Sphinx-4 – 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.

Problem: researchers approach the problem of core speech recognition research, they are often faced with the problem of needing to develop an entire system from scratch, even if they only want to explore one facet of the field. Open source speech recognition systems are available, such as HTK [1], ISIP [2], AVCSR [3] and earlier versions of the Sphinx systems [4] – [6]. The available systems are typically optimized for a single approach to speech system design. As a result, these systems intrinsically create barriers to future research that departs from the original purpose of the system. In addition, some of these systems are encumbered by licensing agreements that make entry into the research arena difficult for non-academic institutions.

To facilitate new innovation in speech recognition research, we formed a distributed, crossdiscipline team to create Sphinx-4 [7]: an open source platform that incorporates state-of-the art methodologies and also addresses the needs of emerging research areas. Given our technical goals as well as our diversity (e.g., we used different operating systems on different machines, etc.), we wrote Sphinx-4 in the JavaTMprogramming language, making it available to a large variety of development platforms.

First and foremost, Sphinx-4 is a modular and pluggable framework that incorporates design patterns from existing systems, with sufficient flexibility to support emerging areas of research interest. The framework is modular in that it comprises separable components dedicated to specific tasks, and it is pluggable in that modules can be easily replaced at run time. To exercise the framework, and to provide researchers with a working system, Sphinx-4 also includes a variety of modules that implement state-of-the-art speech recognition techniques.

The remainder of this document describes the Sphinx-4 framework and implementation, and also includes a discussion of our experiences with Sphinx-4 to date.

Methods : The traditional approach to speech recognition system design has been to create an entire system optimized around a particular methodology. As evidenced by past research systems such as Dragon [8], Harpy [9], Sphinx and others, this approach has proved to be quite valuable in that the resulting systems have provided foundational methods for speech recognition research.

In the same light, however, each of these systems was largely dedicated to exploring a single specific ground breaking area of speech recognition. For example, Baker introduced hidden Markov models (HMMs) with his Dragon system, [8], [10] and earlier predecessors

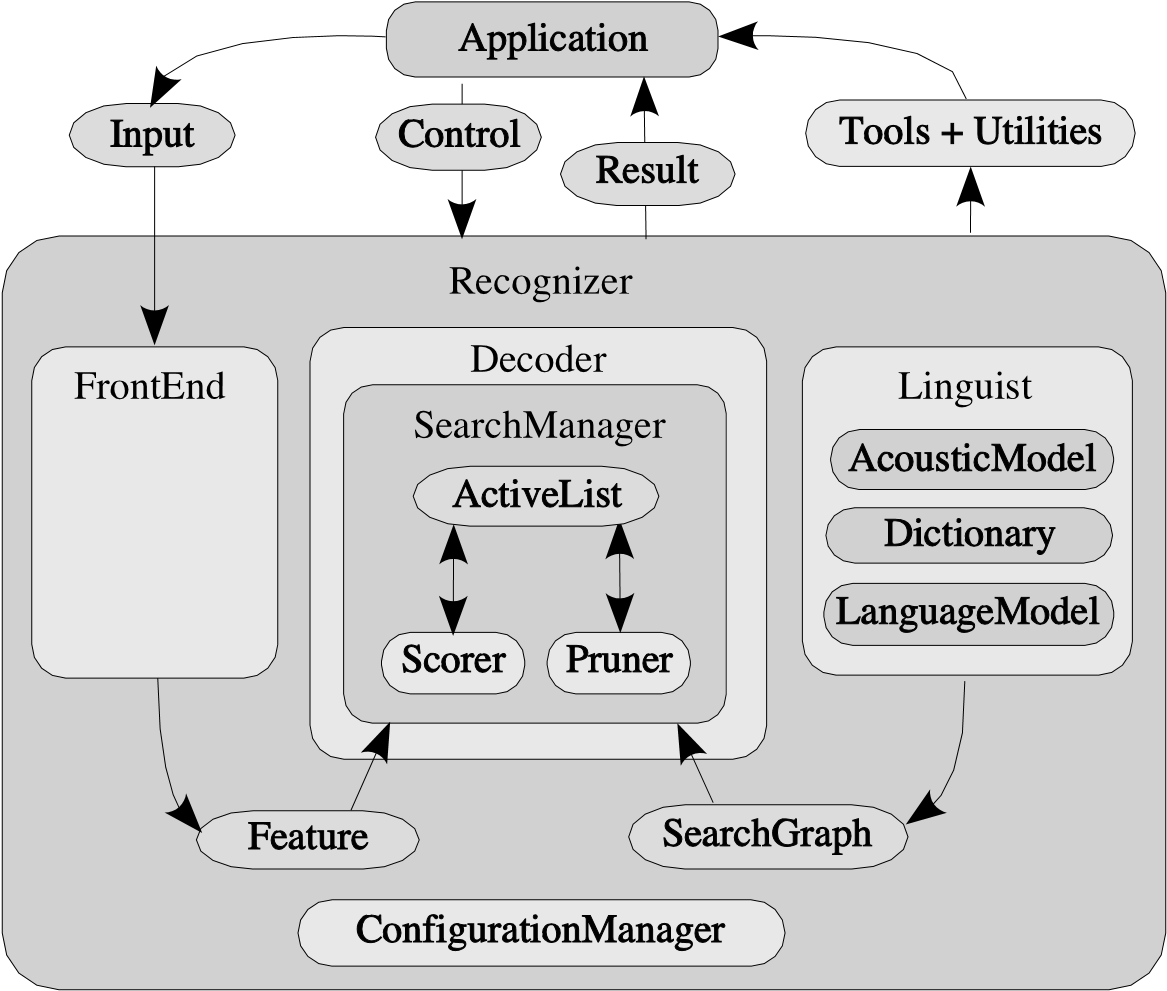


Fig. 1. Sphinx-4 Decoder Framework. The main blocks are the FrontEnd, the Decoder, and the Linguist. Supporting blocks include the ConfigurationManager and the Tools blocks. The communication between the blocks, as well as communication with an application, is depicted.

of Sphinx explored variants of HMMs such as discrete HMMs [4], semicontinuous HMMs [5] , and continuous HMMs [11]. Other systems explored specialized search strategies such as using lex tree searches for large N-Gram models [12].

Because they were focused on such fundamental core theories, the creators of these systems tended to hardwire their implementations to a high degree. For example, the predecessor Sphinx systems restrict the order of the HMMs to a constant value and also fix the unit context to a single left and right context. Sphinx-3 eliminated support for context free grammars (CFGs) due to the specialization on large N-Gram models. Furthermore, the decoding strategy of these systems tended to be deeply entangled with the rest of the system. As a result of these constraints, the systems were difficult to modify for experiments in other areas.

Design patterns for these systems emerged over time, however, as exemplified by Jelinek’s source-channel model [13] and Huang’s basic system architecture [14]. In developing Sphinx-4, one of our primary goals was to develop a framework that supported these design patterns, yet also allowed for experimentation in emerging areas of research.

The Sphinx-4 framework has been designed with a high degree of flexibility and modularity. Figure 1 shows the overall architecture of the system. Each labeled element in Figure 1 represents a module that can be easily replaced, allowing researchers to experiment with different module implementations without needing to modify other portions of the system.

There are three primary modules in the Sphinx-4 framework: the *FrontEnd*, the *Decoder*, and the *Linguist*. The FrontEnd takes one or more input signals and parameterizes them into a sequence of *Features*. The Linguist translates any type of standard language model, along with

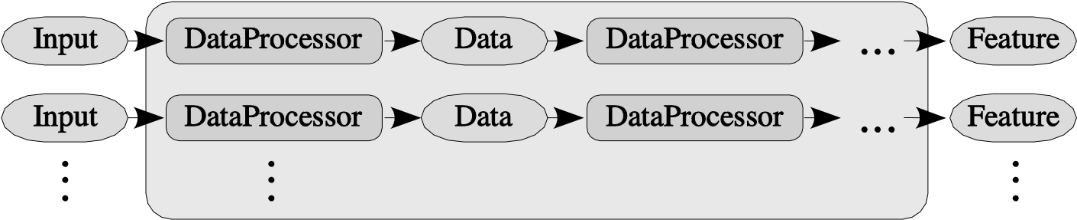


Fig. 2. Sphinx-4 FrontEnd. The FrontEnd comprises one or more parallel chains of communicating DataProcessors.

pronunciation information from the *Dictionary* and structural information from one or more sets of *AcousticModels*, into a *SearchGraph*. The *SearchManager* in the Decoder uses the Features from the FrontEnd and the SearchGraph from the Linguist to perform the actual decoding, generating *Results*. At any time prior to or during the recognition process, the application can issue *Controls* to each of the modules, effectively becoming a partner in the recognition process.

The Sphinx-4 system is like most speech recognition systems in that it has a large number of configurable parameters, such as search beam size, for tuning the system performance. The Sphinx-4 *ConfigurationManager* is used to configure such parameters. Unlike other systems, however, the ConfigurationManager also gives Sphinx-4 the ability to dynamically load and configure modules at run time, yielding a flexible and pluggable system. For example, Sphinx-4 is typically configured with a FrontEnd (seeSection IV) that produces Mel-Frequency Cepstral Coefficients (MFCCs) [15]. Using the ConfigurationManager, however, it is possible to reconfigure Sphinx-4 to construct a different FrontEnd that produces Perceptual Linear Prediction coefficients (PLP) [16] without needing to modify any source code or to recompile the system.

To give applications and developers the ability to track decoder statistics such as word error rate [17], run time speed, and memory usage, Sphinx-4 provides a number of *Tools*. As with the rest of the system, the Tools are highly configurable, allowing users to perform a wide range of system analysis. Furthermore, the Tools also provide an interactive run time environment that allows users to modify the parameters of the system while the system is running, allowing for rapid experimentation with various parameters settings.

Sphinx-4 also provides support for *Utilities* that support application-level processing of recognition results. For example, these utilities include support for obtaining result lattices, confidence scores, and natural language understanding.

IV. FRONTEND

The purpose of the FrontEnd is to parameterize an *Input* signal (e.g., audio) into a sequence of output *Features*. As illustrated in Figure 2, the FrontEnd comprises one or more parallel chains of replaceable communicating signal processing modules called *DataProcessors*. Supporting multiple chains permits simultaneous computation of different types of parameters from the same or different input signals. This enables the creation of systems that can simultaneously decode using different parameter types, such as MFCC and PLP, and even parameter types derived from non-speech signals such as video [3].

Like the ISIP [2] system, each DataProcessor in the FrontEnd provides an input and an output that can be connected to another DataProcessor, permitting arbitrarily long sequences of chains. The inputs and outputs of each DataProcessor are generic *Data* objects that encapsulate processed input data as well as markers that indicate data classification events such as end-point detection. The last DataProcessor in each chain is responsible for producing a Data object composed of parameterized signals, called *Features*, to be used by the Decoder.

Like the AVCSR system [3], Sphinx-4 permits the ability to produce parallel sequences of features. Sphinx-4 is unique, however, in that it allows for an arbitrary number of parallel streams.

The communication between blocks follows a pull design. With a pull design, a DataProcessor requests input from an earlier module only 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. This pull design enables the processors to perform buffering, allowing consumers to look forwards or backwards in time.

The ability to look forwards or backwards in time not only permits the Decoder to perform frame-synchronous Viterbi searches [18], but also allows the decoder to perform other types of searches such as depth-first and A\* [19].

Within the generic FrontEnd framework, the Sphinx-4 provides a suite of DataProcessors that implement common signal processing techniques. These implementations include support for the following: reading from a variety of input formats for batch mode operation, reading from the system audio input device for live mode operation, preemphasis, windowing with a raised cosine transform (e.g., Hamming and Hanning windows), discrete Fourier transform (via FFT), mel frequency filtering, bark frequency warping, discrete cosine transform ( DCT), linear predictive encoding (LPC), end pointing, cepstral mean normalization (CMN), mel-cepstra frequency coefficient extraction (MFCC), and perceptual linear prediction coefficient extraction ( PLP ).

Using the ConfigurationManager described in Section III, users can chain the Sphinx-4 DataProcessors together in any manner as well as incorporate DataProcessor implementations of their own design. As such, the modular and pluggable nature of Sphinx-4 not only applies to the higher-level structure of Sphinx-4, but also applies to the higher-level modules themselves (i.e., the FrontEnd is a pluggable module, yet also consists of pluggable modules itself).

V. LINGUIST

The *Linguist* generates the SearchGraph that is used by the decoder during the search, while at the same time hiding the complexities involved in generating this graph. As is the case throughout Sphinx-4, the Linguist is a pluggable module, allowing people to dynamically configure the system with different Linguist implementations.

A typical Linguist implementation constructs the SearchGraph using the language structure as represented by a given LanguageModel and the topological structure of the AcousticModel (HMMs for the basic sound units used by the system). The Linguist may also use a Dictionary (typically a pronunciation lexicon) to map words from the LanguageModel into sequences ofAcousticModel elements. When generating the SearchGraph, the Linguist may also incorporate sub-word units with contexts of arbitrary length, if provided.

By allowing different implementations of the Linguist to be plugged in at run time, Sphinx4 permits individuals to provide different configurations for different system and recognition requirements. For instance, a simple numerical digits recognition application might use a simple Linguist that keeps the search space entirely in memory. On the other hand, a dictation application with a 100K word vocabulary might use a sophisticated Linguist that keeps only a small portion of the potential search space in memory at a time.

The Linguist itself consists of three pluggable components: the LanguageModel, the Dictionary, and the AcousticModel, which are described in the following sections.

1. *LanguageModel*

The LanguageModel module of the Linguist provides word-level language structure, which can be represented by any number of pluggable implementations. These implementations typically fall into one of two categories: graph-driven grammars and stochastic N-Gram models. The graph-driven grammar represents a directed word graph where each node represents a single word and each arc represents the probability of a word transition taking place. The stochastic N-Gram models provide probabilities for words given the observation of the previous n-1 words.

The Sphinx-4 LanguageModel implementations support a variety of formats, including the following:

* + SimpleWordListGrammar: defines a grammar based upon a list of words. An optional parameter defines whether the grammar “loops” or not. If the grammar does not loop, then the grammar will be used for isolated word recognition. If the grammar loops, then it will be used to support trivial connected word recognition that is the equivalent of a unigram grammar with equal probabilities.
  + JSGFGrammar: supports the JavaTMSpeech API Grammar Format (JSGF) [20], which defines a BNF-style, platform-independent, and vendor-independent Unicode representation of grammars.
  + LMGrammar: defines a grammar based upon a statistical language model. LMGrammar generates one grammar node per word and works well with smaller unigram and bigram grammars of up to approximately 1000 words.
  + FSTGrammar: supports a finite-state transducer (FST) [21] in the ARPA FST grammar format.
  + SimpleNGramModel: provides support for ASCII N-Gram models in the ARPA format. The SimpleNGramModel makes no attempt to optimize memory usage, so it works best with small language models.
  + LargeTrigramModel: provides support for true N-Gram models generated by the CMUCambridge Statistical Language Modeling Toolkit [22]. The LargeTrigramModel optimizes memory storage, allowing it to work with very large files of 100MB or more.

1. *Dictionary*

The *Dictionary* provides pronunciations for words found in the LanguageModel. The pronunciations break words into sequences of sub-word units found in the AcousticModel. The Dictionary interface also supports the classification of words and allows for a single word to be in multiple classes.

Sphinx-4 currently provides implementations of the Dictionary interface to support the CMU Pronouncing Dictionary [23]. The various implementations optimize for usage patterns based on the size of the active vocabulary. For example, one implementation will load the entire vocabulary at system initialization time, whereas another implementation will only obtain pronunciations on demand.

1. *AcousticModel*

The *AcousticModel* module provides a mapping between a unit of speech and an HMM that can be scored against incoming features provided by the FrontEnd. As with other systems, the mapping may also take contextual and word position information into account. For example, in the case of triphones, the context represents the single phonemes to the left and right of the given phoneme, and the word position represents whether the triphone is at the beginning, middle, or end of a word (or is a word itself). The contextual definition is not fixed by Sphinx-4, allowing for the definition of AcousticModels that contain allophones as well as AcousticModels whose contexts do not need to be adjacent to the unit.

Typically, the Linguist breaks each word in the active vocabulary into a sequence of contextdependent sub-word units. The Linguist then passes the units and their contexts to the AcousticModel, retrieving the HMM graphs associated with those units. It then uses these HMM graphs in conjunction with the LanguageModel to construct the SearchGraph.

Unlike most speech recognition systems, which represent the HMM graphs as a fixed structure in memory, the Sphinx-4 HMM is merely a directed graph of objects. In this graph, each node corresponds to an HMM state and each arc represents the probability of transitioning from one state to another in the HMM. By representing the HMM as a directed graph of objects instead of a fixed structure, an implementation of the AcousticModel can easily supply HMMs with different topologies. For example, the AcousticModel interfaces do not restrict the HMMs in terms of the number of states, the number or transitions out of any state, or the direction of a transition (forward or backward). Furthermore, Sphinx-4 allows the number of states in an HMM to vary from one unit to another in the same AcousticModel.

Each HMM state is capable of producing a score from an observed feature. The actual code for computing the score is done by the HMM state itself, thus hiding its implementation from the rest of the system, even permitting differing probability density functions to be used per HMM state. The AcousticModel also allows sharing of various components at all levels. That is, the components that make up a particular HMM state such as Gaussian mixtures, transition matrices, and mixture weights can be shared by any of the HMM states to a very fine degree.

As with the rest of Sphinx-4, individuals can configure Sphinx-4 with different implementations of the AcousticModel based upon their needs. Sphinx-4 currently provides a single AcousticModel implementation that is capable of loading and using acoustic models generated by the Sphinx-3 trainer.

1. *SearchGraph*

Even though Linguists may be implemented in very different ways and the topologies of the search spaces generated by these Linguists can vary greatly, the search spaces are all represented as a SearchGraph. Illustrated by example in Figure 3, the SearchGraph is the primary data structure used during the decoding process.

The graph is a directed graph in which each node, called a *SearchState*, represents either an *emitting* or a *non-emitting* state. Emitting states can be scored against incoming acoustic features while non-emitting states are generally used to represent higher-level linguistic constructs such as words and phonemes that are not directly scored against the incoming features. The arcs between states represent the possible state transitions, each of which has a probability representing the likelihood of transitioning along the arc.

The SearchGraph interface is purposely generic to allow for a wide range of implementation choices, relieving the assumptions and hard-wired constraints found in previous recognition systems. In particular, the Linguist places no inherent restrictions on the following:

* + Overall search space topology
  + Phonetic context size
  + Type of grammar (stochastic or rule based)
  + N-Gram language model depth

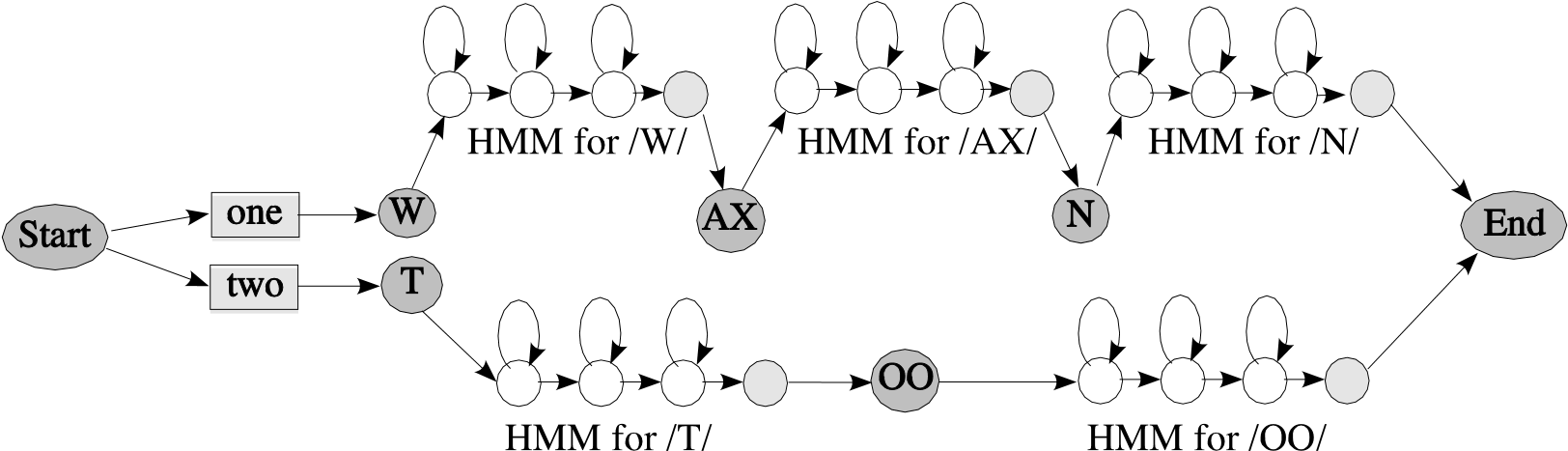


Fig. 3. Example SearchGraph. The SearchGraph is a directed graph composed of optionally emitting SearchStates and SearchStateArcs with transition probabilities. Each state in the graph can represent components from the LanguageModel ( words in rectangles), Dictionary (sub-word units in dark circles) or AcousticModel ( HMMs ).

A key feature of the SearchGraph is that the implementation of the SearchState need not be fixed. As such, each Linguist implementation typically provides its own concrete implementation of the SearchState that can vary based upon the characteristics of the particular Linguist. For instance, a simple Linguist may provide an in-memory SearchGraph where each SearchState is simply a one-to-one mapping onto the nodes of the in-memory graph. A Linguist representing a very large and complex vocabulary, however, may build a compact internal representation of the SearchGraph. In this case, the Linguist would generate the set of successor SearchStates by dynamically expanding this compact representation on demand. The manner in which the SearchGraph is constructed affects the memory footprint, speed, and recognition accuracy. The modularized design of Sphinx-4, however, allows different SearchGraph compilation strategies to be used without changing other aspects of the system. 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.

Expected results: As with the FrontEnd, Sphinx-4 provides several implementations of the Linguist to support different tasks.

The FlatLinguist is appropriate for recognition tasks that use context-free grammars (CFG), finite-state grammars (FSG), finite-state transducers (FST) and small N-Gram language models. The FlatLinguist converts any of these external language model formats into an internal Grammar structure. The Grammar represents a directed word graph where each *GrammarNode* represents a single word, and each arc in the graph represents the probability of a word transition taking place. The FlatLinguist generates the SearchGraph directly from this internal Grammar graph, storing the entire SearchGraph in memory. As such, the FlatLinguist is very fast, yet has difficulty handling grammars with high branching factors.

The DynamicFlatLinguist is similar to the FlatLinguist in that is is appropriate for similar recognition tasks. The main difference is that the DynamicFlatLinguist dynamically creates the SearchGraph on demand, giving it the capability to handle far more perplex grammars. With this capability, however, comes a cost of a modest decrease in run time performance.

The LexTreeLinguist is appropriate for large vocabulary recognition tasks that use large N-Gram language models. The order of the N-Grams is arbitrary, and the LexTreeLinguist will support true N-Gram decoding. The LexTreeLinguist organizes the words in a lex tree [6], a compact method of representing large vocabularies. The LexTreeLinguist uses this lex tree to dynamically generate SearchStates, enabling it to handle very large vocabularies using only a modest amount of memory. The LexTreeLinguist supports ASCII and binary language models generated by the CMU-Cambridge Statistical Language Modeling Toolkit [22].