LECTURE 34: BASIC SEARCH ALGORITHMS

- Objectives:
  - The importance of search in speech recognition
  - General search algorithms
  - Breadth-First vs. Depth-First
  - Beam Search

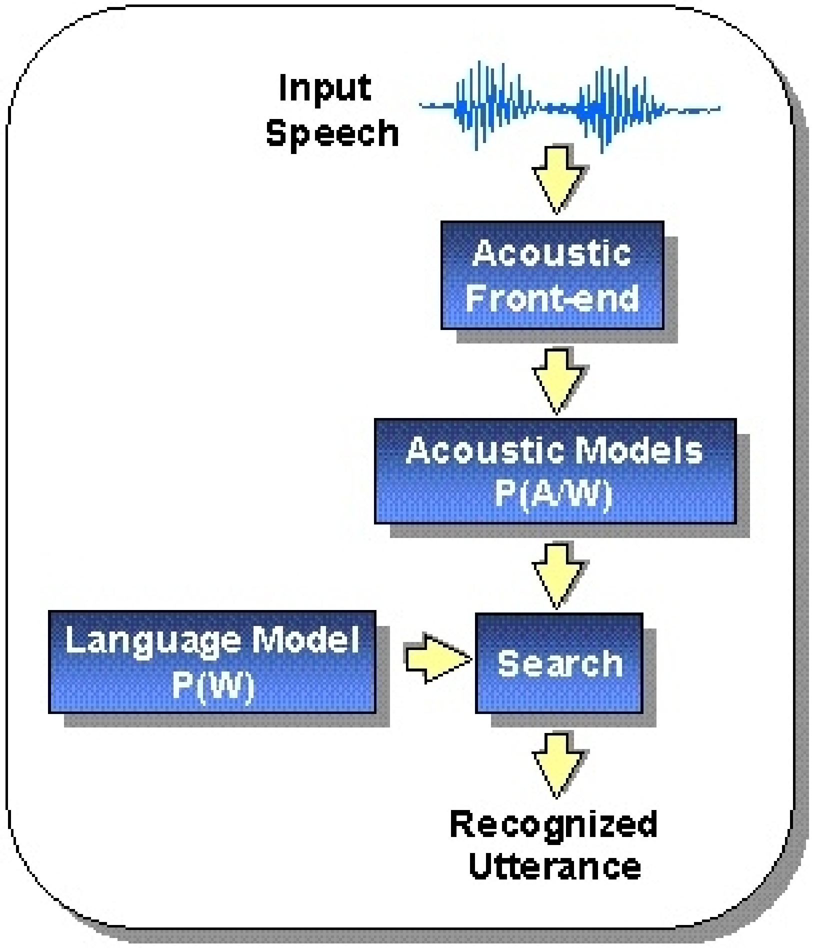
This lecture follows the course textbook closely:

> X. Huang, A. Acero, and H.W. Hon, *Spoken Language Processing - A Guide to Theory, Algorithm, and System Development*, Prentice Hall, Upper Saddle River, New Jersey, USA, ISBN: 0-13-022616-5, 2001.

This material can also be found in most computer science textbooks on algorithms:

> T. Corment, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, McGraw-Hill, New York, New York, USA, ISBN: 0-07-013151-1.

HUMAN LANGUAGE TECHNOLOGY: SPEECH RECOGNITION IS MULTIDISCIPLINARY



- Acoustic Front-End: Signal Processing
- Acoustic Models: Pattern Recognition, Linguistics
- Language Model: Natural Language Processing
- Search: Computational Linguistics, Cognitive Science

SPEECH RECOGNITION REQUIRES GOOD PATTERN RECOGNITION AND SEARCH

- Continuous speech recognition is both a pattern recognition and search problem. Why?
- The decoding process of a speech recognizer finds the most probable sequence of words given the acoustic and language models. Recall our basic equation for speech recognition:

$$P(W|A) = \frac{P(W)P(A|W)}{P(A)}$$

  Search is the process of finding the most probable word sequence:

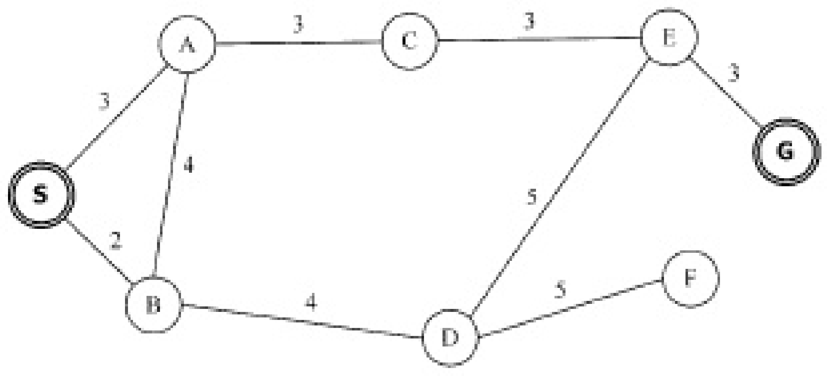
$$\hat{W} = \underset{W}{\operatorname{argmax}}\left[\frac{P(W)P(A|W)}{P(A)}\right]$$

$$= \underset{W}{\operatorname{argmax}}[P(W)P(A|W)]$$

- The complexity of the search algorithm depends heavily on the nature of the search space, which in turn, depends heavily on the language model constraints (e.g., networks vs. N-grams).
- Speech recognition typically uses a hierarchical Viterbi beam search for decoding/recognition, and A^* stack decoding for N-best and word graph generation.

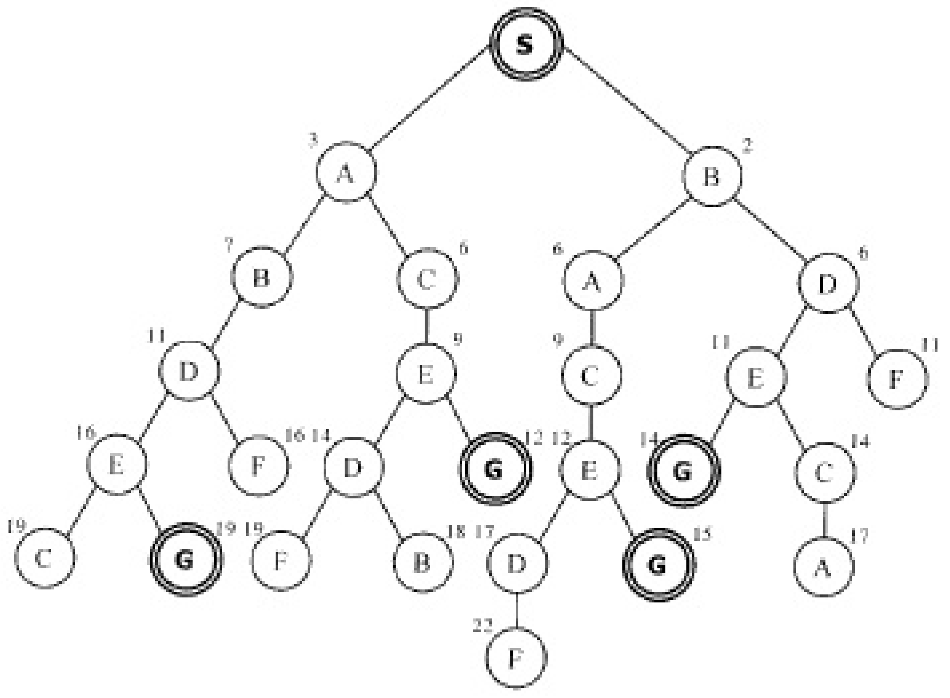
GENERAL GRAPH SEARCH

- Many interesting and useful problems cannot be handled solely by dynamic programming. For example, consider the traveling salesman problem - finding the shortest distance tour covering N cities and only visiting each city once:



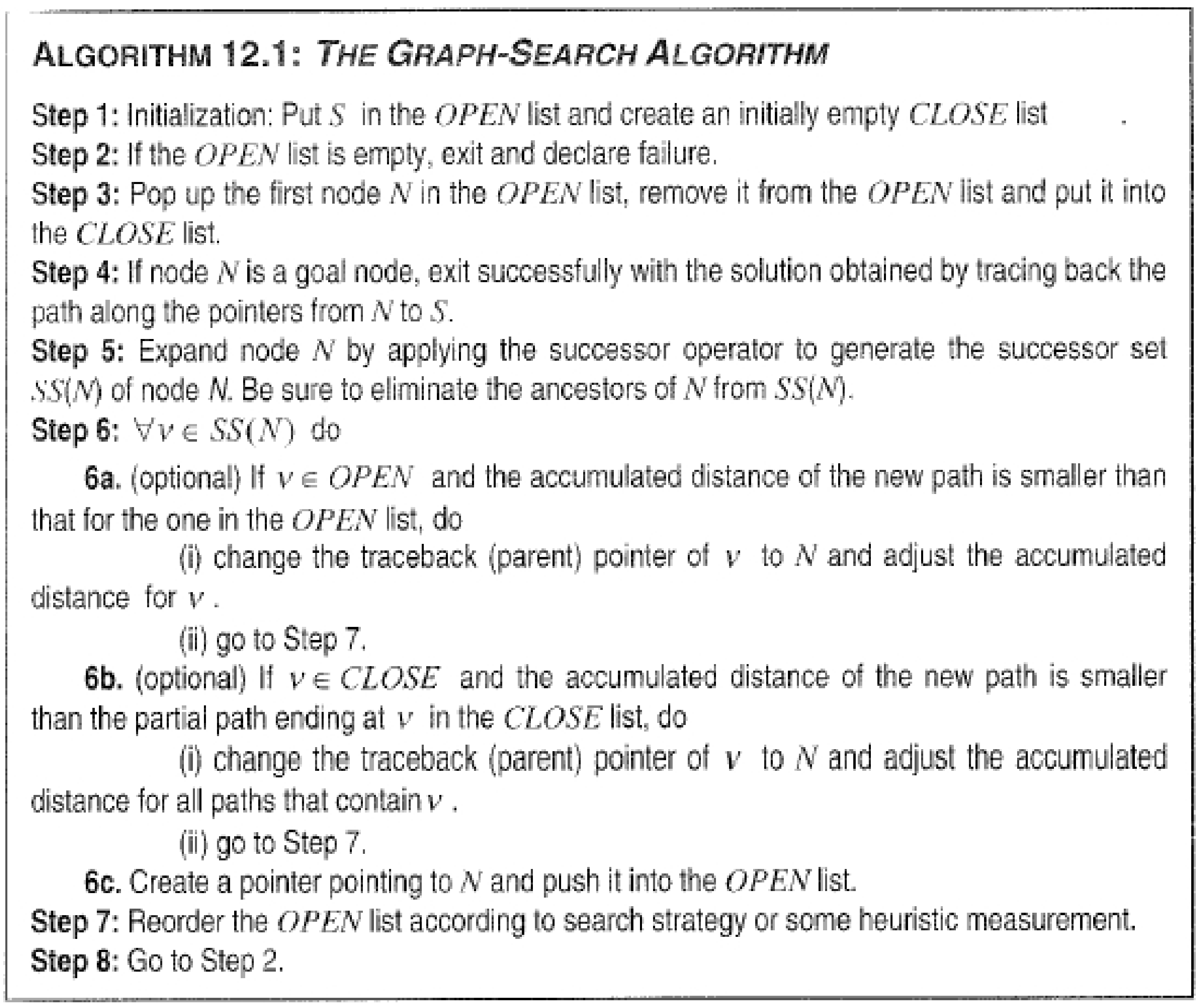
The complexity of an exhaustive search solution to such problems can be $O(N^T)$ - which is prohibitive for speech recognition.

- A search tree solution to the traveling salesman problem is show below:



- The search space is defined by a triplet (S,O,G), where S is the set of initial states, O is a set of operators or rules, and G is a set of goal states.

- A general algorithm for searching such spaces can be defined as follows:



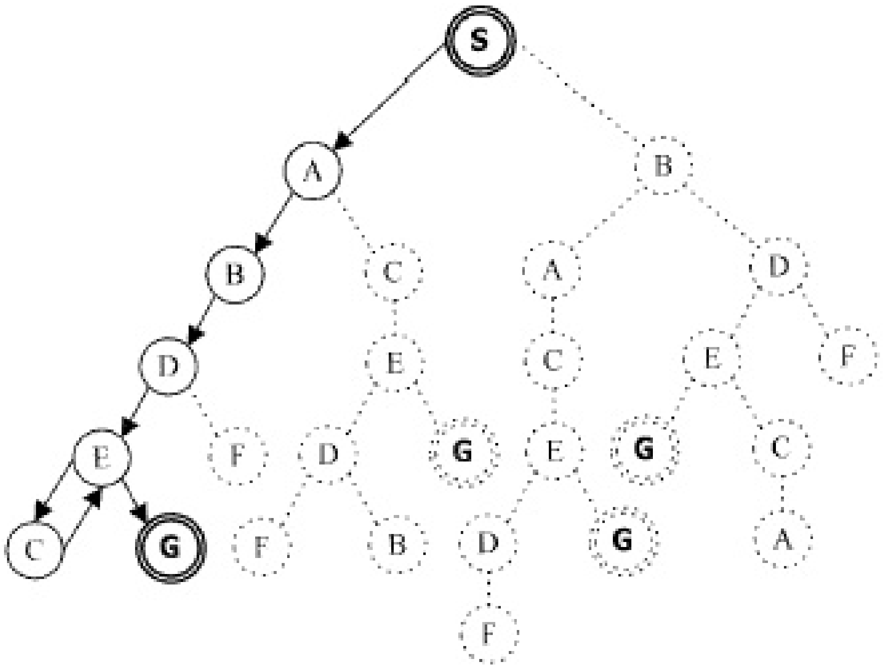
ALGORITHM 12.1: *THE GRAPH-SEARCH ALGORITHM*

```
Step 1: Initialization: Put $S$ in the $OPEN$ list and create an initially empty $CLOSE$ list .
Step 2: If the $OPEN$ list is empty, exit and declare failure.
Step 3: Pop up the first node $N$ in the $OPEN$ list, remove it from the $OPEN$ list and put it into the $CLOSE$ list.
Step 4: If node $N$ is a goal node, exit successfully with the solution obtained by tracing back the path along the pointers from $N$ to $S$.
Step 5: Expand node $N$ by applying the successor operator to generate the successor set $SS(N)$ of node N. Be sure to eliminate the ancestors of $N$ from $SS(N)$.
Step 6: $\forall v \in SS(N)$ do
    6a. (optional) If $v \in OPEN$ and the accumulated distance of the new path is smaller than that for the one in the $OPEN$ list, do
            (i) change the traceback (parent) pointer of $v$ to $N$ and adjust the accumulated distance for $v$.
            (ii) go to Step 7.
    6b. (optional) If $v \in CLOSE$ and the accumulated distance of the new path is smaller than the partial path ending at $v$ in the $CLOSE$ list, do
            (i) change the traceback (parent) pointer of $v$ to $N$ and adjust the accumulated distance for all paths that contain $v$ .
            (ii) go to Step 7.
    6c. Create a pointer pointing to $N$ and push it into the $OPEN$ list.
Step 7: Reorder the $OPEN$ list according to search strategy or some heuristic measurement.
Step 8: Go to Step 2.
```

- An important part of any search algorithm is the *successor operator* which generates the list of all possible nodes that can follow a given node, and computes the distance associated with each of these arcs.

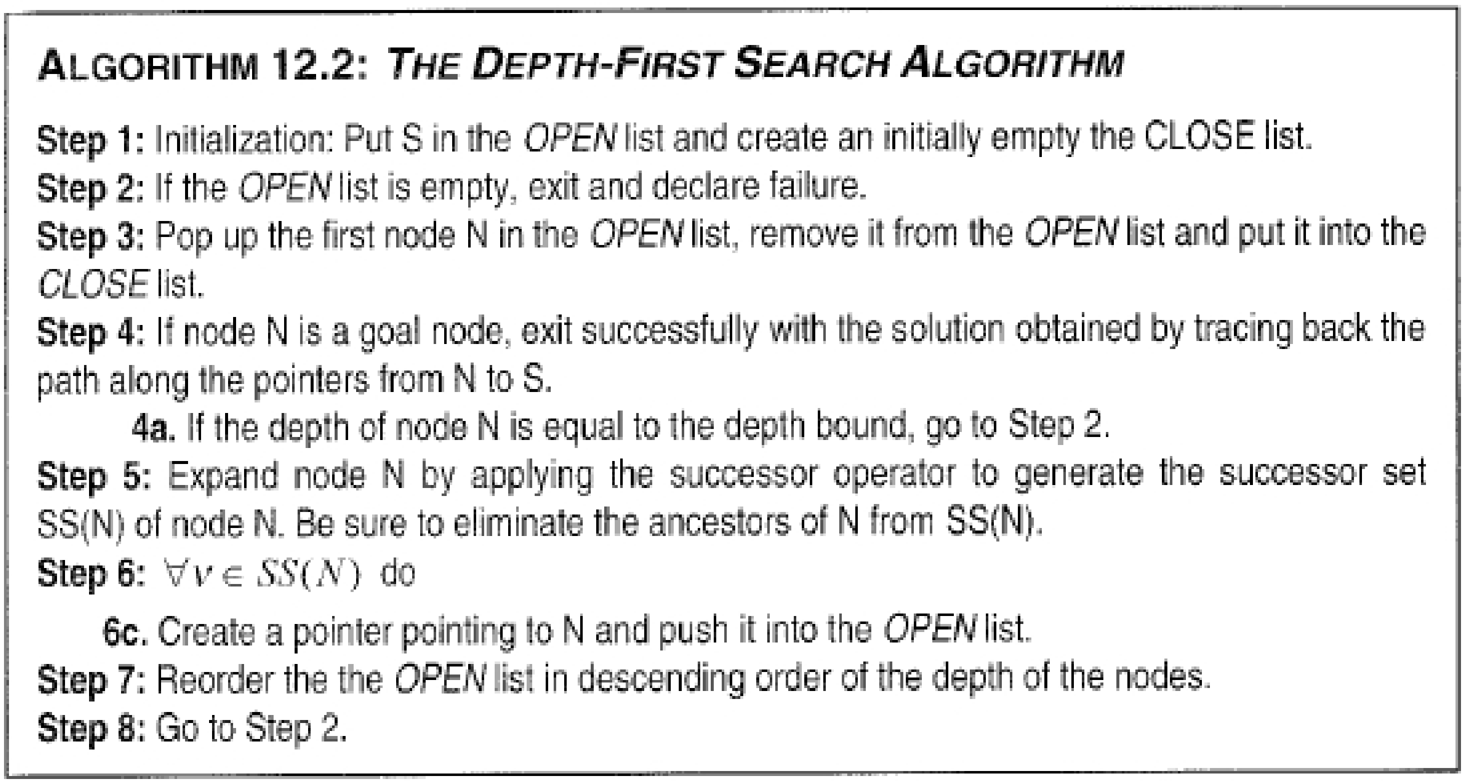
DEPTH-FIRST SEARCH

- Depth-first search explores a single path until its conclusion. If this path does not terminate on a goal state, we backtrack and arbitrarily continue with another path:



  Such a strategy is common for solving problems such as mazes where the first solution that reaches a goal state is acceptable (though this might not be the fastest solution).

- A general algorithm for searching such spaces can be defined as follows:



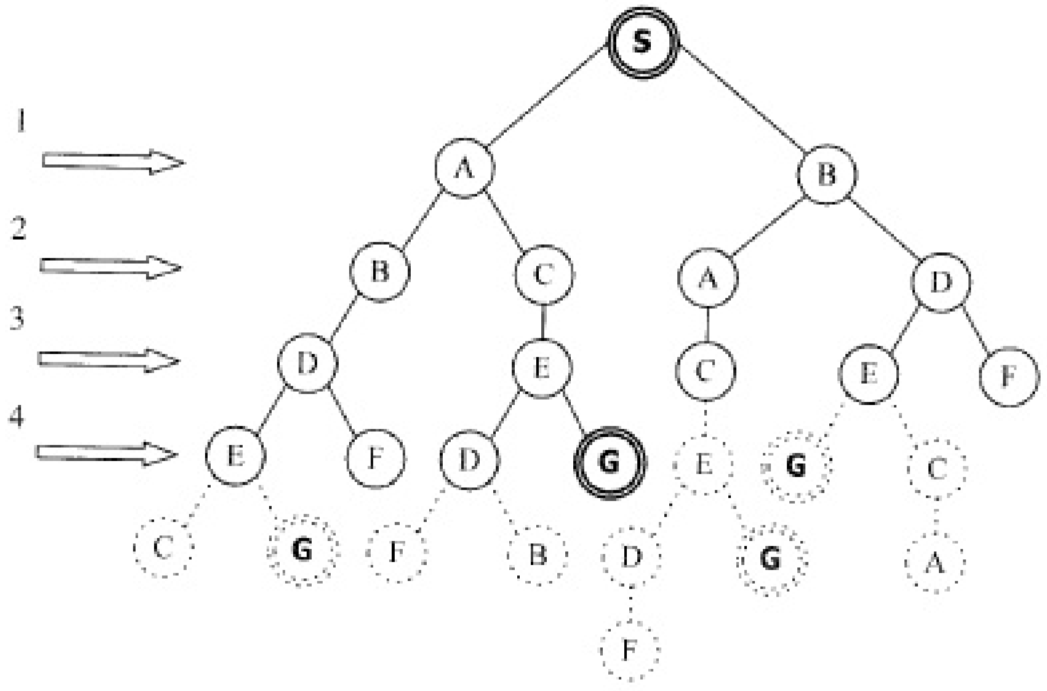
ALGORITHM 12.2: *THE DEPTH-FIRST SEARCH ALGORITHM*

Step 1: Initialization: Put S in the *OPEN* list and create an initially empty the CLOSE list.
Step 2: If the *OPEN* list is empty, exit and declare failure.
Step 3: Pop up the first node N in the *OPEN* list, remove it from the *OPEN* list and put it into the *CLOSE* list.
Step 4: If node N is a goal node, exit successfully with the solution obtained by tracing back the path along the pointers from N to S.
  4a. If the depth of node N is equal to the depth bound, go to Step 2.
Step 5: Expand node N by applying the successor operator to generate the successor set SS(N) of node N. Be sure to eliminate the ancestors of N from SS(N).
Step 6: $\forall v \in SS(N)$ do
  6c. Create a pointer pointing to N and push it into the *OPEN* list.
Step 7: Reorder the the *OPEN* list in descending order of the depth of the nodes.
Step 8: Go to Step 2.

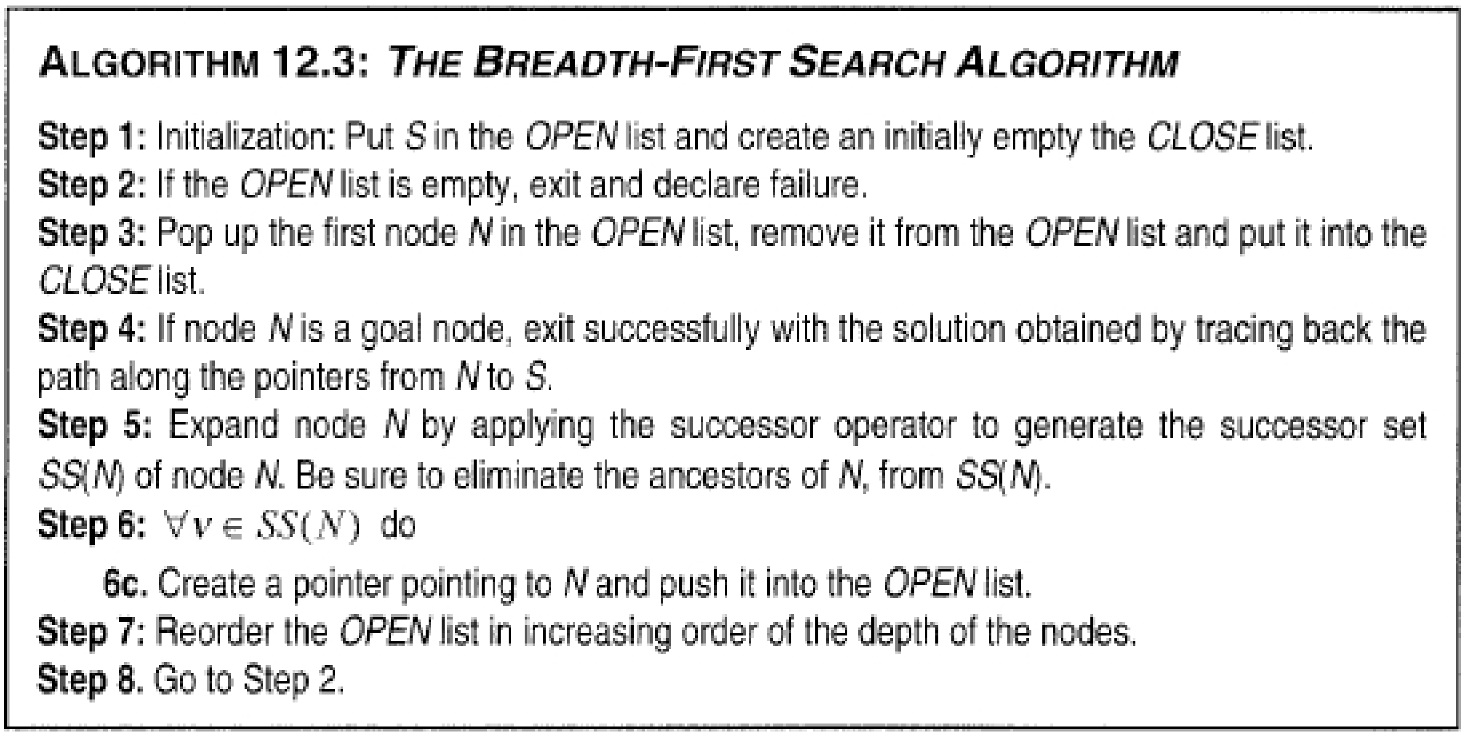
- Depth-first search, as we will see, is useful when dealing with beam search and fast-matching algorithms.

BREADTH-FIRST SEARCH

- Breadth-first search explores all alternatives simultaneously level-by-level:



- A general algorithm for searching such spaces can be defined as follows:



ALGORITHM 12.3: *THE BREADTH-FIRST SEARCH ALGORITHM*

Step 1: Initialization: Put *S* in the *OPEN* list and create an initially empty the *CLOSE* list.
Step 2: If the *OPEN* list is empty, exit and declare failure.
Step 3: Pop up the first node *N* in the *OPEN* list, remove it from the *OPEN* list and put it into the *CLOSE* list.
Step 4: If node *N* is a goal node, exit successfully with the solution obtained by tracing back the path along the pointers from *N* to *S*.
Step 5: Expand node *N* by applying the successor operator to generate the successor set *SS*(*N*) of node *N*. Be sure to eliminate the ancestors of *N*, from *SS*(*N*).
Step 6: $\forall v \in SS(N)$ do
    6c. Create a pointer pointing to *N* and push it into the *OPEN* list.
Step 7: Reorder the *OPEN* list in increasing order of the depth of the nodes.
Step 8. Go to Step 2.

- Breadth-first search is a critical part of a speech recognition system. Why?

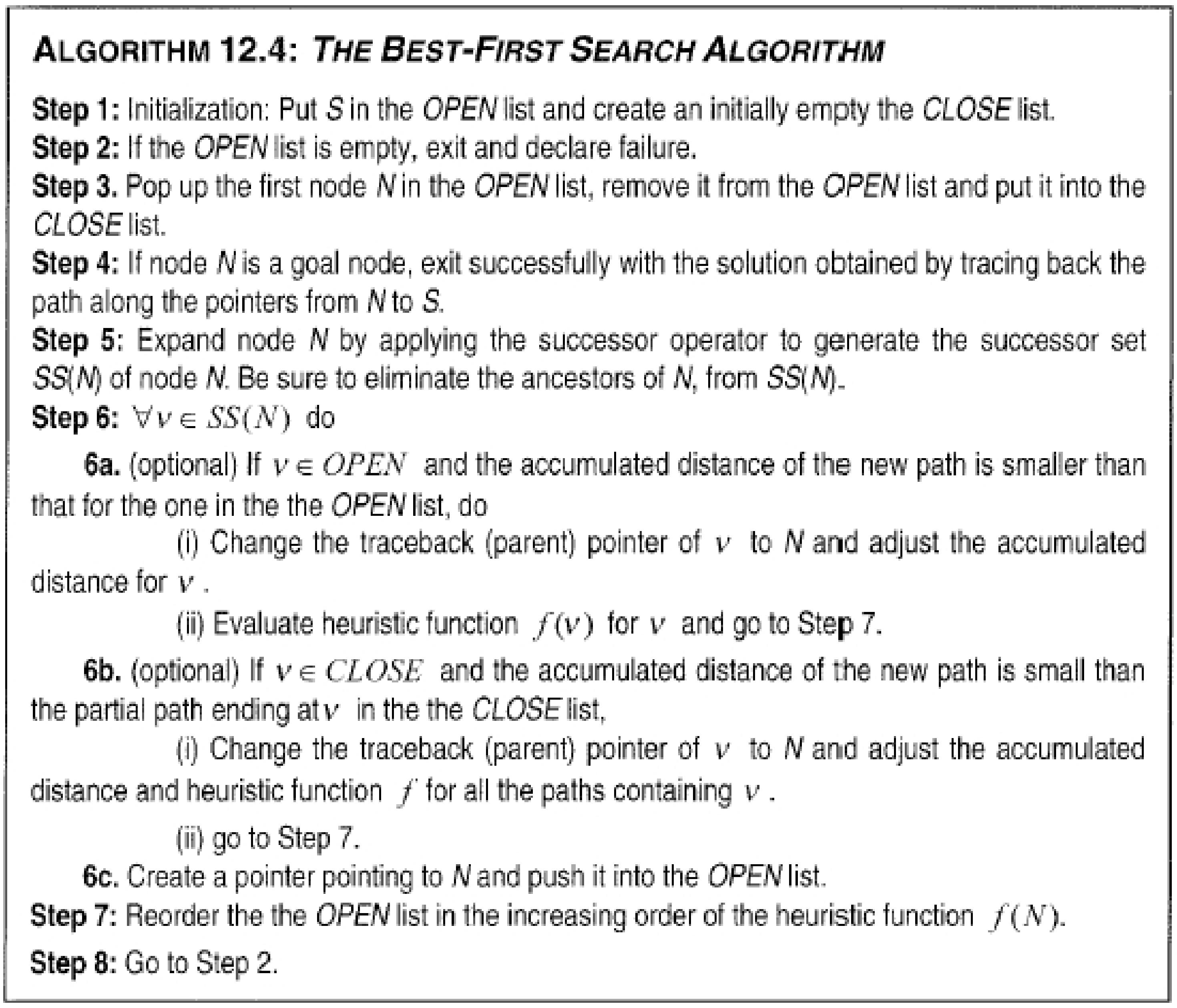
HEURISTIC SEARCH: BEST-FIRST (A^* SEARCH)

- Best-first search uses an evaluation function, h(N), which indicates the relative goodness of pursuing that node. If we combine this with the partial path score, we can define a general evaluation function:

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

which can be used to evaluate hypotheses as they evolve. If we always pursue the best path according to this evaluation function, what are the merits of this approach? What constraints must be placed on this function to guarantee an optimal solution? How would that solution compare to other search algorithms?

- A general algorithm for searching such spaces can be defined as follows:



ALGORITHM 12.4: ***THE BEST-FIRST SEARCH ALGORITHM***

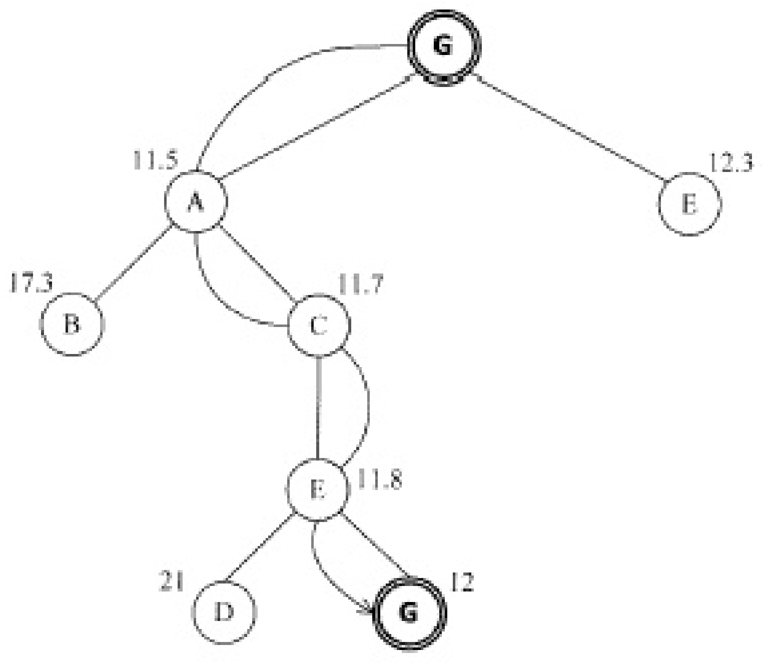
Step 1: Initialization: Put *S* in the *OPEN* list and create an initially empty the *CLOSE* list.
Step 2: If the *OPEN* list is empty, exit and declare failure.
Step 3. Pop up the first node *N* in the *OPEN* list, remove it from the *OPEN* list and put it into the *CLOSE* list.
Step 4: If node *N* is a goal node, exit successfully with the solution obtained by tracing back the path along the pointers from *N* to *S*.
Step 5: Expand node *N* by applying the successor operator to generate the successor set *SS*(*N*) of node *N*. Be sure to eliminate the ancestors of *N*, from *SS*(*N*).
Step 6: $\forall v \in SS(N)$ do

6a. (optional) If $v \in OPEN$ and the accumulated distance of the new path is smaller than that for the one in the the *OPEN* list, do
(i) Change the traceback (parent) pointer of v to *N* and adjust the accumulated distance for v.
(ii) Evaluate heuristic function $f(v)$ for v and go to Step 7.

6b. (optional) If $v \in CLOSE$ and the accumulated distance of the new path is small than the partial path ending at v in the the *CLOSE* list,
(i) Change the traceback (parent) pointer of v to *N* and adjust the accumulated distance and heuristic function f for all the paths containing v.
(ii) go to Step 7.

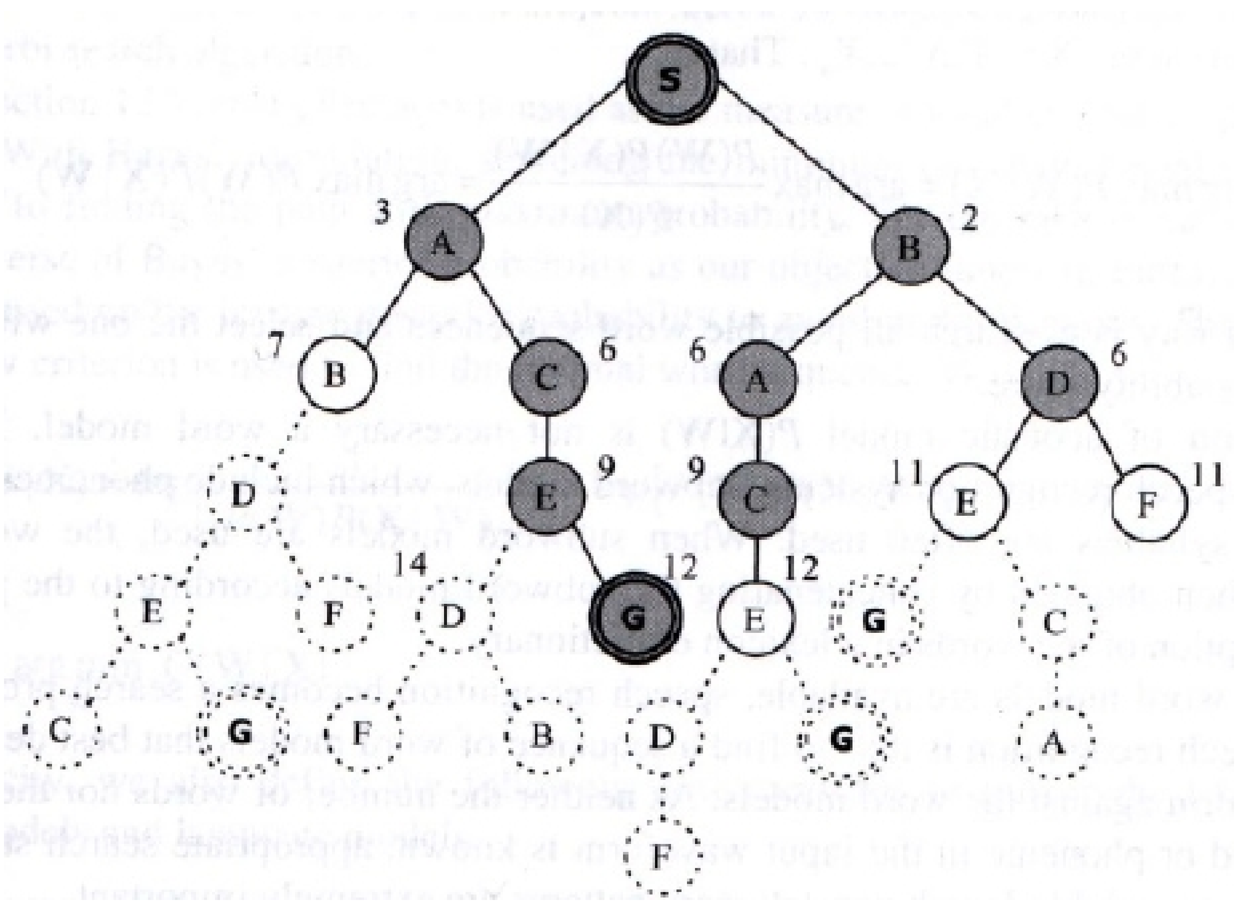
6c. Create a pointer pointing to *N* and push it into the *OPEN* list.
Step 7: Reorder the the *OPEN* list in the increasing order of the heuristic function $f(N)$.
Step 8: Go to Step 2.

- A search algorithm is said to be *admissible* if it can guarantee an optimal solution.
- One possible solution to the traveling salesman problem using best-first search might look like this:

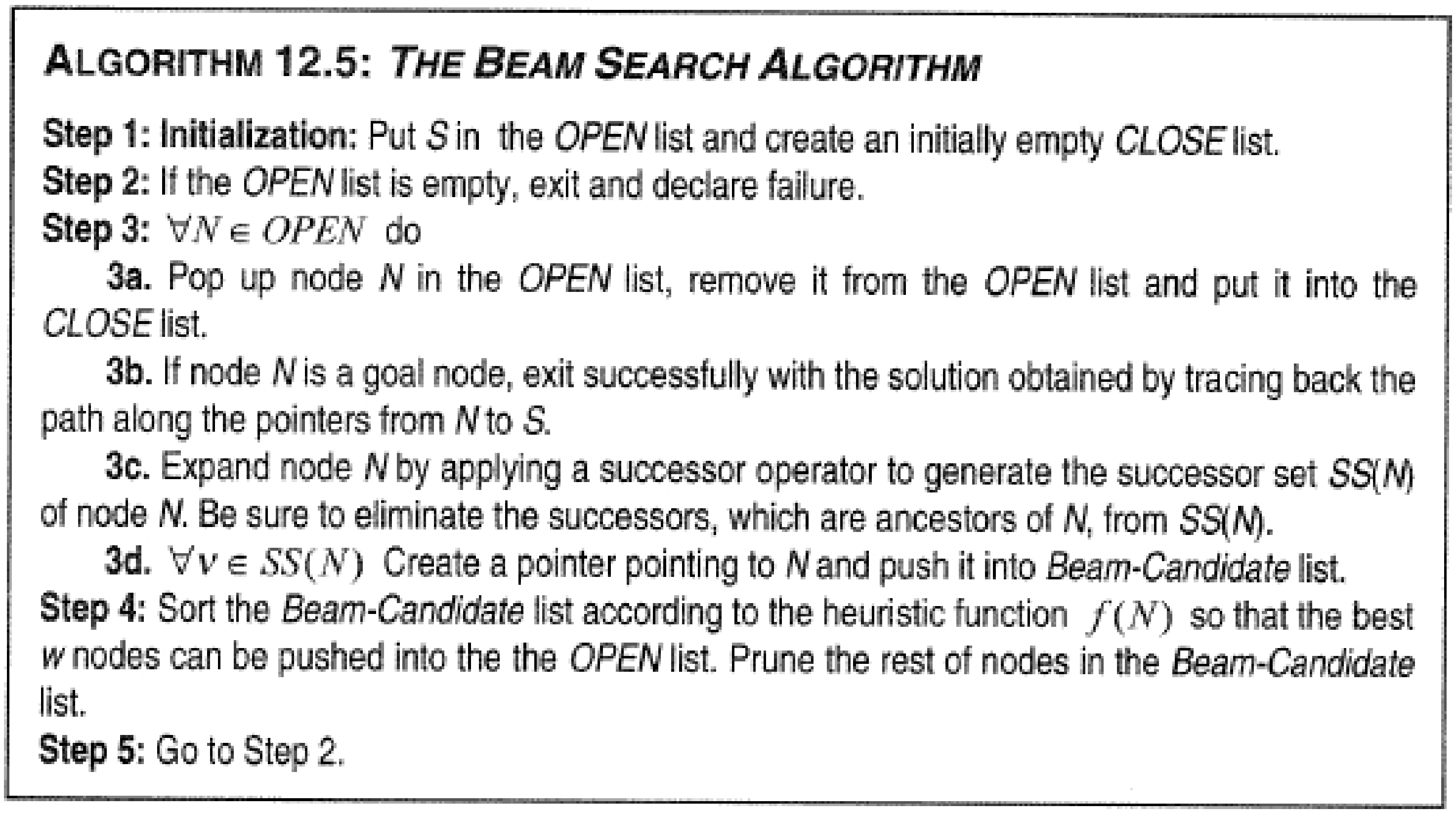


HEURISTIC SEARCH: BEAM SEARCH

- Beam search is another form of heuristic search in which we terminate hypotheses that do not appear to be promising:



- A general algorithm for searching such spaces can be defined as follows:



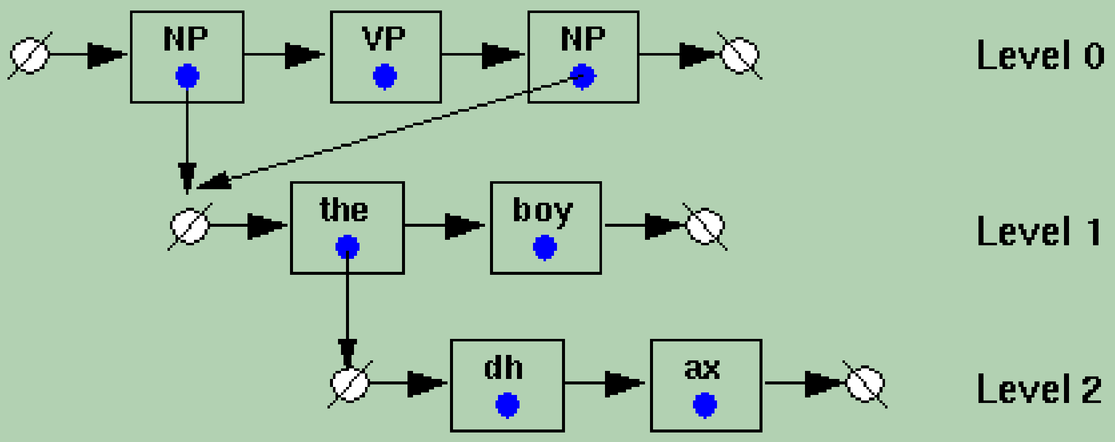
ALGORITHM 12.5: *THE BEAM SEARCH ALGORITHM*

Step 1: Initialization: Put *S* in the *OPEN* list and create an initially empty *CLOSE* list.
Step 2: If the *OPEN* list is empty, exit and declare failure.
Step 3: $\forall N \in OPEN$ do
3a. Pop up node *N* in the *OPEN* list, remove it from the *OPEN* list and put it into the *CLOSE* list.
3b. If node *N* is a goal node, exit successfully with the solution obtained by tracing back the path along the pointers from *N* to *S*.
3c. Expand node *N* by applying a successor operator to generate the successor set *SS*(*N*) of node *N*. Be sure to eliminate the successors, which are ancestors of *N*, from *SS*(*N*).
3d. $\forall v \in SS(N)$ Create a pointer pointing to *N* and push it into *Beam-Candidate* list.
Step 4: Sort the *Beam-Candidate* list according to the heuristic function $f(N)$ so that the best *w* nodes can be pushed into the the *OPEN* list. Prune the rest of nodes in the *Beam-Candidate* list.
Step 5: Go to Step 2.

- Why is beam search very appropriate for speech recognition?

HIERARCHICAL SEARCH: OPTIMAL SEARCH IS SUBTLE

Search Space Specification: DiGraph<SearchNode>



Search Level Specifications: Vector<SearchLevel>

Level 0: Symbols = “NP”, “VP”
beam prune = true
beam width = 1000
Level 1: Symbols = “the”, “boy”, “ran”, ...
use Nsymbol probabilities
Nsymbol length = 3
Level 2: Symbols = “dh”, “ax”, “b”, “oy”, ...
use context dependency
context = 1 on left, 1 on right

Current Search Paths: Vector<DoubleLinkedList<Trace> >

Level 0 Traces	Trace (NP, VP)
Level 1 Traces	Trace (NP, the, dh, ax, boy)
Level 2 Traces	Trace (NP, the, dh)

- To maintain optimality in the search, we must maintain a history of predecessor words *and* states, since the same word sequence can be produced by multiple paths in the network.
- Dynamic expansion of context is generally preferred over precompilation. Why?