OpenFst: An Open-Source, Weighted Finite-State Transducer Library and its Applications to Speech and Language

Part III. *Applications*

FST Applications

Weighted finite-state transducers have been used in speech recognition, optical character recognition, machine translation, text-to-speech synthesis, information extraction, data mining, and fraud detection.

Current OpenFst applications:

- Speech recognition (speech-to-text):
  - Search graphs, phone and word lattices
  - Segmentations, context-dependency representation
  - Lexicons and language models
  - 'Context-free' grammars (with semantic output)
  - Text denormalization
- Speech synthesis (text-to-speech):
  - Tokenization and text normalization
  - Pronunciation models
- Optical character recognition – lexicons, language models, search graphs
- Machine translation – translation models, language models, decoding
- Information extraction – large-scale regular-expression matching

Why (Weighted) Finite-State Transducers?

- Finite-state acceptors and transducers can efficiently represent certain (the *regular* or *rational*) sets and binary relations over string.
- Weights can represent probabilities, costs, etc associated with alternative, uncertain data.
- Optimization operations (determinization, minimization) can be used to minimize redundancy and size.
- Composition can be used to efficiently recognize inputs and cascade transductions.
- OpenFst has been used with FSTs of many millions of states and arcs (there is even a distributed FST representation in development).

Common parts of FST Applications

- Models
  - Constructed
    - Regular expressions
    - Rewrite rules
    - Context-dependency transducer
    - Recognition grammars
  - Learned
    - n-gram language models
    - Pair n-gram language models
    - Weighted edit distance
- Cascades and Search
  - Composition and intersection
  - ShortestPath and ShortestDistance
  - FST optimization

Keyword Detection

- C identifiers: $\{char, const, continue, if, int, else, short, signed, sizeof\}$
- Linear search:

```
if (token == "char") return 1;
if (token == "const") return 1;
if (token == "oontinue") return 1;
if (token == "if") return 1;
if (token == "int") return 1;
if (token == "else") return 1;
if (token == "short") return 1;
if (token == "signed") return 1;
if (token == "sizeof") return 1;
else return 0;
```

Keyword Detection – Binary Search

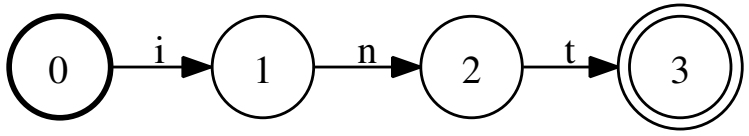
```
vector<string> keywrds;


keywrds.push_back("char");
keywrds.push_back("continue");
keywrds.push_back("else");
keywrds.push_back("if");
keywrds.push_back("int");
keywrds.push_back("short");
keywrds.push_back("signed");
keywrds.push_back("sizeof");


return binary_search(keywrds.begin(), keywrds.end(), token);
```

Keyword Detection – Automata Search

- Search by intersecting token string with automaton.
- Token string must be represented as an FST:



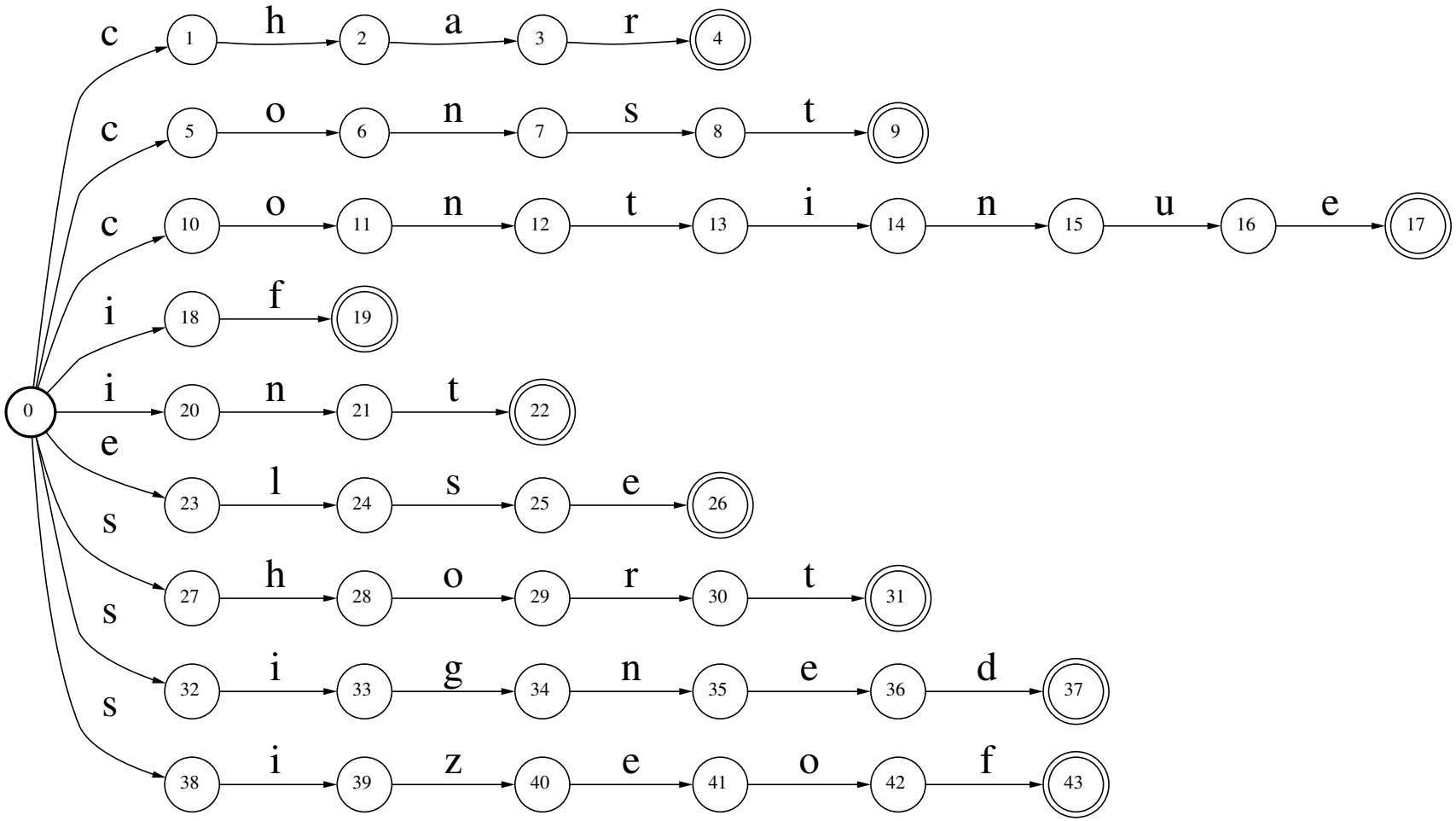
- `Intersect(token, keywords, &result);`

String FSTs

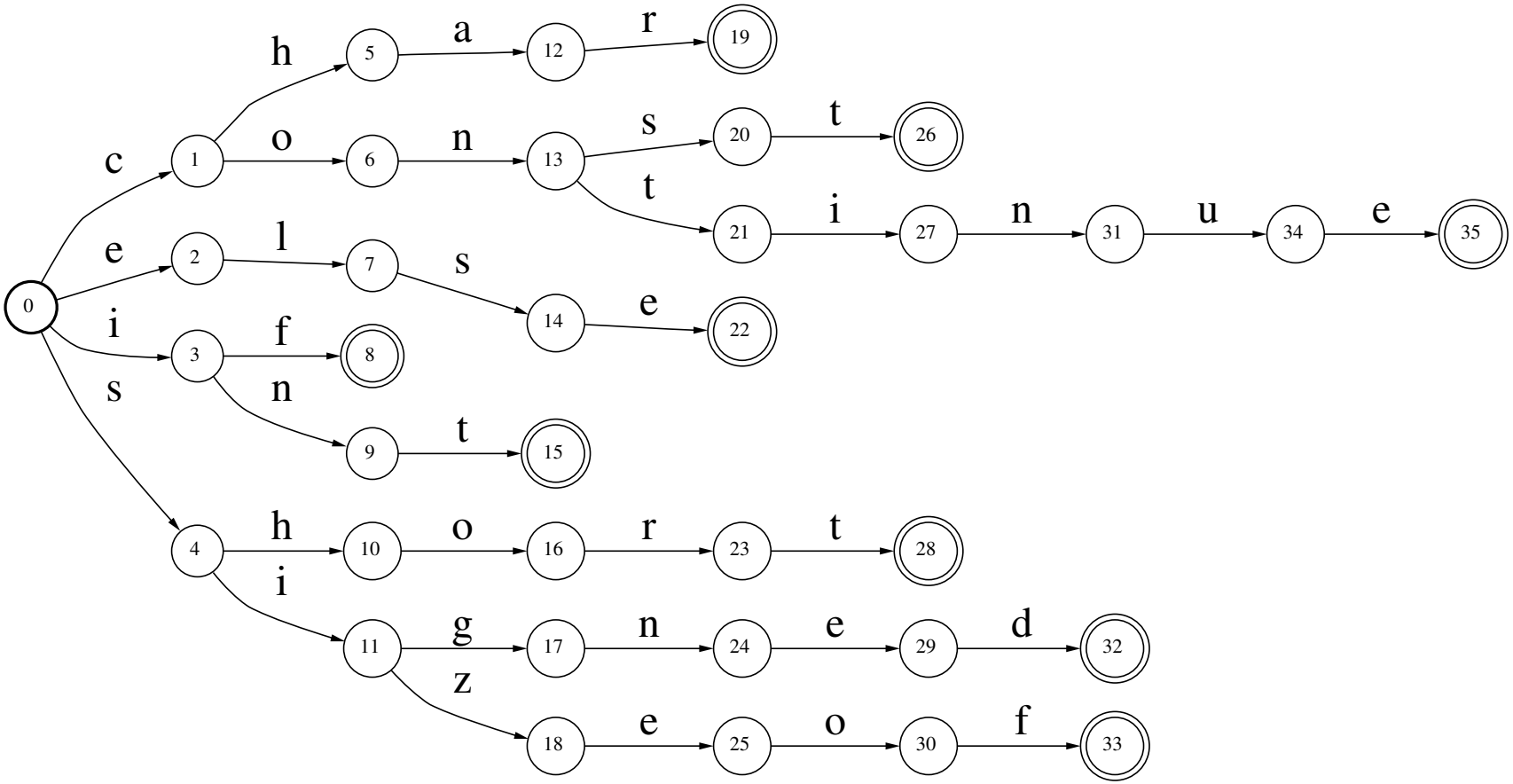
```
#include <fst/fstlib.h>
using fst::SymbolTable;
using fst::StdArc;
using fst::StdVectorFst;

bool StringFst(const vector<string> &symbols,
               const SymbolTable *symtab,
               StdVectorFst *fst) {
  StdVectorFst::StateId q = fst->AddState();
  fst->SetStart(q);
  for (int i = 0; i < symbols.size(); ++i) {
    StdArc::Label lbl = symtab->Find(symbols[i]);
    if (fst::kNoLabel == lbl) return false;
    StdVectorFst::StateId r = fst->AddState();
    fst->AddArc(q, StdArc(lbl, lbl, StdVectorFst::Weight::One(), r));
    q = r;
  }
  fst->SetFinal(q, StdVectorFst::Weight::One());
  return true;
}
```

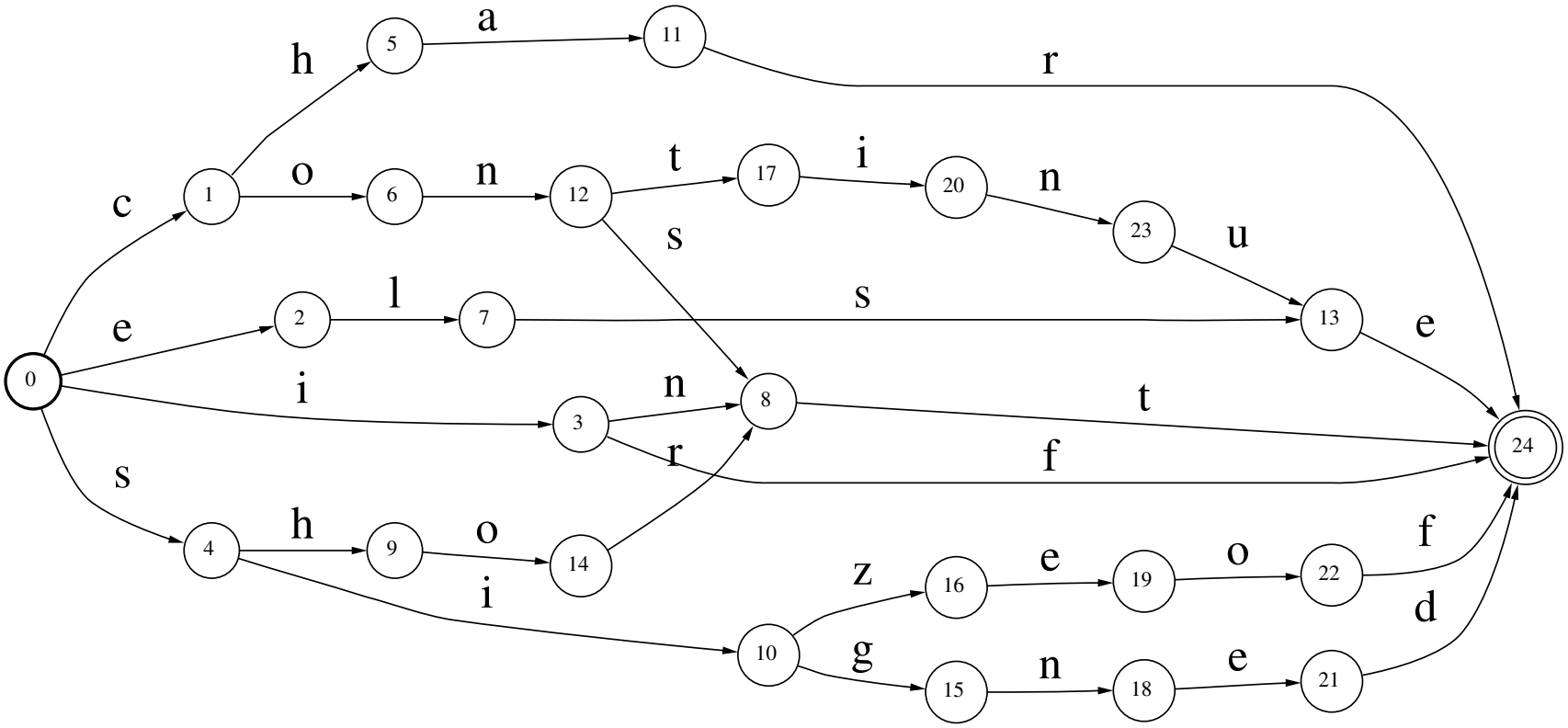
Keyword Detection – Automata Search (continued)



Keyword Detection – Deterministic Search

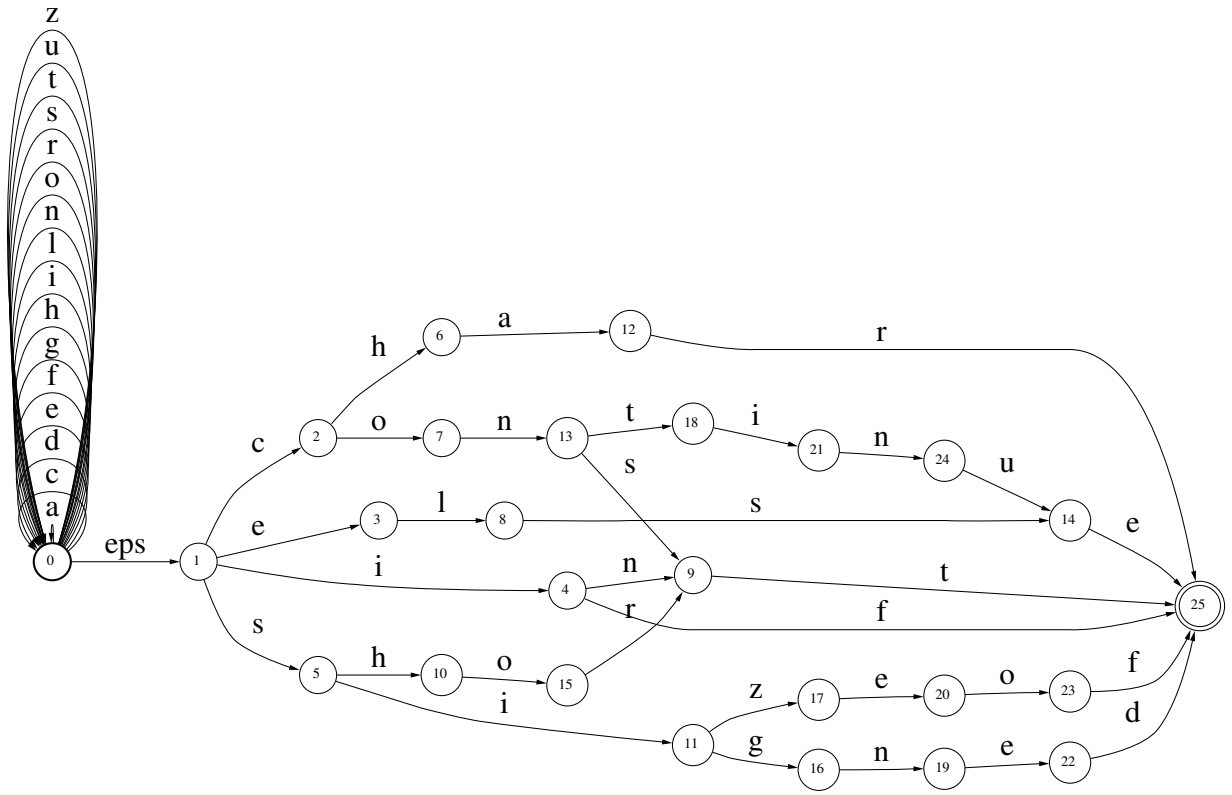


Keyword Detection – Minimal Deterministic Search



Pattern Matching – Dictionaries

- Equation: $M = \Sigma^* D$
- Graphical Representation:



- Determinize using *failure transitions*; search by intersecting token string with automaton.

Pattern Matching – Regular Expressions

- Single Regular Expressions:
  - Compile each regular expression into an automaton. Proceed as above.
- Multiple Regular Expressions:
  - Matching regular patterns $P_1 \ldots P_N$ in string s: if N and $|s|$ are large, then matching every pattern separately is very inefficient
  - Solution: create acceptor for the language $P = \Sigma^*(P_1 \cup ... \cup P_N)$. Insert markers indicating the start and the end of a match. Search by intersecting P with the input text.
- Optimizations/improvements
  - Failure transitions, Aho-Corasick algorithm and its generalizations.
  - Support for weights and outputs.

Pattern Matching – Context-Free Rules

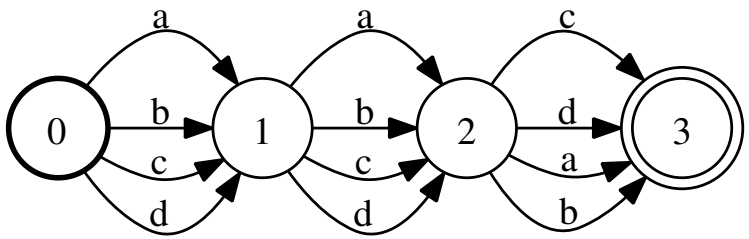
- Context-free rules (with restrictions). Rules may have weights and outputs - supports semantic annotation using JavaScript.
  W3C Standard: http://www.w3.org/TR/speech-grammar.
- Each rule compiled into an FST over the alphabet of terminal and non-terminal symbols.
- Rules are combined into a top-level lazy FST where non-terminals are dynamically replaced by their corresponding rule FSTs.
- Parse using FST composition and shortest-path algorithms.

Hangman Game

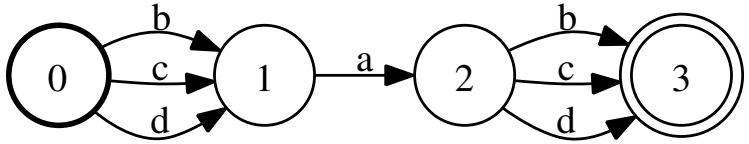
- Guess an orthographic word letter-by-letter, knowing only its length.
- Player 1 guesses one letter at a time.
- If the word contains that letter, Player 2 reveals all occurrences of it.
- Otherwise Player 2 gains one point.
- The game ends when Player 2 reaches a certain number of points or when Plaer 1 has uncovered the word.
- Let's help Player 1, using FSTs.
- Need one FST to represent the state of the game board.

Hangman – Board

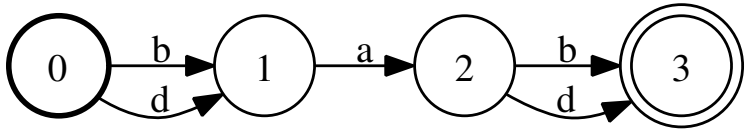
- Initially a sausage:



- After Player 1 correctly guesses an 'a':



- After Player 1 incorrectly guesses a 'c':



- What should Player 1's next guess be?

Hangman – Guessing

- Intersect the game board FST with a weighted dictionary (log semiring).
- Compute the expected number of occurrences of each letter.
- Guess the letter with the highest expected occurrence count.

Computing Expected Counts – Forward/Backward

```
VectorFst<LogWeight> lattice;
Intersect(board, dictionary, &lattice);
Push(&lattice, REWEIGHT_TO_INITIAL);      // Normalize probabilities

vector<LogWeight> alpha, beta;
ShortestDistance(lattice, &alpha);        // ``Forward''
ShortestDistance(lattice, &beta, true);   // ``Backward''

map<Label, LogWeight> letter_counts;
// Not shown: Initialize values in letter_counts to LogWeight::Zero()
for (StateIterator<...> siter(lattice); !siter.Done(); siter.Next()) {
  const int q = siter.Value();
  for (ArcIterator<...> aiter(lattice, q); !aiter.Done(); aiter.Next()) {
    const LogArc &arc = aiter.Value();
    LogWeight gamma = Times(Times(alpha[q], arc.weight), beta[arc.nextstate]);
    letter_counts[arc.ilabel] = Plus(letter_counts[arc.ilabel], gamma);
  }
}
```

Computing Expected Counts – Expectation Semiring

```
typedef ExpectationWeight<LogWeight,
                          PowerWeight<LogWeight, 26> > > LetterWeight;
struct LogToLetterMapper() {
  public Arc<LetterWeight> operator()(const Arc<LogWeigh> &arc) const {
    PowerWeight<LogWeight, 26> tuple = PowerWeight::Zero();
    // Assumes that 'a' corresponds to label 1 etc.
    tuple.SetValue(arc.ilabel, arc.weight);
    return Arc<LetterWeight>(arc.ilabel, arc.olabel,
                             LetterWeight(arc.weight, tuple), arc.nextstate);
  }
};

VectorFst<LogWeight> lattice;
Intersect(board, dictionary, &lattice);
Push(&lattice, REWEIGHT_TO_INITIAL);

VectorFst<LetterWeight> expct_lattice;
Map(lattice, &expct_lattice, LogToLetterMapper());
LetterWeight expct_letter_counts = ShortestDistance(expct_lattice);
```

Weight Reestimation with EM

Baum–Welch algorithm, an instance of the Expectation/Maximization (EM) meta-algorithm:

- Compute the expected arc counts (E step), using either:
  - Forward-Backward algorithm over Log Semiring; or
  - Shortest distance over Expectation Semiring.
- Normalize the arc counts for each state (M step),
  perhaps with smoothing or regularization.

Classic String Edit Distance

- Edit operations with costs: insertion, deletion, substitution
- A least-cost alignment of two strings is any least-cost sequence of edit operations that transforms one string into the other (argmax).
- The edit distance between two strings is the cost of any least-cost sequence of edit operations that transforms one string into the other (max).
- FST-based solution is as efficient as the classical solution:
  $\text{Shortest}\{\text{Distance}, \text{Path}\}(x \circ M \circ y)$
- FST-based solution is much more flexible, since M can be just about anything.

Stochastic Edit Distance Learning

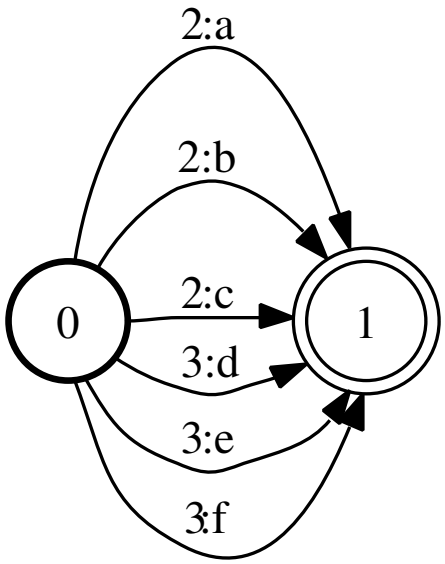
- Often contains a special case where the edit-distance model M is an FST with a single state
- Special case learning algorithm due to Ristad and Yianilos 1998.
- For each pair (x, y) of similar strings, compute the expected arc counts over $x \circ M \circ y$.
- Use this to compute monotonic alignments of e.g. pronunciation dictionaries.

Latent Alignment Models

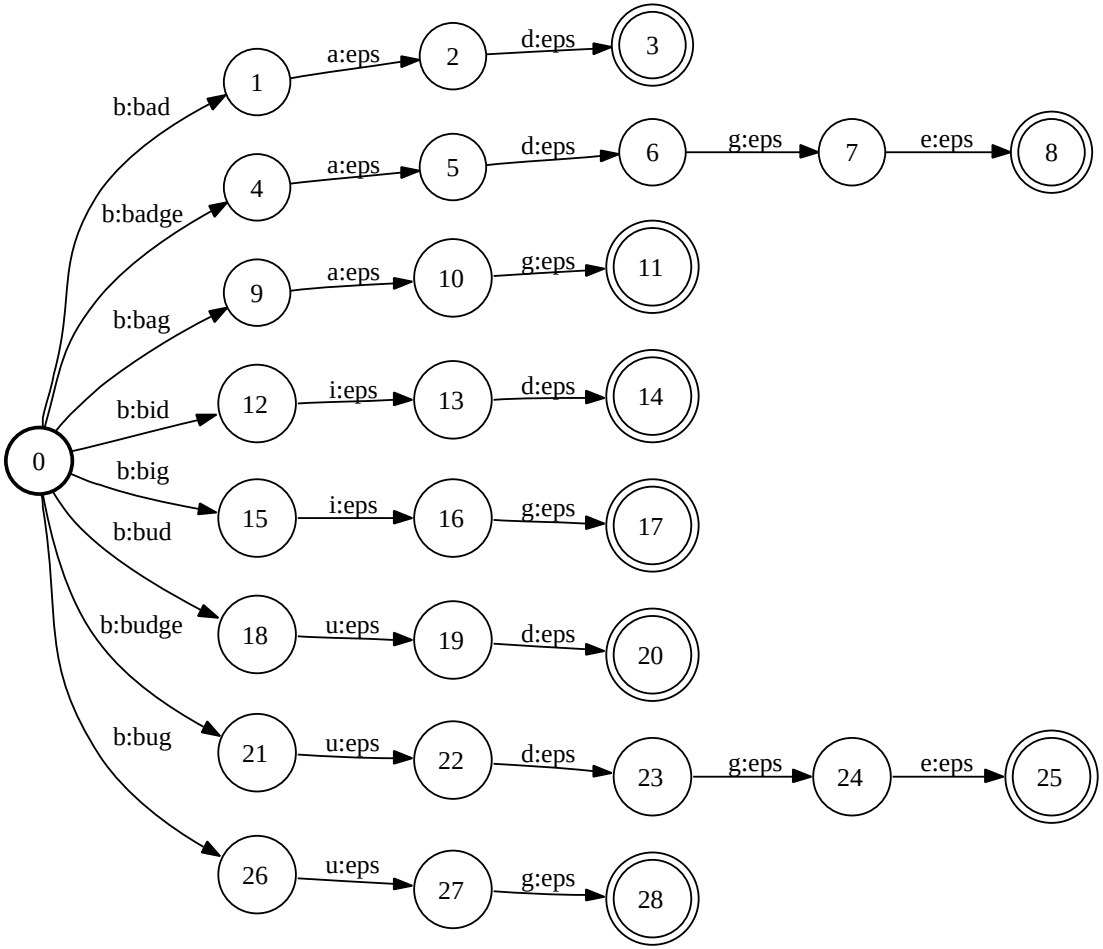
- Many string-to-string transformation problems (translation, transliteration, pronunciation modeling) become simpler when expressed with latent alignments.
- Often simple stochastic alignment models are enough to align a given pair of strings:
  - Train a simple stochastic edit distance model.
  - Use the simple model to impute latent alignments for the input data.
  - Train more complex models on the aligned data.
  - Pair n-gram language models.
- One problem: Pair models are join probability models $\Pr(x, y)$. When cascading multiple models like this, we generally need conditional models like $\Pr(x \mid y)$. Ordinary language modeling tools were not designed with transducers in mind and do not produce conditional models.

Telephone Keypad Input Method

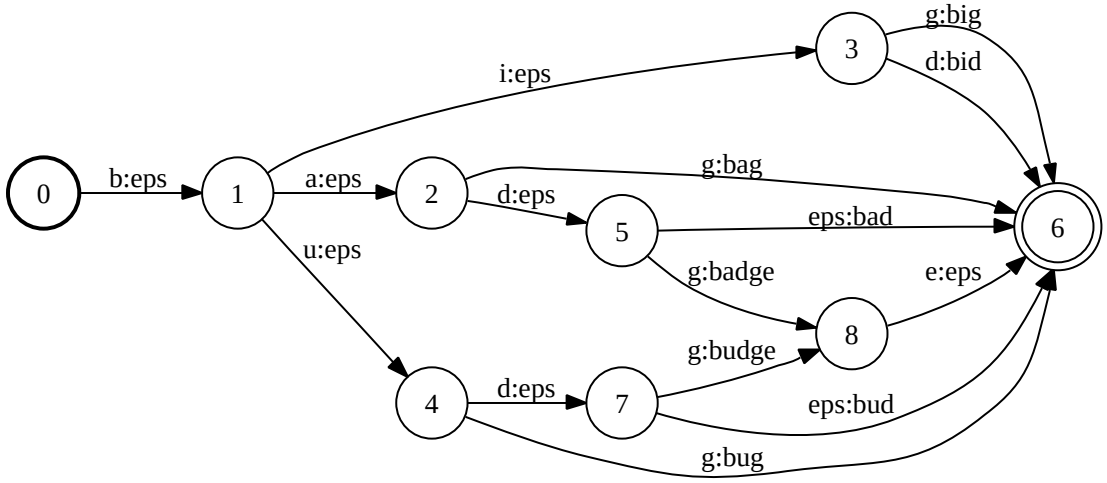
- Example: T9 transducer (T):



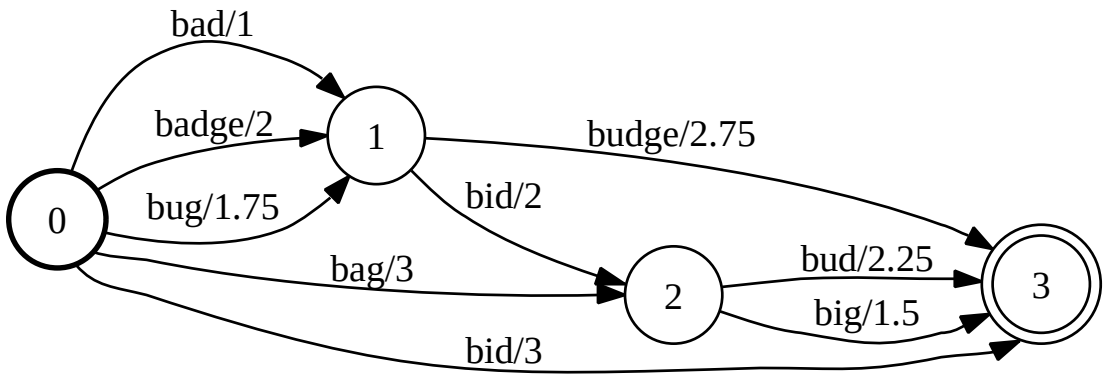
- Dictionary transducer:



- Minimal deterministic dictionary transducer (L):



- Word grammar (G):



- T9 recognition:
  - Construct: $M = T^* \circ L^* \circ G$
  - Compute for input s: ShortestPath$(s \circ M)$.

ASR Problem Definition

Given an utterance, find its most likely written transcription.

Fundamental ideas:

- Utterances are built from sequences of units
- Acoustic correlates of a unit are affected by surrounding units
- Units combine into higher level units — phones → syllables → words
- Relationships between levels can be modeled by weighted graphs
- Recognition: find the best path in a suitable product graph

Maximum-Likelihood Decoding

Overall analysis:

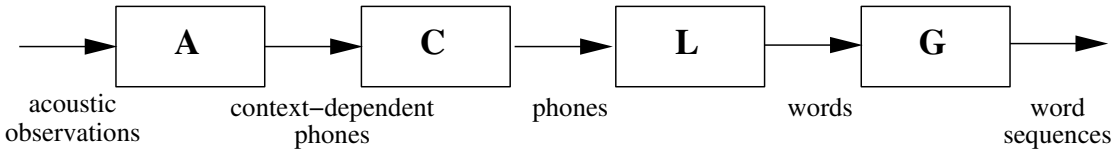
- Acoustic observations: parameter vectors derived by local spectral analysis of the speech waveform at regular (e.g. 10msec) intervals
- Observation sequence $\mathbf{o}$
- Transcriptions $\mathbf{w}$
- Probability $P(\mathbf{o}|\mathbf{w})$ of observing $\mathbf{o}$ when $\mathbf{w}$ is uttered
- Maximum-likelihood decoding:

$$\begin{aligned} \hat{\mathbf{w}} &= \operatorname*{argmax}_{\mathbf{w}} P(\mathbf{w}|\mathbf{o}) = \operatorname*{argmax}_{\mathbf{w}} \frac{P(\mathbf{o}|\mathbf{w})P(\mathbf{w})}{P(\mathbf{o})} \\ &= \operatorname*{argmax}_{\mathbf{w}} \underbrace{P(\mathbf{o}|\mathbf{w})}_{\substack{\text{channel} \\ \text{model}}} \underbrace{P(\mathbf{w})}_{\substack{\text{language} \\ \text{model}}} \end{aligned}$$

Generative Models of Speech

Typical decomposition of $P(\mathbf{o}|\mathbf{w})$ into conditionally-independent mappings between levels:

- Acoustic model $P(\mathbf{o}|\mathbf{p})$: phone sequences $\rightarrow$ observation sequences. Detailed model:
  - $P(o|d)$: distributions $\rightarrow$ observation vectors — *symbolic* $\rightarrow$ *quantitative*
  - $P(\mathbf{d}|m)$: context-dependent phone models $\rightarrow$ distribution sequences
  - $P(\mathbf{m}|\mathbf{p})$: phone sequences $\rightarrow$ model sequences
- Pronunciation model $P(\mathbf{p}|\mathbf{w})$: word sequences $\rightarrow$ phone sequences
- Language model $P(\mathbf{w})$: word sequences



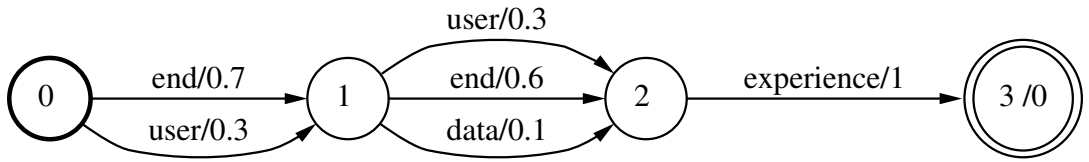
Speech Recognition Problems

- Modeling: how to describe accurately the relations between levels ⇒ *modeling errors*
- Search: how to find the best interpretation of the observations according to the given models ⇒ *search errors*

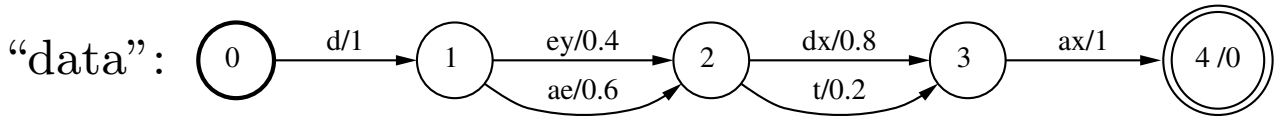
This talk will emphasize the latter topic.

Classical ASR Search I – Network Representation

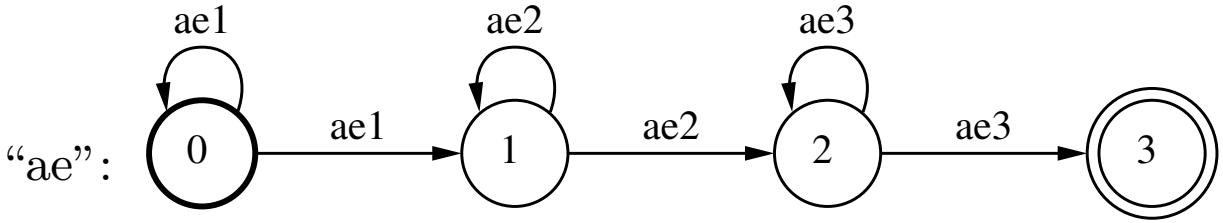
- Grammar: word network



- Lexicon: mapping from word label to phonetic network



- Phone model: mapping from phone label to HMM network



Classical ASR Search II – Network Substitution and Viterbi Search

- Above networks are recursively substituted, either offline or dynamically during recognition, to form a single large network.
- The combined network is time-synchronously (Viterbi) matched to the incoming speech and searched for the best (lowest total cost) matching path which is returned as the hypothesized word string. For improved speed, partial paths that score less than the best path so far (outside the so-called beam) can be pruned, but this potentially creates search errors.

Problems with the Classical ASR Search

The problems with this classical approach for large vocabulary speech recognition include:

1. Context-Dependent Modeling: Context-dependent models are awkward to represent by network substitution. (*Solution:* finite-state transducers and composition.)
2. Network Redundancy and Size: Networks can be highly redundant and very large. (*Solution:* finite-state optimizations – determinization and minimization.)
3. Network Weight Distribution: The distribution of the grammar and pronunciation weights strongly affect pruning efficiency. What is the optimal way to distribute them? (*Solution:* weight pushing)

Context-Dependency Examples

- Context-dependent phone models: Maps from context-independent units to context-dependent ones. Example: $ae/b\_\_d \rightarrow ae_{b,d}$
- Context-dependent allophonic rules: Maps from baseforms to detailed phones. Example: $t/V'\_\_V \rightarrow dx$
- Difficulty: In cross-word contexts – where several words enter and leave a state in the grammar, substitution does not apply.

Weighted Transducers

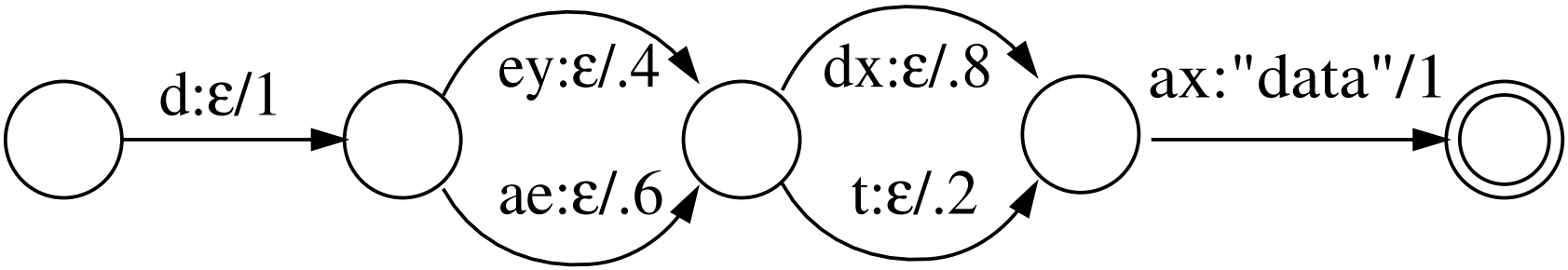
Finite automata with input labels, output labels, and weights.

- Transitions:

$$q \xrightarrow{\mathbf{x}:\mathbf{y}/k} q'$$

  $\mathbf{x} \in \Sigma \cup \{\epsilon\}$, $\mathbf{y} \in \Gamma \cup \{\epsilon\}$, weight k

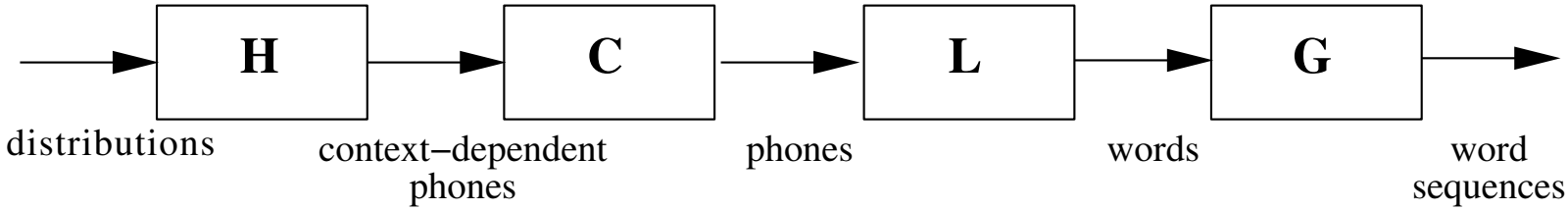
- Example — word pronunciation transducer:



How to Build a Finite-State Language Processor

Operation	purpose	example
Basic element		word pronunciation
Union	alternation	alternative pronunciations
Concatenation	sequencing	compound word
Closure	indefinite repetition	sequence translation
Composition	stage combination	recognition cascade
Determinization	redundancy removal	determinized cascade
Minimization	size reduction	optimized cascade

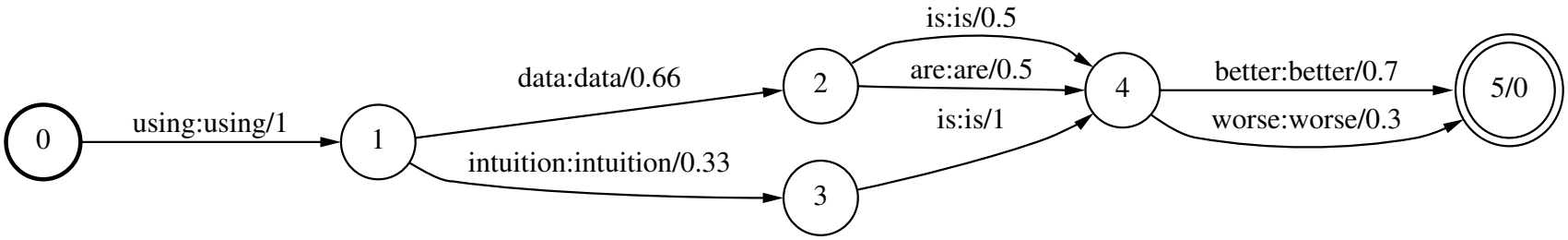
Integrated network



- H: HMM transducer, closure of the union of all HMMs used in acoustic modeling,
- C: context-dependency transducer mapping context-dependent phones to phones,
- L: pronunciation dictionary transducer mapping phonemic transcriptions to word sequences,
- G: language model weighted automaton.

$H \circ C \circ L \circ G$: mapping from sequences of distribution names to word sequences.

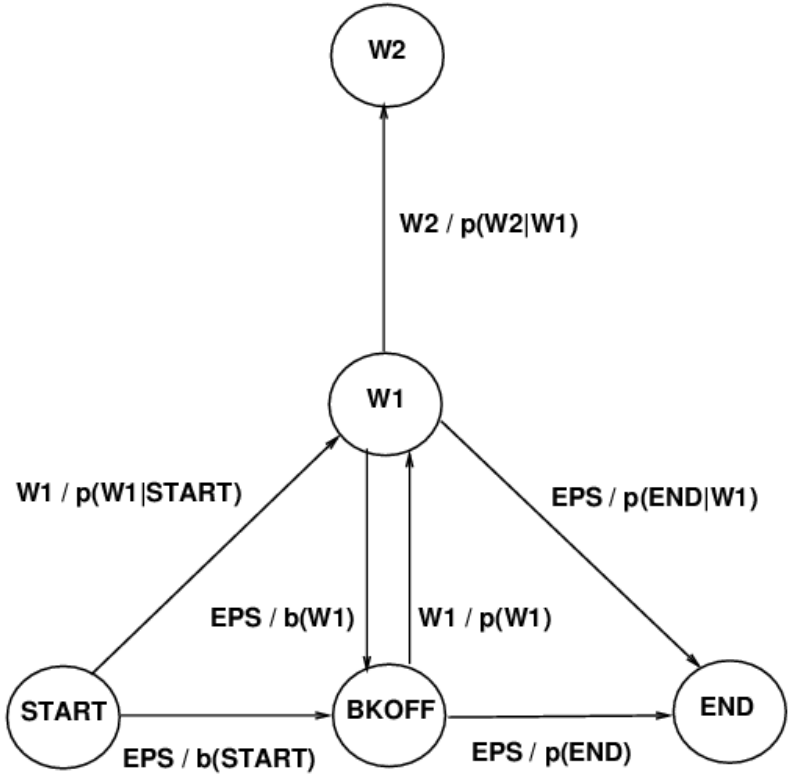
Grammar Acceptor



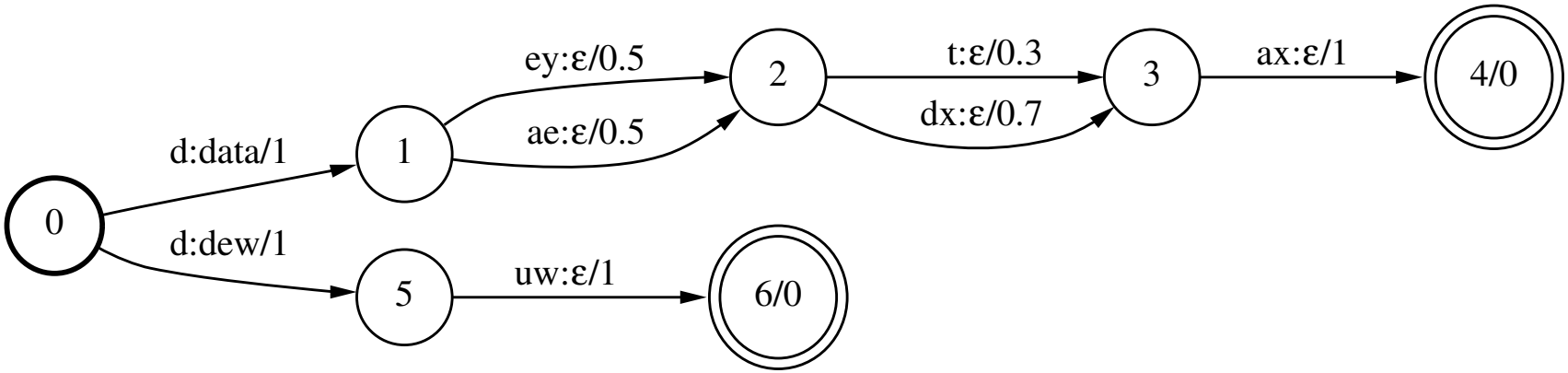
N-Gram Language Models

- Want to estimate the probability of a sentence $w_1, w_2, w_3, w_4, \ldots$
- By the chain rule of probability,
  $\Pr(w_1, w_2, \ldots) = \Pr(w_1) \times \Pr(w_2 \,|\, w_1) \times \Pr(w_3 \,|\, w_1, w_2) \times \Pr(w_4 \,|\, w_1, w_2, w_3) \times \cdots$
- Bigram approximation, plus conventional handling of first word:
  $\Pr(w_1, w_2, \ldots) \approx \Pr(w_1 \,|\, \$) \times \Pr(w_2 \,|\, w_1) \times \Pr(w_3 \,|\, w_2) \times \Pr(w_4 \,|\, w_3) \times \cdots$
- Has a straightforward representation as an FST, where states encode conditioning histories.
- Backoff models can be represented inexactly with epsilon transitions or exactly with failure transitions.

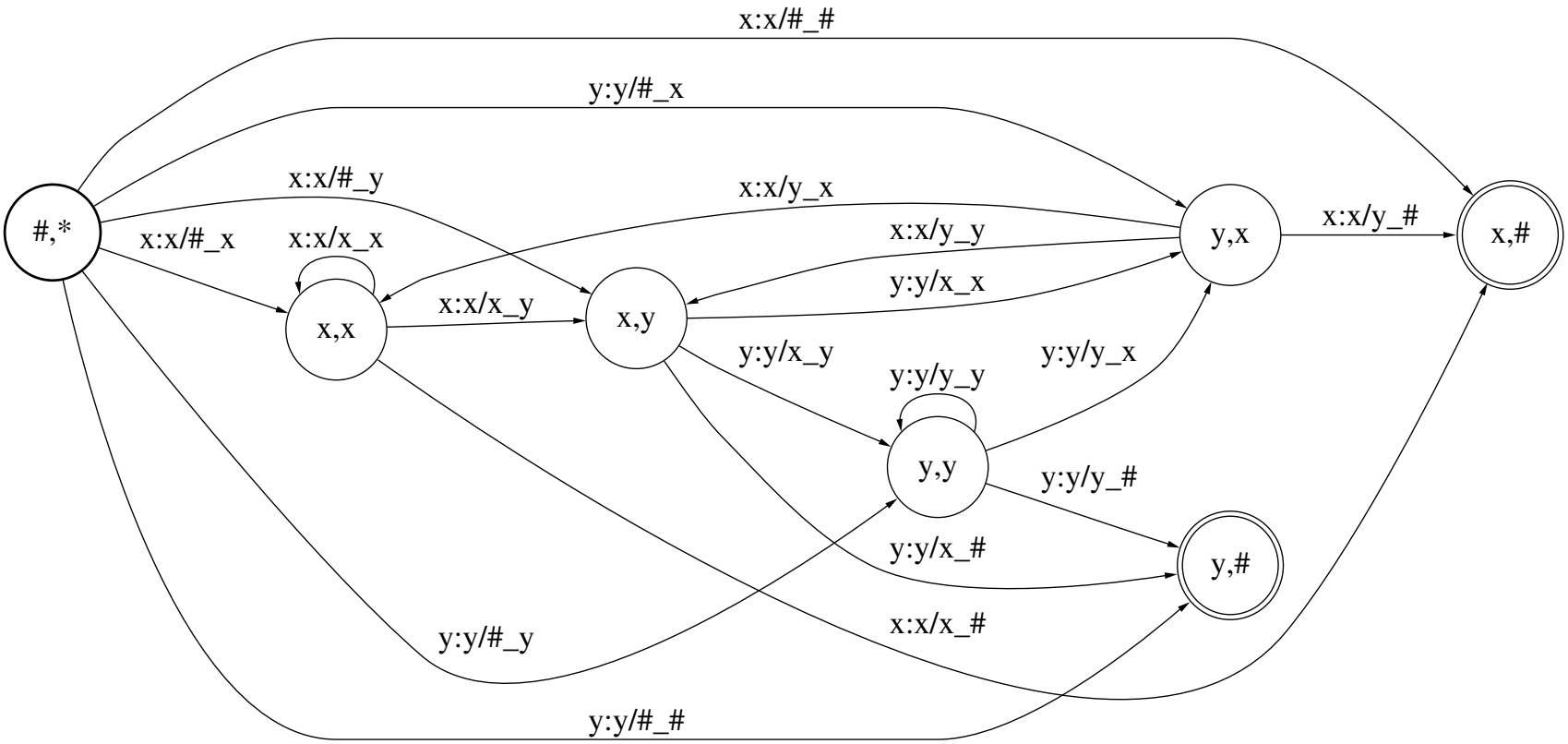
Bigram Grammar



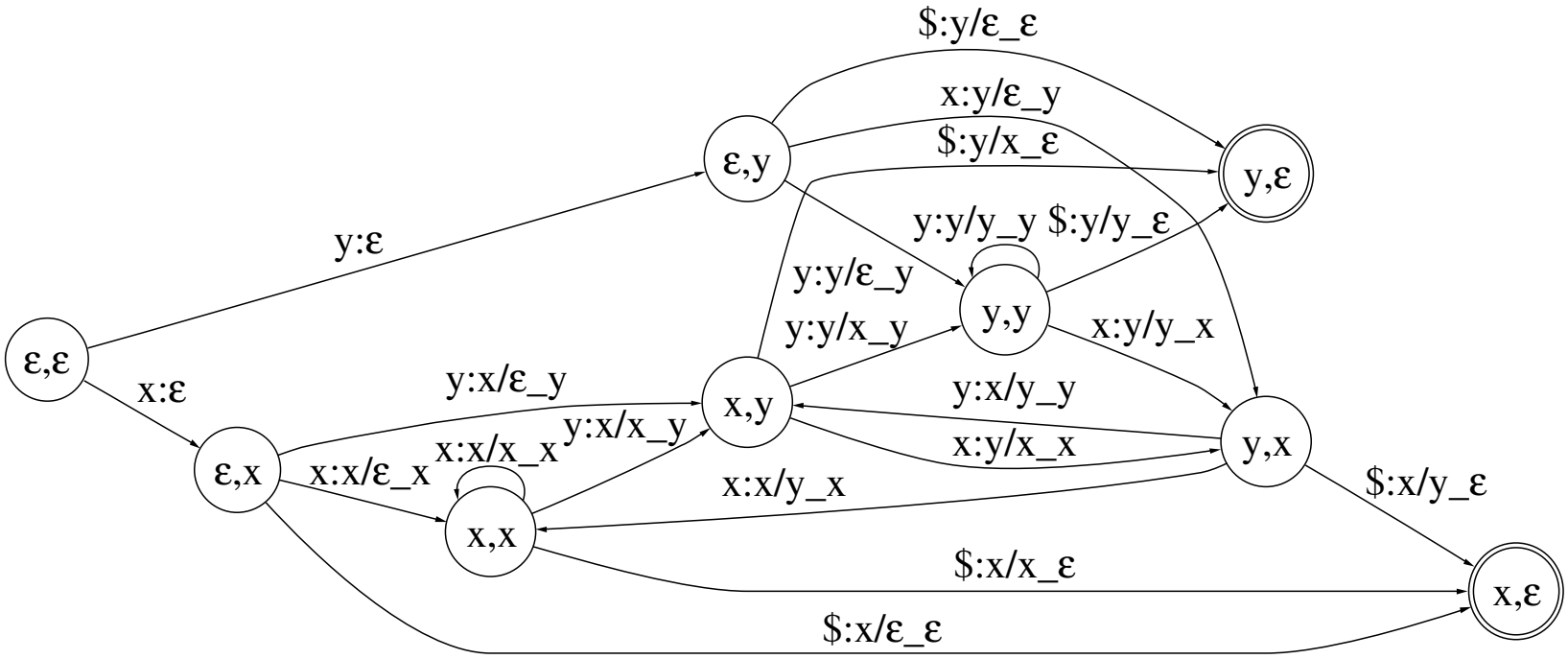
Pronunciation Lexicon Transducer



Context-Dependent Triphone Transducer



Deterministic Context-Dependent Triphone Transducer



Network Construction I: Disambiguation

1. $L \rightarrow \tilde{L}$, auxiliary symbols used to make $L \circ G$ determinizable (homophones, transduction's unbounded delay):

$$\texttt{r eh d } \#_0 \quad \textit{read}$$
$$\texttt{r eh d } \#_1 \quad \textit{red}$$

2. $C \rightarrow \tilde{C}$, self-loops used for further determinizations at the context-dependent level,

3. $H \rightarrow \tilde{H}$, self-loops at initial state, auxiliary context-dependent symbols mapped to new distinct distribution names.

Network Construction II: Combination/Optimization

1. Composition:

$$N = \pi_\epsilon(\tilde{H} \circ \tilde{C} \circ \tilde{L} \circ G)$$

2. Determinization:

$$N = \pi_\epsilon(det(\tilde{H} \circ det(\tilde{C} \circ det(\tilde{L} \circ G))))$$

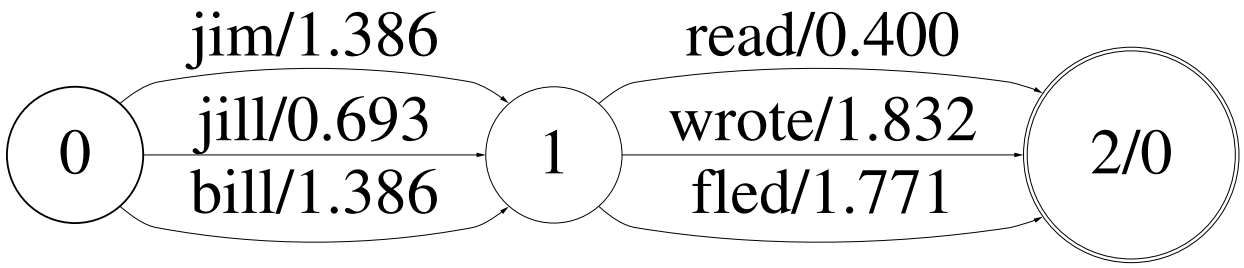
3. Minimization:

$$N = \pi_\epsilon(min(det(\tilde{H} \circ det(\tilde{C} \circ det(\tilde{L} \circ G)))))$$

4. Weight Pushing:

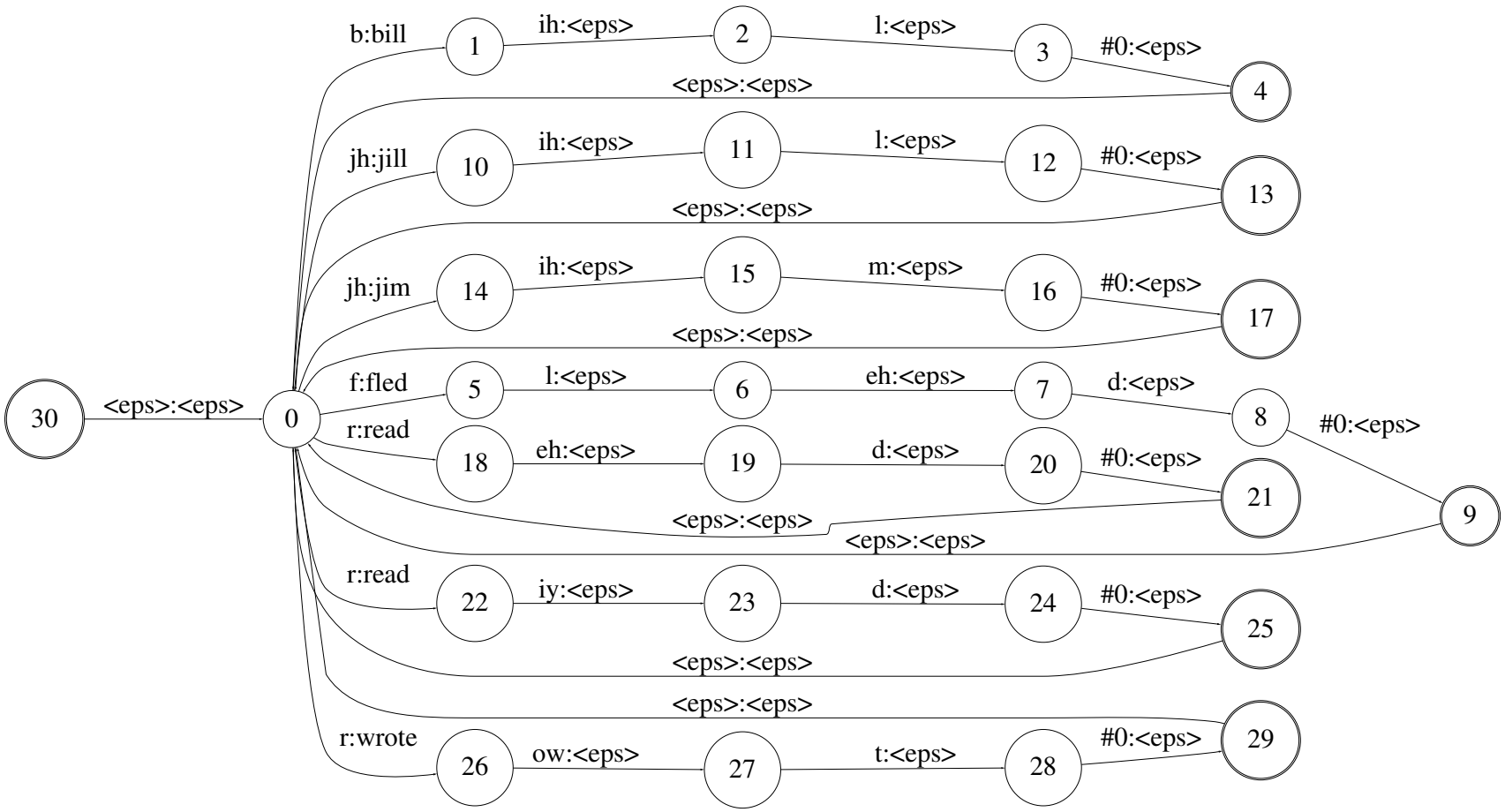
$$N = push(\pi_\epsilon(min(det(\tilde{H} \circ det(\tilde{C} \circ det(\tilde{L} \circ G))))))$$

Network Construction Example I



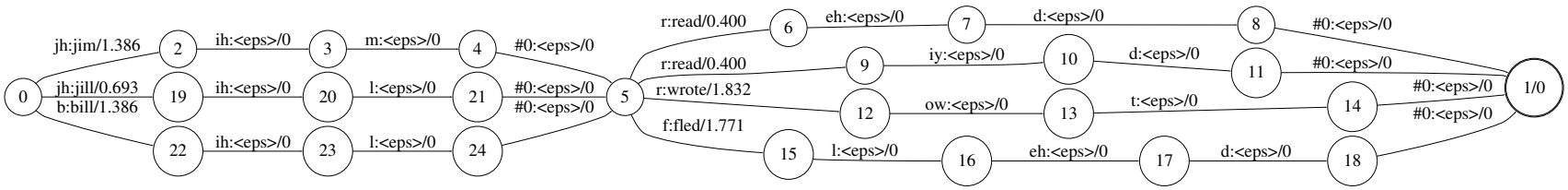
G

Network Construction Example II



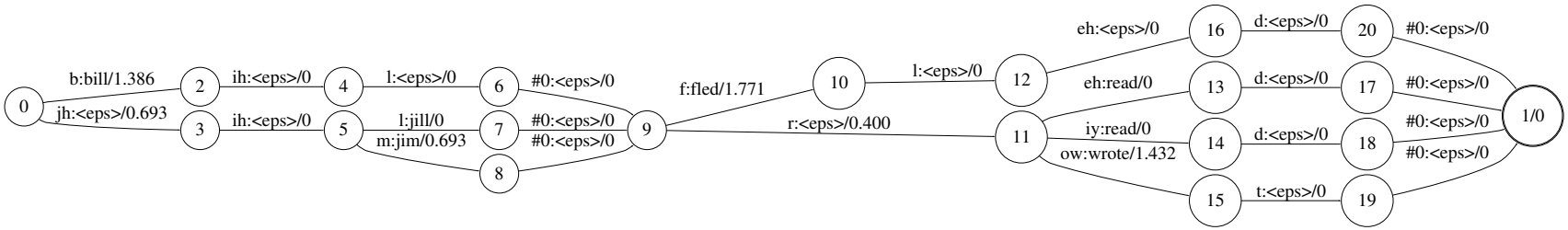
$\tilde{L}$

Network Construction Example III



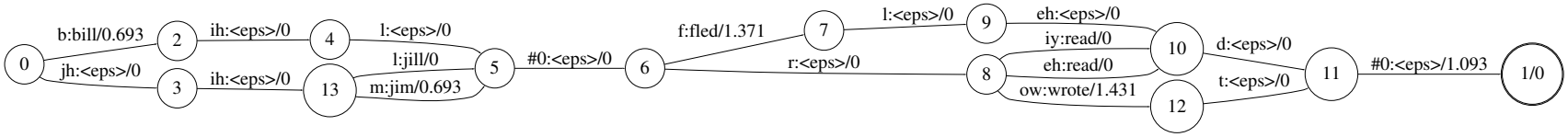
$$\tilde{L} \circ G$$

Network Construction Example IV



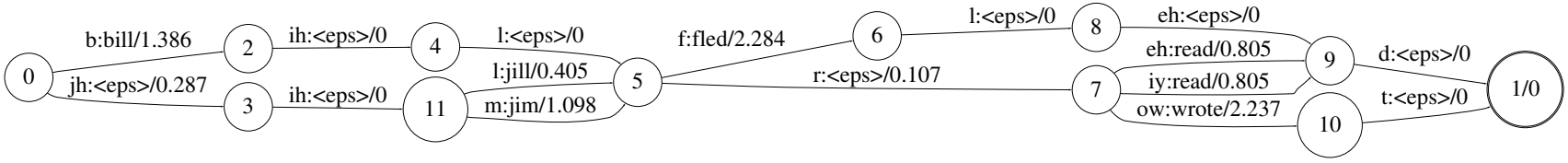
$$det(\tilde{L} \circ G)$$

Network Construction Example V



$$min(det(\tilde{L} \circ G))$$

Network Construction Example VI



$$push(\pi_\epsilon(min(det(\tilde{L} \circ G))))$$

Network Construction – Alternatives

- $C \circ det(L \circ G)$
  Grammar G compiled into optimized network offline, cannot exchange it at run-time.
- $det(C \circ L) \circ G$
  Outermost composition badly behaved.

ASR Recognition Transducer Standardization

- Minimal deterministic weighted transducers: unique up to state renumbering and to any weight and output label redistribution that preserves the total path weights and output strings.
- Weight-pushed transducer: selects a specific weight distribution along paths while preserving total path weights.
- Result is a *standardized* recognition transducer.

Network Construction III: Factoring

1. Idea
   a) Decoder feature: separate representation for variable-length left-to-right HMMs (time and space efficiency),
   b) To take advantage of this feature, *factor* integrated network N:

$$N = H' \circ F$$

2. Algorithm
   a) Replace input of each linear path in N by a single label naming an n-state HMM,
   b) Define *gain* $G(\sigma)$ of the replacement of linear path σ,

$$G(\sigma) = \sum_{\pi \in \mathrm{Lin}(N), i[\pi]=\sigma} |\sigma| - |o[\pi]| - 1$$

   c) Replacement exactly in the cases where it helps reducing the size of the network.

1st-Pass Recognition Networks – 40K NAB Task

network	states	transitions
G	1,339,664	3,926,010
$L \circ G$	8,606,729	11,406,721
$det(L \circ G)$	7,082,404	9,836,629
$C \circ det(L \circ G))$	7,273,035	10,201,269
$det(H \circ C \circ L \circ G)$	18,317,359	21,237,992
F	3,188,274	6,108,907
$min(F)$	2,616,948	5,497,952

1st-Pass Recognition Speed - 40K NAB Eval '95

network	x real-time
$C \circ L \circ G$	12.5
$C \circ det(L \circ G)$	1.2
$det(H \circ C \circ L \circ G)$	1.0
$push(min(F))$	0.7

Recognition speed of the first-pass networks in the NAB $40,000$-word vocabulary task at 83% word accuracy.

2nd-Pass Recognition Speed - 160K NAB Eval '95

network	x real-time
$C \circ L \circ G$	.18
$C \circ det(L \circ G)$	.13
$C \circ push(min(det(L \circ G)))$	.02

Recognition speed of the second-pass networks in the NAB $160,000$-word vocabulary task at 88%.