algorithm called A-STAR can be used to derive the *N* best paths through the graph

Confidence

- Skipping this for now