

DESIGN OF THE CMU SPHINX-4 DECODER

Paul Lamere¹, Philip Kwok¹, William Walker¹,
Evandro Gouvêa², Rita Singh², Bhiksha Raj³, Peter Wolf³

¹Sun Microsystems Laboratories, ²Carnegie Mellon University, ³Mitsubishi Electric Research Laboratories, USA

ABSTRACT

Sphinx-4 is an open source HMM-based speech recognition system written in the Java™ programming language. The design of the Sphinx-4 decoder incorporates several new features in response to current demands on HMM-based large vocabulary systems. Some new design aspects include graph construction for multilevel parallel decoding with multiple feature streams without the use of compound HMMs, the incorporation of a generalized search algorithm that subsumes Viterbi decoding as a special case, token stack decoding for efficient maintenance of multiple paths during search, design of a generalized language HMM graph from grammars and language models of multiple standard formats, that can potentially toggle between flat search structure, tree search structure, *etc.* This paper describes a few of these design aspects, and reports some preliminary performance measures for speed and accuracy.

1. INTRODUCTION

Sphinx-4 is a state-of-art HMM-based speech recognition system being developed on open source (cmusphinx.sourceforge.net) since February 2002. It is the latest addition to Carnegie Mellon University's repository of Sphinx speech recognition systems. The Sphinx-4 decoder has been designed jointly by researchers from CMU, Sun Microsystems Laboratories and Mitsubishi Electric Research Laboratories. Over the past few years, the demands placed on conventional recognition systems have increased significantly. Several functionalities are now additionally desired of a system, such as the ability to perform multistream decoding in a theoretically correct manner, with as much user control on the level of combination as possible, that of at least some degree of basic easy control over the system's performance in the presence of varied and unexpected environmental noise levels and types, portability across a growing number of computational platforms, conformance to widely varying resource requirements, easy restructuring of the architecture for distributed processing *etc.* The importance of good and flexible user interfaces is also clear as speech recognition is increasingly used by non-experts in diverse applications. The design of Sphinx-4 is driven by almost all of these considerations, resulting in a system that is highly modular, portable and easily extensible. Many new features have been incorporated by extending conventional design strategies or implementing new ones. The design is more utilitarian and futuristic than most existing HMM-based systems.

This paper describes a selected set of important design innovations in the Sphinx-4 decoder. The sections are arranged as follows: Section 2 describes the overall architecture of the decoder, and some software aspects. Section 3 describes the design of the graph construction module for decoding with single and multiple parallel feature streams. Section 4 describes the design of the search module, and the generalized classification algorithm used

in it. Section 5 describes the design of the frontend and acoustic scorer. Section 6 gives some performance measures.

2. OVERALL ARCHITECTURE

The Sphinx-4 architecture has been designed with a high degree of modularity. Figure 1 shows the overall architecture of the system. Even within each module shown in Figure 1, the code is extremely modular with easily replaceable functions.

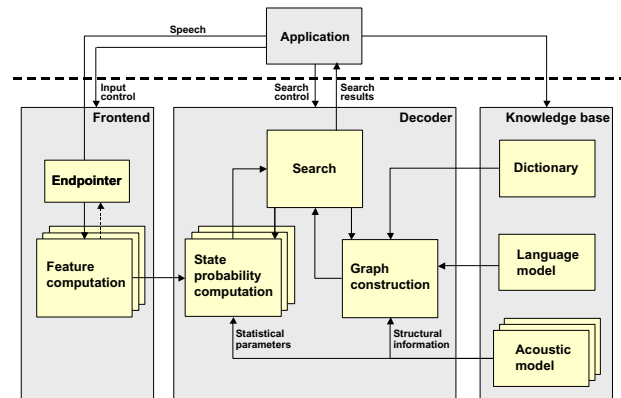


Figure 1. Architecture of the Sphinx-4 system. The main blocks are Frontend, Decoder and Knowledge base. Except for the blocks within the KB, all other blocks are independently replaceable software modules written in the Java™ programming language. Stacked blocks indicate multiple types which can be used simultaneously.

There are three main blocks in the design: the frontend, the decoder, and the knowledge base (KB). These are controllable by an external application. The frontend module takes in speech and parametrizes it. Within it, the endpointer module can either perform endpointing on the speech signal, or on the sequence of feature vectors computed from it. The decoder block performs the actual recognition. It comprises a graph construction module, which translates any type of standard language model provided to the KB by the application into an internal format, and together with information from the dictionary, and structural information from one or more sets of acoustic models, constructs a *Language HMM*. The latter is then used by the search module to determine the structure of the trellis to be searched. The trellis is not explicitly constructed. Rather, it is an implicit entity as in conventional decoders. Search is performed by token passing [2]. The application can tap the information in the tokens to get search results at various levels (such as state, phone or word-level lattices and segmentations). When multiple feature streams are used for recognition, the application can also control the level at which scores from the parallel streams are combined during search, and how each information stream is pruned. The search module requires state output probability values for each feature vector in order to determine the scores of state sequences. State output probabilities

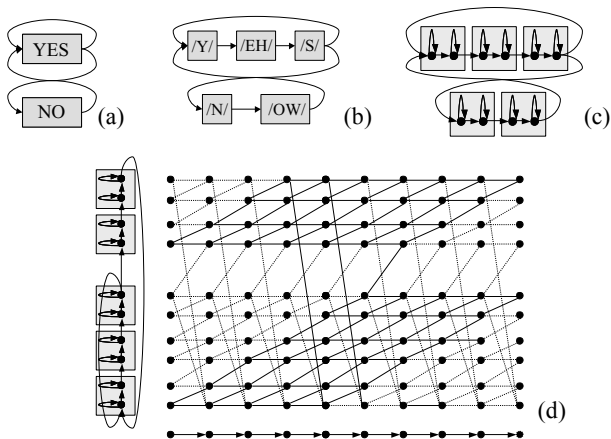


Figure 2. (a) Word graph for a simple language with a two word vocabulary. (b) Phonetic graph derived from (a). (c) Language HMM derived from (b). (d) Trellis formed as the crossproduct of (c) and a linear graph of observation data vectors.

are computed by the state probability computation module, which is the only module that has access to the feature vectors. Score computation is thus an on-demand task, carried out whenever the search module communicates a state identity to the scoring module, for which a score for the current feature (of a specific type) is desired. In Sphinx-4, the graph construction module is called the *linguist*, and the score computation module is called the *acoustic scorer*. The system permits the use of any level of context in the definition of the basic sound units. The modular design also favors any future hardware implementation.

Programming language: The system is entirely developed on the Java™ platform which is highly portable: once compiled, the bytecode can be used on any system that supports the Java platform. This feature permits a high degree of decoupling between all modules. Each module can be exchanged for another without requiring any modification of the other modules. The particular modules to be used can be specified at run time through a command line argument, with no need to recompile the code. The garbage collection (GC) feature of the Java platform simplifies memory management greatly; memory management is no longer done through explicit code. When a structure is no longer needed, the program simply stops referring to it. The GC frees all relevant memory blocks automatically. The Java platform also provides a standard manner of writing multithreaded applications to easily take advantage of hyper-threaded processors and multi-processor machines. In addition, the Javadoc™ tool automatically extracts information from comments in the code and creates html files that provide documentation about the software interface.

3. GRAPH CONSTRUCTION MODULE

In an HMM-based decoder, search is performed through a trellis: a directed acyclic graph (DAG) which is the cross product of the language HMM and time. The language HMM is a directed graph, in which any path from source to sink represents the HMM for a valid word sequence in the given language. Language probabilities are applied at transitions between words. The language HMM graph is a composition of the language structure as represented by a given language model and the topological structure of the acoustic models (HMMs for the basic sound units used by the system).

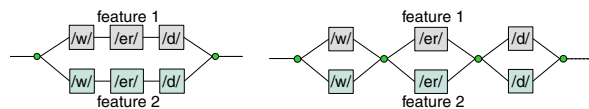


Figure 3. Graph construction with two parallel features. In the left panel scores for words are combined at word boundaries. In the right panel, they are combined at phoneme boundaries.

Figure 2 shows examples of a language HMM, the trellis which is based on it and searched, and the intermediate level graphs which are used implicitly in the composition of the language HMM.

The graph construction module in Sphinx-4, *i.e.* the *linguist*, has two submodules. The first interprets the language model provided by the application as a part of the KB, and converts it into an internal grammar. This permits the external language model provided by the application to have any structure, such as statistical N-gram, context free grammar (CFG), finite state grammar (FSG), finite state transducer (FST), simple word list *etc.* For FSGs and FSTs, the internal grammar is a literal translation of the external representation. The internal grammar representation of word lists links a single source node to all words, and has edges from the outputs of words to a common sink node. A loopback is made from the sink to the source. For N-gram LMs, a fully-connected structure is formed where every word is represented by a node, and there are explicit links from every node to every other node.

The internal grammar is then converted to a language HMM by the second submodule, which is independent of the grammar construction module. Note that in the architecture diagram both modules are represented by a single graph construction module. In forming the language HMM, the word-level network of the internal grammar is expanded using the dictionary and the structural information from the acoustic models. In this graph, sub-word units with contexts of arbitrary length can be incorporated, if provided. We assume, however, that silence terminates any context - sub-word units that are separated from each other by an intermediate silence cannot affect each other. The acoustic models for the sub-word units are incorporated into the final language HMM.

The word graph can be converted to a language HMM either dynamically or statically. In dynamic construction, word HMMs are constructed on demand - when the search reaches the terminal state for a word, the HMMs for words that can follow it are constructed if they have not already been instantiated. During construction, appropriate context dependent sub-word units are used at the word boundaries. In static construction, the entire language HMM is constructed statically. HMMs are constructed for all words in the vocabulary. Each word HMM is composed with several word-beginning and word-ending context dependent phones, each corresponding to a possible crossword context. Each word is connected to every other word by linking appropriate context-dependent crossword units.

The manner in which the language HMM is constructed affects the search. The topology of the language HMM affects the memory footprint, speed and recognition accuracy. The modularized design of Sphinx-4 allows different language HMM compilation strategies to be used without changing other aspects of the search. The choice between static and dynamic construction of language HMMs depends mainly on the vocabulary size, language model complexity and desired memory footprint of the system, and can be made by the application.

The language HMM accommodates parallel feature streams as shown in Figure 3. The design does not use compound HMMs [3] as in conventional systems, but maintains separate HMMs for the individual feature streams. The HMMs for the individual streams are tied at the level at which the application requires unit scores to be combined. Time-synchronization of paths is ensured at the boundaries of combined units, during search.

4. SEARCH MODULE

Search in Sphinx-4 can be performed using the conventional Viterbi algorithm, or a more general algorithm called Bushderby [1], which performs classification based on *free energy*, rather than likelihoods. Likelihoods are used in the computation of free energy, but do not constitute the main objective function used for classification. The theoretical motivations for this algorithm are described in [1]. From an engineering perspective, this amounts to the application of different logic at different nodes in the language HMM. At non-emitting nodes between words, only the score of the best incoming path is retained. At every other node U , a free-energy term, computed from the scores of the set π of all incoming edges incident on U , is derived as the following $1/T$ norm:

$$\text{score}(U) = \left(\sum_{\pi} \text{score}(\pi)^{\frac{1}{T}} \right)^T \quad (1)$$

When $T = 1$, this computes the sum of all incoming scores. At $T = 0$, $\text{score}(U)$ is merely the highest of incoming scores. Thus the search collapses to Viterbi decoding at $T = 0$. Classification for mismatched data can directly be controlled through the Bushderby parameter T , and has been shown to yield significant improvements in recognition performance as compared to the Viterbi algorithm.

Both Viterbi and Bushderby decoding are performed using a token passing algorithm [2]. A token is an object that is associated with a state and contains the overall acoustic and language scores of the path at a given point, a language HMM reference, a reference to an input feature vector, and other relevant information. The language HMM reference allows the search module to relate a token to its state output distribution, context-dependent phonetic unit, pronunciation, word, and grammar state. Every partial hypothesis terminates in an *active token*. At each time step, a subset of these tokens form an *active list*. Each token in the active list generates new tokens that are propagated to successor states. Active lists are selected from currently active nodes in the trellis through pruning. Sphinx-4 performs both relative and absolute pruning, and also pruning for individual features when decoding with parallel feature streams. All pruning thresholds are controllable by the application. The GC automatically reclaims unused tokens, thus simplifying the implementation of the pruner, in that the module can simply stop referring to tokens that are not in the active list. The search module communicates with the state probability estimation module, also called the *acoustic scorer*, to obtain acoustic scores for the current data. Data are only seen by the acoustic scorer.

Search can be performed in either *depth-first* or *breadth-first* manner. Depth-first search is similar to conventional stack decoding, where the most promising tokens are propagated through time sequentially. Thus, partial hypotheses can be of varying lengths. In

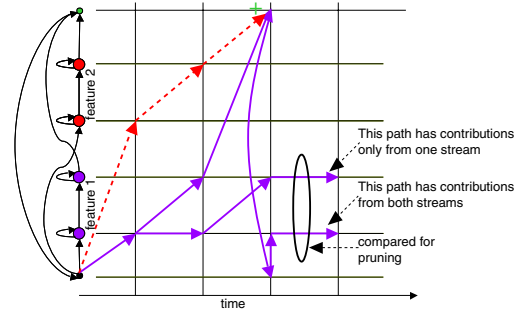


Figure 4. The pruning problem encountered by the search module in decoding parallel feature streams. If pruning is based on combined scores, paths with different contributions from the multiple feature streams get compared for pruning.

breadth-first search, all active tokens are expanded synchronously, making all partial hypotheses equal in length. The Bushderby algorithm applies only to breadth-first search. For Viterbi decoding, when multiple tokens arrive at a state, only the best token is retained. When Bushderby decoding is being performed, however, multiple tokens at emitting states are merged into a single token.

For decoding with parallel feature streams, timing considerations become important. It is necessary to ensure that the paths from the multiple feature streams that are combined enter and exit the units across which they are combined at the same time. The units may be states, phonemes or words. This is because Sphinx-4 uses a factored language HMM for the multiple streams, as shown in Figure 3. This becomes extremely difficult to ensure with conventional Viterbi decoding, where only a single path is retained at any state. It becomes impossible to ensure that the paths that survive through the factored HMMs have compatible timings. To resolve this problem, the search module maintains *token stacks*, keeping multiple tokens at each state. This permits multiple paths to survive through any state.

A second problem with factored implementation of multiple features streams is that of pruning. It is inappropriate to perform pruning based on total path scores alone, since this leads to unbalanced scores. The problem is depicted in Figure 4. To avoid this problem, pruning is done at two levels. Within the states of the HMM for any feature, pruning is performed based only on the total score for that feature, without incorporating the contributions of other features. The beams used for this pruning are typically very wide. At the non-emitting nodes where feature scores are combined, pruning is performed based on the combined scores. This pruning uses much narrower beams. Also, feature scores are combined in a weighted manner as follows:

$$\text{score}(U) = \prod_{feat} \text{score}(feat)^{w_{feat}} \quad (2)$$

The weights w_{feat} can be controlled directly by the application or through any application-enforced learning algorithm.

The result generated by the search module is in the form of a token tree, which can be queried for the best recognition hypotheses, or a set of hypotheses. The tree does not encode the entire recognition lattice. To derive lattices, a secondary table must maintain pointers to word-ending tokens that were pruned.

5. FRONT END AND ACOUSTIC SCORER

Front end: The front end module of Sphinx-4 is parallelizable, and can currently simultaneously compute MFCC and PLP cepstra from speech signals, with easy extensibility to other feature types. The module is organized as a sequence of independent replaceable communicating submodules, each with an input and output that can be tapped. Thus outputs of intermediate submodules can be easily fed into the state computation module, for example. Features computed using independent sources by the application, such as visual features, can also be directly fed into the acoustic scorer, either in parallel with the features computed by the frontend, or independently. Moreover, one can easily add new submodules between two existing ones. It would thus be very easy to add a specific noise cancellation module to the front end.

The communication between blocks follows a *pull* design. In this design, a module requests input from an earlier module when needed, as opposed to the more conventional *push* design, where a module propagates its output to the succeeding module as soon as it is generated. At a global level, in a pull design, speech is processed only when recognition is requested. In a push design, recognition is requested after speech is captured. Each module operates in response to control signals which are interpreted from the data requested from the predecessor. The control signal might indicate the beginning or end of speech, data dropped, *etc.* If the incoming data are speech, they are processed and the output is buffered, waiting for the successor module to request it. Handling of control signals such as start or end of speech are essential for livemode operation. This design allows the system to be used in live or batchmode without modification.

In addition to being responsive to a continuous stream of input speech, the frontend is capable of three other modes of operation: (a) fully endpointed, in which explicit beginning and end points of a speech signal are sensed (b) click-to-talk, in which the user indicates the beginning of a speech segment, but the system determines when it ends, (c) push-to-talk, in which the user indicates both the beginning and the end of a speech segment. Currently, endpoint detection is performed by a simple algorithm that compares the energy level to three threshold levels. Two of these are used to determine start of speech, and one for end of speech.

Acoustic scorer: The acoustic scorer provides state output density values on demand. When the search module requests a score for a given state at a given time, it accesses the feature vector for that time and performs the mathematical operations to compute the score. It matches the acoustic model set to be used against the feature type in case of parallel decoding with parallel acoustic models. There are no restrictions on the allowed topology for the HMMs used in parallel scoring. The scorer retains all information pertaining to the state output densities. Thus, the search module need not know the scoring is done with continuous, semi-continuous or discrete HMMs. Any heuristic algorithms incorporated into the scoring procedure for speeding it up can be performed locally within the scorer.

6. EXPERIMENTAL EVALUATION

The performance of Sphinx-4 is compared with Sphinx-3 on the on the speaker independent portion of the Resource Management database (RM1) [4] in Table 1. Sphinx-3 builds most of the lan-

guage HMM dynamically, whereas the experiments reported with Sphinx-4 utilize static language HMM construction. Since the two operations are fundamentally different, language HMM construction times have been factored out of the real-time numbers reported in Table 1. Acoustic models were 3 state, 8 Gaussians/state HMMs with 1800 tied states, trained with the RM1 training data using the training module of Sphinx-3. Test results are reported using the following statistical N-gram language models: a flat unigram, a unigram, and a bigram. The LMs were created from the LM training data provided with the RM1 database. Sphinx-4 is currently also capable of working from an external FST language model. Although the results with FSTs are consistently better than those with N-gram LMs, we do not report them here, since comparisons with Sphinx-3 are not possible. Table 1 shows both word error rate (WER) and decoding speed (in times real time). All experiments were run on a 1.7GHz Pentium-4 processor running Linux RedHat 7.1, using Java™ 2 Platform, Standard Edition (J2SE™) v. 1.4.1_01 SDK.

Type of N-gram LM	Speed (xRT)		WER (%)	
	S3	S4	S3	S4
Flat unigram	1.2	4.8	18.7	19.3
Unigram	1.1	5.5	13.1	14.5
Bigram	1.1	6.0	1.9	3.3

Table 1: Performance of Sphinx-3 (S3) and Sphinx-4 (S4) on RM1. The speeds do not include loading time, which are different for S3 (dynamic language HMM construction) and S4 (currently static language HMM construction). Sphinx-4 has not been optimized for speed or accuracy.

Sphinx-4 performance has not been optimized on the bigram and trigram tasks at the time of this submission. The evaluation of medium vocabulary tasks is ongoing, and large vocabulary tasks will be approached shortly. The optimized bigram and trigram tasks, and the completed medium and large vocabulary evaluations will be reported by the time the paper is presented. They will also be reported on SourceForge as and when they are completed.

ACKNOWLEDGEMENTS

We thank Prof. Richard Stern at CMU, Robert Sproull at Sun Microsystems Laboratories, and Joe Marks at MERL, for making this team possible. We also thank Sun Microsystems Laboratories and the current management for their continued support and collaborative research funds. Rita Singh was sponsored by the Space and Naval Warfare Systems Center, San Diego, under Grant No. N66001-99-1-8905. The content of this paper does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred.

REFERENCES

- [1] R. Singh, M. Warmuth, B. Raj, and P. Lamere, "Classification with free energy at raised temperatures," in *EURO-SPEECH 2003*, 2003.
- [2] S.J. Young, N.H. Russel, and J.H.S. Russel, "Token passing: A simple conceptual model for connected speech recognition systems," *Tech. Report*, Cambridge Univ. Engg. Dept., 1989
- [3] G. Potamianos, C. Neti, G. Iyengar, and E. Helmut, "Large-vocabulary audio-visual speech recognition by machines and humans," *Proc. EUROSPEECH 2001*, Aalborg, 2001.
- [4] P. Price, W.M. Fisher, J. Bernstein, and D.S. Pallett, "The DARPA 1000-word resource management database for continuous speech recognition," in *Proc. ICASSP 1988*, Vol. 1. pp. 651-654, 1988.