Viterbi Beam Search with Layered Bigrams

David M. Goblirsch

NYNEX Science & Technology, Inc.
500 Westchester Ave.
White Plains, New York 10604
Email: dmgob@nynexst.com

ABSTRACT

We outline an implementation of Viterbi beam search that incorporates *layered bigrams*. Layered bigrams are class bigrams in which some nodes are themselves bigrams, resulting in a recursive structure. The implementation is in C++ and involves a hierarchy of classes. The paper outlines the main concepts and the corresponding C++ classes.

1. Introduction

Bigrams or class bigrams [3] are commonly used to constrain the search space when using the time-synchronous Viterbi search algorithm [4] for large-vocabulary speech recognition. Despite their usefulness, they have some well-known problems. First, by definition, bigrams do not capture dependencies beyond the preceding word or word class. Second, parameter smoothing—necessary for assigning nonzero probabilities to word pairs not seen during training but which should be allowed during recognition—can assign nonzero probabilities to unseen word sequences which truly should not be allowed. For example, having clock-time phrases as part of the top-level training texts can make sequences such as *the o'clock* legal when smoothing.

Trigrams mitigate the first problem by expanding the context to the two preceding words (or classes), but this is still not sufficient to capture many dependencies of interest. One way to capture longer contexts is to construct bigram- or trigram-like models that allow modeling units that are larger than single words, i.e., phrases [5]. By creating modeling units that correspond to semantic concepts such as "dollar amount" and "clock time," it is possible to get a language model that can generate a sequence like *eleven seventy one* in one context but not the other. Furthermore, parameter smoothing can't create combinations such as *the o'clock* if the word *o'clock* is hidden inside a "clock time" subgrammar.

In a LAYERED BIGRAM, each node in the top-level bigram is either a word, a word class (as in conventional class bigrams), or *another bigram*, which may itself have nodes that are bigrams, and so on. In other words, layered bigrams have a recursive structure.

Example templates selected from corpora we use to generate layered bigrams are provided in Section 3 after some prerequisite terminology is established in Section 2.

The layering concept can be extended to layered trigrams, layered quadgrams, and so on. However, our current system only implements layered bigrams, so our discussion will be limited to that case for the rest of this paper.

Layered bigrams have three advantages over conventional class bigrams. First, they allow modeling of longer-term dependencies than just the preceding word or word class (albeit weakly). The dependency is still limited to the preceding node, but if that node is itself a bigram, then we have dependencies on all the words in the phrases modeled by that bigram. Second, layered bigrams provide more control over the search space: parameter smoothing in one bigram cannot create illegal word sequences in another bigram or across bigrams. One also gains the option of turning off smoothing entirely in some bigrams, particularly those for which either a large amount of training data is available or for which complete specifications can be constructed by hand, such as simple clock times, dollar amounts, or city names. Third, the use of layered bigrams is a step toward integrating the speech recognizer with a natural language module, as the recognizer could provide a partial parse of the recognized utterance by tagging word subsequences with the name of the bigram that generated them; an example is provided in Section 3.

Layered bigrams have a major drawback: they are not as easily designed using strictly data-driven approaches as conventional bigrams. They are not too difficult to construct for applications such as ARPA's Air Travel Information System (ATIS) task because such applications have a manageable number of subgrammars corresponding to different, easily identified semantic concepts. However, in general, automatic parsing of corpora into phrases and clustering of phrases will be needed to design layered bigrams optimally, perhaps using techniques described in [2], [3], or [5].

Our definition of a "layered bigram" is somewhat weaker than that of [11] because in our case the probabilities that are internal to a bigram node are independent of its predecessor nodes; there is no conditioning on the children of these nodes.

We have implemented a recognition module based on the time-synchronous Viterbi algorithm, but extended to incorporate layered bigrams. Section 4 provides an overview of the algorithm. We do not discuss the design or evaluation of layered bigrams (e.g., how to

compute perplexity), nor issues such as N-best decoding.

2. Concepts

When a concept is introduced that will later be realized as a C++ class or as a `typedef` for a C++ class, its name will be typeset in SMALL CAPS. After that, the name of the concept will be capitalized. The names of C++ classes and keywords will be typeset using a `typewriter` font.

RULES are language models used to guide the beam search. Our implementation has two types of Rules, GRAMMARS and WORDSETS.

Grammars determine how word sequences can be produced. Grammars are modeled as NETWORKS: directed graphs with some nodes designated as entry nodes and some nodes designated as exit nodes. Each arc is tagged with a log-probability of being traversed, and each node is associated with a Rule, i.e., a WordSet or a Grammar.

A WordSet is a set of WORDS, each with a probability of being chosen, that are considered to be equivalent from the point of view of the containing Grammar in this sense: no matter which word in the WordSet is recognized, the probability distribution over successor nodes in the containing Grammar is the same. WordSets are usually called *word classes* in the literature. WordSets are conceptually just Grammars that have a particularly simple form, but making the distinction leads to a more efficient implementation.

Each time an utterance is to be recognized, a fresh SEARCHTRELLIS is created and associated with a top-level Rule. It is the responsibility of a dialogue manager to switch between top-level Rules. The SearchTrellis is the top-level object controlling the forward beam search and the final backtrace.

In addition to modeling Grammars, Networks are used to represent the acoustic models for words as hidden Markov models (HMMS). In this case the nodes are associated with log-likelihood functions to be evaluated on acoustic observations.

3. Examples

Here is one template from our training set for an ATIS layered bigram. The names of Grammars start with "$" and the names of WordSets start with "%".

```
list flights from $city-name to $city-name on
   %day-name leaving after $clock-time
```

City names are modeled using a Grammar because there are multi-word names, e.g., *San Diego*. By using a Grammar instead of a WordSet (where we could reduce a multi-word name to a single compound word, e.g., *San_Diego*), different city names can share common constituents without resorting to a tree-structured lexicon [12]. For example, one copy of the HMM for *San* can be shared between *San Antonio*, *San Diego*, and *San Francisco*. The WordSet `%day-name` contains the words *Sunday, . . . , Saturday*. Clock times obviously require a Grammar. In fact, the clock-time Grammar has a subgrammar, `$minutes`. An example of a template covered by the clock-time Grammar is

```
%hour-12 $minutes p m
```

where `%hour-12` is a WordSet containing *one* through *twelve*.

The transition probabilities for any Grammar can be found by collecting statistics over a collection of templates. Templates for simple Grammars such as `$city-name` and `$clock-time` can often be enumerated by hand.

One application of subgrammars is to handle contractions. Here are two templates from a corpus we use to train the language model for an application we are developing that will allow a speaker to retrieve information about movies and other home entertainment options:

```
$I+would like a description of $movie-name
$what+is $movie-name about
```

The Grammar `$I+would` can produce the single word *I'd* or the two-word phrase *I would*. Similarly, `$what+is` can produce the single word *what's* or the two-word phrase *what is*.

The recognition module could aid the natural language module by providing a partial parse of the recognized utterance by tagging word subsequences with the name of the WordSet or Grammar that generated them. For example, rather than just returning the text

```
flight number one twenty six leaving at
  one forty five
```

the search module could return

```
flight number $flight-number{one twenty six}
  leaving at
    $clock-time{%hour{one} $minutes{forty five}}
```

However, we have not yet implemented this capability.

4. Implementation

This section outlines the main classes in our implementation. Many auxiliary classes and low-level details have been omitted from this discussion, and only two member functions that are relevant to the forward phase of the beam search are discussed here.

The classes fall into two categories: structural and dynamic. The structural classes define the language and pronunciation models; they are constructed when the recognizer is initialized and fixed throughout its use. The dynamic classes contain the evolving paths and create backtrace cells. Each dynamic class is associated with a structural class.

A class diagram summarizing (but not completely specifying) the classes and their relationships is shown in Figure 1. All inheritance relationships in the diagram are `public`. More details are given in the next two subsections.

4.1. The Structural Classes

`Network<A,N>` is a general purpose template class implementing Networks with labeled nodes and arcs. The type variable `A` is the data type attached to arcs and `N` is the data type attached to nodes.

LogprobNetwork<N> is a template class implementing Networks whose arcs are labeled with log probabilities and whose nodes are labeled with data of an unspecified type N. It is derived from Network<Logprob,N>, where Logprob can be a typedef for a float or something more sophisticated.

Hmm is derived from LogprobNetwork<DensityId>. An object of the class DensityId identifies the log-likelihood function that is associated with an HMM node.

Grammars are LogprobNetworks for which each node essentially contains a pointer to a Rule, i.e., either a WordSet or another Grammar. (We say "essentially" because in our implementation the nodes also contain a probability of being followed by a node not explicitly listed as a successor node in order to handle unseen node sequences using the single backoff-node approach of [10]. For simplicity, Grammar is shown here as being derived from LogprobNetwork<Rule*>.) The nodes contain pointers to Rules because it is through pointers and virtual functions that C++ allows run-time polymorphism.[1]

A WordSet essentially just contains a set of Words and their log probabilities. Although the WordSet concept is motivated by observing that there are sets of words having similar distributional properties, individual words are also placed in WordSet objects, containing just that one word.

Rule is an abstract base class [6], from which Grammar and WordSet are derived. The motivation for this class is to provide a virtual member function create_Paths which is invoked whenever a Grammar node becomes active in the search. C++'s virtual function mechanism can be used to ensure that a Rule* that points to a Grammar will create a Grammar_Paths object and that a Rule* that points to a WordSet will create a WordSet_Paths object.

A class Word does not appear in Figure 1 because Word is not implemented as a class. Instead, it is a typedef for a Standard Template Library (STL) [7] pair, coupling a string (the written representation of the word) with an acoustic model, currently an Hmm:

```
typedef  pair<const String, Hmm>  Word;
```

This is convenient because the dictionary is implemented using an STL map: a container class having key/value pairs as entries.

4.2. The Dynamic Classes

The top-level dynamic class is SearchTrellis. When a new utterance is to be recognized, a new SearchTrellis is created. It holds a pointer to the top-level Rule that will guide the search, and it creates a Rule_Paths object to hold the evolving paths. SearchTrellis also has an optional beginning silence HMM.

A Grammar_Paths object contains the active paths for a Grammar and a WordSet_Paths object contains the active paths for a WordSet. These "_Paths" objects are created and destroyed dynamically, as are the paths they contain. Each "_Paths" object contains a reference to its corresponding structural class.

[1] Because of the Network formulation, our implementation handles Grammars that are more general than bigrams. To date, however, our Grammar networks have been designed by computing bigram statistics over template files, so we have limited our discussion to this case.

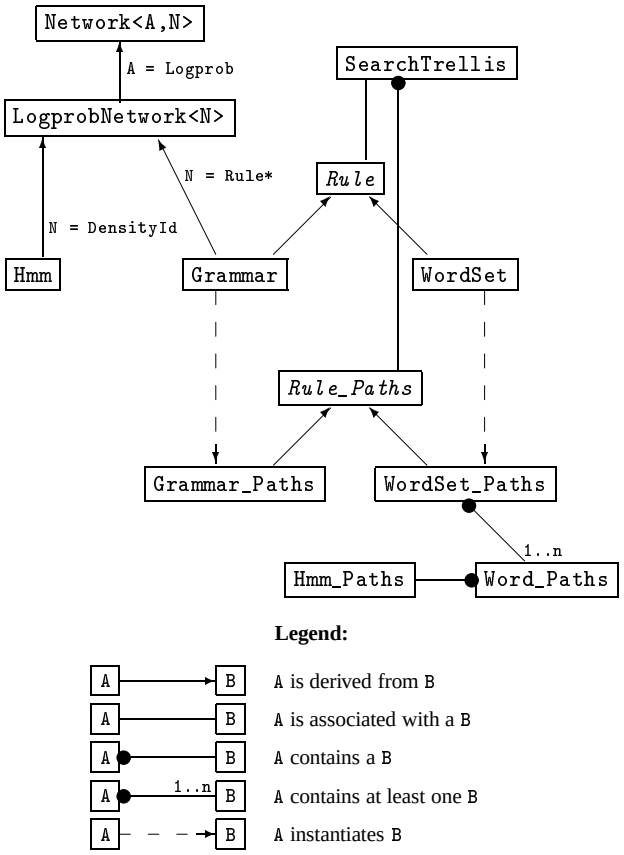

Figure 1: Partial class diagram, using a notation adapted from that of Booch [1].

Rule_Paths is an abstract base class from which Grammar_Paths and WordSet_Paths are derived. This class provides a virtual member function extend_paths because extending the paths for a Grammar involves different steps than for a WordSet. For each acoustic observation, the SearchTrellis is updated by invoking extend_paths on its Rule_Paths object. Understanding the forward phase of the beam search requires understanding what extend_paths does for each of the dynamic classes.

The implementation of WordSet_Paths::extend_paths is straightforward. If there is an active entry path into this WordSet, we first activate any Words that are inactive, i.e., create a Word_Paths object for each Word in the WordSet that doesn't currently have one. Then for each Word that is active, we invoke Word_Paths::extend_paths. If after returning from this function a Word_Paths object has no active paths because of pruning, it is destroyed.

Word_Paths::extend_paths just passes the request on to Hmm_Paths::extend_paths. But if extending the paths of the

HMM yields an active exit node, then it is also the job of `Word_Paths::extend_paths` to create a backtrace cell and add it to the backtrace data structure.

The implementations of `extend_paths` for `Grammar_Paths` and `Hmm_Paths` are patterned after the beam search algorithm described in [8, 9]. In particular, paths for active nodes are maintained in dynamic containers and best-path selection is performed using temporary arrays. For both HMMs and Grammars, extending paths from one time instant to the next takes place in three stages: (1) extend internal paths, (2) if there is an active entry path, extend it to the entry nodes, and (3) incorporate the observation.[2]

In `Hmm_Paths`, all active nodes extend paths to their successor nodes. In `Grammar_Paths`, a node can be active without having an active exit path, so only active nodes that have an active exit path extend that path to their successor nodes.

Extending an active entry path involves the same steps for both HMMs and Grammars. In either case, any entry nodes which were inactive upon invoking this function need to be created.

For `Hmm_Paths`, incorporating the observation just means evaluating the log-likelihood functions associated with the active nodes. For `Grammar_Paths` it requires invoking the member function `extend_paths` on each active node. But some nodes correspond to WordSets and others to Grammars. Since the dynamically created nodes in a `Grammar_Paths` object each contain a `Rule_Paths*` and since `extend_paths` is virtual, the correct version is invoked.

Here is the simple one-best decoding strategy we first used. There is a class `Word_Hyp` (not shown in Figure 1) for which each object represents one word hypothesis. Each time a `Word_Paths` object has an active exit path, a `Word_Hyp` object is created and added to a list containing all of the word hypotheses created at that time instant. The `Word_Paths`' exit path is changed to point back to this newly created `Word_Hyp`. The `SearchTrellis` keeps these lists in an array with one list (possibly empty) for each time instant. When there are no more observations to process, we access the best-scoring exit path of the `SearchTrellis`'s `Rule_Paths`. An attribute of that path is its ending `Word_Hyp`. Because each `Word_Hyp` has a pointer to its predecessor `Word_Hyp`, traceback of the top-scoring answer is trivial. Of course, a more sophisticated approach is required for N-best decoding.

Each path being extended is represented by two values: a score and a path history. Using the simple one-best decoding strategy outlined above, the path history is just a pointer to the most recent `Word_Hyp` on the path.

5. Conclusion

This paper has motivated the use of layered bigrams for speech recognition and outlined an implementation of Viterbi beam search in C++ that incorporates them. A number of interesting problems have not been addressed here. One is the design of layered bigrams. Another is to quantify the relationship between the complexity of the layered bigram and the search's speed and accuracy. We are still in the process of improving the algorithm and evaluating this approach.

[2] "Extending a path" means taking the source node's exit score and adding the log-probability on the outbound arc and then comparing it to the best score received so far by the destination node. The destination node remembers the best score it receives and updates its path history accordingly.

Acknowledgment

I would like to thank my colleagues in the Speech Services Technology group here at NYNEX Science & Technology who carefully reviewed earlier drafts of this paper.

6. REFERENCES

1. Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, second edition, 1994.
2. Eric Brill. *A Corpus-Based Approach to Language Learning*. PhD thesis, University of Pennsylvania, 1993.
3. P. F. Brown, V. J. Della Pietra, P. V. deSouza, J. C. Lai, and R. L. Mercer. Class-based n-gram models of natural language. *Computational Linguistics*, 18(4):467–479, Dec. 1992.
4. Kai-Fu Lee. *Automatic Speech Recognition: The Development of the SPHINX System*. Kluwer Academic Publishers, 1989.
5. Michael K. McCandless and James R. Glass. Emprical acquisition of word and phrase classes in the ATIS domain. In *Proceedings, Eurospeech '93*, volume 2, pages 981–984, 1993.
6. Scott Meyers. *Effective C++*. Addison-Wesley, 1992.
7. David R. Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.
8. H. Ney, D. Mergel, A. Noll, and A. Paeseler. A data-driven organization of the dynamic programming beam search for continuous speech recognition. In *1987 ICASSP Proceedings*, pages 833–836, 1987.
9. Hermann Ney, Dieter Mergel, Andreas Noll, and Annedore Paeseler. Data driven search organization for continuous speech recognition. *IEEE Trans. on Signal Processing*, 40(2):272–281, February 1992.
10. Paul Placeway, Richard Schwartz, Pascale Fung, and Long Nguyen. The estimation of powerful language models from small and large corpora. In *Proceedings, ICASSP-93*, volume 2, pages 33–36, 1993.
11. Stephanie Seneff, Helen Meng, and Victor Zue. Language modelling for recognition and understanding using layered bigrams. In *ICSLP 92 Proceedings*, volume 1, pages 317–320, 1992.
12. Steve Young. Large vocabulary continuous speech recognition: A review. In *Proceedings, IEEE Automatic Speech Recognition Workshop*, pages 3–28, Snowbird, Utah, December 1995.