Search Algorithms for Speech Recognition

Berlin Chen 2004

References

- Books
  1. X. Huang, A. Acero, H. Hon, “Spoken Language Processing”, Chapters 12-13, Prentice Hall, 2001
  2. Chin-Hui Lee, Frank K. Soong and Kuldip K. Paliwal. Automatic Speech and Speaker Recognition, Chapters 13, 16-18, Kluwer Academic Publishers, 1996
  3. John R. Deller, JR. John G. Proakis, and John H. L. Hansen. Discrete-Time Processing of Speech Signals, Chapters 11-12, IEEE Press, 2000
  4. L.R. Rabiner and B.H. Juang. Fundamentals of speech recognition, Chapter 6, Prentice Hall, 1993
  5. Frederick Jelinek. Statistical Methods for Speech Recognition, Chapters 5-6, MIT Press, 1999
  6. N. Nilisson. Principles of Artificial Intelligence, 1982
- Papers
  1. Hermann Ney, “Progress in Dynamic Programming Search for LVCSR,” Proceedings of the IEEE, August 2000
  2. Patrick Kenny, et al, “A*-Admissible heuristics for rapid lexical access,” IEEE Trans. on SAP, 1993
  3. Stefan Ortmanns and Hermann Ney, “A Word Graph Algorithm for Large Vocabulary Continuous Speech Recognition,” Computer Speech and Language (1997) 11,43-72
  4. Jean-Luc Gauvain and Lori Lamel, “Large-Vocabulary Continuous Speech Recognition: Advances and Applications,” Proceedings of the IEEE, August 2000

Introduction

- Template-based: **without statistical modeling/training**
  - Directly compare/align the testing and reference waveforms on their features vector sequences (with different length, respectively) to derive the overall distortion between them
  - **Dynamic Time Warping (DTW)**: warp speech templates in the time dimension to alleviate the distortion

- Model-based: **HMM are using for recognition systems**
  - Concatenate the subword models according to the pronunciation of the words in a lexicon
  - The states in the HMM can be expanded to form the state-search space (HMM state transition network) in the search
  - Apply appropriate search strategies

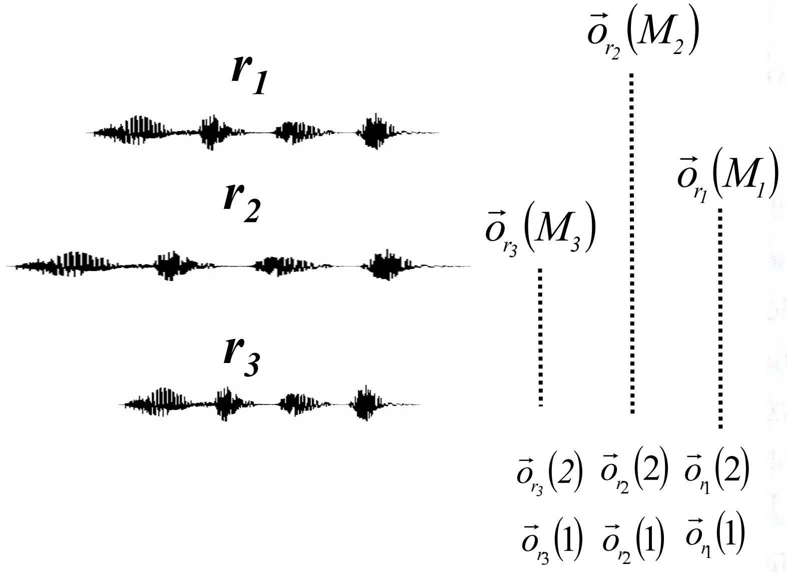
Template-based Speech Recognition

- Dynamic Time Warping (DTW) is simple to implement and fairly effective for small-vocabulary Isolated word speech recognition
  - Use **dynamic programming (DP)** to temporally align patterns to account for differences in speaking rates across speakers as well as across repetitions of the word by the same speakers

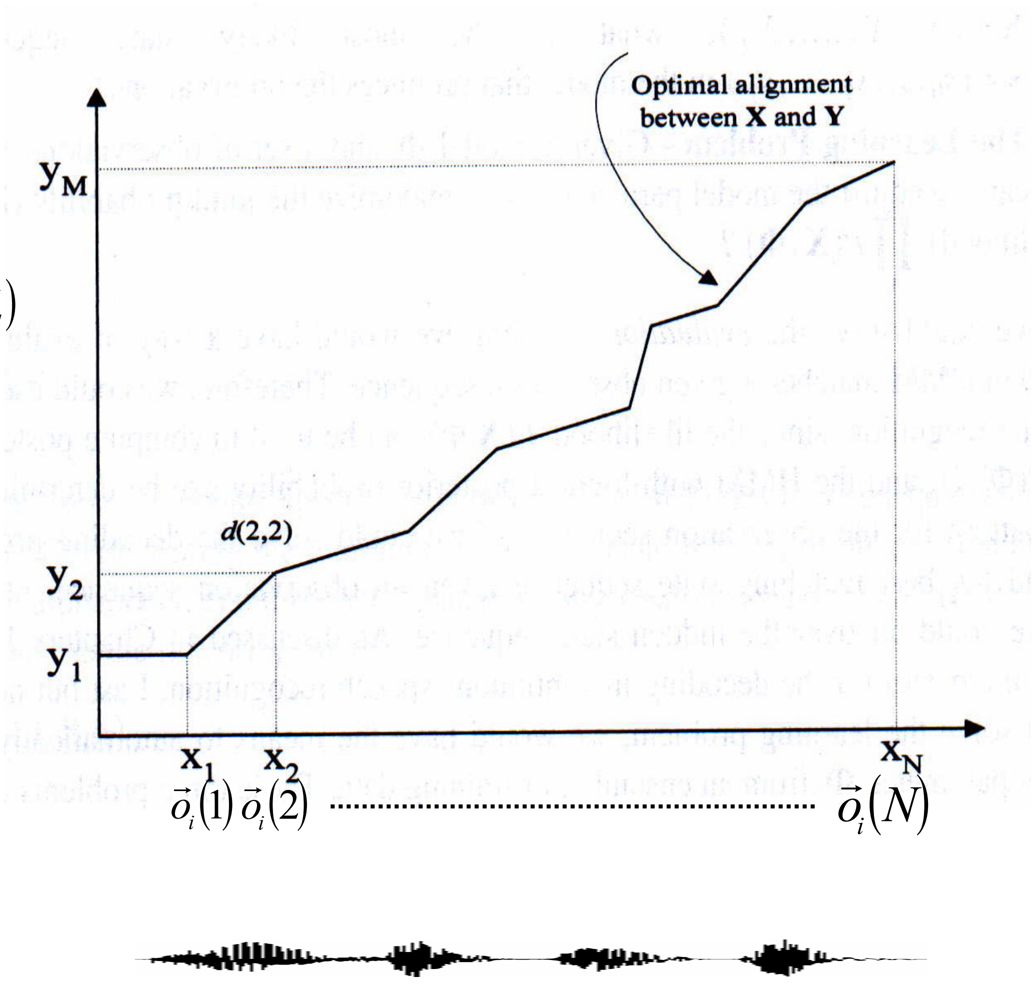
- Drawback
  - Do not have a principled way to derive an averaged template for each pattern from a large training samples
  - A multiplicity of reference templates is required to characterize the variation among different utterances

Template-based Speech Recognition (cont.)

- Example



r_1
r_2
r_3
$\vec{o}_{r_2}(M_2)$
$\vec{o}_{r_1}(M_1)$
$\vec{o}_{r_3}(M_3)$
$\vec{o}_{r_3}(2)$ $\vec{o}_{r_2}(2)$ $\vec{o}_{r_1}(2)$
$\vec{o}_{r_3}(1)$ $\vec{o}_{r_2}(1)$ $\vec{o}_{r_1}(1)$



Optimal alignment between X and Y
y_M
$d(2,2)$
y_2
y_1
x_1
x_2
x_N
$\vec{o}_i(1)\,\vec{o}_i(2)$
$\vec{o}_i(N)$

$$
\begin{aligned}
D_{\min}(i_k, j_k) &= \min_{i_{k-1}, j_{k-1}} D_{\min}\left[(i_k, j_k)\middle|(i_{k-1}, j_{k-1})\right] \\
&= \min_{i_{k-1}, j_{k-1}} D_{\min}(i_{k-1}, j_{k-1}) + d\left[(i_k, j_k)\middle|(i_{k-1}, j_{k-1})\right]
\end{aligned}
$$

$$
D_{\min}\left[(i_k, j_k)\middle|(i_{k-1}, j_{k-1})\right] = D_{\min}(i_{k-1}, j_{k-1}) + d\left[(i_k, j_k)\middle|(i_{k-1}, j_{k-1})\right]
$$

Model-based Speech Recognition

- A search process to uncover the word sequence $\hat{W} = w_1 w_2,\ldots, w_m$ that has the maximum posterior probability $P(W|X)$

$$\hat{W} = \arg\max_{W} P(W|X)$$

$$= \arg\max_{W} \frac{P(W)P(X|W)}{P(X)}$$

$$= \arg\max_{W} \underbrace{P(W)}_{\text{Language Model Probability}}\underbrace{P(X|W)}_{\text{Acoustic Model Probability}}$$

$$W = w_1, w_2, \ldots w_i, \ldots, w_m$$

$$\text{where } w_i \in V : \{v_1, v_2, \ldots\ldots, v_N\}$$

N-gram Language Modeling

Unigram:

$$P(w_1 w_2 .. w_k) \approx P(w_1)P(w_2)\ldots P(w_k),\quad P(w_j) = C(w_j) \big/ \sum_i C(w_i)$$

Bigram:

$$P(w_1 w_2 .. w_k) \approx P(w_1)P(w_2|w_1)\ldots P(w_k|w_{k-1}),\quad P(w_j|w_{j-1}) = C(w_{j-1}w_j) \big/ C(w_{j-1})$$

Trigram:

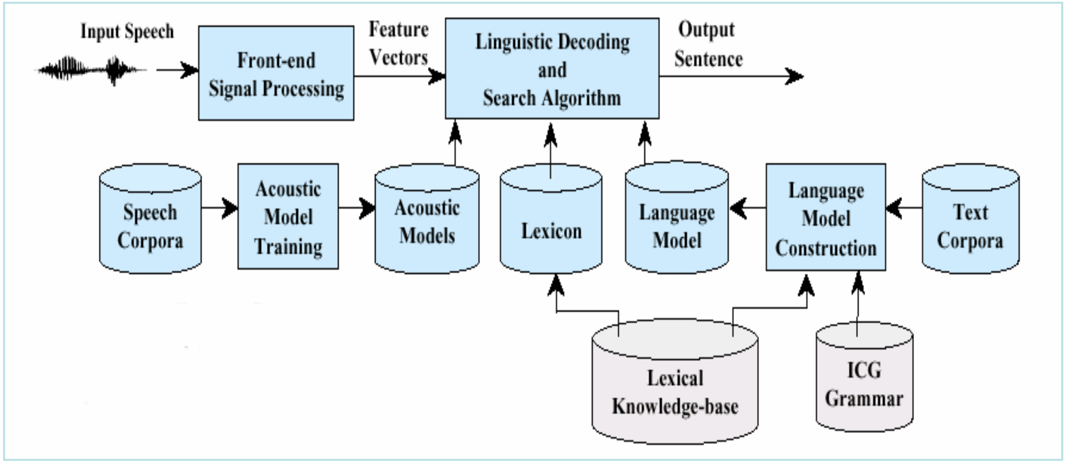
$$P(w_1 w_2 .. w_k) \approx P(w_1)P(w_2|w_1)P(w_3|w_1 w_2)\ldots P(w_k|w_{k-2}w_{k-1}),\quad P(w_k|w_{k-1}w_{k-2}) = C(w_{j-2}w_{j-1}w_j) \big/ C(w_{j-2}w_{j-1})$$

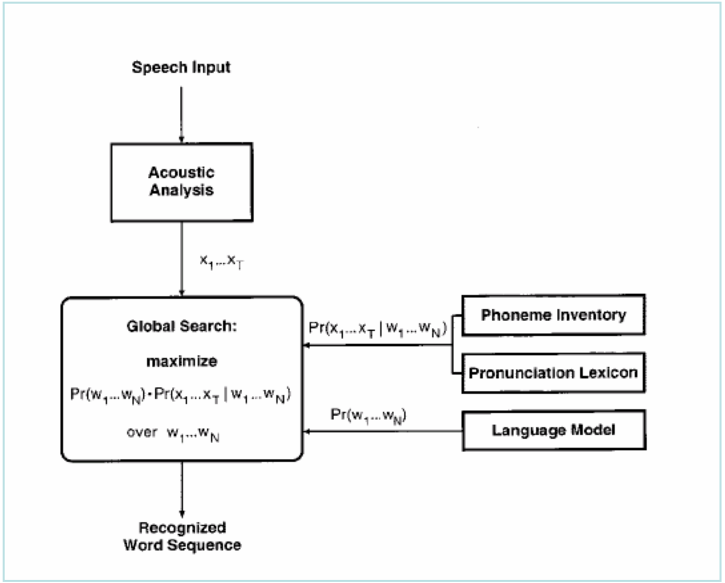
Model-based Speech Recognition (cont.)

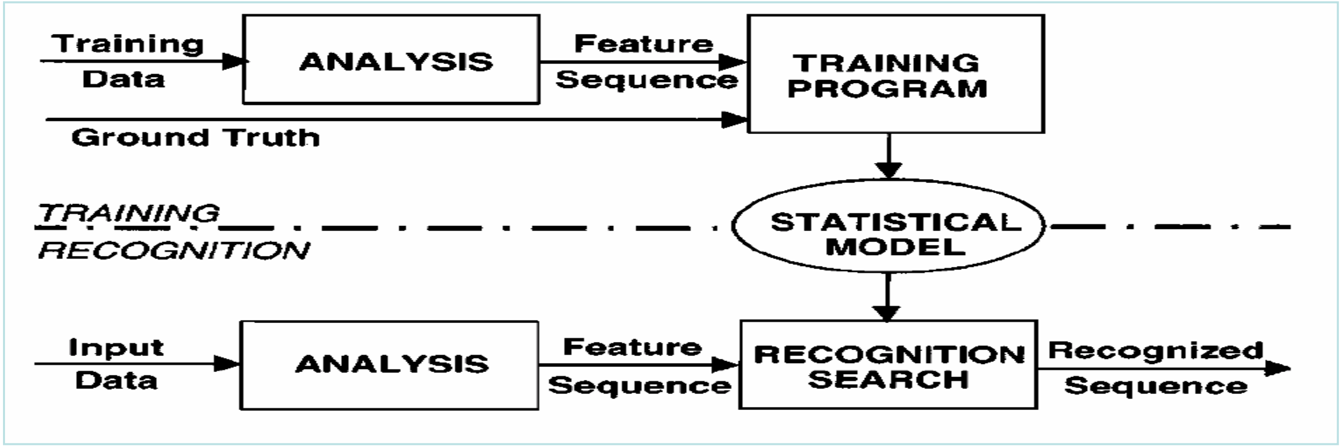
- Therefore, the model-based continuous speech recognition is both a pattern recognition and search problems
  - The acoustic and language models are built upon a statistical pattern recognition framework
  - In speech recognition, making a search decision is also referred as a *decoding* process (or a search process)
    - **Find a sequence of words whose corresponding acoustic and language models best match the input signal**
    - **The search space (complexity) is highly imposed by the language models**
- The model-based continuous speech recognition is usually with the Viterbi (plus beam, or Viterbi beam) search or A* stack decoders
  - The relative merits of both search algorithms were quite controversial in the 1980s

Model-based Speech Recognition (cont.)

- Simplified Block Diagrams







Basic Search Algorithms

What Is “Search”?

- What Is “Search”: Moving around, examining things, and making decisions about whether the sought object has yet been found
  - Classical problems in AI:
    traveling salesman’s problem, *8-queens*, etc.

- The directions of the search process
  - Forward search (reasoning): from initial state to goal state(s)
  - Backward search (reasoning): from goal state(s) to goal state
  - Bidirectional search
    - Seems particular appealing if the number of nodes at each step grows exponential with the depth that need to be explored

What Is “Search”? (cont.)

- Two sategories of search algorithms
  - Uninformed Search (Blind Search)
    - Depth-First Search
    - Breadth-First Search

    Have no sense of where the goal node lies ahead!

  - Informed Search (Heuristic Search)
    - A* search (Best-First Search)

    The search is guided by some domain knowledge (or heuristic information)! (e.g. the predicted distance/cost from the current node to the goal node)
      - *Some heuristic can reduce search effort without sacrificing optimality*

Depth-First Search

- The deepest nodes are expanded first and nodes of equal depth are ordered arbitrary
- Pick up an arbitrary alternative at each node visited
- Stick with this partial path and walks forward from the partial path, other alternatives at the same level are ignored completely
- When reach a dead-end, go back to last decision point ad proceed with another alternative

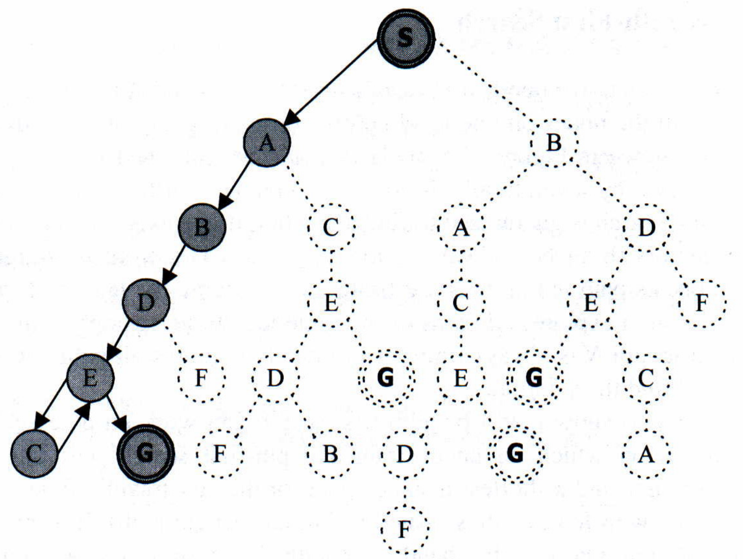


Figure 12.4 The node-expanding procedure of the depth-first search for the path search problem in Figure 12.1. When it fails to find the goal city in node C, it backtracks to the parent and continues the search until it finds the goal city. The gray nodes are those that are explored. The dotted nodes are not visited during the search [42].

- *Depth-first search could be dangerous because it might search an impossible path that is actually an infinite dead-end*

Breadth-First Search

- Examine all the nodes on one level before considering any of the nodes on the next level (depth)
- Breadth-first search is guaranteed to find a solution if one exists
  - But it might not find a short-distance path, it's guaranteed to find one with few nodes visited (minimum-length path)
- Could be inefficient

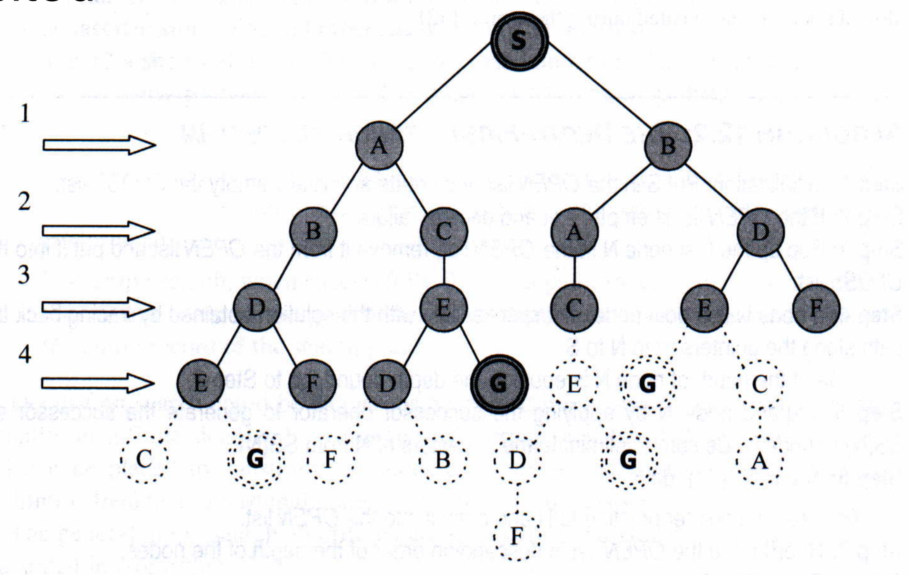


Figure 12.5 The node-expanding procedure of a breadth-first search for the path search problem in Figure 12.1. It searches through each level until the goal is identified. The gray nodes are those that are explored. The dotted nodes are not visited during the search [42].

A* search

- History of A* Search in AI
  - The most studied version of the best-first strategies (Hert, Nilsson,1968)
  - Developed for **additive cost measures** (The cost of a path = sum of the costs of its arcs)
- Properties
  - Can sequentially generate multiple recognition candidates
  - Need a good heuristic function
- Heuristic
  - A technique (domain knowledge) that improves the efficiency of a search process
  - Inaccurate heuristic function results in a less efficient search
  - The heuristic function helps the search to satisfy admissible condition
- Admissibility
  - The property that a search algorithm guarantees to find an optimal solution, if there is one

A* search

- A Simple Example
  - **Problem**: Find a path with highest score form root node “A” to some leaf node (one of “L1”,”L2”,”L3”,”L4”)

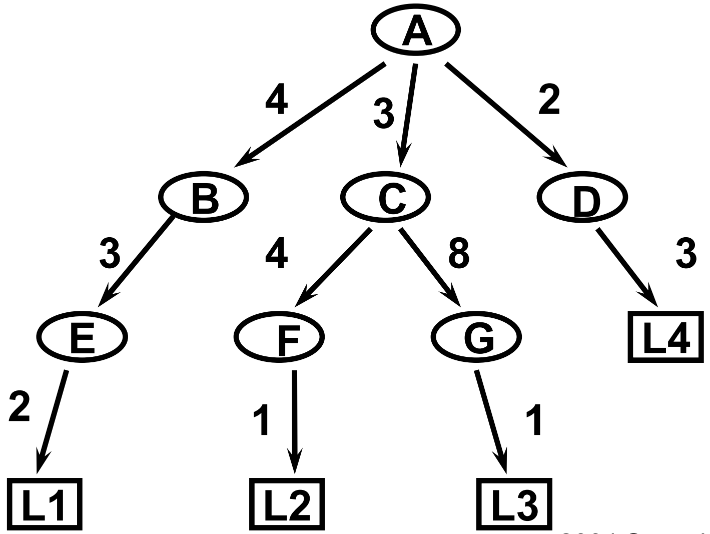
$f(n) = g(n) + h(n)$, evaluation function of node n

$g(n)$: cost from root node to node n, decoded partial path score

$h^*(n)$: exact score from node n to a specific leaf node

$h(n)$: estimated score from node n to goal state, heuristic function

Admissibil ity : $h(n) \geq h^*(n)$

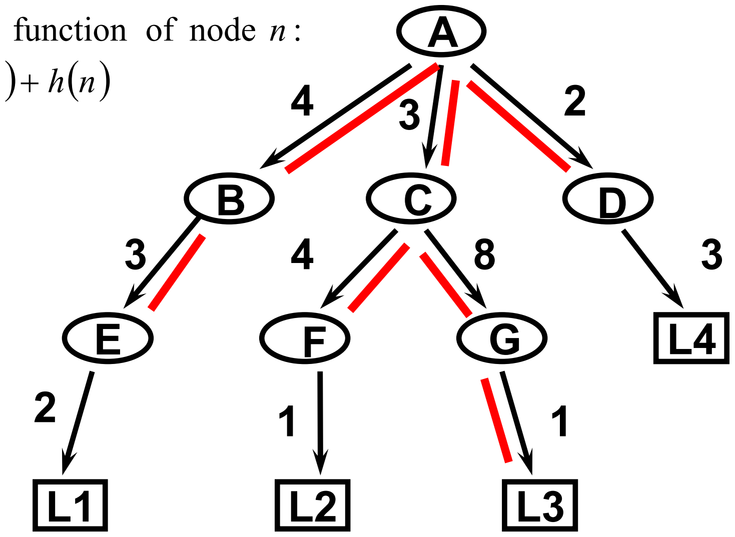


A* search (cont.)

- A Simple Example:

Evaluation function of node n:

$f(n) = g(n) + h(n)$



Proving the Admissibility of A* Algorithm:

Suppose when algorithm terminates, *"G "* is a complete path on the top of the stack and *"p"* is a partial path which presents somewhere on the stack.
There exists a complete path *"P"* passing through *"p"*, which is not equal to *"G"* and is optimal.

Proof:
1. ***"P"* is a complete which passes through *"p"*, *f(P)<=f(p)***
2. **Because *"G"* is on the top of the stack , *f(G)>=f(p)>=f(P)***
3. **Therefore, it makes contrariety !!**

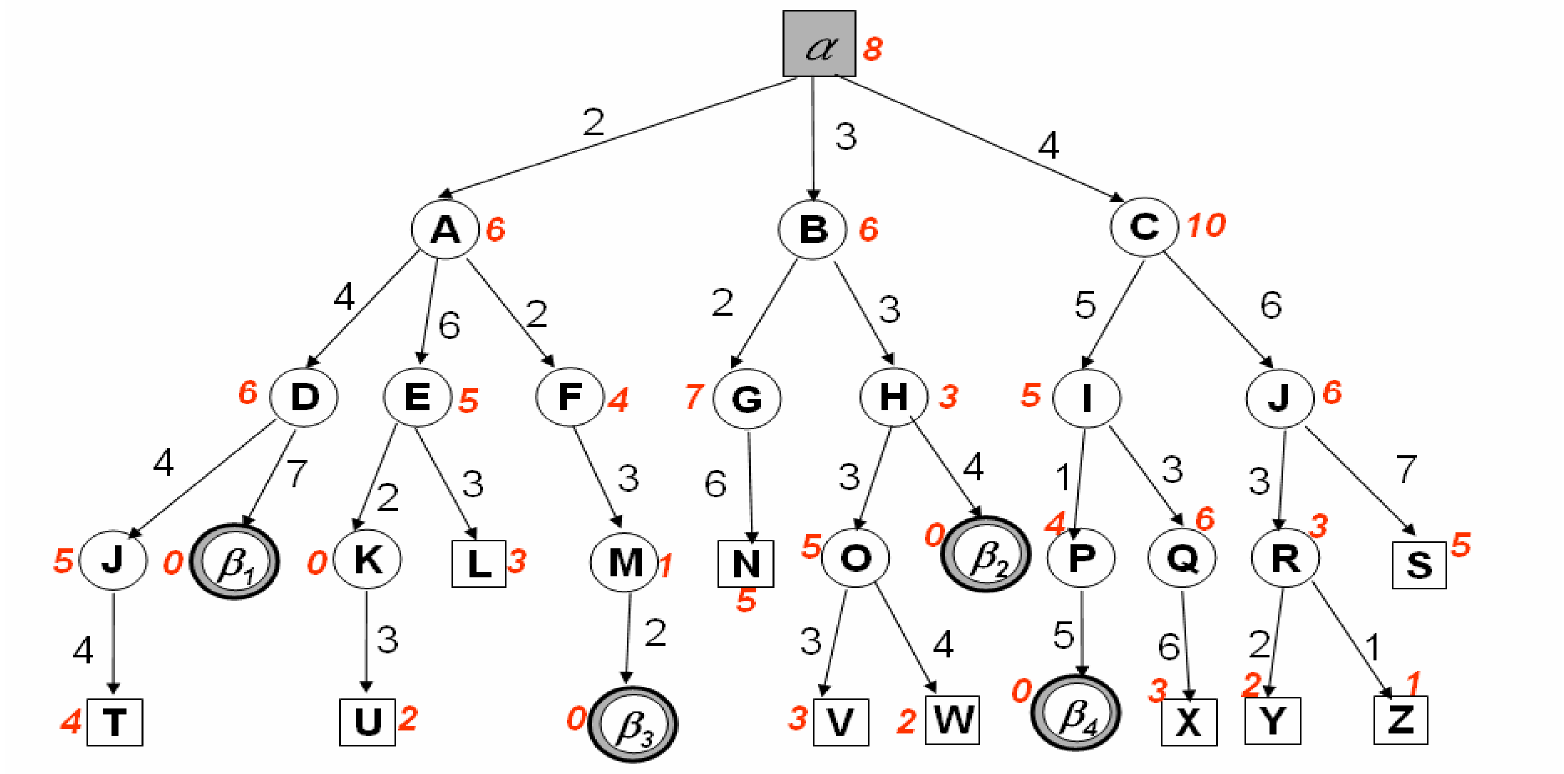
List or Stack*(sorted)*

Stack Top	*Stack Elements*
A(15)	A(15)
C(15)	C(15), B(13), D(7)
G(14)	G(14), B(13), F(9), D(7)
B(13)	B(13), L3(12), F(9), D(7)
L3(12)	L3(12), E(11), F(9), D(7)

Node	g(n)	h(n)	f(n)
A	0	15	15
B	4	9	13
C	3	12	15
D	2	5	7
E	7	4	11
F	7	2	9
G	11	3	14
L1	9	0	9
L2	8	0	8
L3	12	0	12
L4	5	0	5

A* search: Exercises

- Please find a path from the initial stat α to one of the four goal states ($\beta 1$, $\beta 2$, $\beta 3$, $\beta 4$) with the shortest path cost. Each arc is associated with a number representing its corresponding cost to be taken, while each node is associated with a number standing for the expected cost (the heuristic score/function) to one of the four goal



A* search: Exercises (cont.)

- **Problems**
  - What is the first goal state found by the depth-first search, which always selects a node’s left-most child node for path expansion? Is it an optimal solution? What is the total search cost?
  - What is the first goal state found by the bread-first search, which always expends all child nodes at the same level from left to right? Is it an optimal solution? What is the total search cost?
  - What is the first goal state found by the A* search using the path cost and heuristic function for path expansion? Is it an optimal solution? What is the total search cost?
  - What is the search path cost if the A* search was used to sequentially visit the four goal states?

Beam Search

- Widely used search technique for speech recognition systems
  - It’s a breadth-first search and progresses along with the depth
  - Unlike traditional breadth-first search, beam search only expands nodes that are likely to succeed at each level
    - Keep up to m-best nodes at each level (stage)
    - Only these nodes are kept in the beam, the rest are ignored (pruned)

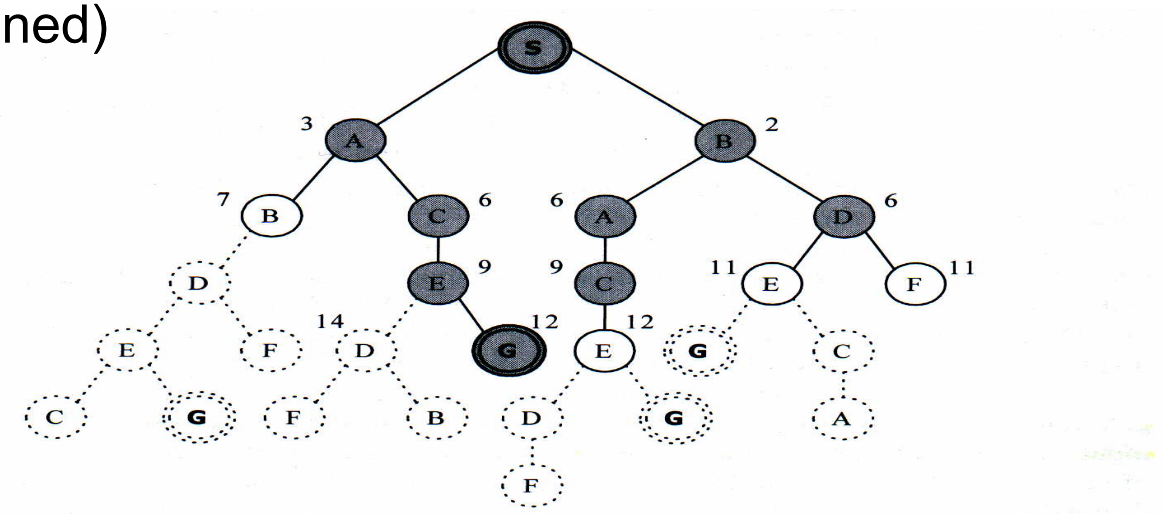
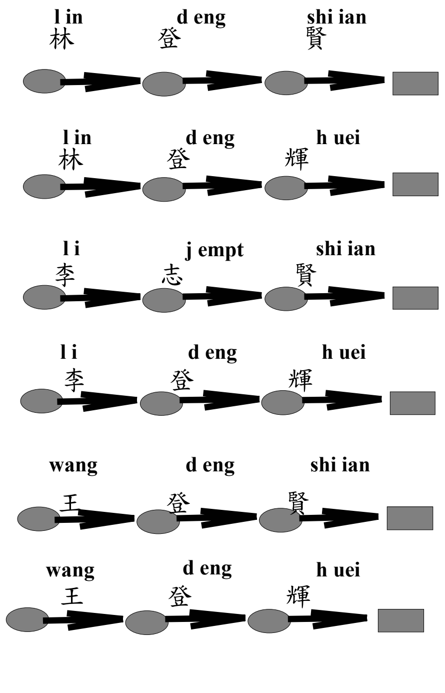


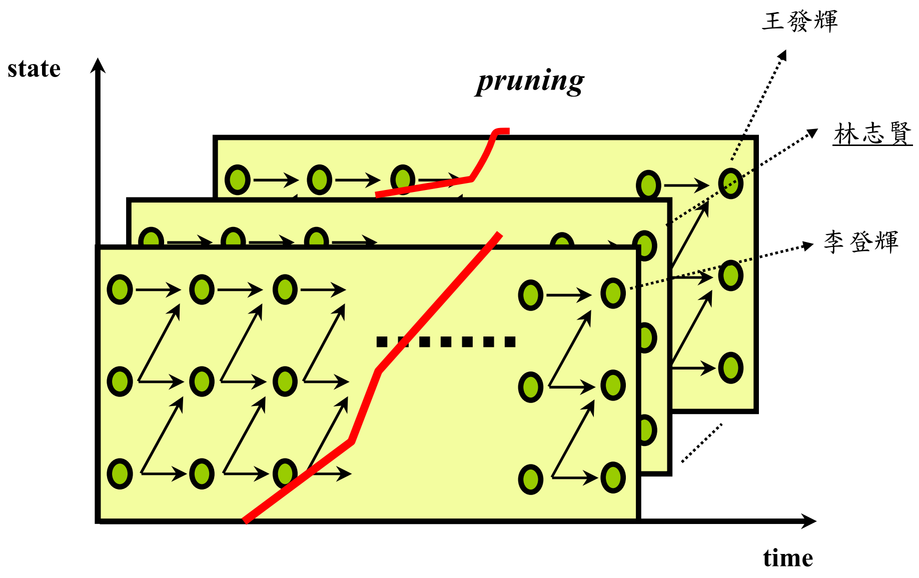
Figure 12.9 Beam search for the city-travel problem. The nodes with gray color are the ones kept in the beam. The transparent nodes were explored but pruned because of higher cost. The dotted nodes indicate all the savings because of pruning [42].

Beam Search (cont.)

- Used to prune unlikely paths in recognition task
- Need some criteria (hypotheses) to prune paths

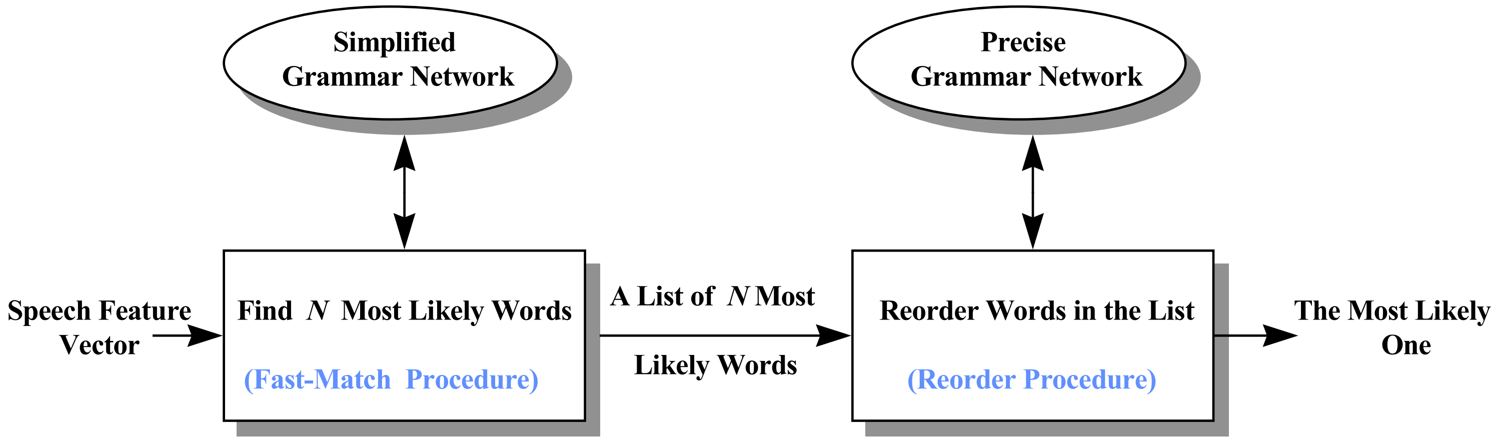


List-lexicon



Fast-Match Search

- Two Stage Processing
  - First stage: use a simplified grammar network (or acoustic models) to generate N likely words
  - Second stage: use a precise grammar network to reorder these N words



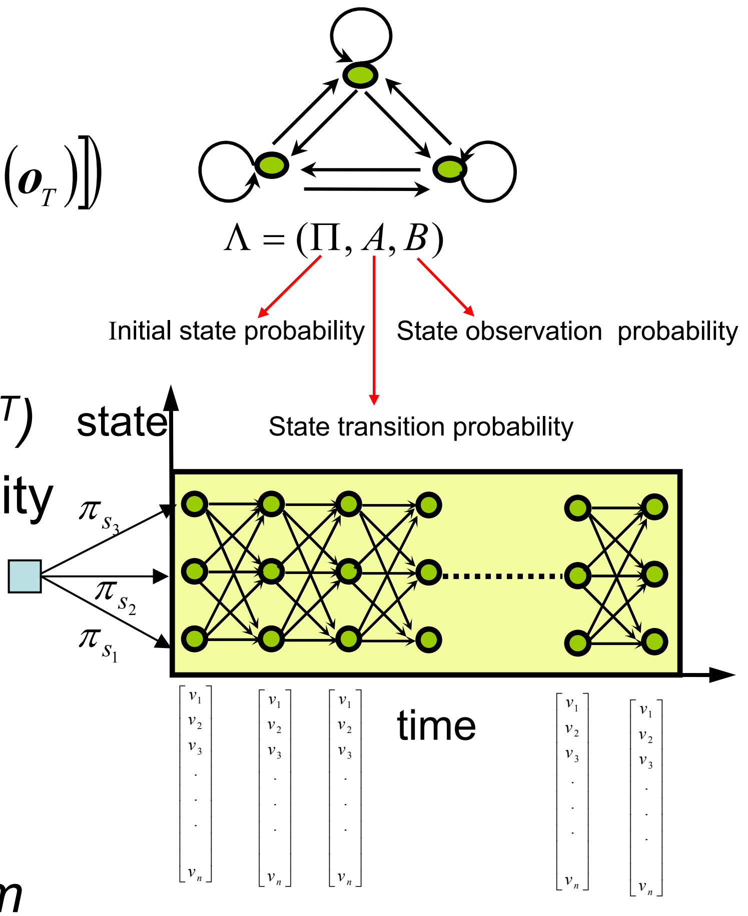
The fast-match algorithm paradigm

Review:
Search Within a Given HMM

Calculating the Probability of an Observation Sequence on an HMM Model

- Direct Evaluation: **without using recursion (DP, dynamic programming) and memory**

$$
\begin{aligned}
P(\boldsymbol{O}|\lambda) &= \sum_{all\,\mathbf{s}} P(\mathbf{s}|\lambda)P(\boldsymbol{O}|\mathbf{s},\lambda) \\
&= \sum_{all\,\mathbf{s}} \left(\left[\pi_{s_1} a_{s_1 s_2} a_{s_2 s_3} \ldots . a_{s_{T-1} s_T}\right]\left[b_{s_1}(\boldsymbol{o}_1) b_{s_2}(\boldsymbol{o}_2) \ldots . b_{s_T}(\boldsymbol{o}_T)\right]\right) \\
&= \sum_{s_1, s_2, \ldots, s_T} \pi_{s_1} b_{s_1}(\boldsymbol{o}_1) a_{s_1 s_2} b_{s_2}(\boldsymbol{o}_2) \ldots . a_{s_{T-1} s_T} b_{s_T}(\boldsymbol{o}_T)
\end{aligned}
$$



  - Huge Computation Requirements: *O(N^T)*
    - Exponential computational complexity

- $\boldsymbol{Complexity} : (2T\text{-}1)N^T\ MUL \quad \approx 2TN^T,\ N^T\text{-}1\ \text{ADD}$

- A more efficient algorithms can be used to evaluate
  - *Forward/Backward Procedure/Algorithm*

Calculating the Probability of an Observation Sequence on an HMM Model (cont.)

- Forward Procedure
  - Base on the HMM assumptions, the calculation of $P\left(s_t \middle| s_{t-1}, \lambda\right)$ and $P\left(o_t \middle| s_t, \lambda\right)$ involves only s_{t-1}, s_t and o_t, so it is possible to compute the likelihood $P(\boldsymbol{O}|\lambda)$ with recursion on t
  - Forward variable : $\alpha_t(i) = P\left(o_1 o_2 \ldots o_t, s_t = i \middle| \boldsymbol{\lambda}\right)$
    - The probability that the HMM is in state *i* at time *t* having generating partial observation $o_1 o_2 \ldots o_t$

Calculating the Probability of an Observation Sequence on an HMM Model (cont.)

- Forward Procedure (Cont.)
  - **Algorithm**

    1. Initialization $\alpha_1(i) = \pi_i b_i(o_1),\ 1 \le i \le N$

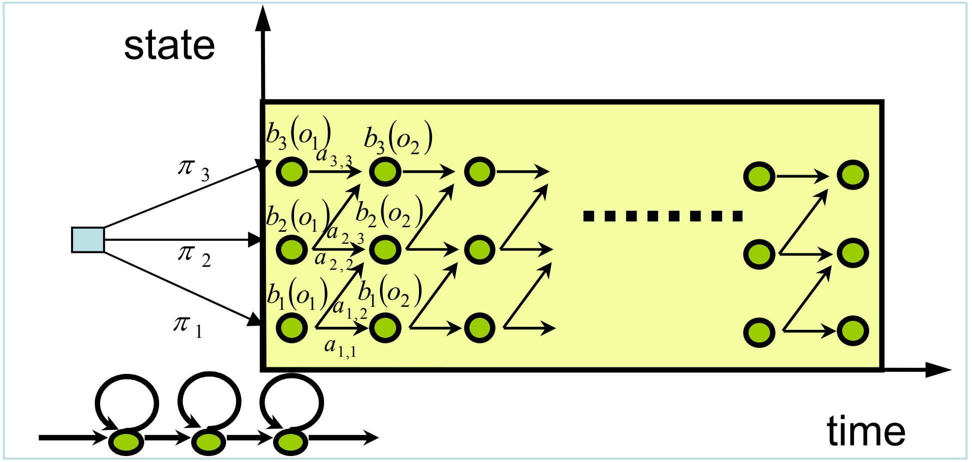
    2. Induction $\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i) a_{ij}\right] b_j(o_{t+1}),\ \ 1 \le t \le T\text{-}1, 1 \le j \le N$

    3. Termination $P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$

  - **Complexity:** *$O(N^2T)$*

    $\mathrm{MUL} : \mathrm{N(N+1)(T-1)} + \mathrm{N} \approx \mathrm{N^2T}$

    $\mathrm{ADD} : \mathrm{(N-1)N(T-1)} \approx \mathrm{N^2T}$



  - **Based on the lattice (trellis) structure**
    - Computed in a *time-synchronous* fashion from *left-to-right*, where each cell for time *t* is completely computed before proceeding to time *t+1*
    - All state sequences, regardless how long previously, merge to *N* nodes (states) at each time instance *t*

Calculating the Probability of an Observation Sequence on an HMM Model (cont.)

- Backward Procedure
  - Backward variable : $\beta_t(i)=P(o_{t+1},o_{t+2},\ldots..,o_T|s_t=i\ ,\ \lambda)$
  - Algorithm

    1. Initialization $\beta_T(i)=1, 1 \le i \le N$
    2. Induction $\beta_t(i)=\sum_{j=1}^{N} a_{ij} b_j(o_{t+1})\beta_{t+1}(j),\ 1 \le t \le T-1, 1 \le j \le N$
    3. Termination $P(\boldsymbol{O}\,|\,\boldsymbol{\lambda})=\sum_{j=1}^{N}\pi_j b_j(o_1)\beta_1(j)$
  - **Complexity:** $O(N^2T)$

    Complexity MUL: $2N^2(T-1) \approx N^2T$; ADD $(N-1)N(T-1) \approx N^2T$

$$
\begin{aligned}
&P(O,q_t=i|\lambda)\\
&=P(O,q_t=i,\lambda)/P(\lambda)\\
&=P(O|q_t=i,\lambda)\cdot P(q_t=i,\lambda)/P(\lambda)\\
&=P(O|q_t=i,\lambda)P(q_t=i|\lambda)\\
&=P(o_1,o_{t+2},..,o_T|q_t=i,\lambda)P(q_t=i|\lambda) \quad [\because \text{observation independence}]\\
&=P(o_{t+1},o_{t+2},..,o_T|q_t=i,\lambda)P(o_1,o_2,..,o_t|q_t=i,\lambda)P(q_t=i|\lambda)\\
&=P(o_{t+1},o_{t+2},..,o_T|q_t=i,\lambda)P(o_1,o_2,..,o_t,q_t=i|\lambda)\\
&=\alpha_t(i)\beta_t(i)
\end{aligned}
$$

$$\boxed{\therefore\ P(O|\lambda)=\sum_{i=1}^{N}P(O,q_t=i|\lambda)=\sum_{i=1}^{N}\alpha_t(i)\beta_t(i)}$$

Choosing an Optimal State Sequence $S=(s_1,s_2,\ldots\ldots, s_T)$ on an HMM Model

- Viterbi Algorithm
  - The Viterbi algorithm can be regarded as the dynamic programming algorithm applied to the HMM or as a modified forward algorithm
    - Instead of summing up probabilities from different paths coming to the same destination state, the Viterbi algorithm picks and remembers the best path
    - Find a single optimal state sequence $\mathbf{S}=(s_1,s_2,\ldots\ldots, s_T)$
  - The Viterbi algorithm also can be illustrated in a trellis framework similar to the one for the forward algorithm

Choosing an Optimal State Sequence $S=(s_1,s_2,\ldots\ldots, s_T)$ on an HMM Model (cont.)

- Viterbi Algorithm (Cont.)
  - **Algorithm**

Find a best state sequence $S=(s_1,s_2,..,s_T)$ for a given observation $O=(o_1,o_2,..,o_T)$?

1. Define a new variable

$$\delta_t(i)=\max_{s_1,s_2,..,s_{t-1}} P\left[s_1,s_2,..,s_{t-1},s_t=i,o_1,o_2,..,o_t \middle| \lambda\right]$$

= the best score along a single path at time t, which accounts for the first t observation and ends in state i

2. By induction $\therefore \delta_{t+1}(j)=\left[\max_{1\le i\le N}\delta_t(i)a_{ij}\right]b_j(o_{t+1})$

$$\psi_{t+1}(j)=\arg\max_{1\le i\le N}\delta_t(i)a_{ij} \quad \text{.... For backtracing}$$

3. We can backtrace from $s_T^*=\arg\max_{1\le i\le N}\delta_T(i)$

  - **Complexity:** *$O(N^2T)$*

Choosing an Optimal State Sequence *S=(s₁,s₂,……, s_T)* on an HMM Model (cont.)

- Viterbi Algorithm (Cont.)
  - **In practice, we calculate the logarithmic value of a given state sequence instead of its real value**

$$1.\ \text{Define}\ \delta_t(i)=\max_{s_1,s_2,..,s_{t-1}} \log P\left[s_1,s_2,..,s_{t-1},s_t=i,o_1,o_2,..,o_t \middle| \lambda\right]$$

$$= \text{the best log score along a single path at time t, which accounts for the first } t \text{ observation and ends in state } i$$

$$\text{By induction} \therefore \delta_{t+1}(j)=\left[\max_{1\le i\le N}\left(\delta_t(i)+\log a_{ij}\right)\right]+\log b_j(o_{t+1})$$

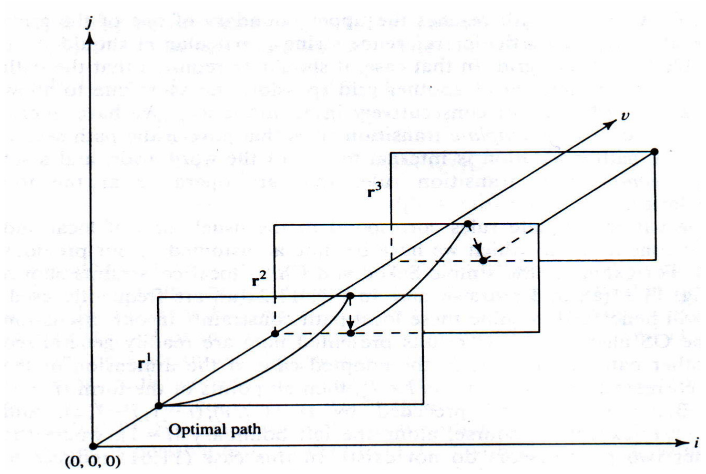
$$\psi_t(j)=\arg\max_{1\le i\le N} \delta_{t-1}(i)+\log a_{ij} \quad \text{......For backtracing}$$

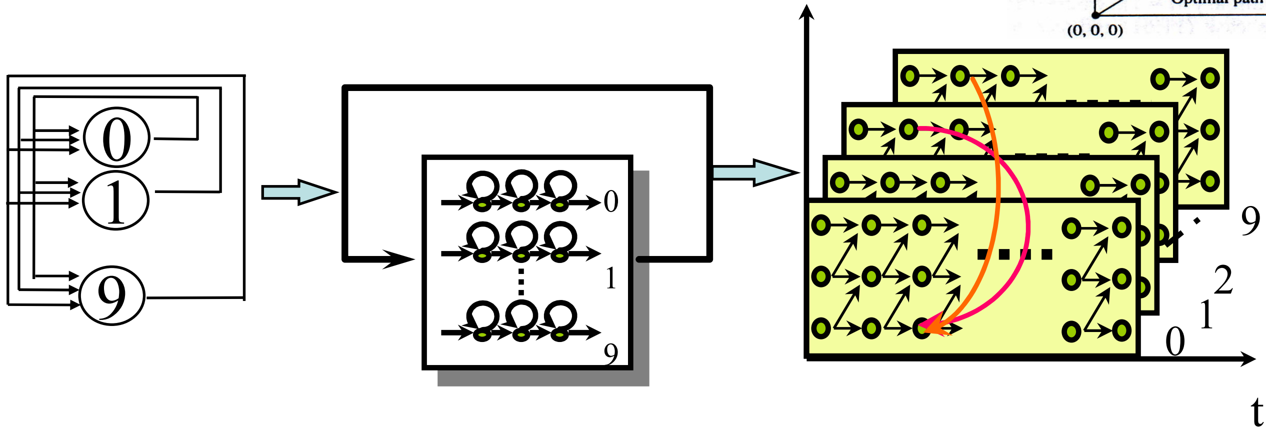
$$\text{We can backtrace from } q_T^* = \arg\max_{1\le i\le N} \delta_T(i)$$

Search in the HMM Networks

Digit/Syllable Recognition

- One-stage Search
  - Unknown number of digits/syllables
  - Search over a 3-dim grid



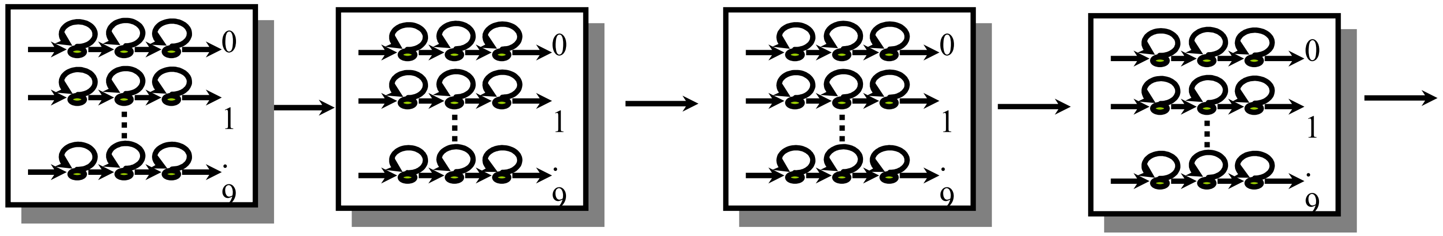


Correct
32561
Recognized
325561
3261

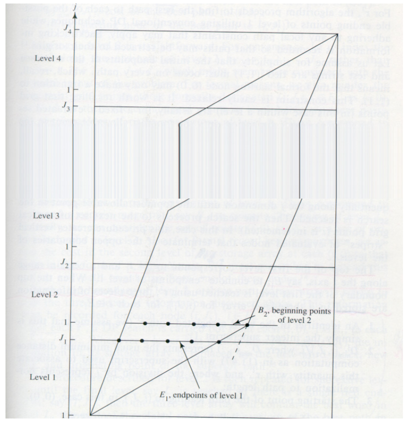
  - At each frame iteration, the maximum value achieved from the end states of all models in previous frame will be propagated and used to compete for the values of the start states of all models
  - May result with substitutions, deletions and insertions

Digit/Syllable Recognition (cont.)

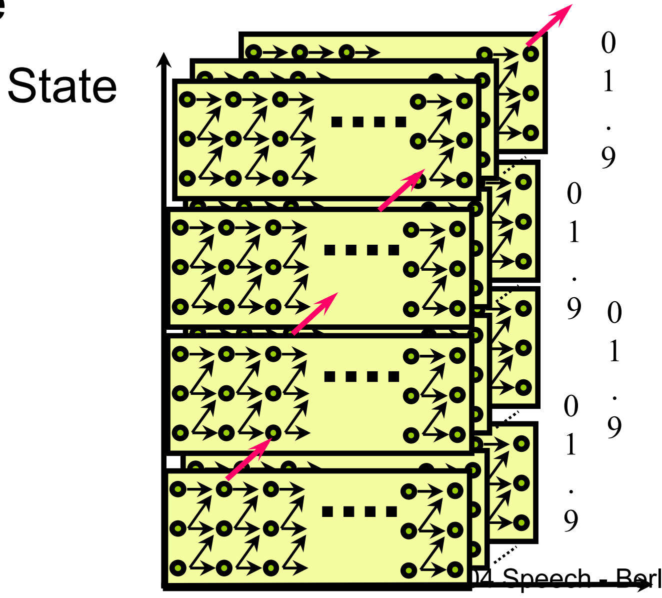
- Level-Building
  - Known number of digits/syllables
  - Higher computation complexity, no deletions and insertions



  - Number of levels: number of digits in an utterance
  - Transitions from the last states of the previous models (previous level) to the first states of specific models (current level)



Level 4
Level 3
Level 2
Level 1
B_2, beginning points of level 2
E_1, endpoints of level 1

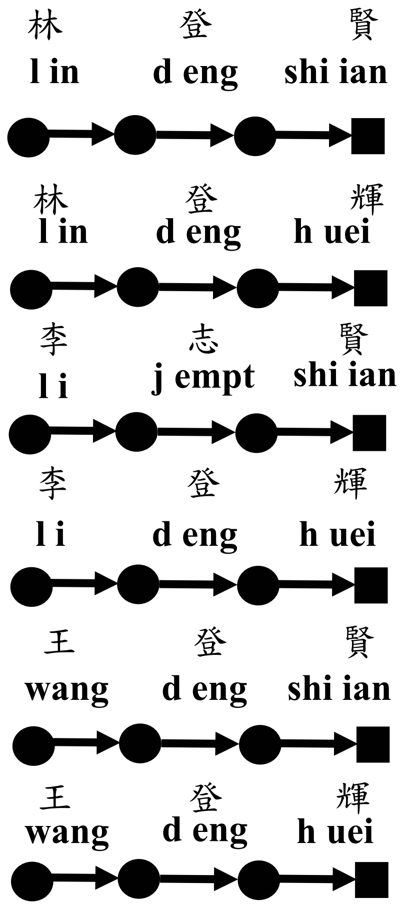


Isolated Word Recognition

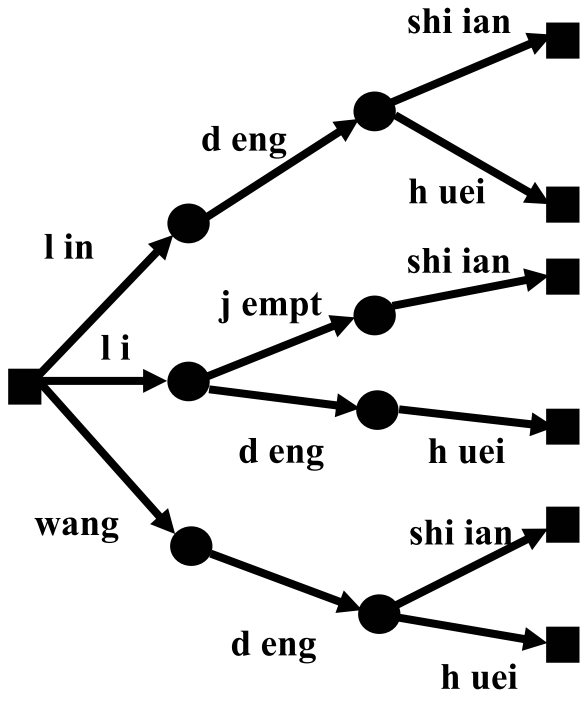
- Word boundaries are known (after endpoint detection)
- Two search structures
  - Lexicon-List (Linear Lexicon)
    - Each word is individually represented as a huge composite HMM by concatenating corresponding subword-level (phone/Initial-Final/syllable) HMMs
    - No sharing of computation between words when performing search
    - The search becomes a simple pattern recognition problem, and the word with the highest forward or Viterbi probability is chosen as the recognition word
  - Tree Structure (Tree Lexicon)
    - Arrange the subword-level (phone/Initial-Final/syllable) representations of the words in vocabulary into a tree structure
    - Each arc stands for an HMM or subword-level modeling
    - Sharing of computation between word as much as possible

Isolated Word Recognition (cont.)

- Two search structures (Cont.)



Linear lexicon ***18 arcs***



Tree lexicon ***13 arcs***

Isolated Word Recognition (cont.)

- More about the Tree Lexicon
  - The idea of using a tree represented was already suggested in 1970s in the CASPERS system and the LAFS system
  - When using such a lexical tree in a *language model* (bigram or trigram) and *dynamic programming*, there are technical details that have to taken into account and require a careful structuring of the search space (especially for continuous speech recognition to be discussed later)
    - Delayed application of language model until reaching tree leaf nodes
    - A copy of the lexical tree for each *alive* language model history in dynamic programming for continuous speech recognition

Continuous Speech Recognition (CSR)

- CSR is rather complicated, since the search algorithm has to consider the possibility of each word starting at arbitrary time frame
- Linear Lexicon Without Language Modeling

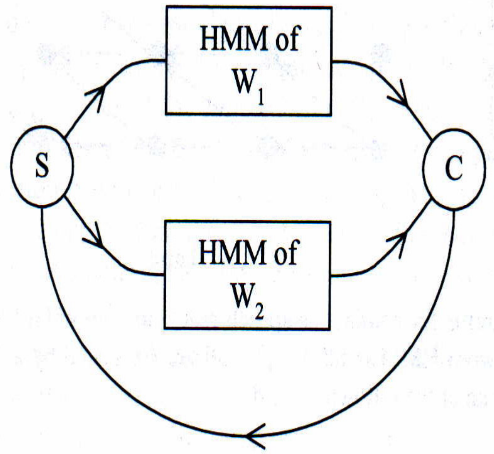


Figure 12.10 A simple example of continuous speech recognition task with two words w_1 and w_2. A uniform unigram language model is assumed for these words. State S is the starting state while state C is a collector state to save fully expanded links between every word pair.

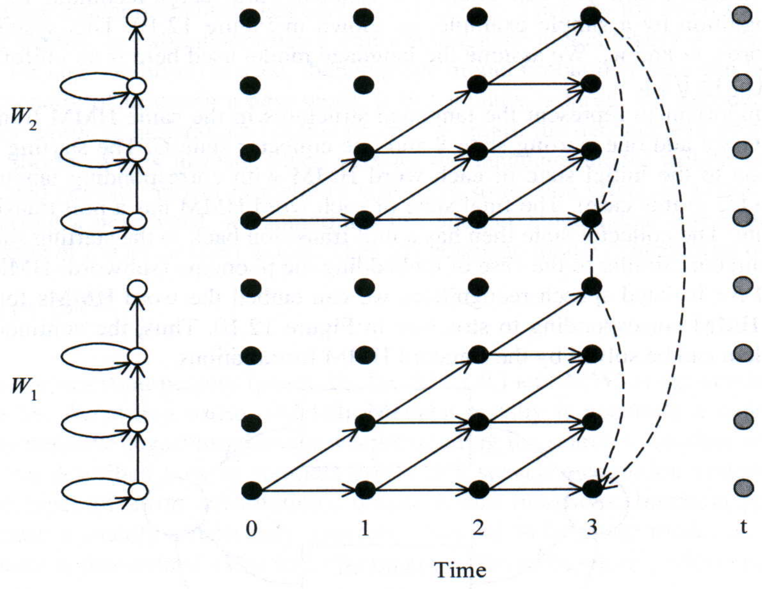


Figure 12.11 HMM trellis for continuous speech recognition example in Figure 12.10. When the final state of the word HMM is reached, a null arc (indicated by a dashed line) is linked from it to the initial state of the following word.

Continuous Speech Recognition (cont.)

- Linear Lexicon With Unigram Language Modeling

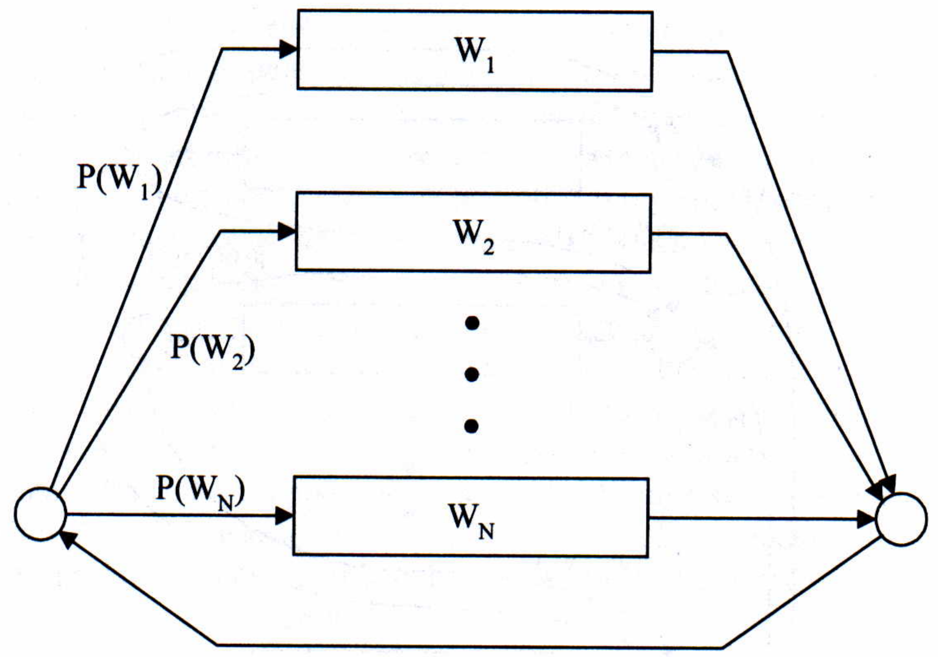


Figure 12.14 A unigram grammar network where the unigram probability is attached as the transition probability from starting state *S* to the first state of each word HMM.

Continuous Speech Recognition (cont.)

- Linear Lexicon With Bigram Language Modeling

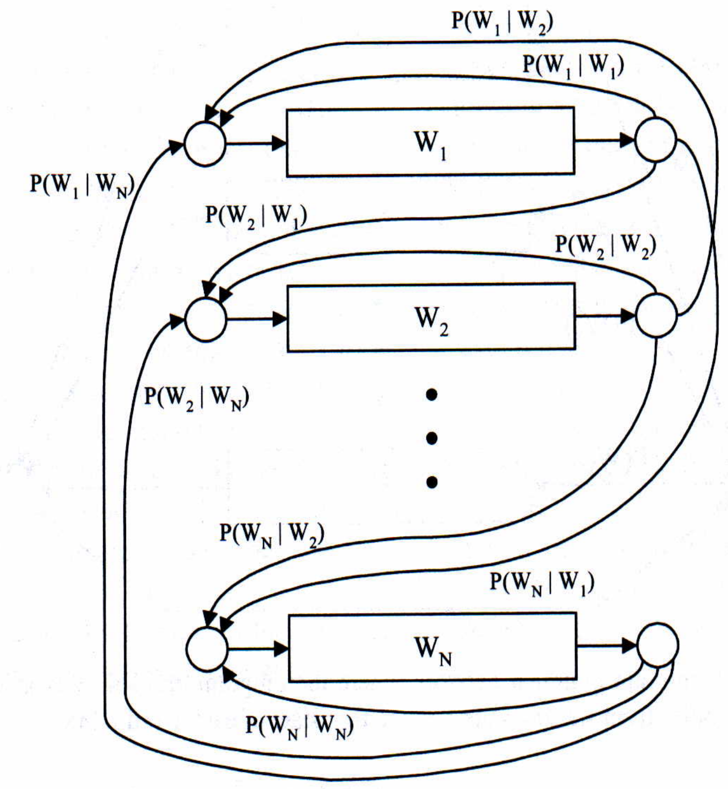


Figure 12.15 A bigram grammar network where the bigram probability $P(w_j | w_i)$ is attached as the transition probability from word w_i to w_j [19].

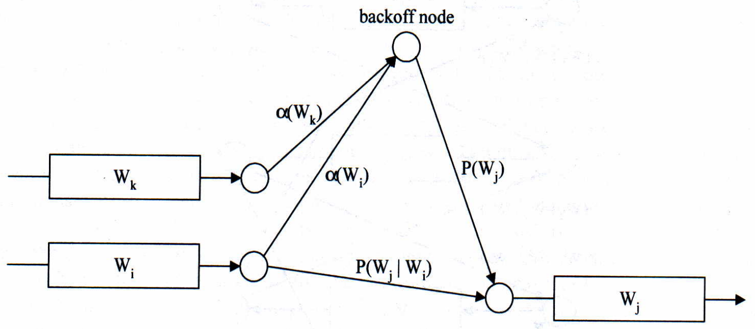


Figure 12.16 Reducing bigram expansion in a search by using the backoff node. In addition to normal bigram expansion arcs for all observed bigrams, the last state of word w_i is first connected to a central backoff node with transition probability equal to backoff weight $\alpha(w_i)$. The backoff node is then connected to the beginning of each word w_j with its corresponding unigram probability $P(w_j)$ [12].

Continuous Speech Recognition (cont.)

- Linear Lexicon With Trigram Language Modeling

language model recombination (keep only n-2 gram history distinct when recombining)

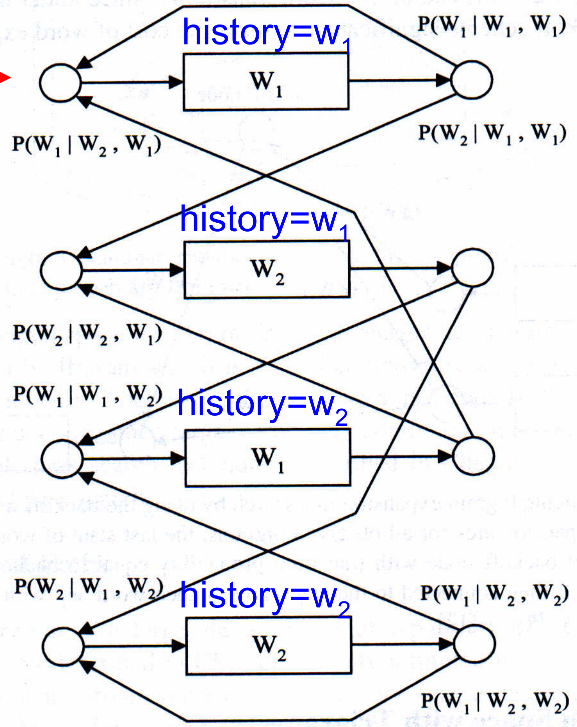


Figure 12.17 A trigram grammar network where the trigram probability $P(w_k | w_i, w_j)$ is attached to transition from grammar state word w_i, w_j to the next word w_k. Illustrated here is a two-word vocabulary, so there are four grammar states in the trigram network [19].

Further Studies on Implementation Techniques for Speech Recognition

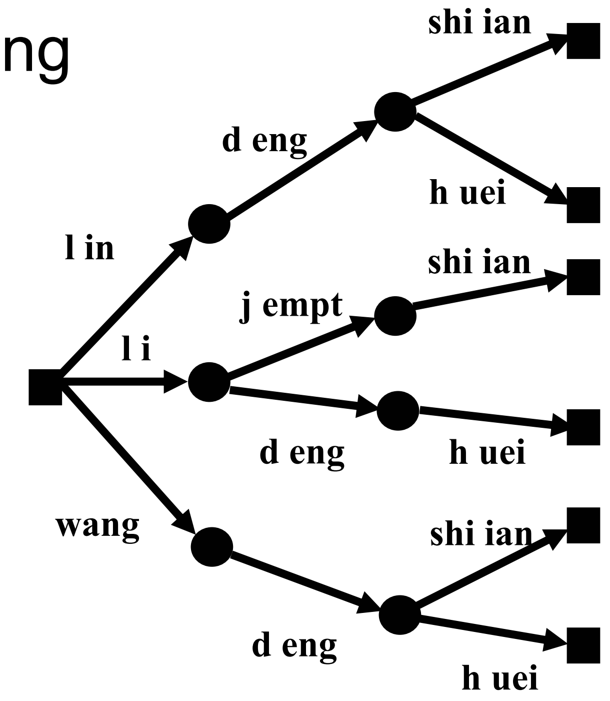
Isolated Word Recognition

Search Strategy: Beam search

- Tree Structure for Pronunciation Lexicon

- Initialization for Dynamic Programming

- Two-Level Dynamic Programming
  - Within HMM
  - Between HMMs (Arc extension)



Isolated Word Recognition

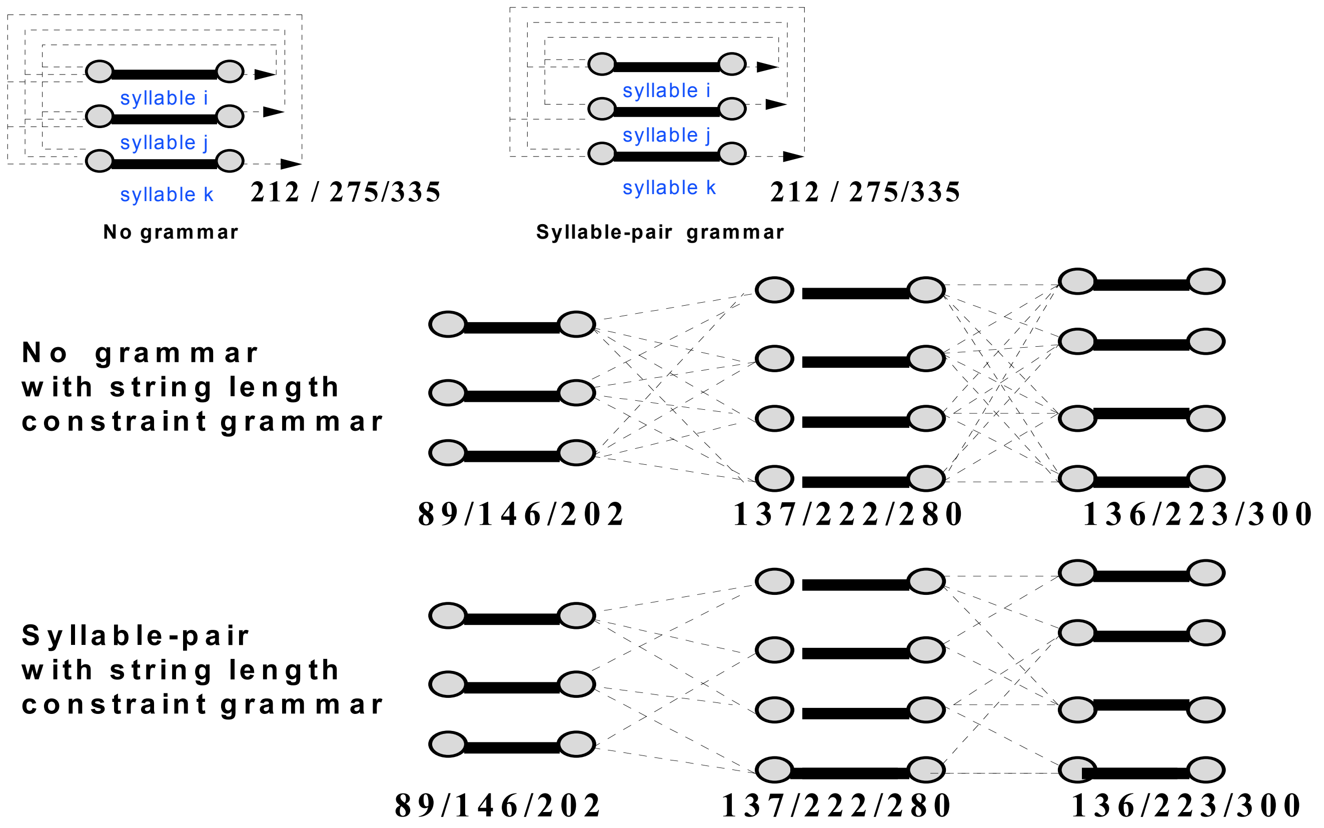
Search Strategy: A* Search

- Applied to Mandarin Isolated Word Recognition
  - Forward Trellis Search (Heuristic Scoring)
    - A forward time-synchronous Viterbi-like trellis search for generating the heuristic score
    - Using a simplified grammar network of different degree grammar type : (Over-generated Grammar)
      - No grammar
      - Syllable-pair grammar
      - No grammar with string length constraint grammar
    - Syllable-pair with string length constraint grammar
  - Backward A* Tree Search
    - A backward time-asynchronous viterbi-like A* tree search for finding the “exact” word
    - A backward syllabic tree without overgenerating the lexical vocabulary

Isolated Word Recognition

Search Strategy: A* Search (cont.)

– **Grammar Networks for Heuristic Scoring**

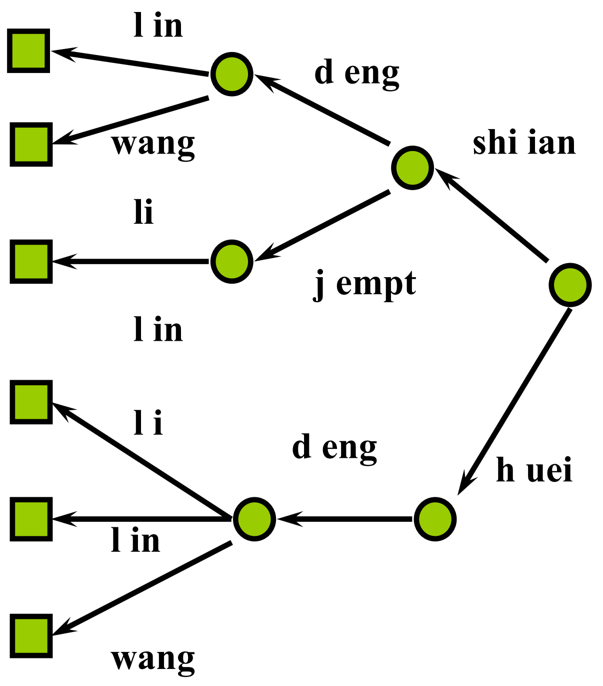


Four types of simplified grammar networks used in the tree search.

Isolated Word Recognition

Search Strategy: A* Search (cont.)

– **Backward Search Tree**



- **Steps in A* Search :**
  - At each iteration of the algorithm-
    - A sorted list (or stack) of partial paths, each with a evaluation function
  - The partial path with the highest evaluation function -
    - Expanded
    - For each one -phone(or one syllable or one arc) extensions permitted by the lexicon, the evaluation functions of the extended paths are calculated
    - And the extended partial paths are inserted into the stack at the appropriate position (sorted according to " evaluation function ")
  - The algorithm terminates -
    - When a complete path (or word) appears on the top of the stack

Keyword Spotting

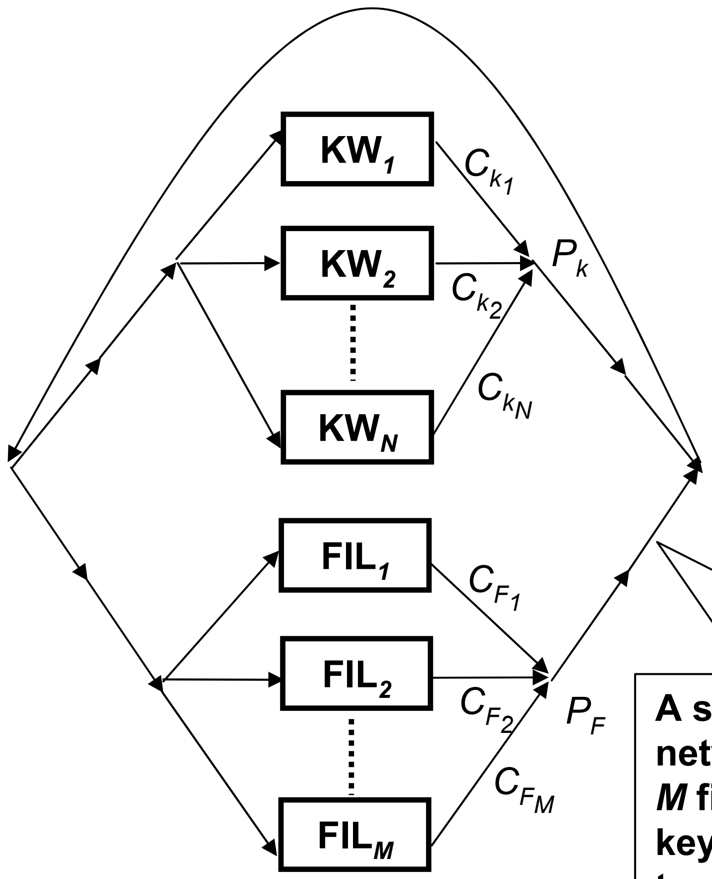
- The Common Aspect of Most Word Spotting Applications
  - It is only necessary to extract partial information from the input speech utterance
  - Many automated speech recognition problems can be loosely described by this requirement
    - *Speech message browsing*
    - *Command spotting*
    - *Telecommunications services (applications)*

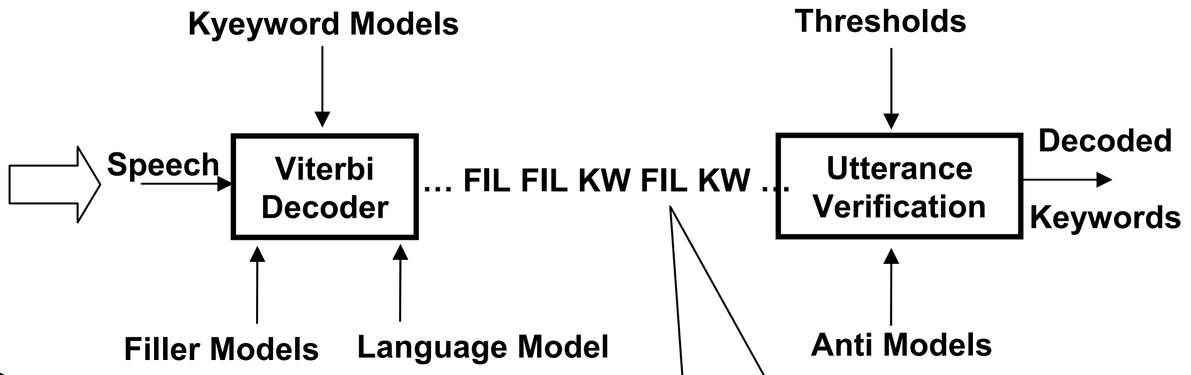
Hesitation,
Repetition,
Out-of-vocabulary words (OOV)

"Mm,...,"
"I wanna talk ..talk to.."
"What?"
幫我找台..台灣銀行的ㄟ電話

Keyword Spotting (cont.)

- General Framework of Keyword Spotting
  - Viterbi Decoding (Continuous Speech Recognition)
  - Utterance Verification (a two-stage approach)



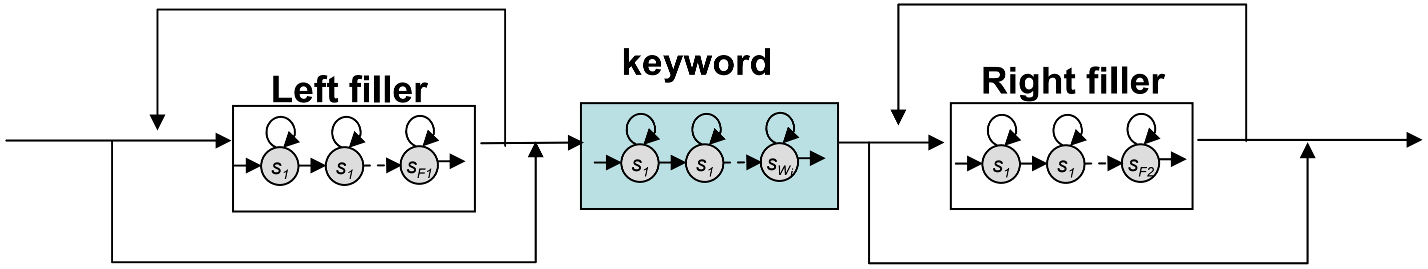


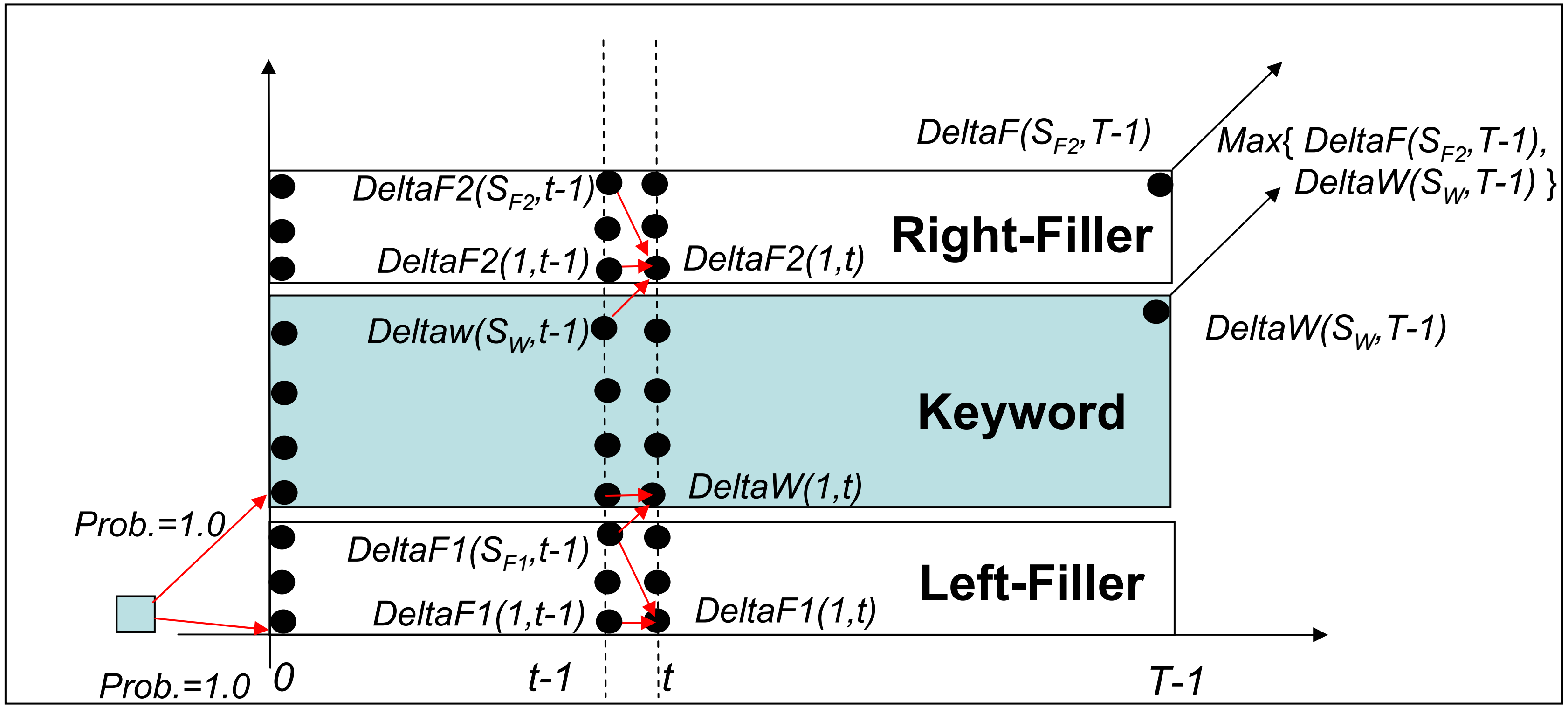
A simple, unconstrained finite state network contains *N* keywords and *M* fillers. Associated with each keyword and filler are word transition penalties.

A continuous stream of keywords and fillers.

Keyword Spotting (cont.)

- Single-keyword Spotting

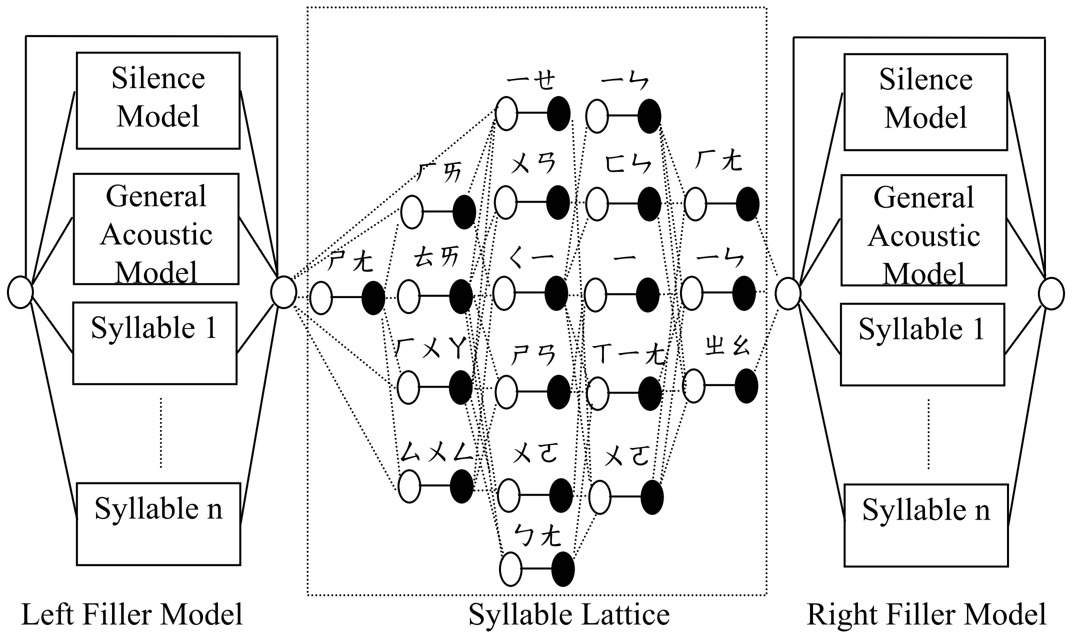




Keyword Spotting (cont.)

Case Study: A* search for Mandarin Keyword Spotting

- Search Framework
  - Forward Heuristic Scoring



$$f(t) = a \cdot sil(t) + b \cdot \overline{syl}(t) + (1-a-b) \cdot fil(t)$$

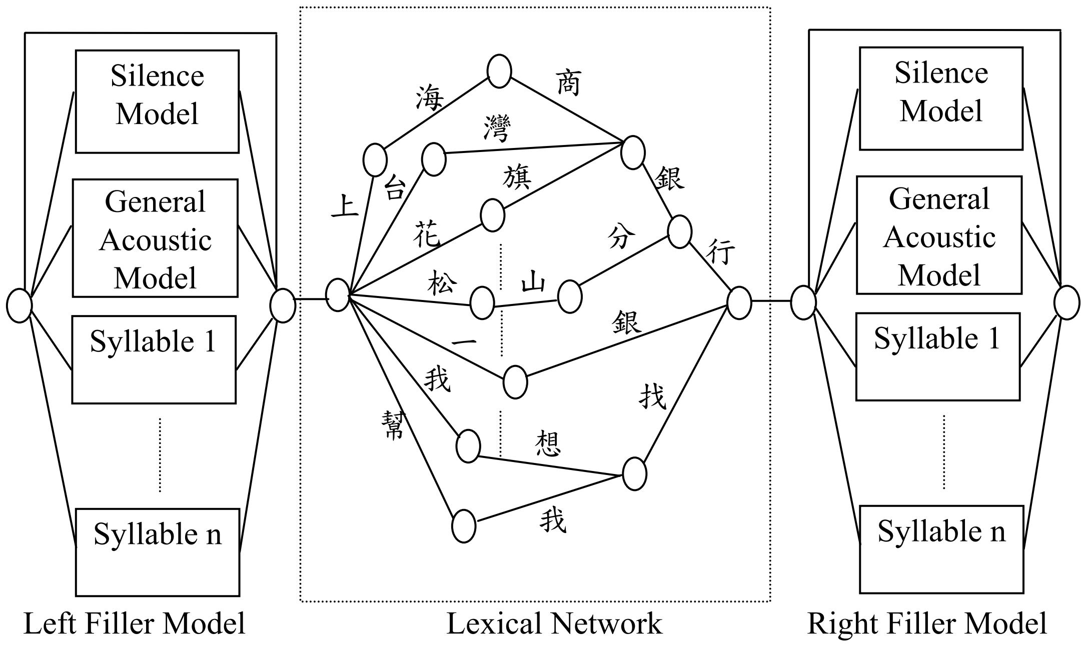
$$h^*(n_k, t) = \underset{0 \le t_1 < t}{MAX} \left[f_L(t_1) + h(n_k, t_1 + 1, t) \right]$$

The structure of the compact syllable lattice and the filler models in the first pass

Keyword Spotting (cont.)

Case Study: A* search for Mandarin Keyword Spotting

- Search Framework
  - Backward Time-Asynchronous A* Search



$$E_p(n_k) = \underset{0<t<T}{MAX}\left[d_p(n_k,t) + h^*(n_k,t-1)\right]$$

$$d_p(n_k,t) = \underset{t<t_2<T}{MAX}\left[g_p(n_k,t,t_2-1) + f_R(t_2)\right]$$

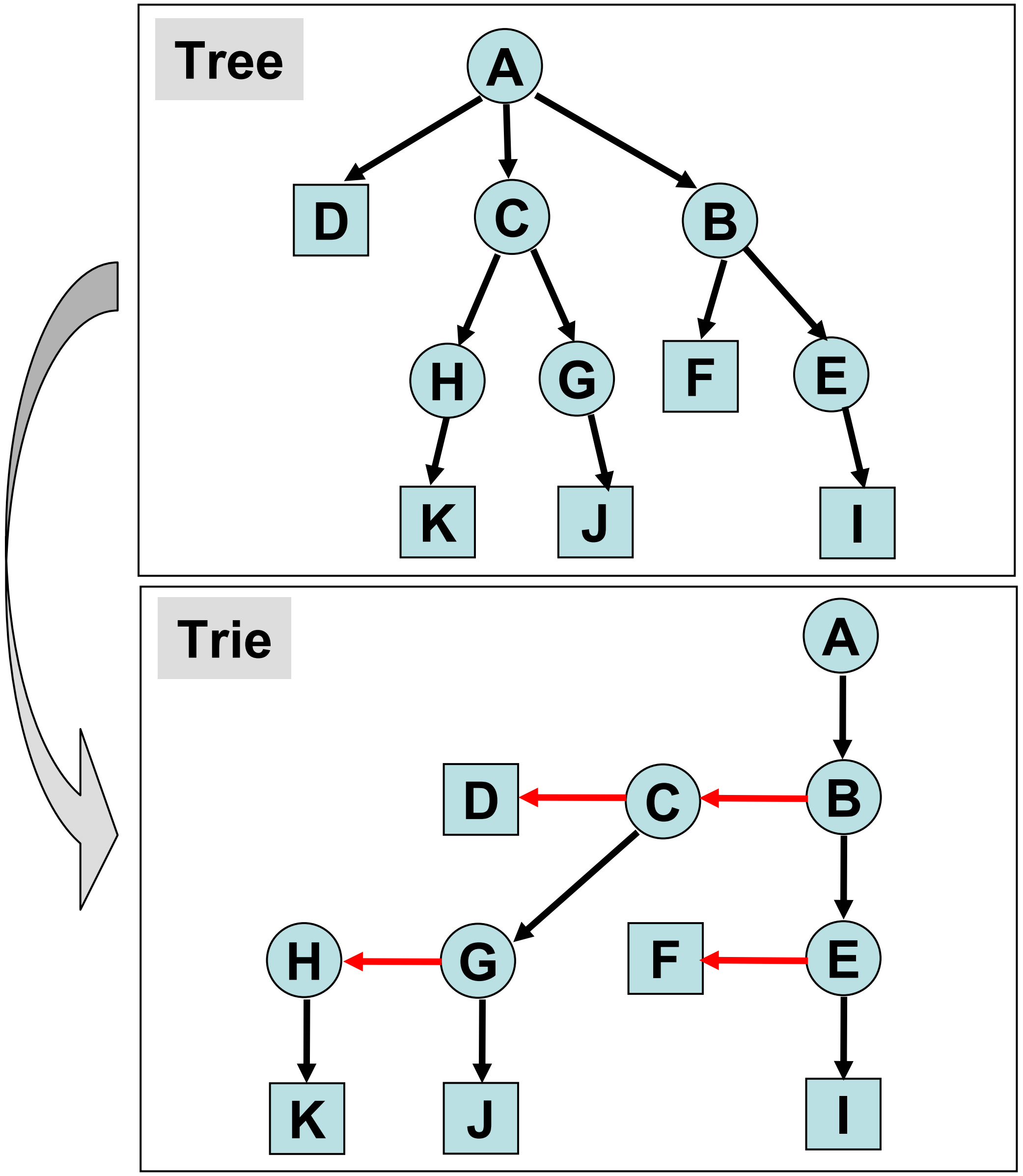
The search framework of key-phrase spotting

Data Structure for the Lexicon Tree

- Trie Structure

```
struct DEF_LEXICON_TREE
{
        short    Model_ID;
        short    WD_NO;
        int      *WD_ID;
        int       Leaf;
        struct   Tree *Child;
        struct   Tree *Brother;
        struct   Tree *Father;

};
```



Data Structure for the Lexicon Tree (cont.)

- Trie Structure

```
Do_Build_Word_Tree(int Word_Pos,int MODEL_LEN,int *Model_ID)
{
  struct Tree *ptr1,*ptr2,*ptrTmp,*TreeNew;
  int i=0,find=-1;
  ptr1=Root;
  while(i<MODEL_LEN)
  {
    ptrTmp=ptr1;   ptr1=ptr1->Child;
    if(ptr1==(struct Tree *) NULL)
    {
      TreeNew=(struct Tree *) malloc(sizeof (struct Tree));
      ptrTmp->Child=ptr1=TreeNew;
      ptr1->Brother=(struct Tree *) NULL;  ptr1->Child=(struct Tree *) NULL;
      ptr1->Father=ptrTmp; ptr1->Model_ID=Model_ID[i];
      if(i==MODEL_LEN-1)
        {
            ptr1->WD_NO=1;
            ptr1->WD_ID=(int  *) malloc((1)*sizeof(int));

        }
       else   ptr1->WD_NO=0;
     }
     else  { ………………….} ;
  }//While Loop
}//Do_Build_Tree
```

Initialization for Two-level DP for the Lexicon Tree

- Initialization: put all the 0-th states of the arcs (HMMs) connecting to the root node into the active state list

```
//-------------Initialization for DP------------
   ActiveTreeStateNo=0;
   ptrTree=Root->Child;
   while(ptrTree!=(struct Tree *) NULL)
   {
      LEX_STATE[PT1][ActiveTreeStateNo].TPTR=ptrTree;
      LEX_STATE[PT1][ActiveTreeStateNo].HMM_state=0;
      LEX_STATE[PT1][ActiveTreeStateNo].Score=(float) 0.0;
      ptrTree=ptrTree->Brother;
      ATreeState++;
   }
   //------------------------------------------
```

```
struct DEF_LEX_STATE
{
    struct  Tree *TPTR;
    short  HMM_state;
    float  Score;
};
```

Dynamic Programming: Within HMM

```
NewActiveTreeStateNo=0;
for(state_no=0;state_no<ActiveTreeStateNo;state_no++)
 {
    cur_HMM=LEX_STATE[PT1][state_no].TPTR->Model_ID; cur_state=LEX_STATE[PT1][state_no].HMM_state;
    if(cur_state!=0)
     {
       FindNewState=-1;//Global Variable
       next_state_no=Find_NewTreeState_POS(Frame_Num,state_no,cur_state,0);
        //看看LEX_STATE[PT2]是否存已在這個Tree-Node:LEX_STATE[PT1]
       if(FindNewState==1)
       {
          Cur_Score=LEX_STATE[PT1][state_no].Score+B_O[Frame_Num][cur_HMM][cur_state]
                          +Model[cur_HMM].Trans[cur_state][cur_state];
          if(Cur_Score>LEX_STATE[PT2][next_state_no].Score) LEX_STATE[PT2][next_state_no].Score=Cur_Score;
       }
     }  //if cur_state !=0
     if(cur_state<Model[cur_HMM].State-2)
     {
       FindNewState=-1;
       next_state_no=Find_NewTreeState_POS(Frame_Num,state_no,cur_state+1,1);
       if(FindNewState==1)
       {
        Cur_Score=LEX_STATE[PT1][state_no].Score+B_O[Frame_Num][cur_HMM][cur_state+1]
                     +Model[cur_HMM].Trans[cur_state][cur_state+1];
         if(Cur_Score>LEX_STATE[PT2][next_state_no].Score)LEX_STATE[PT2][next_state_no].Score=Cur_Score;
       }
     }//if cur_state<Model[cur_HMM].State-2
 }//for ActiveTreeStateNo
```

Dynamic Programming: Within HMM (cont.)

```
int Find_NewTreeState_POS(int Frame_Num,int Index,int cur_state, int type)
{
   int i,cur_HMM;
   float trans;
   if((ActiveNode_Iter=
      NewActiveTreeNodeMAP.find(Bipairx((int)LEX_STATE[PT1][Index].TPTR,cur_state)))!= NewActiveTreeNodeMAP.end())
    {
        FindNewState=1;
        return ActiveNode_Iter->second;
     }
     else
    {
        cur_HMM=LEX_STATE[PT1][Index].TPTR->Model_ID;
        if(type==0)
              trans=Model[cur_HMM].Trans[cur_state][cur_state];
        else
            trans=Model[cur_HMM].Trans[cur_state-1][cur_state];
        LEX_STATE[PT2][NewActiveTreeStateNo].TPTR=LEX_STATE[PT1][Index].TPTR;
        LEX_STATE[PT2][NewActiveTreeStateNo].HMM_state=cur_state;
        LEX_STATE[PT2][NewActiveTreeStateNo].Score=LEX_STATE[PT1][Index].Score
                   +B_O[Frame_Num][cur_HMM][cur_state]+trans;
        NewActiveTreeNodeMAP[Bipairx((int)LEX_STATE[PT2][NewActiveTreeStateNo].TPTR
              ,LEX_STATE[PT2][NewActiveTreeStateNo].HMM_state)]=NewActiveTreeStateNo;
        return NewActiveTreeStateNo++;
     }
}
```

Dynamic Programming: Within HMM (cont.)

- Pruning the HMM states with lower scores

```
 Acoustic_MAX=(float) Min_Delta;
 for(state_no=0;state_no<NewTreeStateNo;state_no++)
   if(LEX_STATE[PT2][state_no].Score>Acoustic_MAX)
       Acoustic_MAX=LEX_STATE[PT2][state_no].Score;


ActiveTreeStateNo=0;
for(state_no=0;state_no<NewTreeStateNo;state_no++)
{
   if((LEX_STATE[PT2][state_no].Score>Acoustic_MAX-Threshold)
   {
       LEX_STATE[PT1][ActiveTreeStateNo]=LEX_STATE[PT2][state_no];
       ActiveTreeStateNo++;
   }
 }
```

Dynamic Programming: Between HMMs

- Arc Extension in the Lexicon Tree

```
State_POS=ActiveTreeStateNo;

for(state_no=0;state_no<State_POS;state_no++)
 {
    cur_HMM=LEX_STATE[PT1][state_no].TPTR->Model_ID;
    cur_state=LEX_STATE[PT1][state_no].HMM_state;
    if(cur_state==Model[cur_HMM].State-2)
      {
        ptrTree=LEX_STATE[PT1][state_no].TPTR->Child;
        while(ptrTree!=(struct Tree *) NULL)
        {
          LEX_STATE[PT1][ActiveTreeStateNo].TPTR=ptrTree;
          LEX_STATE[PT1][ActiveTreeStateNo].HMM_state=0;
          LEX_STATE[PT1][ActiveTreeStateNo].Score=LEX_STATE[PT1][state_no].Score
                    +Model[cur_HMM].Trans[cur_state][cur_state+1];
          ActiveTreeStateNo++;
          ptrTree=ptrTree->Brother;
        }//while
      }
  }//for state_no
```