CHAPTER 12

Basic Search Algorithms

Continuous speech recognition (CSR) is both a pattern recognition and search problem. As described in previous chapters, the acoustic and language models are built upon a statistical pattern recognition framework. In speech recognition, making a search decision is also referred to as decoding. In fact, decoding got its name from information theory (see chapter 3) where the idea is to *decode* a signal that has presumably been encoded by the source process and has been transmitted through the communication channel, as depicted in Figure 1.1. In this chapter, we first review the general decoder architecture that is based on such a source-channel model.

The decoding process of a speech recognizer is to find a sequence of words whose corresponding acoustic and language models best match the input signal. Therefore, the process of such a decoding process with trained acoustic and language models is often referred to as just a *search* process. Graph search algorithms have been explored extensively in the fields of artificial intelligence, operation research, and game theory. In this chapter we present several basic search algorithms, which serve as the basic foundation for CSR.

The complexity of a search algorithm is highly correlated with the search space, which is determined by the constraints imposed by the language models. We discuss the impact of different language models, including finite-state grammars, context-free grammars, and *n*-grams.

Speech recognition search is usually done with the Viterbi or A* stack decoders. The reasons for choosing the Viterbi decoder involve arguments that point to speech as a left-to-right process and to the efficiencies afforded by a time-synchronous process. The reasons for choosing a stack decoder involve its ability to more effectively exploit the A* criteria, which holds out the hope of performing an optimal search as well as the ability to handle huge search spaces. Both algorithms have been successfully applied to various speech recognition systems. The relative meritsof both search algorithms were quite controversial in the 1980s. Lately, with the help of efficient pruning techniques, Viterbi beam search (described in detail in Chapter 13) has been the preferred method for almost all speech recognition tasks. Stack decoding, on the other hand, remains an important strategy to uncover the n-best and lattice structures.

12.1. BASIC SEARCH ALGORITHMS

Search is a subject of interest in artificial intelligence and has been well studied for expert systems, game playing, and information retrieval. We discuss several general graph search methods that are fundamental to spoken language systems. Although the basic concept of graph search algorithms is independent of any specific task, the efficiency often depends on how we exploit domain-specific knowledge.

The idea of search implies moving around, examining things, and making decisions about whether the sought object has yet been found. In general, search problems can be represented using the *state-space search* paradigm. It is defined by a triplet (*S, O, G*), where *S* is a set of initial states, *O* a set of operators (or rules) applied on a state to generate a transition with its corresponding cost to another state, and *G* a set of goal states. A solution in the state-space search paradigm consists in finding a path from an initial state to a goal state. The state-space representation is commonly identified with a directed graph in which each node corresponds to a state and each arc to an application of an operator (or a rule), which transitions from one state to another. Thus the state-space search is equivalent to searching through the graph with some objective function.

Before we present any graph search algorithms, we need to remind the readers of the importance of the dynamic programming algorithm described in Chapter 8. Dynamic programming should be applied whenever possible and as early as possible because (1) unlike any heuristics, it will not sacrifice optimality; (2) it can transform an exponential search into a polynomial search.

12.1.1. General Graph Searching Procedures

Although dynamic programming is a powerful polynomial search algorithm, many interesting problems cannot be handled by it. A classical example is the traveling-salesman's prob-

lem. We need to find a shortest-distance tour, starting at one of many cites, visiting each city exactly once, and returning to the starting city. This is one of the most famous problems in the *NP*-hard class [1, 32]. Another classical example is the *N*-queens problem (typically 8-queens), where the goal is to place *N* queens on an $N \times N$ chessboard in such a way that no queen can capture any other queen; i.e., there is no more than one queen in any given row, column, or diagonal. Many of these puzzles have the same characteristics. As we know, the best algorithms currently known for solving the *NP*-hard problem are exponential in the problem size. Most graph search algorithms try to solve those problems using heuristics to avoid or moderate such a combinatorial explosion.

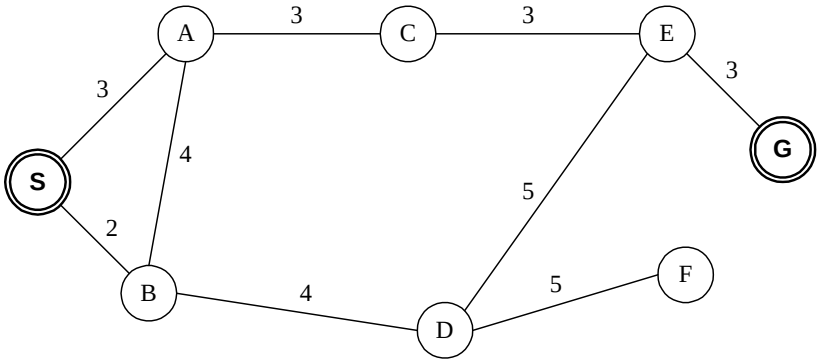

Figure 12.1 A highway distance map for cities S, A, B, C, D, E, F and G. The salesman needs to find a path to travel from city S to city G [42].

Let's start our discussion of graph search procedure with a simple city-traveling problem [42]. Figure 12.1 shows a highway distance map for all the cities. A salesman John needs to travel from the starting city S to the end city G. One obvious way to find a path is to derive a graph that allows orderly exploration of all possible paths. Figure 12.2 shows the graph that traces out all possible paths in the city-distance map shown in Figure 12.1. Although the city-city connection is bi-directional, we should note that the search graph in this case must not contain cyclic paths, because they would not lead to any progress in this scenario.

If we define the search space as the potential number of nodes (states) in the graph search procedure, the search space for finding the optimal state sequence in the Viterbi algorithm (described in Chapter 8) is $N \times T$, where *N* is the number of states for the HMM and *T* is length of the observation. Similary, the search space for John's traveling problem will be 27.

Another important measure for a search graph is the *branching factor*, defined as the average number of successors for each node. Since the number of nodes of a search graph (or tree) grows exponentially with base equal to this branching factor, we certainly need to watch out for search graphs (or trees) with a large branching factor. Sometimes they can be too big to handle (even infinite, as in game playing). We often trade the optimal solution for improved performance and feasibility. That is, the goal for such search problems is to find

one satisfactory solution instead of the optimal one. In fact, most AI (artifical intelligence) search problems belong to this category.

The search tree in Figure 12.2 may be implemented either explicitly or implicitly. In an explicit implementation, the nodes and arcs with their corresponding distances (or costs) are explicitly specified by a table. However, an explicit implementation is clearly impractical for large search graphs and impossible for those with infinite nodes. In practice, most parts of the graph may never be explored before a solution is found. Therefore, a sensible strategy is to dynamically generate the search graph. The part that becomes explicit is often referred to as an *active* search space. Throughout the discussion here, it is important to keep in mind this distinction between the implicit search graph that is specified by the start node S and the explicit partial search graphs that are actually constructed by the search algorithm.

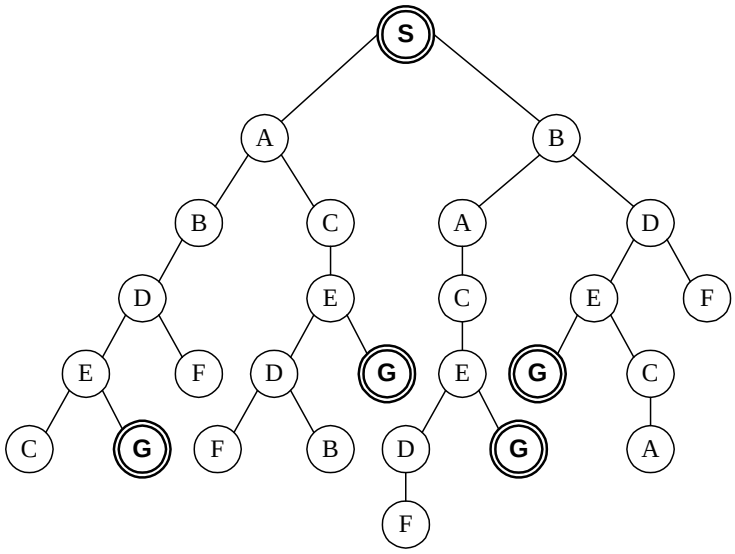

Figure 12.2 The search tree (graph) for the salesman problem illustrated in Figure 12.1. The number next to each goal nodes is the accumulated distance from start city to end city [42].

To expand the tree, the term *successor operator* (or *move generator*, as it is often called in game search) is defined as an operator that is applied to a node to generate all of the successors of that node and to compute the distance associated with each arc. The successor operator obviously depends on the topology (or rules) of the problem space. Expanding the starting node *S*, and successors of *S*, ad infinitum, gradually makes the implicitly defined graph explicit. This recursive procedure is straightforward, and the search graph (tree) can be constructed without the extra book-keeping. However, this process would only generate a search tree where the same node might be generated as a part of several possible paths.

For example, node *E* is being generated in four different paths. If we are interested in finding an optimal path to travel from *S* to *G*, it is more efficient to merge those different paths that lead to the same node *E*. We can pick the shortest path up to *C*, since everything

following *E* is the same for the rest of the paths. This is consistent with the dynamic programming principle – when looking for the best path from *S* to *G*, all partial paths from *S* to any node *E*, other than the best path from *S* to *E*, should be discarded. The dynamic programming merge also eliminates cyclic paths implicitly, since a cyclic path cannot be the shortest path. Performing this extra bookkeeping (merging different paths leading into the same node) generates a search graph rather than a search tree.

Although a graph search has the potential advantage over a tree search of being more efficient, it does require extra bookkeeping. Whether this effort is justified depends on the individual problem one has to address.

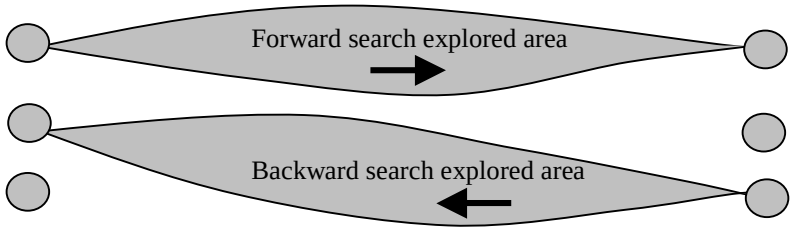

Figure 12.3 A bad case for bi-directional search, where the forward search and the backward search crossed each other [42].

Most search strategies search in a forward direction, i.e., build the search graph (or tree) by starting with the initial configuration (the starting state *S*) from the root. In the general AI literature, this is referred to as *forward reasoning* [43], because it performs rule-base reasoning by matching the left side of rules first. However, for some specific problem domains, it might be more efficient to use *backward reasoning* [43], where the search graph is built from the bottom up (the goal state *G*). Possible scenarios include:

- *There are more initial states than goal states*. Obviously it is easy to start with a small set of states and search for paths leading to one of the bigger sets of states. For example, suppose the initial state *S* is the hometown for John in the city-traveling problem in Figure 12.1 and the goal state *G* is an unfamiliar city for him. In the absence of a map, there are certainly more locations (neighboring cities) that John can identify as being close[1] to his home city *S* than those he can identify as being close to an unfamiliar location. In a sense, all of those locations being identified as close to John's home city *S* are equivalent to the initial state *S*. This means John might want to consider reasoning backward from the unfamiliar goal city *G* for the trip planning.

[1] *Being close* means that, once John reaches one of those neighboring cities, he can easily remember the best path to return home. It is similar to the killer book for chess play. Once the player reaches a particular board configuration, he can follow the killer book for moves that can guarantee a victory.

- *The branching factor for backward reasoning is smaller than that for forward reasoning*. In this case it makes sense to search in the direction with lower branching factor.

It is in principle possible to search from both ends simultaneously, until two partial paths meet somewhere in the middle. This strategy is called *bi-directional search* [43]. Bi-directional search seems particularly appealing if the number of nodes at each step grows exponentially with the depth that needs to be explored. However, sometimes bi-directional search can be devastating. The two searches may cross each other, as illustrated in Figure 12.3.

ALGORITHM 12.1: THE GRAPH-SEARCH ALGORITHM

1. Initialization: Put S in the *OPEN* list and create an initially empty *CLOSE* list
2. If the *OPEN* list is empty, exit and declare failure.
3. Pop up the first node N in the *OPEN* list, remove it from the *OPEN* list and put it into the *CLOSE* list.
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.
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 $SS(N)$.
6. $\forall \nu \in SS(N)$ do

6a. (optional) If $\nu \in OPEN$ 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 ν to N and adjust the accumulated distance for ν.

(ii) goto Step **7.**

6b. (optional) If $\nu \in CLOSE$ and the accumulated distance of the new path is small than the partial path ending at ν in the *CLOSE* list, do

(i) change the trace back (parent) pointer of ν to N and adjust the accumulated distance for all paths that contain ν.

(ii) goto Step **7.**

6c. Create a pointer pointing to N and push it into the *OPEN* list
7. Reorder the *OPEN* list according to search strategy or some heuristic measurement.
8. Goto Step **2.**

The process of explicitly generating part of an implicitly defined graph forms the essence of our general graph search procedure. The procedure is summarized in Algorithm 12.1. It maintains two lists: *OPEN*, which stores the nodes waiting for expansion; and *CLOSE*, which stores the already expanded nodes . Steps 6a and 6b are basically the bookkeeping process to merge different paths going into the same node by picking the one that has the minimum distance. Step 6a handles the case where ν is in the *OPEN* list and thus is

not expanded. The merging process is straightforward, with a single comparison and change of trace-back pointer if necessary. However, when v is in the *CLOSE* list and thus is already expanded in Step 6b, the merging requires additional forward propagation of the new score if the current path is found to be better than the best subpath already in the *CLOSE* list. This forward propagation could be very expensive. Fortunately, most of the search strategy can avoid such a procedure if we know that the already expanded node must belong in the best path leading to it. We discuss this in Section 12.5.

As described earlier, it may not be worthwhile to perform bookkeeping for a graph search, so Steps 6a and 6b are optional. If both steps are omitted, the graph search algorithm described above becomes a tree search algorithm. To illustrate different search strategies, tree search is used as the basic graph search algorithm in the sections that follows. However, you should note that all the search methods described here could be easily extended to graph search with the extra bookkeeping (merging) process as illustrated in Steps 6a and 6b of Algorithm 12.1.

12.1.2. Blind Graph Search Algorithms

If the aim of the search problem is to find an acceptable path instead of the best path, blind search is often used. *Blind search* treats every node in the *OPEN* list the same and blindly decides the order to be expanded without using any domain knowledge. Since blind search treats every node equally, it is often referred to as *uniform search* or *exhaustive search*, because it exhaustively tries out all possible paths. In AI, people are typically not interested in blind search. However, it does provide a lot of insight into many sophisticated heuristic search algorithms. You should note that blind search does not expand nodes randomly. Instead, it follows some systematic way to explore the search graph. Two popular types of blind search are depth-first search and breadth-first search.

12.1.2.1. Depth-First Search

When we are in a maze, the most natural way to find a way out is to mark the branch we take whenever we reach a branching point. The marks allow us to go back to a choice point with an unexplored alternative; withdraw the most recently made choice and undo all consequences of the withdrawn choice whenever a dead end is reached. Once the alternative choice is selected and marked, we go forward based on the same procedure. This intuitive search strategy is called *backtracking*. The famous *N*-queens puzzle [32] can be handily solved by the backtracking strategy.

Depth-first search picks an arbitrary alternative at every node visited. The search sticks with this partial path and works forward from the partial path. Other alternatives at the same level are ignored completely (for the time being) in the hope of finding a solution based on the current choice. This strategy is equivalent to ordering the nodes in the *OPEN* list by their depth in the search graph (tree). The deepest nodes are expanded first and nodes of equal depth are ordered arbitrarily.

Although depth-first search hopes the current choice leads to a solution, sometimes the current choice could lead to a dead-end (a node which is neither a goal node nor can be expanded further). In fact, it is desirable to have many short dead-ends. Otherwise the algorithm may search for a very long time before it reaches a dead-end, or it might not ever reach a solution if the search space is infinite. When the search reaches a dead-end, it goes back to the last decision point and proceeds with another alternative.

Figure 12.4 shows all the nodes being expanded under the depth-first search algorithm for the city-traveling problem illustrated in Figure 12.1. The only differences between the graph search and the depth-first search algorithms are:

1. The graph search algorithm generates all successors at a time (although all except one are ignored first), while backtracking generates only one successor at a time.
2. The graph search, when successfully finding a path, saves only one path from the starting node to the goal node, while depth-first search in general saves the entire record of the search graph.

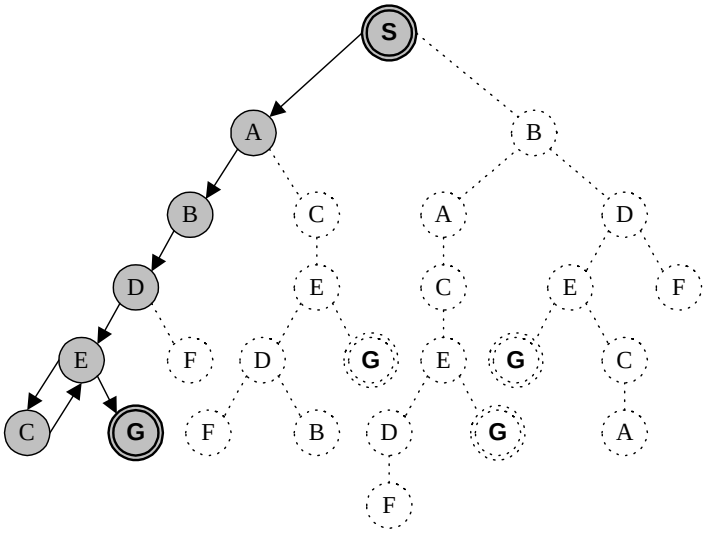

Figure 12.4 The node-expanding procedure of the depth-first search for the path search problem in Figure 12.1. When it fails to find the goal city in node *C*, it backtracks to the parent and continues the search until it finds the goal city. The gray nodes are those that explored. The dotted nodes are 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. To prevent exploring of paths that are too long, a depth bound can be placed to constraint the nodes to be expanded, and any node reaching that depth limit is treated as a terminal node (as if it had no successor).

The general graph search algorithm can be modified into a depth-first search algorithm as illustrated in Algorithm 12.2.

ALGORITHM 12.2 THE DEPTH-FIRST SEARCH ALGORITHM

1. Initialization: Put S in the *OPEN* list and create an initially empty the CLOSE list
2. If the *OPEN* list is empty, exit and declare failure.
3. Pop up the first node N in the *OPEN* list, remove it from the *OPEN* list and put it into the *CLOSE* list.
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, goto Step 2.
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).
6. $\forall v \in SS(N)$ do
   6c. Create a pointer pointing to N and push it into the *OPEN* list
7. Reorder the the *OPEN* list in descending order of the depth of the nodes.
8. Goto Step 2.

12.1.2.2. Breadth-first Search

One natural alternative to the depth frist search strategy is breadth-first search. *Breadth-first search* examines all the nodes on one level before considering any of the nodes on the next level (depth). As shown in Figure 12.5, node *B* would be examined just after node *A*. The search moves on level-by-level, finally discovering G on the fourth level.

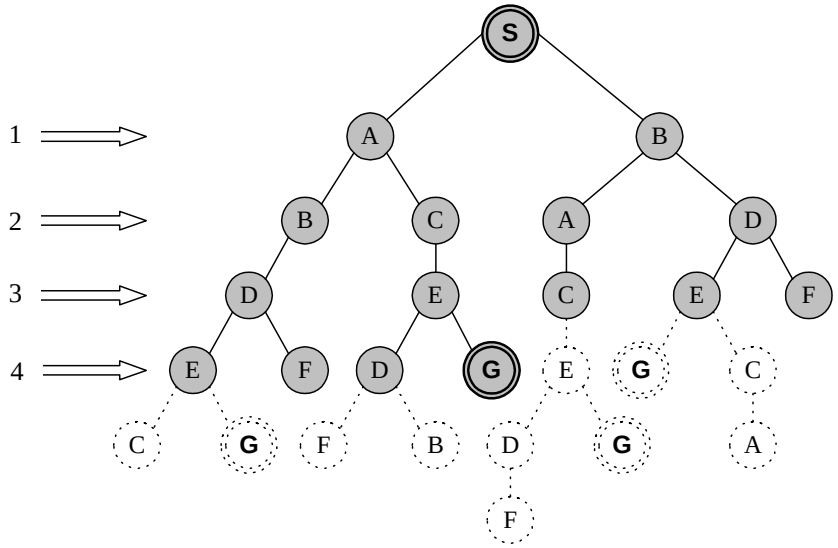

Figure 12.5 The node-expanding procedure of a breadth-first search for the path search problem in Figure 12.1. It searches through each level until the goal is identified. The gray nodes are those that are explored. The dotted nodes are not visited during the search [42].

Breadth-first search is guaranteed to find a solution if one exists, assuming that a finite number of successors (branches) always follow any node. The proof is straightforward. If there is a solution, its path length must be finite. Let's assume the length of the solution is *M*. Breadth-first search explores all paths of the same length increasingly. Since the number of paths of fixed length *N* is always finite, it eventually explores all paths of length *M*. By that time it should find the solution.

It is also easy to show that a breadth-first search can work on a search tree (graph) with infinite depth on which an unconstrained depth-first search will fail. Although a breadth-first might not find a shortest-distance path for the city-travel problem, it is guaranteed to find the one with fewest cities visited (minimum-length path). In some cases, it is a very desirable solution. On the other hand, a breadth-first search may be highly inefficient when all solutions leading to the goal node are at approximately the same depth. The breadth-first search algorithm is summarized in Algorithm 12.3.

ALGORITHM 12.3: THE BREADTH-FIRST SEARCH ALGORITHM

1. Initialization: Put *S* in the *OPEN* list and create an initially empty the *CLOSE* list
2. If the *OPEN* list is empty, exit and declare failure.
3. Pop up the first node *N* in the *OPEN* list, remove it from the *OPEN* list and put it into the *CLOSE* list.
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*.
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*).
6. $\forall v \in SS(N)$ do
    6c. Create a pointer pointing to *N* and push it into the *OPEN* list
7. Reorder the *OPEN* list in increasing order of the depth of the nodes
8. Goto Step **2.**

12.1.3. Heuristic Graph Search

Blind search methods, like depth-first search and breadth-first search, have no sense (or guidance) of where the goal node lies ahead. Consequently, they often spend a lot of time searching in hopeless directions. If there is guidance, the search can move in the direction that is more likely to lead to the goal. For example, you may want to find a driving route to the World Trade Center in New York. Without a map at hand, you can still use a straight-line distance estimated by eye as a hint to see if you are closer to the goal (World Trade Center). This *hill-climbing* style of guidance can help you to find the destination much more efficiently.

Blind search only finds one arbitrary solution instead of the optimal solution. To find the optimal solution with depth-first or breadth-first search, you must not stop searching when the first solution is discovered. Instead, the search needs to continue until it reaches all

the solutions, so you can compare them to pick the best. This strategy for finding the optimal solution is called *British Museum search* or *brute-force search*. Obviously, it is unfeasible when the search space is large. Again, to conduct selective search and yet still be able to find the optimal solution, some guidance on the search graph is necessary.

The guidance obviously comes from domain-specific knowledge. Such knowledge is usually referred to as *heuristic* information, and search methods taking advantage of it are called *heuristic search* methods. There is usually a wide variety of different heuristics for the problem domain. Some heuristics can reduce search effort without sacrificing optimality, while other can greatly reduce search effort but provide only sub-optimal solutions. In most practical problems, the choice of different heuristics is usually a tradeoff between the quality of the solution and the cost of finding the solution.

Heuristic information works like an evaluation function $h(N)$ that maps each node N to a real number, and which serves to indicate the relative goodness (or cost) of continuing the search path from that node. Since in our city-travel problem, straight-line distance is a natural way of measuring the goodness of a path, we can use the heuristic function $h(N)$ for the distance evaluation as:

$h(N)$=Heuristic estimate of the remaining distance from node N to goal G (12.1)

Since $g(N)$, the distance of the partial path to the current node N is generally known, we have:

$g(N)$=The distance of the partial path already traveled from root S to node N (12.2)

We can define a new heuristic function, $f(N)$, which estimates the total distance for the path (not yet finished) going through node N.

$$f(N) = g(N) + h(N) \tag{12.3}$$

A heuristic search method basically uses the heuristic function $f(N)$ to re-order the *OPEN* list in the Step 7 of Algorithm 12.1. The node with the best heuristic value is explored first (expanded first). Some heuristic search strategies also prune some unpromising partial paths forever to save search space. This is why heuristic search is often referred to as heuristic pruning.

The choice of the heuristic function is critical to the search results. If we use one that overestimates the distance of some nodes, the search results may be suboptimal. Therefore, heuristic functions that do not overestimate the distance are often used in search methods aiming to find the optimal solution.

To close this section, we describe two of the most popular heuristic search methods: . best-first (or A^* Search) [32, 43] and beam search [43]. They are widely used in many components of spoken language systems.

12.1.3.1. Best-First (A^* Search)

Once we have a reasonable heuristic function to evaluate the goodness of each node in the *OPEN* list, we can explore the best node (the node with smallest $f(N)$ value) first, since it offers the best hope of leading to the best path. This natural search strategy is called *best-first search*. To implement best-first search based on the Algorithm 12.1, we need to first evaluate $f(N)$ for each successor before putting the successors in the *OPEN* list in Step 6. We also need to sort the elements in the *OPEN* list based on $f(N)$ in Step 7, so that the best node is in the front-most position waiting to be expanded in Step 3. The modified procedure for performing best-first search is illustrated in Algorithm 12.4. To avoid duplicating nodes in the *OPEN* list, we include Steps 6a and 6b to take advantage of the dynamic programming principle. They perform the needed bookkeeping process to merge different paths leading into the same node.

ALGORITHM 12.4: THE BEST-FIRST SEARCH ALGORITHM

1. Initialization: Put *S* in the *OPEN* list and create an initially empty the *CLOSE* list
2. If the *OPEN* list is empty, exit and declare failure.
3. Pop up the first node *N* in the *OPEN* list, remove it from the *OPEN* list and put it into the *CLOSE* list.
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*.
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*).
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 goto 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 path containing v.

(ii) goto Step **7.**

6c. Create a pointer pointing to *N* and push it into the *OPEN* list

7. Reorder the the *OPEN* list in the increasing order of the heuristic function $f(N)$
8. Goto Step **2.**

A search algorithm is said to be *admissible* if it can guarantee to find an optimal solution, if one exists. Now we show that if the heuristic function $h(N)$ of estimating the remaining distance from N to goal node G is not an underestimate[2] of the true distance from N to goal node G, the best-first search illustrated in Algorithm 12.4 is admissible. In fact, when $h(N)$ satisfies the above criterion, the best-first algorithm is called A* (pronounced as /*eh*/-star) Search.

The proof can be carried out informally as follows. When the frontmost node in the *OPEN* list is the goal node G in Step 4, it immediately implies that

$$\forall v \in OPEN \quad f(v) \geq f(G) = g(G) + h(G) = g(G) \tag{12.4}$$

Equation (12.4) says that the distance estimate of any incomplete path are no shorter than the first found complete path. Since the distance estimate for any incomplete path is underestimated, the first found complete path in Step 4 must be the optimal path. A similar argument can also be used to prove that the Step 6b is actually not necessary for admissible heuristic functions; that is, there cannot be another path with a shorter distance from the starting node to a node that has been expanded. This is a very important feature since Step 6b is in general very expensive and it requires significant updates of many already expanded paths.

The A* search method is actually a family of search algorithms. When $h(N) = 0$ for all *N*, the search degenerates into an uninformed search[3] [40]. In fact, this type of uninformed search is the famous *branch-and-bound search* algorithm that is often used in many *operations research* problems. Branch-and-bound search always expands the shortest path leading into an open node until there is a path reaching the goal that is of length no longer than all incomplete paths terminating at open nodes. If $g(N)$ is further defined as the depth of the node *N*, the use of heuristic function $f(N)$ makes the search method identical to breadth-first search. In Section 12.1.2.2, we mention that breadth-first search is guaranteed to find minimum length path. This can certainly be derived from the admissibility of the A* search method.

When the heuristic function is close to the true remaining distance, the search can usually find the optimal solution without too much effort. In fact, when the true remaining distances for all nodes are known, the search can be done in a totally greedy fashion without any search at all; i.e., the only path explored is the solution. Any non-zero heuristic function is then called an informed heuristic function, and the search using such a function is called informed search. A heuristic function h_1 is said to be more informed than a heuristic function h_2 if the estimate h_1 is everywhere larger than h_2 and yet still admissible (underestimate). Finding an informed admissible heuristic function (guaranteed to underestimate for all nodes) is in general a difficult task. The heuristic often requires extensive analysis of the domain-specific knowledge and knowledge representation.

---

[2] For admissibility, we actually require only that the heuristic function not overestimate the distance from *N* to *G*. Since it is very rare to have an exact estimate, we use underestimate throughout this chapter without loss of generality. Sometimes we refer to an underestimate function as a lower-bound estimate of the true value.

[3] In some literature an uninformed search is referred to as uniform-cost search.

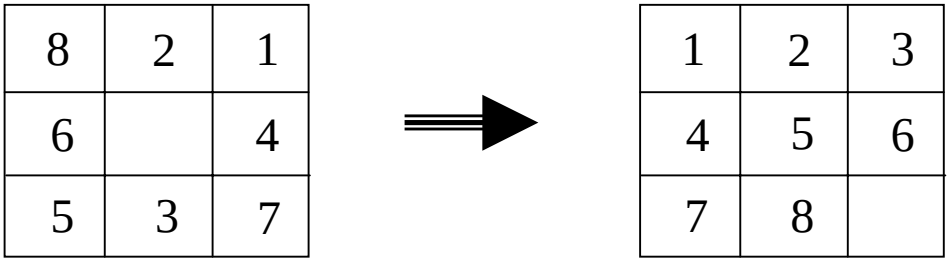

Figure 12.6 Initial and goal configurations for the 8-puzzle problem.

Let's look at a simple example – the 8-puzzle problem. The 8-puzzle consists of eight numbered, movable tiles set in a 3×3 frame. One cell of this frame is always empty, so it is possible to move an adjacent numbered tile into the empty cell. A solution for the 8-puzzle is to find a sequence of moves to change the initial configuration into a given goal configuration as shown in Figure 12.6. One choice for an informed admissible heuristic function h_1 is the number of misplaced tiles associated with the current configuration. Since each misplaced tile needs to move at least once to be in the right position, this heuristic function is clearly a lower bound of the true movements remaining. Based on this heuristic function, the value for the initial configuration will be 7 in Figure 12.6. If we examine this problem further, a more informed heuristic function h_2 can be defined as the sum of all row and column distances of all misplaced tiles and their goal positions. For example, the row and column distance between the tile 8 in the initial configuration and the goal position is 2 + 1= 3 which indicates that one must to move tile 8 at least 3 times in order for it to be in the right position. Based on the heuristic function h_2, the value for the initial configuration will be 16 in Figure 12.6, so h_2 is again admissible.

In our city-travel problem, one natural choice for the underestimating heuristic function of the remaining distance between node *N* and goal *G* is the straight-line distance since the true distance must be no shorter than the straight-line distance.

Figure 12.7 shows an augmented city-distance map with straight-line distance to goal node attached to each node. Accordingly, the heuristic search tree can be easily constructed for improved efficiency. Figure 12.8 shows the search progress of applying the A^* search algorithm for the city-traveling problem by using the straight-line distance heuristic function to estimate the remaining distances.

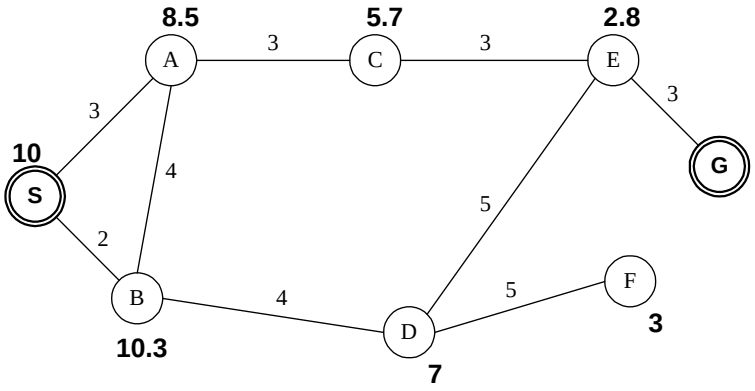

Figure 12.7 The city-travel problem augmented with heuristic information. The numbers beside each node indicate the straight-line distance to the goal node G [42].

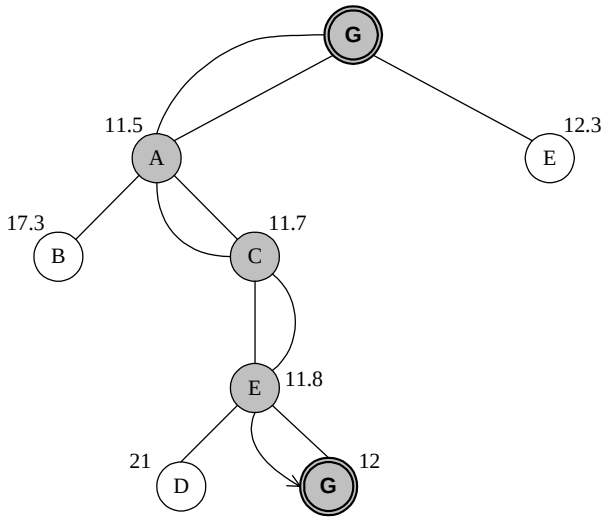

Figure 12.8 The search progress of applying A* search for the city-travel problem. The search determines that path S-A-C-E-G is the optimal one. The number beside the node is *f* values on which the sorting of the *OPEN* list is based [42].

12.1.3.2. Beam Search

Sometimes, it is impossible to find any effective heuristic estimate, as required in A* search, particularly when there is very little (or no) information about the remaining paths. For example, in real-time speech recognition, there is little information about what the speaker will utter for the remaining speech. Therefore, an efficient uninformed search strategy is very important to tackle this type of problem.

Breadth-first style search is an important strategy for heuristic search. A breadth-first search virtually explores all the paths with the same depth before exploring deeper paths. In practice, paths of the same depth are often easier to compare. It requires fewer heuristics to rank the goodness of each path. Even with uninformed heuristic function ($h(N) = 0$), the direct comparison of g (distance so far) of the paths with the same length should be a reasonable choice.

Beam search is a widely used search technique for speech recognition systems [26, 31, 37]. It is a breadth-first style search and progresses along with the depth. Unlike traditional breadth-first search, however, beam search only expands nodes that are likely to succeed at each level. Only these nodes are kept in the beam, and the rest are ignored (pruned) for improved efficiency.

In general, a beam search only keeps up to *w* best paths at each stage (level), and the rest of the paths are discarded. The number *w* is often referred to as beam width. The number of nodes explored remains manageable in beam search even if the whole search space is gigantic. If a beam width *w* is used in a beam search with an average branching factor *b*, only $w \times b$ nodes need to be explored at any depth, instead of the exponential number needed for breadth-first search. Suppose that a beam width of 2 is used for the city-travel problem and the same kind of heuristic function (straight-line distance) is used; Figure 12.9 illustrates how beam search progresses to find the path. We can also see that the beam search saved a large number of unneeded nodes, as shown by the dotted lines.

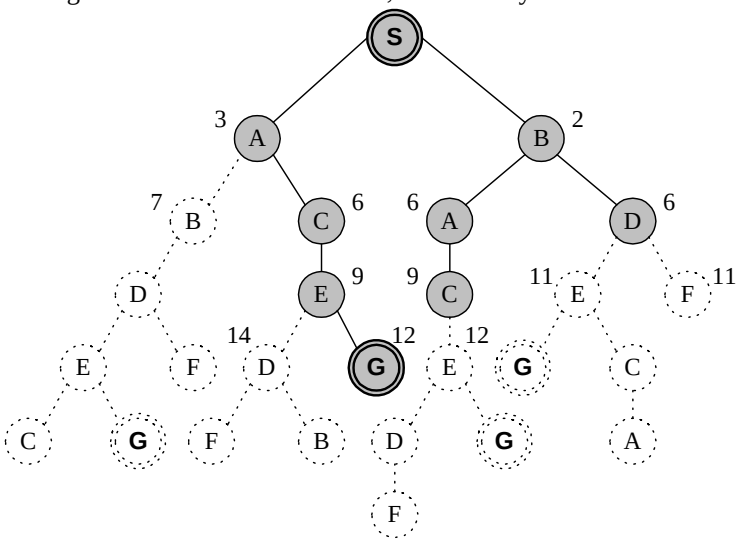

Figure 12.9 Beam search for the city-travel problem. The nodes with gray color are the ones kept in the beam. The dashed nodes were explored but pruned because of higher cost. The dotted nodes indicate all the savings because of pruning [42].

The beam search algorithm can be easily modified from the breadth-first search algorithm and is illustrated in Algorithm 12.5. For simplicity, we do not include the merging step here. In Algorithm 12.5, Step 4 obviously requires sorting, which is time consuming if the

number $w \times b$ is huge. In practice, the beam is usually implemented as a flexible list where nodes are expanded if their heuristic functions $f(N)$ are within some threshold (a.k.a beam threshold) of the best node (the smallest value) at the same level. Thus, we only need to identify the best node and then prune away nodes that are outside of the threshold. Although this makes the beam size change dynamically, it significantly reduces the effort for sorting of the *Beam-Candidate* list. In fact, by adjusting the beam threshold, the beam size can be controlled indirectly and yet kept manageable.

Unlike A* search, beam search is an approximate heuristic search method that is not admissible. However, it has a number of unique merits. Because of its simplicity in both its search strategy and its requirement of domain-specific heuristic information, it has become one of the most popular methods for complicated speech recognition problems. It is particularly attractive when integration of different knowledge sources is required in a time-synchronous fashion. It has the advantages of providing a consistent way of exploring nodes level by level and of offering minimally needed communication between different paths. It is also very suitable for parallel implementation because of its breath-first search nature.

ALGORITHM 12.5: THE BEAM SEARCH ALGORITHM

1. Initialization: Put *S* in the *OPEN* list and create an initially empty the *CLOSE* list
2. If the *OPEN* list is empty, exit and declare failure.
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

4. Sort the *Beam-Candidate* list according to the heuristic function $f(N)$ so that the best *w* nodes can be push into the the *OPEN* list. Prune the rest of nodes in the *Beam-Candidate* list.
5. Goto Step **2.**

12.2. SEARCH ALGORITHMS FOR SPEECH RECOGNITION

As described in Chapter 9, the decoder is basically a search process to uncover the word sequence $\hat{\mathbf{W}} = w_1 w_2 ... w_m$ that has the maximum posterior probability $P(\mathbf{W}|\mathbf{X})$ for the given acoustic observation $\mathbf{X} = X_1 X_2 ... X_n$. That is,

$$\hat{\mathbf{W}} = \arg\max_{\mathbf{W}} P(\mathbf{W} \mid \mathbf{X}) = \arg\max_{\mathbf{W}} \frac{P(\mathbf{W})P(\mathbf{X} \mid \mathbf{W})}{P(\mathbf{X})} = \arg\max_{\mathbf{W}} P(\mathbf{W})P(\mathbf{X} \mid \mathbf{W}) \qquad (12.5)$$

One obvious way is to search all possible word sequences and select the one with best posterior probability score.

The unit of acoustic model $P(\mathbf{X}|\mathbf{W})$ is not necessary a word model. For large-vocabulary speech recognition systems, subword models, which include phonemes, demisyllables, and syllable are often used. When subword models are used, the word model $P(\mathbf{X}|\mathbf{W})$ is then obtained by concatenating the subword models according to the pronunciation transcription of the words in a lexicon or dictionary.

When word models are available, speech recognition becomes a search problem. The goal for speech recognition is thus to find a sequence of word models that best describes the input waveform against the word models. As neither the number of words nor the boundary of each word or phoneme in the input waveform is known, appropriate search strategies to deal with these variable-length nonstationary patterns are extremely important.

When HMMs are used for speech recognition systems, the states in the HMM can be expanded to form the state-search space in the search. In this chapter, we use HMMs as our speech models. Although the HMM framework is used to describe the search algorithms, all techniques mentioned in this and the following chapter can be used for systems based on other modeling techniques, including template matching and neural networks. In fact, many search techniques had been invented before HMMs were applied to speech recognition. Moreover, the HMMs state transition network is actually general enough to represent the general search framework for all modeling approaches.

12.2.1. Decoder Basics

The lessons learned from dynamic programming or the Viterbi algorithm introduced in Chapter 8 tell us that the exponential blind search can be avoided if we can store some intermediate optimal paths (results). Those intermediate paths are used for other paths without being recomputed each time. Moreover, the beam search described in the previous section shows you that efficient search is possible if appropriate pruning is employed to discard highly unlikely paths. In fact, all the search techniques use two strategies: sharing and pruning. *Sharing* means that intermediate results can be kept, so that they can be used by other paths without redundant re-computation. *Pruning* means that unpromising paths can be discarded reliably without wasting time in exploring them further.

Search strategies based on dynamic programming or the Viterbi algorithm with the help of clever pruning, have been applied successfully to a wide range of speech recognition tasks [31], ranging from small-vocabulary tasks, like digit recognition, to unconstraint large-vocabulary (more than 60,000 words) speech recognition. All the efficient search algorithms we discuss in this chapter and the next are considered as variants of dynamic programming or the Viterbi search algorithm.

In Section 12.1, cost (distance) is used as the measure of goodness for graph search algorithms. With Bayes' formulation, searching the minimum-cost path (word sequence) is equivalent to finding the path with maximum probability. For the sake of consistency, we use the inverse of Bayes' posterior probability as our objective function. Furthermore, loga-

rithms are used on the inverse posterior probability to avoid multiplications. That is, the following new criterion is used to find the optimal word sequence $\hat{\mathbf{W}}$:

$$C(\mathbf{W} \mid \mathbf{X}) = \log\left[\frac{1}{P(\mathbf{W})P(\mathbf{X} \mid \mathbf{W})}\right] = -\log\left[P(\mathbf{W})P(\mathbf{X} \mid \mathbf{W})\right] \tag{12.6}$$

$$\hat{\mathbf{W}} = \arg\min_{\mathbf{W}} C(\mathbf{W} \mid \mathbf{X}) \tag{12.7}$$

For simplicity, we also define the following cost measures to mirror the likelihood for acoustic models and language models:

$$C(\mathbf{X} \mid \mathbf{W}) = -\log\left[P(\mathbf{X} \mid \mathbf{W})\right] \tag{12.8}$$

$$C(\mathbf{W}) = -\log\left[P(\mathbf{W})\right] \tag{12.9}$$

12.2.2. Combining Acoustic And Language Models

Although Bayes' equation [Eq. (12.5)] suggests that the acoustic model probability (conditional probability) and language model probability (prior probability) can be combined through simple multiplication, in practice some weighting is desirable. For example, when HMMs are used for acoustic models, the acoustic probability is usually underestimated, owing to the fallacy of the Markov and independence assumptions. Combining the language model probability with an underestimated acoustic model probability according to Eq. (12.5) would give the language model too little weight. Moreover, the two quantities have vastly different dynamic ranges particularly when continuous HMMs are used. One way to balance the two probability quantities is to add a *language model weight* LW to raise the language model probability $P(\mathbf{W})$ to that power $P(\mathbf{W})^{LW}$ [4, 25]. The language model weight LW is typically determined empirically to optimize the recognition performance on a development set. Since the acoustic model probabilities are underestimated, the language model weight LW is typically >1.

Language model probability has another function as a penalty for inserting a new word (or existing words). In particular, when a uniform language model (every word has a equal probability for any condition) is used, the language model probability here can be viewed as purely the penalty of inserting a new word. If this penalty is large, the decoder will prefer fewer longer words in general, and if this penalty is small, the decoder will prefer a greater number of shorter words instead. Since varying the language model weight to match the underestimated acoustic model probability will have some side effect of adjusting the penalty of inserting a new word, we sometimes use another independent *insertion penalty* to adjust the issue of longer or short words. Thus the language model contribution will becomes:

$$P(\mathbf{W})^{LW} IP^{N(\mathbf{W})} \tag{12.10}$$

where IP is the insertion penalty (generally $0 < IP \leq 1.0$) and $N(\mathbf{W})$ is the number of words in sentence $\mathbf{W}$. According to Eq. (12.10), insertion penalty is generally a constant that is added to the negative-logarithm domain when extending the search to another new word. In Chapter 9, we described how to compute errors in a speech recognition system and introduced three types of error: substitutions, deletions and insertions. Insertion penalty is so named because it usually affects only insertions. Similar to language model weight, the insertion penalty is determined empirically to optimize the recognition performance on a development set.

12.2.3. Isolated Word Recognition

With isolated word recognition, word boundaries are known. If word HMMs are available, the acoustic model probability $P(\mathbf{X}|\mathbf{W})$ can be computed using the forward algorithm introduced in Chapter 8. The search becomes a simple pattern recognition problem, and the word $\hat{\mathbf{W}}$ with highest forward probability is then chosen as the recognized word. When subword models are used, word HMMs can be easily constructed by concatenating corresponding phoneme HMMs or other types of subword HMMs according to the procedure described in Chapter 9.

12.2.4. Continuous Speech Recognition

Search in continuous speech recognition is rather complicated, even for a small vocabulary, since the search algorithm has to consider the possibility of each word starting at any arbitrary time frame. Some of the earliest speech recognition systems took a two-stage approach towards continuous speech recognition, first hypothesizing the possible word boundaries and then using pattern matching techniques for recognizing the segmented patterns. However, due to significant cross-word co-articulation, there is no reliable segmentation algorithm for detecting word boundaries other than doing recognition itself.

Let's illustrate how you can extend the isolated-word search technique to continuous speech recognition by a simple example, as shown in Figure 12.10. This system contains only two words, w_1 and w_2. We assume the language model used here is an uniform unigram ($P(w_1) = P(w_2) = 1/2$).

It is important to represent the language structures in the same HMM framework. In Figure 12.10, we add one starting state S and one collector state C. The starting state has a null transition to the initial state of each word HMM with corresponding language model probability (1/2 in this case). The final state of each word HMM has a null transition to the collector state. The collector state then has a null transition back to the starting state in order to allow recursion. Similar to the case of embedding the phoneme (subword) HMMs into the word HMM for isolated speech recognition, we can embed the word HMMs for w_1 and w_2

into a new HMM corresponding to structure in Figure 12.10. Thus, the continuous speech search problem can be solved by the standard HMM formulations.

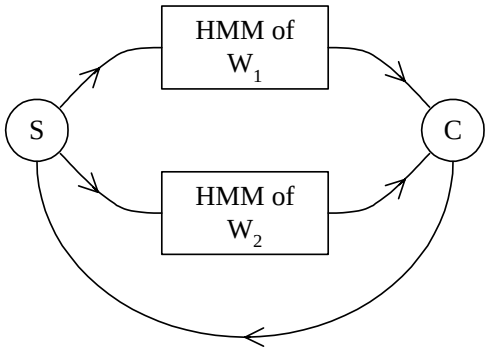

Figure 12.10 A simple example of continuous speech recognition task with two words w_1 and w_2. A uniform unigram language model is assumed for these words. State S is the starting state while state C is a collector state to save full expanded links between every word pairs.

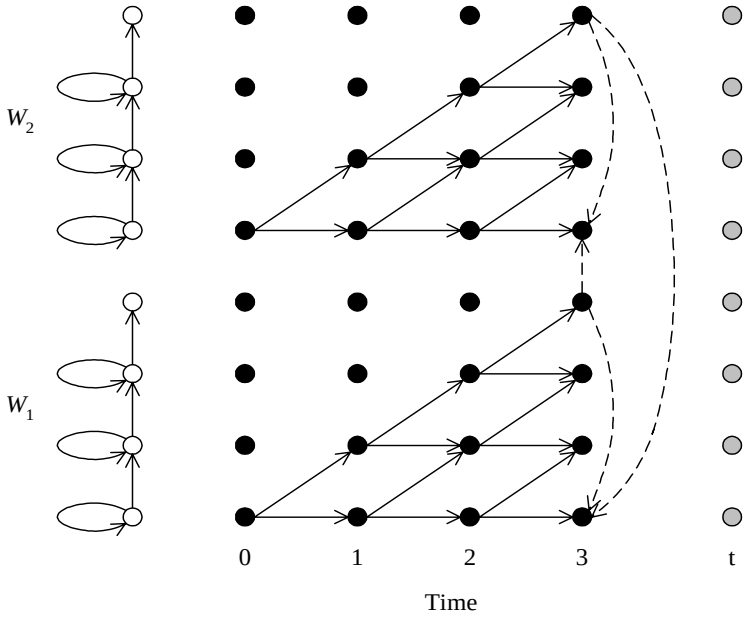

Figure 12.11 HMM trellis for continuous speech recognition example in Figure 12.10. When the final state of the word HMM is reached, a null arc (indicated by a dashed line) is linked from it to the initial state of the following word.

The composite HMMs shown in Figure 12.10 can be viewed as a stochastic finite state network with transition probabilities and output distributions. The search algorithm is essen-

tially producing a match between the acoustic the observation **X** and a path[4] in the stochastic finite state network. Unlike isolated word recognition, continuous speech recognition needs to find the optimal word sequence $\hat{\mathbf{W}}$. The Viterbi algorithm is clearly a natural choice for this task since the optimal state sequence $\hat{\mathbf{S}}$ corresponds to the optimal word sequence $\hat{\mathbf{W}}$. Figure 12.11 shows the HMM Viterbi trellis computation for the two-word continuous speech recognition example in Figure 12.10. There is a cell for each state in the stochastic finite state network and each time frame *t* in the trellis. Each cell $C_{s,t}$ in the trellis can be connected to a cell corresponding to time *t* or *t*+1 and to states in the stochastic finite state network that can be reached from *s*. To make a word transition, there is a null transition to connect the final state of each word HMM to the initial state of the next word HMM that can be followed. The trellis computation is done *time-synchronously* from left to right; i.e., the each cell for time *t* is completely computed before proceeding to time *t*+1.

12.3. LANGUAGE MODEL STATES

The state-space is a good indicator of search complexity. Since the HMM representation for each word in the lexicon is fixed, the state-space is determined by the language models. According to Chapter 11, every language model (grammar) is associated with a state machine (automata). Such a state machine is expanded to form the state-space for the recognizer. The states in such a state machine are referred to as *language models states*. For simplicity, we will use the concepts of state-space and language model states interchangeably. The expansion of language model states to HMM states will be done implicitly. The language model states for isolated word recognition are trivial. They are just the union of the HMM states of each word. In this section we look at the language model states for various grammars for continuous speech recognition.

12.3.1. Search Space with FSM and CFG

As described in Chapter 8, the complexity for the Viterbi algorithm is $O(N^2T)$ where *N* is the total numberof states in the composite HMM and *T* is the length of input observation. A full time-synchronous Viterbi search is quite efficient for moderate tasks (vocabulary $\leq$ 500). We have already demonstrated in Figure 12.11 how to search for a two-word continuous speech recognition task with a uniform language model. The uniform language model, which allows all words in the vocabulary to follow every word with the same probability, is suitable for connected-digit task. In fact, most small vocabulary tasks in speech recognition applications usually use a finite state grammar (FSG).

Figure 12.12 shows a simple example of an FSM. Similar to the process described in Section 12.2.3 and 12.2.4, each of the word arcs in an FSG can be expanded as a network of phoneme (subword) HMMs. The word HMMs are connected with null transitions with the

[4] A path here means a sequence of states and transitions.

grammar state. A large finite state HMM network that encodes all the legal sentences can be constructed based on the expansion procedure. The decoding process is achieved by performing time-synchronous Viterbi search on this composite finite state HMM.

In practice, FSGs are sufficient for simple tasks. However, when an FSG is made to satisfy the constraints of sharing of different sub-grammars for compactness and support for dynamic modifications, the resulting non-deterministic FSG is very similar to context-free grammar (CFG) in terms of implementation. The CFG grammar consists of a set of productions or rules, which expand nonterminals into a sequence of terminals and nonterminals. Nonterminals in the grammar tend to refer to high-level task specific concepts such as dates, names, and commands. The terminals are words in the vocabulary. A grammar also has a non-terminal designated as its start state.

Although efficient parsing algorithms, like chart parsing (described in Chapter 11), are available for CFG, they are not suitable for speech recognition, which requires left-to-right processing. A context-free grammar can be formulated with a recursive transition network (RTN). RTNs are more powerful and complicated than the finite state machines described in Chapter 11 because they allow arc labels to refer to other networks as well as words. We use Figure 12.13 to illustrate how to embed HMMs into a recursive transition network.

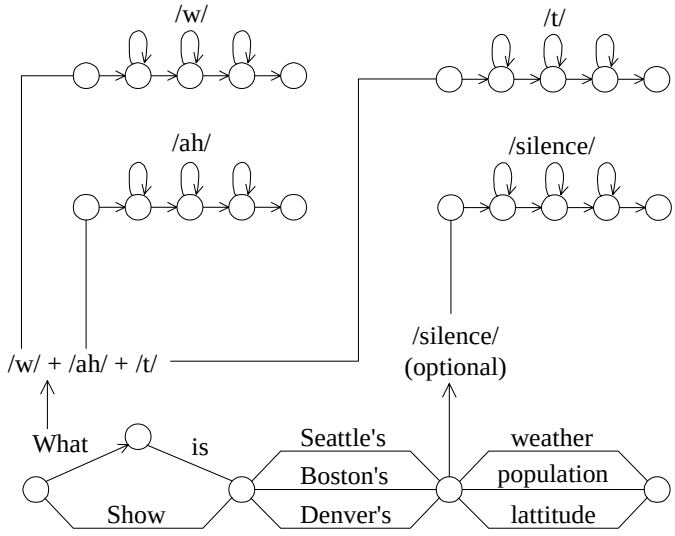

Figure 12.12 An illustration of how to compile a speech recognition task with finite state grammar into a composite HMM.

Figure 12.13 is an RTN representation of the following CFG:

```
S→ NP VP
NP→ sam | sam davis
VP → VERB tom
VERB → likes | hates
```

There are three types of arcs in an RTN, as shown in Figure 12.13: CAT(*x*), PUSH (*x*), and POP(*x*). The CAT(*x*) arc indicates that *x* is a terminal node (which is equivalent to a word arc). Therefore, all the CAT(*x*) arcs can be expanded by the HMM network for *x*. The word HMM can again be a composite HMM built from phoneme (or subword) HMMs. Similar to the finite state grammar case in Figure 12.12, each grammar state acts as a state with incoming and outgoing null transitions to connect word HMMs in the CFG.

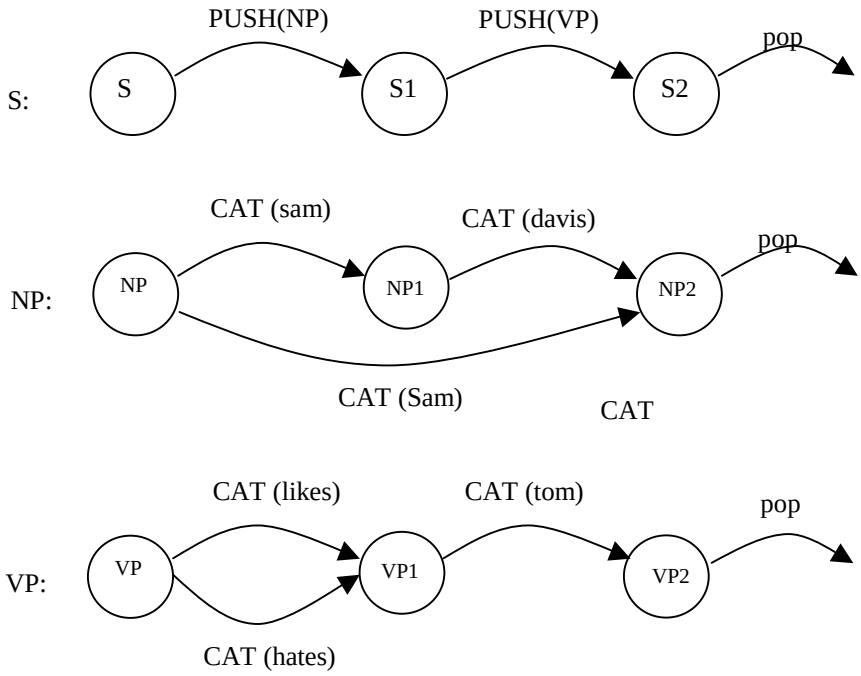

Figure 12.13 An simple RTN example with three types of arcs: CAT(*x*), PUSH (*x*), POP.

During decoding, the search pursues several paths through the CFG at the same time. Associated with each of the paths is a grammar state that describes completely how the path can be extended further. When the decoder hypothesizes the end of the current word of a path, it asks the CFG module to extend the path further by one word. There may be several alternative successor words for the given path. The decoder considers all the successor word possibilities. This may cause the path to be extended to generate several more paths to be considered ,each with its own grammar state.

Readers should note that the same word might be under consideration by the decoder in the context of different paths and grammar states at the same time. For example, there are two word arcs CAT (Sam) in Figure 12.13. Their HMM states should be considered as distinct states in the trellis because they are in completely different grammar states. Two different states in the trellis also means that different paths going into these two states cannot be merged. Since these two partial paths will lead to different successive paths, the search deci-

sion needs to be postponed until the end of search. Therefore, when embedding HMMs into word arcs in the grammar network, the HMM state will be assigned a new state identity, although the HMM parameters (transition probabilities and output distributions) can still be shared across different grammar arcs.

Each path consists of a stack of production rules. Each element of the stack also contains the position within the production rule of the symbol that is currently being explored. The search graph (trellis) started from the initial state of CFG (state S). When the path needs to be extended, we look at the next arc (symbol in CFG) in the production. When the search enters a CAT(x) arc (terminal), the path gets extended with the terminal, and the HMM trellis computation is performed on the CAT(x) arc to match the model x against the acoustic data. When the final state of the HMM for x is reached, the search moves on via the null transition to the destination of the CAT(x) arc. When the search enters a PUSH(x) arc, it indicates a nonterminal symbol x is encountered. In effect, the search is about to enter a subnetwork of x, the destination of the PUSH(x) arc is stored in a last-in first-out (LIFO) stack. When the search reaches a POP arc that signals the end of the current network, the control should jump back to the calling network. In the other words, the search returns to the state extracted from the top of the LIFO stack. Finally, when we reach the end of the production rule at the very bottom of the stack, we have reached an accepting state in which we have seen a complete grammatical sentence. For our decoding purpose, that is the state we want to pick as the best score at the end of time frame T to get the search result.

The problem of connected word recognition by finite state or context-free grammars is that the number of states increases enormously when it is applied to more complex grammars. Moreover it remains a challenge to generate such FSGs or CFGs from a large corpus, either manually or automatically. As mentioned in Chapter 11, it is questionable whether FSG or CFG is adequate to describe natural languages or unconstrained spontaneous languages. Instead, n-gram language models are often used for natural languages or unconstrained spontaneous languages. In the next section we investigate how to integrate various n-grams into continuous speech recognition.

12.3.2. Search Space with the Unigram

The simplest n-gram is the unigram that is memory-less and only depends on the current word.

$$P(\mathbf{W}) = \prod_{i=1}^{n} P(w_i) \tag{12.11}$$

Figure 12.14 shows such a unigram grammar network. The final state of each word HMM is connected to the collector state by a null transition, with probability 1.0. The collector state is then connected to the starting state by another null transition, with transition probability equal to 1.0. For word expansion, the starting state is connected to the initial state of each word HMM by a null transition, with transition probability equal to the corresponding unigram probability. Using the collector state and starting state for word expansion

allows efficient expansion because it first merges all the word-ending paths[5] (only the best one survives) before expansion. It can cut the total cross-word expansion from N^2 to N.

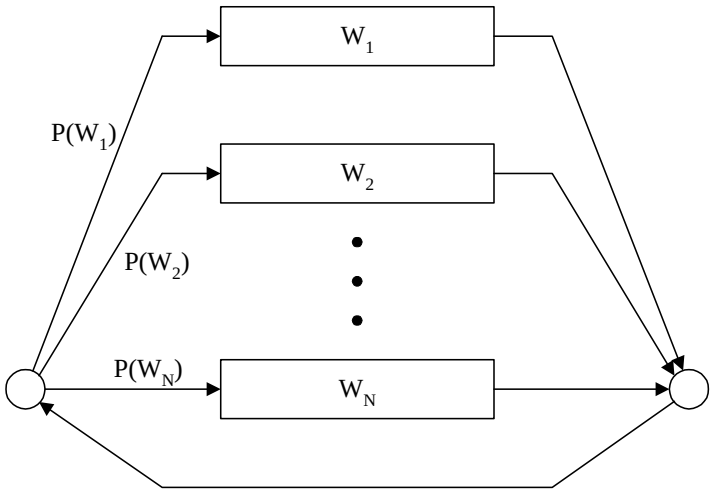

Figure 12.14 A unigram grammar network where the unigram probability is attached as the transition probability from starting state *S* to the first state of each word HMM.

12.3.3. Search Space with Bigrams

When the bigram is used, the probability of a word depends only on the immediately preceding word. Thus, the language model score is:

$$P(\mathbf{W}) = P(w_1 | < s >) \prod_{i=2}^{n} P(w_i \mid w_{i-1}) \tag{12.12}$$

where $< s >$ represents the symbol of starting of a sentence.

Figure 12.15 shows a grammar network using a bigram language model. Because of the bigram constraint, the merge-and-expand framework for unigram search no longer applies here. Instead, the bigram search needs to perform expand-and-merge. Thus, bigram expansion is more expensive than unigram expansion. For a vocabulary size *N*, the bigram would need N^2 word-to-word transitions in comparison to *N* for the unigram. Each word transition has a transition probability equal to the correspondent bigram probability. Fortunately, the total number of states for bigram search is still proportional to the vocabulary size *N*.

[5] In graph search, a partial path still under consideration is also referred as a theory, although we will use paths instead of theories in this book.

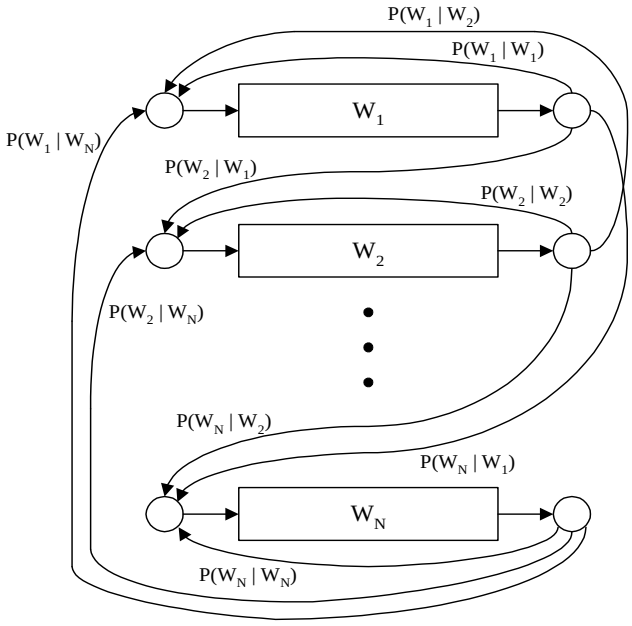

Figure 12.15 A bigram grammar network where the bigram probability $P(w_j \mid w_i)$ is attached as the transition probability from word w_i to w_j [19].

Because the search space for bigram is kept manageable, bigram search can be implemented very efficiently. Bigram search is a good compromise between efficient search and effective language models. Therefore, bigram search is arguably the most widely used search technique for unconstrained large-vocabulary continuous speech recognition. Particularly for the multiple-pass search techniques described in Chapter 13, a bigram search is often used in the first pass search.

12.3.3.1. Backoff Paths

When the vocabulary size *N* is large, the total bigram expansion N^2 can become computationally prohibitive. As described in Chapter 11, only a limited number of bigrams are observable in any practical corpora for a large vocabulary size. Suppose the probabilities for unseen bigrams are obtained through Katz's backoff mechanism. That is, for unseen bigram $P(w_j \mid w_i)$,

$$P(w_j \mid w_i) = \alpha(w_i)P(w_j) \tag{12.13}$$

where $\alpha(w_i)$ is the backoff weight for word w_i.

Using the backoff mechanism for unseen bigrams, the bigram expansion can be significantly reduced [12]. Figure 12.16 shows the new word expansion scheme. Instead of full

bigram expansion, only observed bigrams are connected by direct word transitions with correspondent bigram probabilities. For backoff 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 transition probability equal to its corresponding unigram probability $P(w_j)$. Readers should note that there are now two paths from w_i to w_j for an observed bigram $P(w_j \mid w_i)$. One is the direct link representing the observable bigram $P(w_j \mid w_i)$, and the other is the two-link backoff path representing $\alpha(w_i)P(w_j)$. For a word pair whose bigram exists, the two-link backoff path is likely to be ignored since the backoff unigram probability is almost always smaller than the observed bigram $P(w_j \mid w_i)$. Suppose there are only N_b different observable bigrams, this scheme requires $N_b + 2N$ instead of N^2 word transitions. Since under normal circumstance, $N_b << N^2$, this backoff scheme significantly reduce the cost of word expansion.

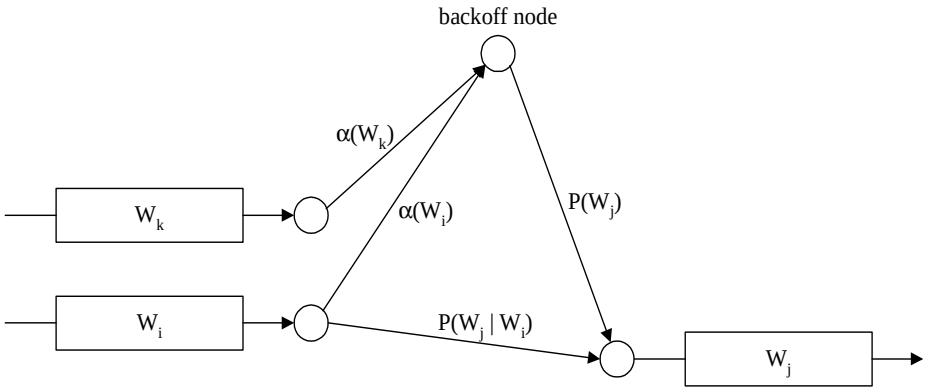

Figure 12.16 Reducing bigram expansion in a search by using the backoff node. In additional to normal bigram expansion arcs for all observed bigrams, the last state of word w_i is first connected to a central backoff node with transition probability equal to backoff weight $\alpha(w_i)$. The backoff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

12.3.4. Search Space with Trigrams

For a trigram language model, the language model score is:

$$P(\mathbf{W}) = P(w_1 \mid < s >)P(w_2 \mid < s >, w_1)\prod_{i=3}^{n} P(w_i \mid w_{i-2}, w_{i-1}) \tag{12.14}$$

The search space is considerably more complex, as shown in Figure 12.17. Since the equivalence grammar class is the previous two words w_i and w_j, the total number of grammar states is N^2. From each of these grammar states, there is a transition to the next word [19].

Obviously, it is very expensive to implement large-vocabulary trigram search given the complexity of the search space. It becomes necessary to dynamically generate the trigram search graph (trellis) via a graph search algorithm. The other alternative is to perform a multiple-pass search strategy, in which the first-pass search uses less detailed language models, like bigrams to generate an *n*-best list or word lattice, and then a second pass detailed search can use trigrams on a much smaller search space. Multiple-pass search strategy is discussed in Chapter 13.

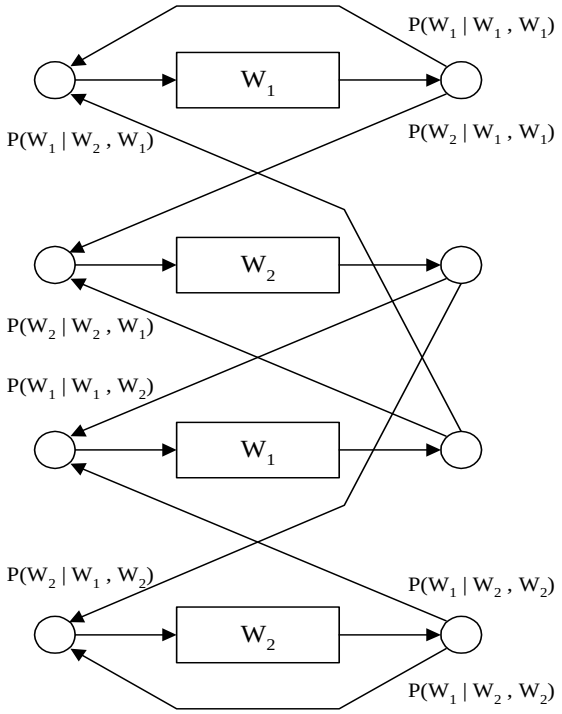

Figure 12.17 A trigram grammar network where the trigram probability $P(w_k \mid w_i, w_j)$ is attached to transition from grammar state word w_i, w_j to the next word w_k. Illustrated here is a two-word vocabulary, so there are four grammar states in the trigram network [19].

12.3.5. How to Handle Silences Between Words

In continuous speech recognition, there are unavoidable pauses (silences) between words or sentences. The pause is often referred to as silence or a non-speech event in continuous

speech recognition. Acoustically, the pause is modeled by a silence model[6] that models background acoustic phenomena. The silence model is usually modeled with a topology flexible enough to accommodate a wide range of lengths, since the duration of a pause is arbitrary.

It can be argued that silences (pauses) are actually linguistic distinguishable events, which contribute to prosodic and meaning representation. For example, people are likely to pause more often in phrasal boundaries. However, these patterns are so far not well understood for unconstrained natural speech (particularly for spontaneous speech). Therefore, the design of almost all automatic speech recognition systems today allows silences (pauses) occurring just about anywhere between two lexical tokens or between sentences. It is relatively safe to assume that people pause a little bit between sentences to catch breath, so the silences between sentences are assumed mandatory while silences between words are optional. In most systems, silence (any pause) is often modeled as a special lexicon entry with special language model probability. This special language model probability is also referred to as silence insertion penalty that is set to adjust the likelihood of inserting such an optional silence between words.

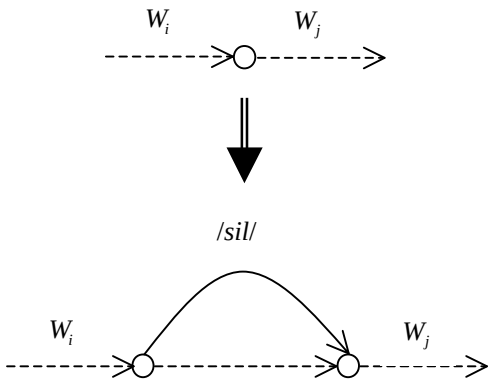

Figure 12.18 Incorporating optional silence (a non-speech event) in the grammar search network where the grammar state connecting different words are replaced by two parallel paths. One is the original null transition directly from one word to the other, while the other first goes through the silence word to accommodate the optional silence.

It is relatively straightforward to handle the optional silence between words. We need only to replace all the grammar states connecting words with a small network like the one shown in Figure 12.18. This arrangement is similar to that of the optional silence in training continuous speech, described in Chapter 9. The small network contains two parallel paths. One is the original null transition acting as the direct transition from one word to another, while the other path will need to go through a silence model with the silence insertion penalty attached in the transition probability before going to the next word.

[6] Some researchers extend the context-dependent modeling to silence models. In that case, there are several silence models based on surrounding contexts.

One thing to clarify in the implementation of Figure 12.18 is that this silence expansion needs to be done for every grammar state connecting words. In the unigram grammar network of Figure 12.14, since there is only one collector node to connect words, the silence expansion is required only for this collector node. On the other hand, in the bigram grammar network of Figure 12.15, there is a collector node for every word before expanding to the next word. In this case, the silence expansion is required for every collector node. For a vocabulary size $|V|$, this means there are $|V|$ numbers of silence networks in the grammar search network. This requirement lies in the fact that in bigram search we cannot merge paths before expanding into the next word. Optional silence can then be regarded as part of the search effort for the previous word, so the word expansion needs to be done after finishing the optional silence. Therefore, we treat each word having two possible pronunciations, one with the silence at the end and one without. This viewpoint integrates silence in the word pronunciation network like the example shown in Figure 12.19.

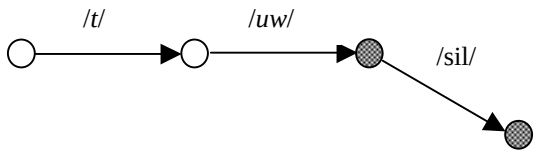

Figure 12.19 An example of treating silence is part of the pronunciation network of word TWO. The shaded nodes represent possible word-ending nodes; one without silence and the other one with silence.

For efficiency reasons, a single silence is sometimes used for large-vocabulary continuous speech recognition using higher order *n*-gram language model. Theoretically, this could be a source of pruning errors.[7] However, the error could turn out to be so small as to be negligible because there are, in general, very few pauses between word for continuous speech. On the other hand, the overhead of using multiple silences should be very minimal because it is less likely to visit those silence models at the end of words due to pruning.

12.4. TIME-SYNCHRONOUS VITERBI BEAM SEARCH

When HMMs are used for acoustic models, the acoustic model score (likelihood) used in search is by definition the forward probability. That is, all possible state sequences must be considered. Thus,

$$P(\mathbf{X} \mid \mathbf{W}) = \sum_{all \text{ possible } s_0^T} P(\mathbf{X}, s_0^T \mid \mathbf{W}) \tag{12.15}$$

where the summation is to be taken over all possible state sequences **S** with the word sequence **W** under consideration. However, under the trellis framework (as in Figure 12.11),

[7] Speech recognition errors due to sub-optimal search or heuristic pruning are referred to as *pruning errors*, which will be described in details in Chapter 13.

more bookkeeping must be performed since we cannot add scores with different word sequence history. Since the goal of decoding is to uncover the best word sequence, we could approximate the summation with the maximum to find the best state sequence instead. The Bayes decision rule, Eq. (12.5) becomes

$$\hat{\mathbf{W}} = \arg\max_{\mathbf{w}} P(\mathbf{W})P(\mathbf{X} \mid \mathbf{W}) \cong \arg\max_{\mathbf{w}} \left\{ P(\mathbf{W}) \max_{S_0^T} P(\mathbf{X}, s_0^T \mid \mathbf{W}) \right\} \tag{12.16}$$

Equation (12.16) is often referred to as the *Viterbi approximation*. It can be literally translated to "the *most likely word sequence* is approximated by the *most likely state sequence*". Viterbi search is then sub-optimal. Although the search results by using forward probability and Viterbi probability could in principle be different, in practice this is rarely the case. We use this approximation for the rest of this chapter.

The Viterbi search has already been discussed as a solution to one of the three fundamental HMM problems in Chapter 8. It can be executed very efficiently via the same trellis framework. To briefly reiterate, the Viterbi search is a time-synchronous search algorithm that completely processes time t before going on to time t+1. For time t, each state is updated by the best score (instead of the sum of all incoming paths) from all states in at time t-1. This is why it is often called *time-synchronous Viterbi search*. When one update occurs, it also records the backtracking pointer to remember the most probable incoming state. At the end of search, the most probable state sequence can be recovered by tracing back these backtracking pointers. The Viterbi algorithm provides an optimal solution for handling nonlinear time warping between hidden Markov models and acoustic observation, word boundary detection and word identification in continuous speech recognition. This unified Viterbi search algorithm serves as the basis for all search algorithms as described in the rest of the chapter.

It is necessary to clarify the backtracking pointer for time-synchronous Viterbi search for continuous word recognition. We are generally not interested in the optimal state sequence for speech recognition.[8] Instead, we are only interested in the optimal word sequence indicated by Eq. (12.16). Therefore, we use the backtrack pointer just to remember the word history for the current path, so the optimal word sequence can be recovered at the end of search. To be more specific, when we reach the final state of a word, we create a history node containing the word identity and current time index and append this history node to the existing backtrack pointer. This backtrack pointer is then passed onto the successor node if it is the optimal path leading to the successor node for both intra-word and inter-word transition. The side benefit of keeping this backtrack pointer is that we no longer need to keep the entire trellis during the search. Instead, we only need space to keep two successive time slices (columns) in the trellis computation (the previous time slice and the current time slice) because all the backtracking information is now kept in the backtrack pointer. This simplification is a significant benefit in the implement of a time-synchronous Viterbi search.

Time-synchronous Viterbi search can be considered as a *breadth-first search* with dynamic programming. Instead of performing a tree search algorithm, the dynamic program-

[8] While we are not interested in optimal state sequences for ASR, they are very useful to derive phonetic segmentation, which could be provide important information for developing ASR systems.

ming principle helps create a search graph where multiple paths leading to the same search state are merged by keeping the best path (with minimum cost). The Viterbi trellis is a representation of the search graph. Therefore, all the efficient techniques for graph search algorithms can be applied to time-synchronous Viterbi search. Although so far we have described the trellis in an explicit fashion – the whole search space needs to be explored before the optimal path can be found, it is not necessary to do so. When the search space contains an enormous number of states, it becomes impractical to pre-compile the composite HMM entirely and store it in the memory. It is preferable to dynamically build and allocate portions of the search space sufficient to search the promising paths. By using the graph search algorithm described in Section 12.1.1, only part of the entire Viterbi trellis is generated explicitly. By constructing the search space dynamically, the computation cost of the search is proportional only to the number of active hypotheses, independent of the overall size of the potential search space. Therefore, dynamically generated trellises are key to heuristic Viterbi search for efficient large-vocabulary continuous speech recognition, as described in Chapter 13.

12.4.1. The Use of Beam

Based on Chapter 8, the search space for Viterbi search is $O(NT)$ and the complexity is $O(N^2T)$, where N is the total number of HMM states and T is the length of the utterance. For large-vocabulary tasks these numbers are astronomically large even with the help of dynamic programming. In order to avoid examining the overwhelming number of possible cells in the HMM trellis, a heuristic search is clearly needed. Different heuristics generate or explore portions of the trellis in different ways.

A simple way to prune the search space for breadth-first search is the beam search described in Section 12.1.3.2. Instead of retaining all candidates (cells) at every time frame, a threshold T is used to keep only a subset of promising candidates. The state at time t with the lowest cost $D_{\min}$ is first identified. Then each state at time t with cost $> D_{\min} + T$ is discarded from further consideration before moving on to the next time frame t+1. The use of the beam alleviates the need to process all the cells. In practice, it can lead to substantial savings in computation with little or no loss of accuracy.

Although beam search is a simple idea, the combination of time-synchronous Viterbi and beam search algorithms produces the most powerful search strategy for large-vocabulary speech recognition. Comparing paths with equal length under a time-synchronous search framework makes beam search possible. That is, for two different word sequences $\mathbf{W}_1$ and $\mathbf{W}_2$, the posterior probability $P(\mathbf{W}_1 | \mathbf{x}_0^t)$ and $P(\mathbf{W}_2 | \mathbf{x}_0^t)$ are always compared based on the same partial acoustic observation $\mathbf{x}_0^t$. This makes the comparison straightforward because the denominator $P(\mathbf{x}_0^t)$ in Eq. (12.5) is the same for both terms and can be ignored. Since the score comparison for each time frame is fair, the only assumption of beam search is that an optimal path should have good enough partial-path score for each time frame to survive under beam pruning.

The time-synchronous framework is one of the aspects of Viterbi beam search that is critical to its success. Unlike the time-synchronous framework, time-asynchronous search algorithms such as stack decoding require the normalization of likelihood scores over feature streams of different time lengths. This, as we will see in Section 12.5, is the Achilles heel of that approach.

The straightforward time-synchronous Viterbi beam search is ineffective in dealing with the gigantic search space of high perplexity tasks. However, with a better understanding of the linguistic search space and the advent of techniques for obtaining *n*-best lists from time-synchronous Viterbi search, described in Chapter 13, time-synchronous Viterbi beam search has turned out to be surprisingly successful in handling tasks of all sizes and all different types of grammars, including FSG, CFG, and n-gram [2, 14, 18, 28, 34, 38, 44]. Therefore, it has become the predominant search strategy for continuous speech recognition.

12.4.2. Viterbi Beam Search

To explain the time-synchronous Viterbi beam search in a formal way [31], we first define some quantities:

$D(t; s_t; w) \equiv$ total cost of the best path up to time t that ends in state s_t of grammar word state w.

$h(t; s_t; w) \equiv$ backtrack pointer for the best path up to time t that ends in state s_t of grammar word state w.

Readers should be aware that w in the two quantities above represents a grammar word state in the search space. It is different from just the word identity since the same word could occur in many different language model states, as in the trigram search space shown in Figure 12.17.

There are two types of dynamic programming (DP) transition rules [30], namely intra-word and inter-word transition. The intra-word transition is just like the Viterbi rule for HMMs and can be expressed as follows:

$$D(t; s_t; w) = \min_{s_{t-1}} \{ d(\mathbf{x}_t, s_t \mid s_{t-1}; w) + D(t-1; s_{t-1}; w) \} \tag{12.17}$$

$$h(t; s_t; w) = h(t-1, b_{\min}(t; s_t; w); w) \tag{12.18}$$

where $d(\mathbf{x}_t, s_t \mid s_{t-1}; w)$ is the cost associated with taking the transition from state s_{t-1} to state s_t while generating output observation $\mathbf{x}_t$, and $b_{\min}(t; s_t; w)$ is the optimal predecessor state of cell $D(t; s_t; w)$. To be specific, they can be expressed as follows:

$$d(\mathbf{x}_t, s_t \mid s_{t-1}; w) = -\log P(s_t \mid s_{t-1}; w) - \log P(\mathbf{x}_t \mid s_t; w) \tag{12.19}$$

$$b_{\min}(t; s_t; w) = \arg\min_{s_{t-1}} \{ d(\mathbf{x}_t, s_t \mid s_{t-1}; w) + D(t-1; s_{t-1}; w) \} \tag{12.20}$$

The inter-word transition is basically a null transition without consuming any observation. However, it needs to deal with creating a new history node for the backtracking pointer. Let's define $F(w)$ as the final state of word HMM *w* and $I(w)$ as the initial state of word HMM *w*. Moreover, state η is denoted as the pseudo initial state. The inter-word transition can then be expressed as follows:

$$D(t;\eta;w) = \min_{v}\{\log P(w \mid v) + D(t;F(v);v)\} \tag{12.21}$$

$$h(t;\eta;w) = \langle v_{\min}, t\rangle :: h(t, F(v_{\min}); v_{\min}) \tag{12.22}$$

where $v_{\min} = \arg\min_{v}\{\log P(w \mid v) + D(t;F(v);v)\}$ and :: is a link appending operator.

The time-synchronous Viterbi beam search algorithm assumes that all the intra-word transitions are evaluated before inter-word null transitions take place. The same time index is used intentionally for inter-word transition since the null language model state transition does not consume an observation vector. Since the initial state $I(w)$ for word HMM *w* could have a self-transition, the cell $D(t;I(w);w)$ might already have active path. Therefore, we need to perform the following check to advance the inter-word transitions.

$$\begin{aligned} &\text{if } D(t;\eta;w) < D(t;I(w);w) \\ &\quad D(t;I(w);w) = D(t;\eta;w) \text{ and } h(t;I(w);w) = h(t;\eta;w) \end{aligned} \tag{12.23}$$

The time-synchronous Viterbi beam search can be summarized as in Algorithm 12.6. For large-vocabulary speech recognition, the experimental results shows that only a small percentage of the entire search space (the beam) needs to be kept for each time interval *t* without increasing error rates. Empirically, the beam size has typically been found to be between 5% and 10% of the entire search space. In Chapter 13 we describe strategies of using different level of beams for more effectively pruning.

12.5. STACK DECODING (A* SEARCH)

If some reliable heuristics are available to guide the decoding, the search can be done in a depth-first fashion around the best path early on, instead of wasting efforts on unpromising paths via the time-synchronous beam search. Stack decoding represents the best attempt to use A* search instead of time-synchronous beam search for continuous speech recognition. Unfortunately, as we will discover in this section, such a heuristic function $h(\bullet)$ (defined in Section 12.1.3) is very difficult to attain in continuous speech recognition, so search algorithms based on A* search are in general less efficient than time-synchronous beam search.

Stack decoding is a variant of the heuristic A* search based on the forward algorithm, where the evaluation function is based on the forward probability. It is a tree search algorithm, which takes a slightly different viewpoint than the time-synchronous Viterbi search. Time-synchronous beam search is basically a breadth-first search, so it is crucial to control

the number of all possible language model states as described in Section 12.3. In a typical large-vocabulary Viterbi search with n-gram language models, this number is determined by the equivalent classes of language model histories. On the other hand, stack decoding as a tree search algorithm treats the search as a task for finding a path in a tree whose branches correspond to words in the vocabulary *V*, non-terminal nodes correspond to incomplete sentences, and terminal nodes correspond to complete sentences. The search tree has a constant branching factor of |*V*|, if we allow every word to be followed by every word. Figure 12.20 illustrated such a search tree for a vocabulary with three words [19].

ALGORITHM 12.6 TIME-SYNCHRONOUS VITERBI BEAM SEARCH

Initialization: For all the grammar word states *w* which can start a sentence,

$$D(0; I(w); w) = 0$$

$$h(0; I(w); w) = null$$

Induction: For time $t = 1$ to T do

For all active states do

Intra-word transitions according to Eq. (12.17) and (12.18)

$$D(t; s_t; w) = \min_{s_{t-1}} \{ d(\mathbf{x}_t, s_t \mid s_{t-1}; w) + D(t-1; s_{t-1}; w) \}$$

$$h(t; s_t; w) = h(t-1, b_{\min}(t; s_t; w); w)$$

For all active word-final states do

Inter-word transitions according to Eq. (12.21), (12.22) and (12.23)

$$D(t; \eta; w) = \min_{v} \{ \log P(w \mid v) + D(t; F(v); v) \}$$

$$h(t; \eta; w) = \langle v_{\min}, t \rangle :: h(t, F(v_{\min}); v_{\min})$$

if $D(t; \eta; w) < D(t; I(w); w)$

$D(t; I(w); w) = D(t; \eta; w)$ and $h(t; I(w); w) = h(t; \eta; w)$

Pruning: Find the cost for the best path and decide the beam threshold

Prune unpromising hypotheses

Termination: Pick the best path among all the possible final states of grammar at time T

Obtain the optimal word sequence according to the backtracking pointer $h(t; \eta; w)$

An important advantage of stack decoding is its consistency with the forward-backward training algorithm. Viterbi search is a graph search, and paths cannot be easily summed because they may have different word histories. In general, the Viterbi search finds the optimal state sequence instead of optimal word sequence. Therefore, Viterbi approximation is necessary to make the Viterbi search feasible, as described in Section 12.4. Stack decoding is a tree search, so each node has a unique history, and the forward algorithm can be used within word model evaluation. Moreover, all possible beginning and ending times (shaded areas in Figure 12.21) beginning and ending times are considered [24]. With stack

decoding, it is possible to use an objective function that searches for the optimal word string, rather than the optimal state sequence. Furthermore, it is in principle natural for stack decoding to accommodate long-range language models if the heuristics can guide the search to avoid exploring the overwhelmingly large unpromising grammar states.

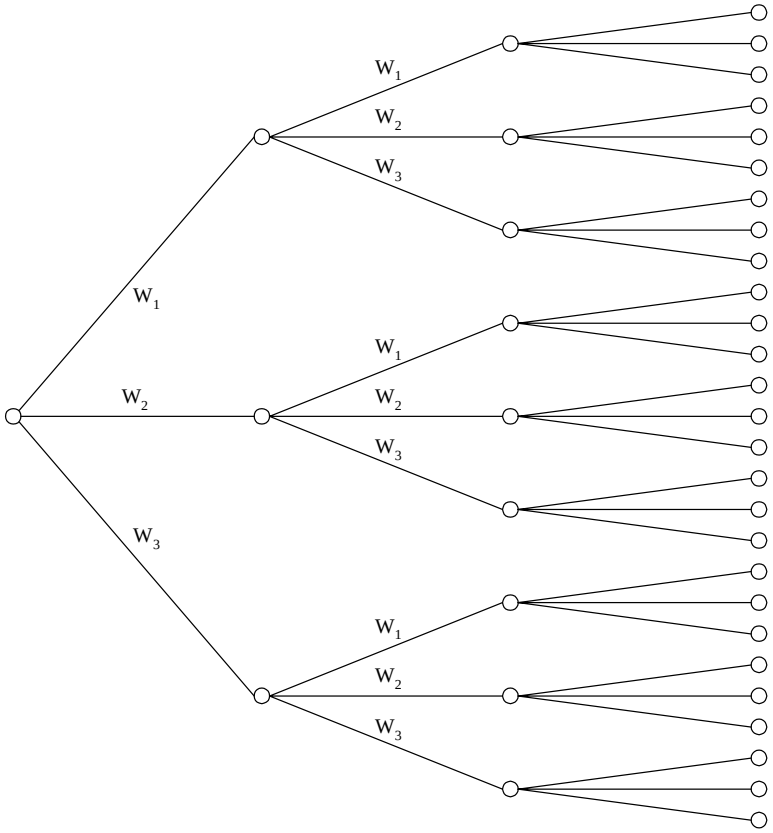

Figure 12.20 A stack decoding search tree for a vocabulary size of three [19].

By formulating stack decoding in a tree search framework, the graph search algorithms described in Section 12.1 can be directly applied to stack decoding. Obviously, blind-search methods, like depth-first and breadth-first search, that do not take advantage of goodness measurement of how close we are getting to the goal, are usually computationally infeasible in practical speech recognition systems. A* search is clearly attractive for speech recognition given the hope of a sufficient heuristic function to guide the tree search in a favorable direction without exploring too many unpromising branches and nodes. In contrast to the Viterbi search, it is not time-synchronous and extends paths of different lengths.

The search begins by adding all possible one-word hypotheses to the *OPEN* list. Then the best path is removed from the *OPEN* list, and all paths from it are extended, evaluated,

and placed back in the *OPEN* list. This search continues until a complete path that is guaranteed to be better than all paths in the *OPEN* list has been found.

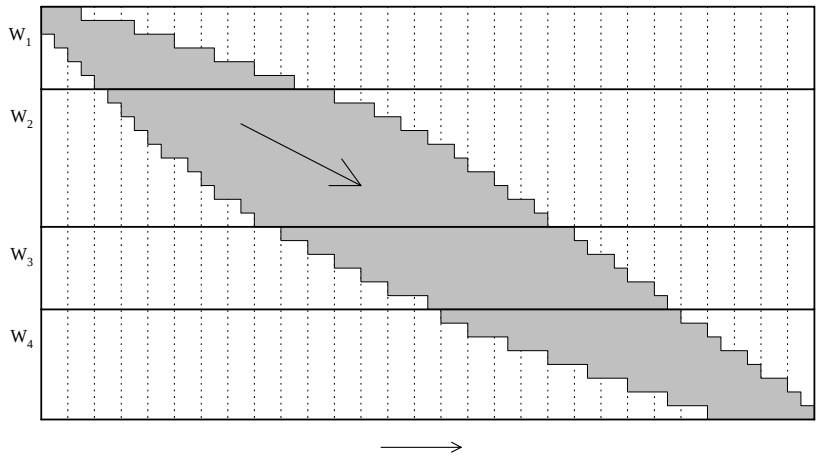

Figure 12.21 The forward trellis space for stack decoding. Each grid point corresponds to a trellis cell in the forward computation. The shaded area represents the values contributing to the computation of the forward score for the optimal word sequence $w_1, w_2, w_3, \ldots$ [24].

Unlike Viterbi search, where the acoustic probabilities being compared are always based on the same partial input, it is necessary to compare the goodness of partial paths of different lengths to direct the A* tree search. Moreover, since stack decoding is done asynchronously, we need an effective mechanism to determine when to end a phone/word evaluation and move on to the next phone/word. Therefore, the heart and soul of the stack decoding is clearly in

1. Finding an effective and efficient heuristic function for estimating the future remaining input feature stream and
2. Determining when to extend the search to the next word/phone.

In the following section we describe these two critical components. Readers will note that the solution to these two issues are virtually the same - using a normalization scheme to compare paths of different lengths.

12.5.1. Admissible Heuristics for Remaining Path

The key issue in heuristic search is the selection of an evaluation function. As described in Section 12.1.3, the heuristic function of the path H_N going through node *N* includes the cost up to the node and the estimate of the cost to the target node from node *N*. Suppose path H_N

is going through node *N* at time *t*; then the evaluation for path H_N can be expressed as follows:

$$f(H_N^t) = g(H_N^t) + h(H_N^{t,T}) \tag{12.24}$$

where $g(H_N^t)$ is the evaluation function for the partial path of H_N up to time *t*, and $h(H_N^{t,T})$ is the heuristic function of the remaining path from $t+1$ to *T* for path H_N. The challenge for stack decoders is to devise an admissible function for $h(\bullet)$.

According to Section 12.1.3.1, an admissible heuristic function is one that always under estimates the true cost of the remaining path from $t+1$ to *T* for path H_N. A trivially admissible function is the zero function. In this case, it results in a very large OPEN list. In addition, since the longer paths tend to have higher cost because of the gradually accumulated cost, the search is likely to be conducted in a breadth-first fashion, which functions very much like a plain Viterbi search. The evaluation function $g(\bullet)$ can be obtained easily by using the HMM forward score as the true cost up to current time *t*. However, how can we find an admissible heuristic function $h(\bullet)$? We present the basic concept here [19, 35].

The goal of $h(\bullet)$ is to find the expected cost for the remaining path. If we can obtain the expected cost per frame ψ for the remaining path, the total expected cost, $(T-t)*\psi$, is simply the product of ψ and the length of the remaining path. One way to find such expected cost per frame is to gather statistics empirically from training data.

1. After the final training iteration, perform Viterbi forced alignment[9] with each training utterance to get an optimal time alignment for each word.
2. Randomly select an interval to cover the number of words ranging from two to ten. Denote this interval as $[i \ldots j]$
3. Compute the average acoustic cost per frame within this selected interval according to the following formula and save the value in a set Λ.

$$\frac{-1}{j-i}\log P(\mathbf{x}_i^j \mid \mathbf{w}_{i\ldots j}) \tag{12.25}$$

   where $\mathbf{w}_{i\ldots j}$ is the word string corresponding to interval $[i \ldots j]$
4. Repeat Steps 2 and 3 for the entire training set.
5. Define $\psi_{\min}$ and ψ_{avg} as the minimum and average value found in set Λ.

[9] Viterbi forced alignment means that the Viterbi is performed on the HMM model constructed from the known word transcription. The term "forced" is used because the Viterbi alignment is forced to be performed on the correct model. Viterbi forced alignment is a very useful tool in spoken language processing as it can provide the optimal state-time alignment with the utterances. This detailed alignment can then be used for different purposes, including discriminant training, concatenated speech synthesis, etc.

Clearly, $\psi_{\min}$ should be a good under-estimate of the expected cost per frame for the future unknown path. Therefore, the heuristic function $h(H_N^{t,T})$ can be derived as:

$$h(H_N^{t,T}) = (T-t)\psi_{\min} \tag{12.26}$$

Although $\psi_{\min}$ is obtained empirically, stack decoding based on Eq. (12.26) will generally find the optimal solution. However, the search using $\psi_{\min}$ usually runs very slowly, since Eq. (12.26) always under-estimates the true cost for any portion of speech. In practice, a heuristic function like ψ_{avg} that may over-estimate has to be used to prune more hypotheses. This speeds up the search at the expense of possible search errors, because ψ_{avg} should represent the average cost per frame for any portion of speech. In fact, there is an argument that one might be able to use a heuristic function even less than ψ_{avg}. The argument is that ψ_{avg} is derived from the correct path (training data) and the average cost per frame for all paths during search should be less than ψ_{avg} because the paths undoubtedly include correct and incorrect ones.

12.5.2. When to Extend New Words

Since stack decoding is executed asynchronously, it becomes necessary to detect when a phone/word ends, so that the search can extend to the next phone/word. If we have a cost measure that indicates how well an input feature vector of any length matches the evaluated model sequence, this cost measure should drop slowly for the correct phone/word and rise sharply for an incorrect phone/word. In order to do so, it implies we must be able to compare hypotheses of different lengths.

The first thing that comes to mind for this cost measure is simply the forward score $-\log P(\mathbf{x}_1^t, s_t \mid w_1^k)$, which represents the likelihood of producing acoustic observation $\mathbf{x}_1^t$ based on word sequence w_1^k and ending at state s_t. However, it is definitely not suitable because it is deemed to be smaller for a shorter acoustic input vector. This causes the search to almost always prefer short phones/words, resulting in many insertion errors. Therefore, we must derive some normalized score that satisfies the desired property described above. The normalized cost $\hat{C}(\mathbf{x}_1^t, s_t \mid w_1^k)$ can be represented as follows [6, 24]:

$$\hat{C}(\mathbf{x}_1^t, s_t \mid w_1^k) = -\log\left[\frac{P(\mathbf{x}_1^t, s_t \mid w_1^k)}{\gamma^t}\right] = -\log\left[P(\mathbf{x}_1^t, s_t \mid w_1^k)\right] + t\log\gamma \tag{12.27}$$

where γ ($0 < \gamma < 1$) is a constant normalization factor.

Suppose the search is now evaluating a particular word w_k; we can define $\hat{C}_{\min}(t)$ as the minimum cost for $\hat{C}(\mathbf{x}_1^t, s_t \mid w_1^k)$ for all the states of w_k, and $\alpha_{\max}(t)$ as the maximum forward probability for $P(\mathbf{x}_1^t, s_t \mid w_1^k)$ for all the states of w_k. That is,

$$\hat{C}_{\min}(t) = \min_{s_t \in w_k} \left[\hat{C}(\mathbf{x}_1^t, s_t \mid w_1^k) \right] \tag{12.28}$$

$$\alpha_{\max}(t) = \max_{s_t \in w_k} \left[P(\mathbf{x}_1^t \mid w_1^k, s_t) \right] \tag{12.29}$$

We want $\hat{C}_{\min}(t)$ to be near 0 just so long as the phone/word we are evaluating is the correct one and we have not gone beyond its end. On the other hand, if the phone/word we are evaluating is the incorrect one or we have already passed its end, we want the $\hat{C}_{\min}(t)$ to be rising sharply. Similar to the procedure of finding the admissible heuristic function, we can set the normalized factor γ empirically during training so that $\hat{C}_{\min}(T) = 0$ when we know the correct word sequence **W** that produces acoustic observation sequence $\mathbf{x}_1^T$. Based on Eq. (12.27), γ should be set to:

$$\gamma = \sqrt[T]{\alpha_{\max}(T)} \tag{12.30}$$

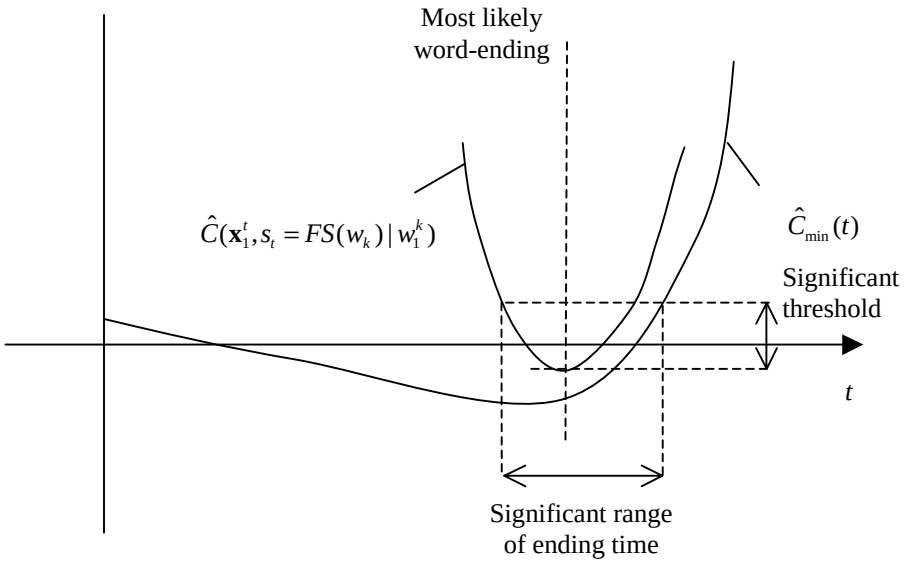

Figure 12.22 $\hat{C}_{\min}(t)$ and $\hat{C}(\mathbf{x}_1^t, s_t = FS(w_k) \mid w_1^k)$ as functions of time *t*. The valley region represents possible ending times for the correct phone/word.

Figure 12.22 shows a plot of $\hat{C}_{\min}(t)$ as a function of time for correct match. In addition, the cost for the final state $FS(w_k)$ of word w_k, $\hat{C}(\mathbf{x}_1^t, s_t = FS(w_k) \mid w_1^k)$, which is the score for w_k-ending path, is also plotted. There should be a valley centered around 0 for $\hat{C}(\mathbf{x}_1^t, s_t = FS(w_k) \mid w_1^k)$, which indicates the region of possible ending time for the correct phone/word. Sometimes a stretch of acoustic observations match may better than the average cost, pushing the curve below 0. Similarly, a stretch of acoustic observations may match worse than the average cost, pushing the curve above 0.

There is an interesting connection between the normalized factor γ and the heuristic estimate of the expected cost per frame, ψ, defined in Eq. (12.25). Since the cost is simply the logarithm on the inverse posterior probability, we get the following equation.

$$\psi = \frac{-1}{T} \log P(\mathbf{x}_1^T \mid \hat{\mathbf{W}}) = -\log\left[\alpha_{\max}(T)^{1/T}\right] = -\log\gamma \tag{12.31}$$

Equation (12.31) reveals that these two quantities are basically the same estimate. In fact, if we subtract the heuristic function $f(H_N^t)$ defined in Eq. (12.24) by the constant $\log\left(e^T\right)$, we get exactly the same quantity as the one defined in Eq. (12.27). Decisions on which path to extend first based on the heuristic function and when to extend the search to the next word/phone are basically centered on comparing partial theories with different lengths. Therefore, the normalized cost $\hat{C}(\mathbf{x}_1^t, s_t \mid w_1^k)$ can be used for both purposes.

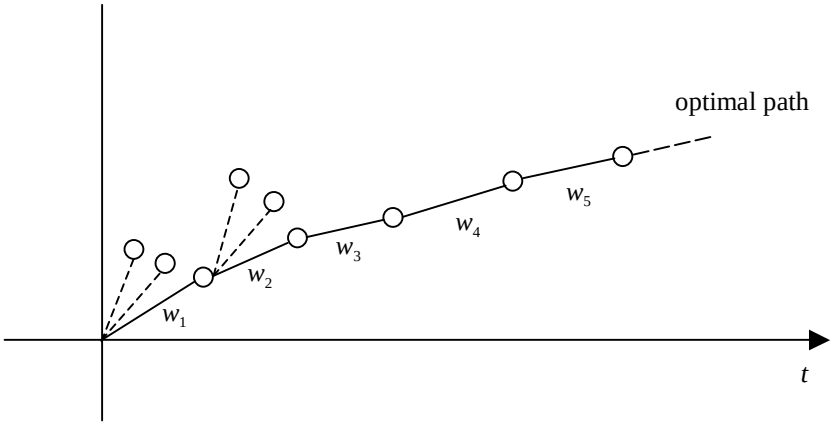

Figure 12.23 Unnormalized cost $C(\mathbf{x}_1^t, s_t \mid w_1^k)$ for optimal path and other competing paths as a function of time.

Based on the connection we have established, the heuristic function, $f(H_N^t)$, which estimate the goodness of a path is simply replaced by the normalized evaluation function

$\hat{C}(\mathbf{x}_1^t, s_t \mid w_1^k)$. If we plot the un-normalized cost $C(\mathbf{x}_1^t, s_t \mid w_1^k)$ for the optimal path and other competing paths as the function time *t*, the cost values increase as paths get longer (illustrated in Figure 12.23) because every frame adds some non-negative cost to the overall cost. It is clear that using un-normalized cost function $C(\mathbf{x}_1^t, s_t \mid w_1^k)$ generally results in a breadth-first search. What we want is an evaluation that decreases slightly along the optimal path, and hopefully increases along other competing paths. Clearly, the normalized cost function $\hat{C}(\mathbf{x}_1^t, s_t \mid w_1^k)$ fulfills this role, as shown in Figure 12.24.

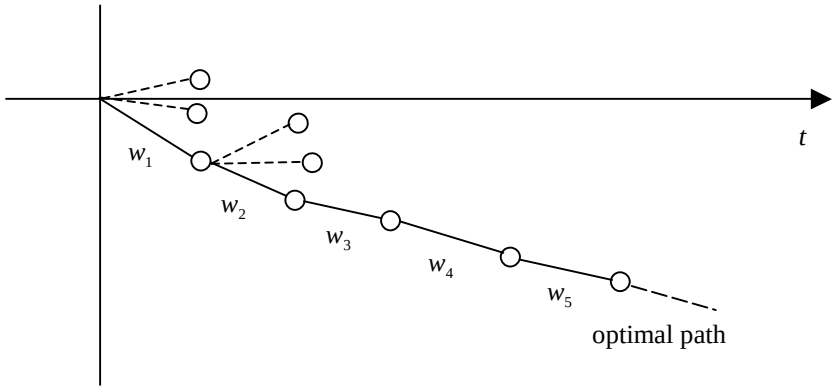

Figure 12.24 Normalized cost $\hat{C}(\mathbf{x}_1^t, s_t \mid w_1^k)$ for the optimal path and other competing paths as a function of time.

Equation (12.30) is a context-less estimation of the normalized factor, which is also referred to as zero-order estimate. To improve the accuracy of the estimate, you can use context-dependent higher-order estimates like [24]:

$\gamma_i = \gamma(\mathbf{x}_i)$ first-order estimate

$\gamma_i = \gamma(\mathbf{x}_i, \mathbf{x}_{i-1})$ second-order estimate

$\gamma_i = \gamma(\mathbf{x}_i, \mathbf{x}_{i-1}, \ldots, \mathbf{x}_{i-N+1})$ n-order estimate

Since the normalized factor *e* is estimated from the training data that is also used to train the parameters of the HMMs, the normalized factor e_i tends to be an over estimate. As a result, $\alpha_{\max}(t)$ might rise slowly for test data even when the correct phone/word model is evaluated. This problem is alleviated by introducing some other scaling factor $\delta < 1$ so that $\alpha_{\max}(t)$ falls slowly for test data for when evaluating the correct phone/word model. The best solution for this problem is to use an independent data set other than the training data to derive the normalized factor γ_i.

12.5.3. Fast Match

Even with an efficient heuristic function and mechanism to determine the ending time for a phone/word, stack decoding could still be too slow for large-vocabulary speech recognition tasks. As described in Section 12.5.1, an effective underestimated heuristic function for the remaining portion of speech is very difficult to derive. On the other hand, a heuristic estimate for the immediate short segment that usually corresponds to the next phone or word may be feasible to attain. In this section, we describe the fast-match mechanism that reduces phone/word candidates for detailed match (expansion).

In asynchronous stack decoding, the most expensive step is to extend the best subpath. For a large-vocabulary search, it implies the calculation of $P(\mathbf{x}_t^{t+k} \mid w)$ over the entire vocabulary size $|V|$. It is desirable to have a fast computation to quickly reduce the possible words starting at a given time *t* to reduce the search space. This process is often referred to as *fast match* [15, 35]. In fact, fast match is crucial to stack decoding, of which it becomes an integral part. Fast match is a method for the rapid computation of a list of candidates that constrain successive search phases. The expensive *detailed match* can then be performed after fast match. In this sense, fast match can be regarded as an additional pruning threshold to meet before new word/phone can be started.

Fast match, by definition, needs to use only a small amount of computation. However, it should also be accurate enough not to prune away any word/phone candidates that participate in the best path eventually. Fast match is in general characterized by the approximations that are made in the acoustic/language models in order to reduce computation. There is an obvious trade-off between these two objectives. Fortunately, many systems [15] have demonstrated that one needs to sacrifice very little accuracy in order to speed up the computation considerably.

Similar to *admissibility* in A* search, there is also an *admissibility* property in fast match. A fast match method is called admissible if it never prunes away the word/phone candidates that participate in the optimal path. In order words, a fast match is admissible if the recognition errors that appear in a system using the fast match followed by a detailed match are those that would appear if the detailed match was carried out for all words/phones in the vocabulary. Since fast match can be applied to either word or phone level, as we describe in the next section, we explain the admissibility for the case of word-level fast match for simplicity. The same principle can be easily extended to phone-level fast match.

Let V be the vocabulary and $C(\mathbf{X} \mid w)$ be the cost of a detailed match between input $\mathbf{X}$ and word *w*. Now $F(\mathbf{X} \mid w)$ is an estimator of $C(\mathbf{X} \mid w)$ that is accurate enough and fast to compute. A word list selected by fast match estimator can be attained by first computing $F(\mathbf{X} \mid w)$ for each word *w* of the vocabulary. Suppose w_b is the word for which the fast match has a minimum cost value:

$$w_b = \arg\min_{w \in V} F(\mathbf{X} \mid w) \tag{12.32}$$

After computing $C(\mathbf{X} \mid w_b)$, the detailed match cost for w_b, we form the fast match word list, Λ, from the word w in the vocabulary such that $F(\mathbf{X} \mid w)$ is no greater than $C(\mathbf{X} \mid w_b)$. In other words,

$$\Lambda = \{ w \in V \mid F(\mathbf{X} \mid w) \le C(\mathbf{X} \mid w_b) \} \tag{12.33}$$

Similar to the admissibility condition for A* search [3, 33], the fast match estimator $F(\bullet)$ conducted in the way described above is admissible if and only if $F(\mathbf{X} \mid w)$ is always an under-estimator (lower bound) of detailed match $C(\mathbf{X} \mid w)$. That is,

$$F(\mathbf{X} \mid w) \le C(\mathbf{X} \mid w) \qquad \forall \mathbf{X}, w \tag{12.34}$$

The proof is straightforward. If the word w_c has a lower detailed match cost $C(\mathbf{X} \mid w_c)$, you can prove that it must be included in the fast match list Λ because

$$C(\mathbf{X} \mid w_c) \le C(\mathbf{X} \mid w_b) \text{ and } F(\mathbf{X} \mid w_c) \le C(\mathbf{X} \mid w_c) \Rightarrow F(\mathbf{X} \mid w_c) \le C(\mathbf{X} \mid w_b)$$

Therefore, based on the definition of Λ, $w_c \in \Lambda$

Now the task is to find an admissible fast match estimator. Bahl et al. [6] proposed one fast match approximation for discrete HMMs. As we will see later, this fast match approximation is indeed equivalent to a simplification of the HMM structure. Given the HMM for word w and an input sequence x_1^T of codebook symbols describing the input signal, the probability that the HMM w produces the VQ sequence x_1^T is given by (according to Chapter 8):

$$P(x_1^T \mid w) = \sum_{s_1, s_2, \ldots s_T} \left[P_w(s_1, s_2, \ldots s_T) \prod_{i=1}^{T} P_w(x_i \mid s_i) \right] \tag{12.35}$$

Since we often use Viterbi approximation instead of the forward probability, the equation above can be approximated by:

$$P(x_1^T \mid w) \cong \max_{s_1, s_2, \ldots s_T} \left[P_w(s_1, s_2, \ldots s_T) \prod_{i=1}^{T} P_w(x_i \mid s_i) \right] \tag{12.36}$$

The detailed match cost $C(\mathbf{X} \mid w)$ can now be represented as:

$$C(\mathbf{X} \mid w) = \min_{s_1, s_2, \ldots s_T} \left\{ -\log \left[P_w(s_1, s_2, \ldots s_T) \prod_{i=1}^{T} P_w(x_i \mid s_i) \right] \right\} \tag{12.37}$$

Since the codebook size is finite, it is possible to compute, for each model w, the highest output probability for every VQ label c among all states s_k in HMM w. Let's define $m_w(c)$ to be the following:

$$m_w(c) = \max_{s_k \in w} P_w(c \mid s_k) = \max_{s_k \in w} b_k(c) \tag{12.38}$$

We can further define the $q_{\max}(w)$ as the maximum state sequence with respect to *T*, i.e., the maximum probability of any complete path in HMM *w*.

$$q_{\max}(w) = \max_{T} \left[P_w(s_1, s_2, \ldots s_T) \right] \tag{12.39}$$

Now let's define the fast match estimator $F(\mathbf{A} \mid w)$ as the following:

$$F(\mathbf{X} \mid w) = -\log \left[q_{\max}(w) \prod_{i=1}^{T} m_w(x_i) \right] \tag{12.40}$$

It is easy to show the fast match estimator $F(\mathbf{X} \mid w) \le C(\mathbf{X} \mid w)$ is admissible based on Eq. (12.38) to Eq. (12.40).

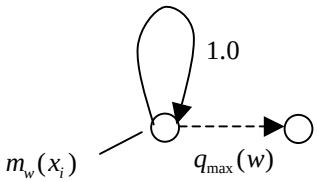

Figure 12.25 The equivalent one-state HMM corresponding to fast match computation defined in Eq. (12.40) [15].

The fast match estimator defined in Eq. (12.40) requires *T*+1 additions for a vector sequence of length *T*. The operation can be viewed as equivalent to the forward computation with a one-state HMM of the form shown in Figure 12.25. This correspondence can be interpreted as a simplification of the original multiple-state HMM into such a one-state HMM. It thus explains why fast match can be computed much faster than detailed match. Readers should note that this HMM is not actually a true HMM by strict definition, because the output probability distribution $m_w(c)$ and the transition probability distribution do not add up to one.

The fast match computation defined in Eq. (12.40) discards the sequence information with the model unit since the computation is independent of the order of input vectors. Therefore, one needs to decide the acoustic unit for fast match. In general, the longer the unit, the faster the computation is, and, therefore, the larger the under-estimation of detailed match scores $C(\mathbf{X} \mid w)$. It thus becomes a trade-off between accuracy and speed.

Now let's analyze the real speedup by using fast match to reduce the vocabulary *V* to the list Λ, followed by the detailed match. Let $|V|$ and $|\Lambda|$ be the sizes for the vocabulary *V* and the fast match short list Λ. Suppose t_f and t_d are the times required to compute one fast match score and one detailed match score for one word, respectively. Then, the total time required for the fast match followed by the detailed match is $t_f |V| + t_d |\Lambda|$ whereas

the time required in doing the detailed match alone for the entire vocabulary is $t_d |V|$. The speed up ratio is then given as follows:

$$\frac{1}{\left(\frac{t_f}{t_d}+\frac{|\Lambda|}{|V|}\right)} \tag{12.41}$$

We need t_f to be much smaller than t_d and $|\Lambda|$ to be much smaller than $|V|$ to have a significant speed-up using fast match. Using our admissible fast match estimator in Eq. (12.40), the time complexity of the computation for $F(\mathbf{X}|w)$ is T instead of N^2T for $D(\mathbf{X}|w)$, where N is the number of states in the detailed acoustic model. Therefore, the t_f/t_d saving is about N^2.

In general, in order to make $|\Lambda|$ much smaller than $|V|$, one needs a very accurate fast match estimator that could result in $t_f \approx t_d$. This is why we often relax the constraint of admissibility, although it is a nice principle to adhere to. In practice, most real-time speech recognition systems don't necessarily obey the admissibility principle with the fast match. For example, Bahl et al. [10], Laface et al., [22] and Roe et al., [36] used several techniques to construct off-line groups of acoustically similar words. Armed with this grouping, they can use an aggressive fast match to select only a very short list of words and words acoustically similar to the words in this list are added to form the *NV* list for further detailed match processing. By doing so, they are able to report a very efficient fast match method that misses the correct word only 2% of the time. When non-admissible fast match is used, one needs to minimize the additional search error introduced by fast match empirically.

Bahl et al. [6] use a one-state HMM as their fast match units and a tree-structure lexicon similar to the lexical tree structures introduced in Chapter 13 to construct the short word list Λ for next-word expansion in stack decoding. Since the fast match tree search is also done in an asynchronous way, the ending time of each phone is detected using normalized scores similar to those described in Section 12.5.2. It is based on the same idea that this normalized score rises slowly for the correct phone, while it drops rapidly once the end of phone is encountered (so the model is starting to go toward the incorrect phones). During the asynchronous lexical tree search, the unpromising hypotheses are also pruned away by a pruning threshold that is constantly changing once a complete hypothesis (a leaf node) is obtained. On a 20,000-word dictation task, such a fast match scheme was about 100 times faster than detailed match and achieved real-time performance on a commercial workstation with only 0.34% increase in the word error rate being introduced by the fast match process.

12.5.4. Stack Pruning

Even with efficient heuristic functions, mechanism to determine the ending time for phone/word, and fast match, stack decoding might still be too slow for large-vocabulary speech recognition tasks. A beam within the stack, which saves only a small number of

promising hypotheses in the *OPEN* list, is often used to reduce search effort. This *stack pruning* is very similar to beam search. A predetermined threshold ε is used to eliminate hypotheses whose cost score is much worse than the best path so far.

Both fast match and stack pruning could introduce search errors where the eventual optimal path is thrown away prematurely. However, the impact could be reduced to a minimum by empirically adjusting the thresholds in both methods.

The implementation of stack decoding is, in general, more complicated, particularly when some inevitable pruning strategies are incorporated to make the search more efficient. The difficulty of devising both an effectively admissible heuristic function for $h(\bullet)$ and an effective estimation of normalization factors for boundary determination have limited the advantage that stack decoders have over Viterbi decoders. Unlike stack decoding, time-synchronous Viterbi beam search can use an easy comparison of same-length path without heuristic determination of word boundaries. As described in the earlier sections, these simple and unified features of Viterbi beam search allows researchers to incorporate various sound techniques to improve the efficiency of search. Therefore, time-synchronous Viterbi Beam search enjoys a much broader popularity in the speech community. However, the principle of stack decoding is essential particularly for n-best and lattice search. As we describe in Chapter 13, stack decoding plays a very crucial part in multiple-pass search strategies for n-best and lattice search because the early pass is able to establish a near-perfect estimate of the remaining path.

12.5.5. Multistack Search

Even with the help of normalized factor γ or heuristic function $h(\bullet)$, it is still more effective to compare hypotheses of the same length than those of different lengths, because hypotheses with the same length are compared based on the true forward matching score. Inspired by the time-synchronous principle in Viterbi beam search, researchers [8, 35] propose a variant stack decoding based on multiple stacks.

Multistack search is equivalent to a best-first search algorithm running on multiple stacks time-synchronously. Basically, the search maintains a separate stack for each time frame *t*, so it never needs to compare hypotheses of different lengths. The search runs time-synchronously from left to right just like time-synchronous Viterbi search. For each time frame *t*, multistack search extracts the best path out the *t*-stack, computes one-word extensions, and places all the new theories into the corresponding stacks. When the search finishes, the top path in the last stack is our optimal path. Algorithm 12.7 illustrates the mutlistack search algorithm.

This time-synchronous multistack search is designed based on the fact that by the time the t^{th} stack is extended; it already contains the best path that could ever be placed into it. This phenomenon is virtually a variant of the dynamic programming principle introduced in Chapter 8. To make multistack more efficient, some heuristic pruning can be applied to reduce the computation. For example, when the top path of each stack is extended for one more word, we could only consider extensions between minimum and maximum duration. On the other hand, when some heuristic pruning is integrated into the multistack search, one

might need to use a small beam in Step 2 of Algorithm 12.7 to extend more than just the best path to guarantee the admissibility.

ALGORITHM 12.7 MULTISTACK SEARCH

1. Initialization: for each word v in vocabulary V
   for $t = 1, 2, \ldots, T$
   Compute $C(\mathbf{x}_1^t \mid v)$ and insert it to t^{th} stack
2. Iteration: for $t = 1, 2, \ldots, T-1$
   Sort the t^{th} stack and pop the top path $C(\mathbf{x}_1^t \mid w_1^k)$ out the stack
   for each word v in vocabulary V
   for $\tau = t+1, t+2, \ldots, T$
   Extend the path $C(\mathbf{x}_1^t \mid w_1^k)$ by word v to get $C(\mathbf{x}_1^\tau \mid w_1^{k+1})$
   where $w_1^{k+1} = w_1^k \,||\, v$ and || means string concatenation
   Place $C(\mathbf{x}_1^\tau \mid w_1^{k+1})$ in τ^{th} stack
3. Termination: Sort the T^{th} stack and the top path is the optimal word sequence

12.6. HISTORICAL PERSPECTIVE AND FURTHER READING

Search has been one of the most important topics in artificial intelligence (AI) since the origins of the field. It plays the central role in general problem solving [29] and computer games. [43], Nilsson's *Principles of Artificial Intelligence* [32] and Barr and Feigenbaum's *The Handbook of Artificial Intelligence* [11] contain a comprehensive introduction to state-space search algorithms. A* search was first proposed by Hart et al. [17]. A* was thought to be derived from Dijkstra's algorithm [13] and Moore's algorithm [27]. A* search is similar to the *branch-and-bound* algorithm [23, 39], widely used in *operations research*. The proof of admissibility of A* search can be found in [32].

The application of *beam search* in speech recognition was first introduced by the HARPY system [26]. It wasn't widely popular until BBN used it for their BYBLOS system [37]. There are some excellent papers with detailed description of the use of time-synchronous Viterbi beam search for continuous speech recognition [24, 31]. Over the years, many efficient implementations and improvements have been introduced for time-synchronous Viterbi beam search, so real-time large-vocabulary continuous speech recognition can be realized on a general-purpose personal computer.

On the other hand, stack decoding was first developed by IBM [9]. It is successfully used in IBM's large-vocabulary continuous speech recognition systems [3, 16]. Lacking of time-synchronous framework, comparing theories of different lengths and extending theories is more complex as described in this chapter. Because of the complexity of stack decoding, far fewer publications and systems are based on it than on Viterbi beam search [16, 19,

20, 35]. With the introduction of multi-stack search [8], stack decoding in essence has actually come very close to time-synchronous Viterbi beam search.

Stack decoding is typically integrated with fast match methods to improve its efficiency. Fast match was first implemented for isolated word recognition to obtain a list of potential word candidates [5, 7]. The paper by Gopalakrishnan et al.'s paper [15] contains a comprehensive description of fast match techniques to reduce the word expansion for stack decoding. Besides the fast match techniques described in this chapter , there are a number of alternative approaches [5, 21, 41]. Waast's fast match [41], for example, is based on a binary classification tree built automatically from data that comprise both phonetic transcription and acoustic sequence.

REFERENCES

[1] Aho, A., J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, 1974, Addison-Wesley Publishing Company.

[2] Alleva, F., X. Huang, and M. Hwang, "An Improved Search Algorithm for Continuous Speech Recognition," *Int. Conf. on Acoustics, Speech and Signal Processing*, 1993, Minneapolis, MN pp. 307-310.

[3] Bahl, L.R. and e. al, "Large Vocabulary Natural Language Continuous Speech Recognition," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1989, Glasgow, Scotland pp. 465-467.

[4] Bahl, L.R., *et al.*, "Language-Model/Acoustic Channel Balance Mechanism," *IBM Technical Disclosure Bulletin*, 1980, **23**(7B), pp. 3464-3465.

[5] Bahl, L.R., *et al.*, "Obtaining Candidate Words by Polling in a Large Vocabulary Speech Recognition System," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1988 pp. 489-492.

[6] Bahl, L.R., *et al.*, "A Fast Approximate Acoustic Match for Large Vocabulary Speech Recognition," *IEEE Trans. on Speech and Audio Processing*, 1993(1), pp. 59-67.

[7] Bahl, L.R., *et al.*, "Matrix Fast Match: a Fast Method for Identifying a Short List of Candidate Words for Decoding," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1989, Glasgow, Scotland pp. 345-347.

[8] Bahl, L.R., P.S. Gopalakrishnan, and R.L. Mercer, "Search Issues in Large Vocabulary Speech Recognition," *Proc. of the 1993 IEEE Workshop on Automatic Speech Recognition*, 1993, Snowbird, UT.

[9] Bahl, L.R., F. Jelinek, and R. Mercer, "A Maximum Likelihood Approach to Continuous Speech Recognition," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 1983(2), pp. 179-190.

[10] Bahl, L.R., *et al.*, "Constructing Candidate Word Lists Using Acoustically Similar Word Groups," *IEEE Trans. on Signal Processing*, 1992(1), pp. 2814-2816.

[11] Barr, A. and E. Feigenbaum, *The Handbook of Artificial Intelligence: Volume I*, 1981, Addison-Wesley.

[12] Cettolo, M., R. Gretter, and R.D. Mori, "Knowledge Integration" in *Spoken Dialogues with Computers*, R.D. Mori, Editor 1998, London, pp. 231-256, Academic Press.

[13] Dijkstra, E.W., "A Note on Two Problems in Connection with Graphs," *Numerische Mathematik*, 1959, **1**, pp. 269-271.

[14] Gauvain, J.L., *et al.*, "The LIMSI Speech Dictation System: Evaluation on the ARPA Wall Street Journal Corpus," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1994, Adelaide, Australia pp. 129-132.

[15] Gopalakrishnan, P.S. and L.R. Bahl, "Fast Match Techniques" in *Automatic Speech and Speaker Recognition*, C.H. Lee, F.K. Soong, and K.K. Paliwal, eds. 1996, Norwell, MA, pp. 413-428, Kluwer Academic Publishers.

[16] Gopalakrishnan, P.S., L.R. Bahl, and R.L. Mercer, "A Tree Search Strategy for Large-Vocabulary Continuous Speech Recognition," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1995, Detroit, MI pp. 572-575.

[17] Hart, P.E., N.J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. on Systems Science and Cybernetics*, 1968, **4**(2), pp. 100-107.

[18] Huang, X., *et al.*, "Microsoft Windows Highly Intelligent Speech Recognizer: Whisper," *IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1995 pp. 93-96.

[19] Jelinek, F., *Statistical Methods for Speech Recognition*, 1998, Cambridge, MA, MIT Press.

[20] Kenny, P., *et al.*, "A* -Admissible Heuristics for Rapid Lexical Access," *IEEE Trans. on Speech and Audio Processing*, 1993, **1**, pp. 49-58.

[21] Kenny, P., *et al.*, "A New Fast Match for Very Large Vocabulary Continuous Speech Recognition," *IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1993, Minneapolis, MN pp. 656-659.

[22] Laface, P., L. Fissore, and F. Ravera, "Automatic Generation of Words toward Flexible Vocabulary Isolated Word Recognition," *Proc. of the Int. Conf. on Spoken Language Processing*, 1994, Yokohama, Japan pp. 2215-2218.

[23] Lawler, E.W. and D.E. Wood, "Branch-and-Bound Methods: A Survey," *Operations Research*, 1966(14), pp. 699-719.

[24] Lee, K.F. and F.A. Alleva, "Continuous Speech Recognition" in *Recent Progress in Speech Signal Processing*, S. Furui and M. Sondhi, eds. 1990, Marcel Dekker, Inc.

[25] Lee, K.F., H.W. Hon, and R. Reddy, "An Overview of the SPHINX Speech Recognition System," *IEEE Trans. on Acoustics, Speech and Signal Processing*, 1990, **38**(1), pp. 35-45.

[26] Lowerre, B.T., *The HARPY Speech Recognition System*, PhD Thesis in *Computer Science Department* 1976, Carnegie Mellon University, .

[27] Moore, E.F., "The Shortest Path Through a Maze," *Int. Symp. on the Theory of Switching*, 1959, Cambridge, MA, Harvard University press pp. 285-292.

[28] Murveit, H., *et al.*, "Large Vocabulary Dictation Using SRI's DECIPHER Speech Recognition System: Progressive Search Techniques," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1993, Minneapolis, MN pp. 319-322.

[29] Newell, A. and H.A. Simon, *Human Problem Solving*, 1972, Englewood Cliffs, NJ, Prentice Hall.

[30] Ney, H. and X. Aubert, "Dynamic Programming Search: From Digit Strings to Large Vocabulary Word Graphs" in *Automatic Speech and Speaker Recognition*, C.H. Lee, F. Soong, and K.K. Paliwal, eds. 1996, Boston, pp. 385-412, Kluwer Academic Publishers.

[31] Ney, H. and S. Ortmanns, *Dynamic Programming Search for Continuous Speech Recognition*, in *IEEE Signal Processing Magazine*, 1999. pp. 64-83.

[32] Nilsson, N.J., *Principles of Artificial Intelligence*, 1982, Berlin, Germany, Springer Verlag.

[33] Nilsson, N.J., *Artificial Intelligence: A New Synthesis*, 1998, Academic Press/Morgan Kaufmann.

[34] Normandin, Y., R. Cardin, and R.D. Mori, "High-Performance Connected Digit Recognition Using Maximum Mutual Information Estimation," *IEEE Trans. on Speech and Audio Processing*, 1994, **2**(2), pp. 299-311.

[35] Paul, D.B., "An Efficient A* Stack Decoder Algorithm for Continuous Speech Recognition with a Stochastic Language Model," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1992, San Francisco, California pp. 25-28.

[36] Roe, D.B. and M.D. Riley, "Prediction of Word Confusabilities for Speech Recognition," *Proc. of the Int. Conf. on Spoken Language Processing*, 1994, Yokohama, Japan pp. 227-230.

[37] Schwartz, R., *et al.*, "Context-Dependent Modeling for Acoustic-Phonetic Recognition of Speech Signals," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1985, Tampa, FLA pp. 1205-1208.

[38] Steinbiss, V., *et al.*, "The Philips Research System for Large-Vocabulary Continuous-Speech Recognition," *Proc. of the European Conf. on Speech Communication and Technology*, 1993, Berlin, Germany pp. 2125-2128.

[39] Taha, H.A., *Operations Research: An Introduction*, 6th ed, 1996, Prentice Hall.

[40] Tanimoto, S.L., *The Elements of Artificial Intelligence : An Introduction Using Lisp*, 1987, Computer Science Press, Inc.

[41] Waast, C. and L.R. Bahl, "Fast Match Based on Decision Tree," *Proc. of the European Conf. on Speech Communication and Technology*, 1995, Madrid, Spain pp. 909-912.

[42] Winston, P.H., *Artificial Intelligence*, 1984, Reading, MA, Addison-Wesley.

[43] Winston, P.H., *Artificial Intelligence*, 3rd ed, 1992, Reading, MA, Addison-Wesley.

[44] Woodland, P.C., *et al.*, "Large Vocabulary Continuous Speech Recognition Using HTK," *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 1994, Adelaide, Australia pp. 125-128.