An Unsupervised Head-Dependency Language Model

by

Philip D. Sarin

Submitted to the Department of Electrical Engineering
and Computer Science in partial fulfillment of the
requirements for the degree of

Master of Engineering in Electrical Engineering and
Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 5, 2000
[June 2000]

Author .....
Department of Electrical Engineering and Computer Science
April 14, 2000

Certified by .....
Professor Robert C. Berwick
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

An Unsupervised Head-Dependency Language Model

by

Philip D. Sarin

Submitted to the Department of Electrical Engineering
and Computer Science

May 5, 2000

in Partial Fulfillment of the Requirements for the Degree of Master
of Engineering in Electrical Engineering and Computer Science

Abstract

Language models are an important component of speech recognition. They aim to predict the probability of any word sequence. Traditional n-gram language models consider only adjacent lexical dependencies and so ignore longer-spanning dependencies. One might capture these long-spanning dependencies by augmenting traditional models with head-dependency information. A *head* word is the critical element in a phrase. *Dependents* modify the head. Prior research has shown that head-dependency relationships acquired from an annotated corpus can improve n-gram models. We propose a probabilistic parser and language model which incorporate head-dependency relationships obtained from the unsupervised algorithm in [10]. Preliminary information theoretic measures suggest that a model which combines n-grams and unsupervised head-dependencies will outperform traditional n-gram models.

We describe a probabilistic model. We discuss the algorithms and data structures necessary to implement the model efficiently in a chart parser. We also outline strategies for pruning and reestimation. Our results reveal that the automatically acquired head-dependency information yields a small improvement, through interpolation, over traditional models. A shift-reduce parser outperforms our new chart parser, but further research on pruning strategies for the chart parser needs to be done.

Thesis Supervisor: Robert C. Berwick
Title: Professor

Table of Contents

List of Figures

List of Tables

Acknowledgements

This thesis marks the close of my five years at MIT. I would like to acknowledge those who have helped me reach this stage in life, and on whom I have the privilege to rely.

My father Raj Sarin, my mother Amita Sarin, and my brother Dave Sarin have sacrificed so much for my education and well-being. I simply cannot imagine where I would be without their love and support.

My crew from Washington, DC has inspired me to raise my standards for the past ten years. Although we may never have the opportunity to reconvene in Room Q, I am greatly indebted to all of them for their friendship.

I thank all of my friends at MIT for enriching my life here. They are wonderful and caring people. Robert Pinder, in particular, has been a dear and trusted friend for every day of my five years at MIT.

Of course, I owe thanks also to those who have helped me with my thesis. Peter Ju, my roommate for three summers and one fall at AT&T, has been a good friend and companion. Thanks to Beppe Riccardi, Srini Bangalore, Mark Beutnagel, Alistair Conkie, Candy Kamm, Rick Rose, Juergen Schroeter, Ann Syrdal, and all the others at AT&T. Thanks also to Professor Berwick at MIT for supervising my thesis work.

Finally, thanks to everything-remains-raw@mit.edu for their silent, tolerant support.

Chapter 1

Introduction

Our goal is to build a better language model for speech recognition. Language models make word predictions to help speech recognizers differentiate between ambiguous acoustic input. N-gram models, the traditional language models for speech recognition, have the weakness that they only consider local lexical dependencies when making word predictions. Overcoming this problem is a fundamental challenge in language modeling. We describe a new language model which incorporates long distance dependencies in the form of head-dependency relationships. We aim to integrate head-dependency information obtained from the unsupervised algorithm of [10] with traditional models. An unsupervised statistical parser described herein will enable the construction of head-dependency language models and the evaluation of their performance. We hypothesize that language models which combine n-gram and head-dependency features will outperform traditional models.

We have two goals. First is to demonstrate that the head-dependency relationships can be acquired through unsupervised learning. Second, we seek to prove that these relationships are useful for language modeling. Previous research on head-dependency parsers and language models has relied on annotated corpora.

In this chapter, we first discuss the background of speech recognition, language modeling, and head dependency relationships. Next, we cover previous work on head-dependency language models and unsupervised language learning. Finally, we outline the rest of the thesis.

1.1 Background

1.1.1 Speech Recognition and Language Modeling Basics

A speech recognizer transcribes speech into text. That is, given an acoustic signal A, the recognizer determines the word sequence $\hat{W}$ such that

$$\hat{W} = \arg max_W P(W|A) = \arg max_W P(A|W) \times P(W) \tag{1.1}$$

The language model computes P(W) for any word sequence W. This probability can be broken down as

$$P(w_1 w_2 \ldots w_N) = \prod_{i=1}^{N} P(w_i | w_1 \ldots w_{i-1}) \tag{1.2}$$

To limit the number of parameters to estimate, traditional language models assume that only a fixed number of words of left context are needed to predict the following word. Such models are *n-gram language models*:

$$P(w_1 w_2 \ldots w_N) = \prod_{i=1}^{N} P(w_i | w_{i-n+1} \ldots w_{i-1}) \tag{1.3}$$

N-gram models have the clear deficiency that they only consider dependencies between adjacent words. Consider predicting the word `within` in the sentence `Art Monk will enter the professional football hall of fame within three years.` A 3-gram, or trigram, language model would estimate the probability of `within` using `of` and `fame`, neither of which strongly suggests the presence of `within`. The verb `enter` is a much stronger predictor.

Increasing n to include `within` would lead to an exponential explosion of the number of parameters and to poor statistical estimates. We instead aim to augment n-gram predictions by using head-dependencies, which are discussed in more detail below.

1.1.2 Head-Dependency Relationships

Head-dependency relationships are one manner of capturing long-spanning dependencies. Head-dependency relationships represent links between key words or phrases, known as *heads*, and the *dependents* which augment them [6]. A head is the critical element of a phrase. Heads themselves can be dependents of other heads. That is, head-dependencies have recursive structure. The Riccardi et al. algorithm allows automatic head-dependency annotation of a corpus.

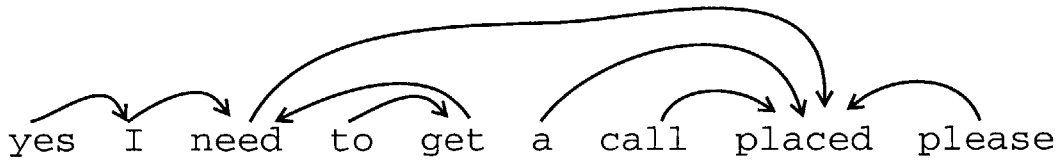


Figure 1.1: Head-dependency links for an example sentence from the *How May I Help You* corpus.

Figure 1.1 shows the head-dependency links chosen by the Riccardi et al. algorithm for the sentence `yes I need to get a call placed here`. An arrow represents a link from dependent to head. For example, `to` is a dependent of `get`. The head words of a sentence have a tree structure. The tree's root, the head word of the whole sentence, is `placed` and its dependents are `need` and `please`. Head-dependency relationships, then, extract the most critical parts of the sentence.

There are other ways of capturing syntactic structure. A common approach is phrase structure grammars which break sentences into syntactic constituents such as noun phrases and verb phrases. This approach has disadvantages. First, placing words and phrases into any number of predefined categories will require learning using an annotated corpus. Second, introducing new nonterminal[1] symbols increases the parameter space of the stochastic language model. The heuristics in [10] can find head-dependencies in an unsupervised and language-independent fashion.

1. Terminal symbols are lexical elements. A nonterminal symbol maps into sequences of terminal and nonterminal symbols. Example of nonterminals in a phrase-structure grammar are "noun phrase" and "verb phrase."

1.1.3 Head-dependency heuristics

According to [10], four characteristics indicate head-dependency information in an unannotated corpus.

1. **Word frequency:** Words which occur most frequently in a corpus are less likely to be heads.

2. **Word 'complexity':** In languages with phonetic orthography, such as English, longer words can be considered more complex. Words which are neither too short nor too long are likely to be heads.

3. **Optionality:** Some words, such as modifiers, are more likely to be optional in a sentence. Optional words are less likely to be heads. Riccardi et al. determine whether a word (e.g., "black") is optional by comparing the distribution of trigrams with the word in the middle (e.g., "the black cat") to the distribution of the bigrams composed by the outer two words in the trigrams (e.g., "the cat").

4. **Distance:** In general, two words which occur far apart in a sentence are less likely to be head and dependent than two words which are close together.

By taking a weighted sum of the above metrics, Riccardi et. al produce a ranked list of head words. With this information, they have created a deterministic head-dependency parser.

The deterministic parser operates in two steps. First, it divides a sentence into chunks, or phrases, based on a language's function words, which typically occur at phrase boundaries. In languages such as English, the right-most word in a chunk is the chunk's head word. Other languages, such as Japanese, have the left-most word in a chunk as the head. There are two options for parsing within a chunk. In a flat structure, every word in a chunk

depends on the chunk's head. In a ladder structure, every word depends on the words to its right (in English).

Figure 1.2: Within-chunk structures. Flat (left) and ladder (right).

Next, the head words are ranked according to the heuristics above. The word most likely to be a head word is the head of the sentence. The procedure is repeated recursively to the words on the left of the head, and to the words on the right of the head. The head words of the left and right subsentences are both considered dependents of the sentence's head word.

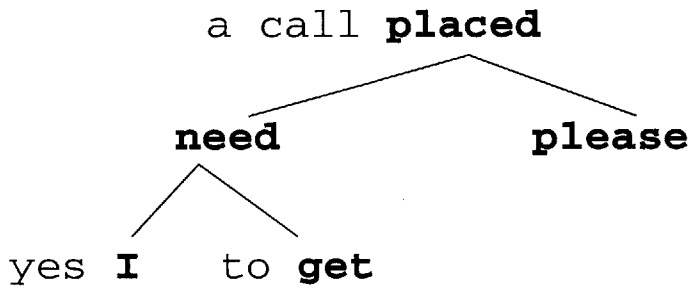


Figure 1.3: Hierarchical derivation of the sentence in Figure 1.1.

Figure 1.3 shows the derivation of the sentence, `yes I need to get a call placed pleased`, according to the Riccardi et al. algorithm. Notice that there are five chunks (`yes I`, `need`, `to get`, `a call placed`, `please`), whose head words are in boldface. As Figure 1.1 on page 14 shows, a flat within chunk structure was used for this sentence.

1.1.4 Parsers

A probabilistic parser is needed to determine head-dependency derivations for unseen sentences. It will output joint probabilities P(W, T) for a word sequence W and each derivation[1] T. From this quantity, it is easy to get P(W).

$$P(W) = \sum_{T} P(W, T) \tag{1.4}$$

The parser's probability values will be trained on a corpus annotated by the deterministic parser described above. We will then reestimate the probabilities with parses generated by our parser. As our parser forms part of a language model for real-time automatic speech recognition, it must build parses incrementally from left to right as input is being received. Our parser is discussed in detail in Chapter 4.

1.2 Previous Research

1.2.1 Link Grammars vs. Phrase Structure Grammars

In [11], Riccardi and Bangalore developed techniques to acquire phrase-structure grammar automatically. "Phrase-grammar" language models developed based on these techniques showed minor improvements in perplexity, a measure commonly used to evaluate language models [12]. In addition, [12] shows the phrase-grammar models generalized better on unseen data than n-gram models. As noted earlier, phrase-structure grammars have the drawback that they require the introduction of nonterminals and an increase in vocabulary size. Other types of grammars do not have this limitation.

Sleator and Temperly propose *link grammars*, which are fully lexical grammars expressed by relationships between words [13]. Head-dependency grammars are a subset of link grammars. Link grammars have several advantages over phrase-structure grammars. They allow easier addition of vocabulary and simplified handling of exceptional cases. De Marcken places the blame for poor performance of unsupervised grammar acquisition on the choice of phrase-structure grammars. He argues that link grammars better represent the structure of natural language [7].

1.2.2 Previous Attempts at Statistical Parsing

---

1. The terms "parse tree" and "derivation" are used interchangeably in this thesis.

Statistical parsers trained with supervised learning algorithms have had encouraging results [3] [2]. Chelba and Jelinek [2] have trained a statistical head-dependency parser with the annotated UPenn Treebank corpus. They combine their parser with an n-gram language model. Their results show that a head-dependency model exhibits better perplexity results than a trigram model. Interpolating between the two leads to even better perplexity values. Speech recognizer word accuracy also improves.

Previous attempts also used different parsing algorithms. The parser in [2] was a stack-based shift/reduce parser. The chart parser in [3] could consider more parses efficiently, but this parser was not applied towards speech recognition and did not have a left-to-right restriction. PGLR parsing in [1] extends the extremely powerful LR parsing technique to natural languages. LR parsers are shift/reduce parsers which rely on a precompiled state machine to improve speed. PGLR research, however, has required part-of-speech labeled text and grammars consisting of parts of speech and not lexical items. With a lexical grammar, a PGLR state machine is computationally too expensive to construct. We discuss parsing techniques in greater detail in a subsequent chapter.

1.3 Outline

The next chapter discusses preliminary statistical results which suggest that the Riccardi et. al. heuristics will show language model improvement. Chapter 3 outlines the probabilistic model. Chapter 4 explains our chart parser and the techniques employed to add efficient probabilistic machinery to a chart parser. Chapter 5 covers reestimation. In Chapter 6, we present and discuss our results. We conclude in the final chapter.

Chapter 2

Preliminary Statistics

Experiments with n-gram models and head-dependencies learned by the Riccardi et al. algorithm indicate that these two types of relationships complement each other. We used mutual information to measure the suitability of various word predictors. We also calculated conditional mutual information to gauge the benefits of combining these predictors.

2.1 Mutual Information

Mutual information measures the degree of dependence between two random variables, in bits. If A and B have mutual information 1, that means that the value of B reduces A's entropy, or the uncertainty in A's value, by one bit. This property is reflected in (2.1), the equation for calculating mutual information. More information on mutual information and entropy is available in [4].

$$I(A; B) = H(A) - H(A \mid B) = H(B) - H(B \mid A) \tag{2.1}$$

Conditional mutual information, $I(A; B \mid C)$ conveys the amount of new knowledge A's value gives about B, or vice versa, given that C's value is already known. As (2.3) shows one can view this measure as this gain in knowledge about A that B provides in addition to C, or the gain in knowledge about B that A provides in addition to C.

$$I(A; B \mid C) = H(A \mid C) - H(A \mid B, C) = H(B \mid C) - H(B \mid A, C) \tag{2.2}$$

$$I(A; B \mid C) = I(A; B, C) - I(A; C) = I(A, C; B) - I(B; C) \tag{2.3}$$

2.2 Experimental Results

Using the *How May I Help You* corpus[1], we calculate the predictive effects of head words and random words in the sentence. In order to predict the effectiveness of a left-to-right parser, we also perform some measurements which consider only words drawn from

1. *How May I Help You* is a database of spontaneous speech responses to the prompt "How May I Help You?" for a call-routing application. See [5].

a word's left context. These measurements include the previous head word, the previous two head words, and the previous non-head word. We compare these predictors to each other and to n-grams, and we evaluate combining n-grams with other predictors.

Legend:
b(w) -- previous word
t(w) -- previous two words
ph(w) -- previous head (not necessarily of w)
pht(w) -- previous two heads (not necessarily of w)
pd(w) -- previous dependent (non-head word)
h(w) -- head of W
r(w) -- random other word in the sentence
pr(w) -- random word before W in the sentence
bold -- highest value in column
underline -- highest value in column among measures which consider only previous words

Table 2.1: Information Characteristics of Various Measures Compared to Bigrams

A(w)	I(w; A(w))	I(w; A(w) | b(w))	I(w; b(w), A(w))	combined b(w), A(w) improvement over b(w) alone
b(w)	**3.32**	0	3.32	0%
r(w)	1.12	1.77	5.09	53%
h(w)	2.72	**2.33**	**5.65**	**70%**
pr(w)	1.20	1.46	4.77	44%
ph(w)	2.47	1.37	4.68	41%
pd(w)	2.53	1.08	4.39	43%

Table 2.2: Information Characteristics of Various Measures Compared to Trigrams

A(w)	I(w; A(w))	I(w; A(w) | t(w))	I(w; t(w), A(w))	combined t(w), A(w) improvement over t(w) alone
t(w)	**4.71**	0	4.71	0%
h(w)	2.72	**1.68**	**6.40**	**36%**
pht(w)	3.89	1.35	6.07	29%

N-grams display the best predictive abilities. The trigram predictor contributes 4.71 bits of information while the bigram gives 3.32 bits. Automatically acquired head words also provide a significant amount of information: 2.72 bits. All other predictors, including the random words, also give considerable information.

Furthermore, the experiments reveal that the information which n-grams and head-dependencies predict is largely not redundant. Combining n-grams and head words contributes 70% more information over bigrams, and 36% over trigrams. There is still a significant information gain over n-grams when we restrict ourselves to predictors from the left context. Combining the previous head with n-grams gives a 41% improvement over bigrams. The previous two heads improve 29% over trigrams. These results suggest that models which combine head-dependency information with n-grams should outperform n-gram language models.

Previous work by Wu et al. in [15] on a Chinese annotated supports these results. Wu et al. conclude that the closest preceding word modified by w is w's best predictor when bigram information is already known. This predictor is similar to ph(w), and Table 2.1 reflects that ph(w) is second to pr(w) in its information gain over the bigram predictor. No baseline corpus of random words is considered in the Wu et al. analysis.

There are limitations with our analysis. First, the Wu et al. analysis operates on a Japanese corpus, and not on our English corpus. Ideally, we would like to compare our results to those of a hand-annotated version of our corpus. More importantly, the preliminary analysis does not measure the predictive effect of the Riccardi et al. tree structure of the left context. Our language model will have a more sophisticated approximation of that structure than this analysis. We expect that our actual language model predictors will be more effective than the ones used in this chapter. Since even a random previous word pro-

vides considerable information (See Table 2.1.), there is good reason to expect our model to improve upon the n-gram.

Chapter 3

Probabilistic Model

Our probabilistic model determines the probability of any word sequence W by summing up the joint probabilities P(W, T) for all parses T. This chapter discusses how we calculate P(W, T), and describes a simple parser closely related to our model. The next chapter discusses a more sophisticated parser which efficiently implements our model. We begin by defining the terms used in this chapter.

3.1 Important Terms

3.1.1 Word Parse

A *word parse* is a sequence of binary trees known as *word parse trees*, each of whose nodes is labeled with a head word. Figure 3.1 shows a word parse consisting of four word-parse trees.

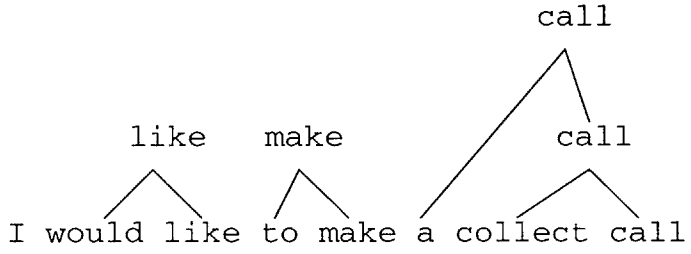


Figure 3.1: An example word parse.

The word parse can be decomposed into two parts, a word sequence W and a parse sequence T. W is the sequence of the labels of the leaves of the word parse plus an invisible terminating symbol </S>. For the example in Figure 3.1, W is I would like to make a collect call </S>. T is the underlying structure of the word parse. HEAD(p), the head of a word parse tree p, is the label of the root node of t ("call" for the right-most tree in Figure

3.1). We define FIRST(p) to be the label of the left-most leaf of p ("a" for the right-most tree in Figure 3.1).

3.1.2 Parse Actions and Parse Stacks

A parse sequence T is a sequence of *parse actions* which operate on a *parse stack* and an *input sequence*. A *parse stack* is a stack of *word parse trees*. Originally, the parse stack is empty, and the input sequence is some word sequence W. Parse actions take place at every time step where the first time step is zero. There are three parse actions:

1. **SHIFT**. Move a word from the input sequence to the top of the parse stack. The time step is the number of words which have been shifted, and so a SHIFT action ends a time step.

2. **ADJOIN-LEFT.** Pop off p_0 and $p_{(-1)}$ from the parse stack, the top and second items respectively of the parse stack. Push onto the stack a node whose left child is p_{-1}, whose right child is p_0, and whose head is HEAD($p_{(-1)}$).

3. **ADJOIN-RIGHT**. The same as ADJOIN_LEFT, but the head-word of the new node is HEAD(p_0).

STACK **INPUT SEQUENCE**

Time Step 0

I would like to make a collect call </S>

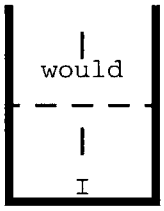
SHIFT action

TIME STEP 1

would like to make a collect call </S>

SHIFT action

TIME STEP 2

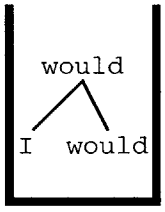


like to make a collect call </S>

STACK **INPUT SEQUENCE**

TIME STEP 2 (continued)

ADJOIN-RIGHT action



like to make a collect call </S>

SHIFT action

Figure 3.2: Time steps 0 through 2 for the word parse in Figure 3.1.

Figure 3.2 gives an example of the first few parse actions for the word parse in Figure 3.1. Time step 0 consists of only one SHIFT operation, the only possible operation at that time step. Time step 1 also contains a lone SHIFT. At time step 2, an ADJOIN-RIGHT

replaces the I and would leaf word parse trees with one word parse tree which adjoins them and has head would.

There are two important parsing rules:

1. A SHIFT action is the end of a time step.

2. When the symbol </S> is shifted onto the stack, parsing ends. No parse actions are ever taken after that point.

3.2 Computing P(W)

We compute P(W) by summing all P(W, T) for all parse sequences T.

$$P(W) = \sum_{T} P(W, T) \tag{3.1}$$

We can compute the word parse probability P(W, T) by the following equation:

$$P(W, T) = \prod_{i=1}^{n+1} \left(P(w_i | W_{i-1} T_{i-1}) \times \prod_{j=1}^{k} P(t_i^j | W_{i-1} T_{i-1} w_i t_i^1 \ldots t_i^{j-1}) \right) \tag{3.2}$$

Here, t_i^j is the jth parsing action taken at time step i. We employ the following equivalence classifications:

$$P(w_i | W_{i-1} T_{i-1}) = P(w_i | h_{-(n-1)} \ldots h_{(-1)} h_0, \text{SHIFT}) \tag{3.3}$$

$$P(t_i^j | W_{i-1} T_{i-1} w_i t_i^1 \ldots t_i^{j-1}) = P(t_i^j | h_{-(m-1)} \ldots h_{(-1)} h_0) \tag{3.4}$$

Here, h_0 is the head word of the stack top word parse tree, h_{-1} is the head word of the second to top tree, etc. The head words of items on the stack are called *exposed heads*. Words are predicted by n previous exposed heads, and parse actions are predicted by m previous exposed heads.

3.3 Initial Estimation

We initially estimate probabilities by maximum likelihood. We compute from training data counts of all events needed to estimate probabilities.

1. $C(w_i, h_{-(n-1)} \ldots h_{(-1)} h_0, \text{SHIFT})$

2. $C(h_{-(n-1)} \ldots h_{(-1)} h_0, \text{SHIFT})$

3. $C(t_i^j, h_{-(m-1)} \ldots h_{(-1)} h_0)$

4. $C(h_{-(m-1)} \ldots h_{(-1)} h_0)$

We estimate not only all of these counts, but also their degenerate cases with smaller histories. These other counts are used for backoff, discussed in the next chapter.

The actual conditional probability estimation is straightforward.

$$P_{\text{ML}}(A|B) = \frac{C(A,B)}{C(B)} \tag{3.5}$$

3.4 Unseen Events

Our model must be able to make word and parse action predictions in stack configurations unseen in training. To do so, we implement separate backoff strategies for word prediction and parse prediction events. If the size of the history H is greater than 0,

$$P(w|H, \text{SHIFT}) = \alpha(H) P_{\text{ML}}(w|H, \text{SHIFT}) + (1 - \alpha(H)) P(w|\text{CHOP}(H), SHIFT) \tag{3.6}$$

$$P(t|H) = \beta(H) P_{\text{ML}}(t|H) + (1 - \beta(H)) P(t|\text{CHOP}(H)) \tag{3.7}$$

Here, P_{ML} is the maximum likelihood probability estimate. CHOP(H) is the history H without its least recent item. Notice that both (3.6) and (3.7) are recursive. Their base cases, where the size of H is zero, are below.

$$P(w|H, \text{SHIFT}) = P_{\text{ML}}(w|H, \text{SHIFT}) \tag{3.8}$$

$$P(t|H) = P_{\text{ML}}(t|H) \tag{3.9}$$

$$\tag{3.10}$$

3.5 A Simple Parser

A straightforward implementation of our model is a shift-reduce parser. Its prime advantage is ease of implementation and validation. Each parse for a particular word sequence

in a shift/reduce parser is represented by a parse stack[1]. We would need to maintain a parse stack for *each* parse that is being considered for a particular word sequence. Considering *all* possible parse stacks is computationally infeasible, and so we employ a pruning strategy.

We would like a wide variety of parses to survive pruning. Flat parses, those with very few ADJOIN-RIGHT and ADJOIN-LEFT actions, are very similar in nature to n-grams. Deeper parses incorporate more long-spanning dependencies than flat parses, but a deep parse is normally less probable than a flat parse because the former involves more parse actions. Therefore, a naive pruning strategy would contain mostly flat parses. In order to capture the long-spanning dependencies of deep parses, we must specifically preserve them in our pruning strategy.

We shall use the method from [2]. All parses of a word sequence will be placed into buckets based on the total number of ADJOIN-LEFT and ADJOIN-RIGHT actions. For a 25 word sentence, then, there will be 25 buckets. The n-gram parse, which contains no adjoin actions, would be placed in bucket 0. All parses which contain 24 adjoin actions will be in bucket 24. We prune each bucket based on two criteria:

1. $\varepsilon_{\text{bucket}}$: We only keep word parses T for which $\frac{P(W,T)}{P(W,T_{max})} \geq \varepsilon_{bucket}$, where T_{max} is the parse of W with the highest probability.

2. $\text{MAX}_{\text{bucket}}$: Each bucket contains at most $\text{MAX}_{\text{bucket}}$ word parses.

In this manner, a mixture of flat and deep parses survive pruning.

---

1. A shift/reduce parser can operate on any context free grammar. The parser derives its name from the two actions it supports. A *shift* action moves a symbol from the input sequence to the top of a parse stack, just as the SHIFT we have defined. A *reduce* action takes symbols from the top of the parse stack and replaces them with a tree headed by some non-terminal symbol. In our parser, there are only two types of reduce actions: ADJOIN-LEFT and ADJOIN-RIGHT. We abuse notation by not differentiating between terminal and non-terminal symbols. We can do this because our grammar is fully lexical. For more information, consult [1].

Chapter 4

The Chart Parser

4.1 Limitations of Shift-Reduce Parsers

Shift-reduce parsers discard a great deal of information in pruning. With our probabilistic model, only the head words of word parse trees are needed to predict words or parse actions. Therefore, two parse stacks of the same size whose corresponding elements have the same head words are equivalent for predictive purposes in our probabilistic model. We will either eliminate one of these stacks in pruning, or eliminate another stack instead. Ideally, though, we would merge these two stacks with dynamic programming.

Another limitation of shift-reduce parsers is their inability to process *word lattices* efficiently. A word lattice is an acyclic directed graph which serves as a compact representation for a set of word string hypotheses. Ideally, our parser should be able to parse word lattices while exploiting their compactness.

In this chapter, we propose a chart parser whose data structure permits dynamic programming at the expense of the shift-reduce parser's simplicity. In order to implement our model efficiently with a chart parser, we introduce several calculations and computational shortcuts. We also hoped to be able to use a chart parser to implement word lattice parsing. Due to time constraints, we decided instead to focus on single hypothesis parsing. The result is more complex than the basic parser presented in the previous chapter.

A chart parser has a more compact representation than a shift-reduce parser, and maintains all possible parses in a single data structure. This approach allows dynamic programming to consolidate word parse trees with the same head words. In effect, this feature allows a chart parser to consider many more parses. Chart parsers, however, do not readily

produce parse stacks. Implementing the probabilistic computations required by our model, then, is more difficult with a chart parser than with a shift/reduce parser.

4.2 Chart Parsing Overview

The fundamental data structure in a chart parser is the *chart*. A chart contains *constituents*, or word parse trees which span a part of the input sequence. Leaf constituents have a span of length one. Every non-leaf constituent has a span greater than length one, and it has a left and a right subconstituent in the chart. A non-leaf node inherits its head from one of its subconstituents. The chart keeps track of all the constituents, and which sections of the input each constituent spans.

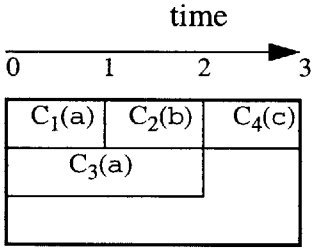


Figure 4.1: An example chart. The head words of each constituent are in parentheses.

Figure 4.1 shows an example chart containing four constituents. C_1, C_2, and C_4 are leaf constituents with heads `a`, `b`, and `c` respectively. C_3 is a non-leaf constituent with head `a`. Figure 4.1 does not show C_3's subconstituents, but it is easy to determine that they must be C_1 and C_2. By looking at the leaf constituents, we can tell this chart shows parses for the word sequence `a b c`.

A chart represents several word parses simultaneously. There is no explicit representation of a parse stack, however. A word parse is a path of constituents C_1, C_2, ..., C_k where

- C_1 has starting point 0.
- C_k has end point m, where m is the number of words in the sequence.
- For $1 \leq i < k$, the end point of C_i is the starting point of C_{i+1}.

There are two word parses in Figure 4.1. The first is the sequence of constituents C_1, C_2, C_4. The second is the sequence C_3, C_4. Notice that C_4 has two *word parse prefixes*, or paths leading to its beginning point from 0.

Basic Chart Parsing Algorithm

The basic chart parsing algorithm requires another data structure, the *agenda*, which maintains a list of constituents scheduled for addition to the chart.

While the input sequence is not empty,

1.0 If the agenda is empty, read word *i*, the next word, from the input sequence and create for that word a new leaf constituent spanning from *i*-1 to *i*. Add that constituent to the agenda.

2.0 While the agenda is not empty

- 2.1 Take a constituent C from the agenda.
- 2.2 For all constituents C' in the chart whose ending position is the same as C's starting position
  - 2.2.1 Perform an ADJOIN-LEFT action. Create a new constituent C_L whose left subconstituent is C', whose right subconstituent is C, and whose head is HEAD(C'). Add C_L to the agenda.
  - 2.2.2 Perform an ADJOIN-RIGHT action. Create a new constituent C_R whose left subconstituent is C', whose right subconstituent is C, and whose head is HEAD(C). Add C_R to the agenda.

This algorithm explores all possible parses for a word sequence. Before a constituent C is added to the chart, the algorithm builds all possible constituents for which C can be the right subconstituent. Clearly, we cannot efficiently find all possible parses for a word

sequence, and so we shall augment this algorithm with a pruning strategy later in this chapter.

4.3 Computing P(*W*) With a Chart Parser

Computing probabilities with a chart parser is hard because we cannot access all parse stacks cheaply. Below, we present a framework for implementing our probability computations and show how to compute P(*T*, *W*) for all parses of a particular word sequence. Next, we show how to obtain P(*W*) efficiently. We finally demonstrate how to build P(*W*) incrementally when constituents are added. In subsequent sections, we outline ways to make the parser more efficient with dynamic programming and with pruning. We also extend the machinery in this section to implement reestimation.

4.3.1 The ϕ Computation

In order to simplify our calculations, we would like to store as much probability information as possible with each constituent. In shift/reduce parsing, we could give each parse stack a probability which changes as the stack is modified. Each constituent in a chart parser could belong to several parse stacks, and so we cannot store *all* the necessary probability information inside constituents.

For a given history *H* of exposed heads and a constituent C, ϕ(C, *H*) is the contribution which C gives to a word parse. The history is a sequence of exposed heads preceding C, and, by convention, the history includes C's head as well. If there are fewer previous exposed heads in the chart than the history requires, than we fill the remaining symbols in the history with the start symbol `<s>`. The length of the history needed is the maximum of the lengths of the word and parse histories chosen. That is, it is the maximum of *n* and *m* from (3.3) and (3.4). We shall call this quantity *histsize*. If we choose a model with word and parse history sizes both two, then C_4 in Figure 4.1 on page 31 has two histories: `[b, c]` and `[a, c]`. The computation of ϕ is below.[1]

For a leaf constituent, C_L:

1. If C_L starts at position 0 in the chart, then there is only one possible history h. It has most recent symbol HEAD(C_L) and preceding symbols <s>.

$$\phi(C_L, H) = P(w|H, \text{SHIFT}) \tag{4.1}$$

2. Otherwise, C_L does not start at 0. We will delay computing any probabilities for C_L.

$$\phi(C_L, H) = 1 \tag{4.2}$$

For a non-leaf constituent C with left subconstituent C_1 and right subconstituent C_2:

We must account for the following events in C's ϕ values for a history H:

1. C_1 is created. ($\phi(C_1, H)$)

2. A SHIFT action occurs when C_1 is on the top of the stack. (Not accounted for in $\phi(C_2)$. Computed by (3.4).)

3. The next word is FIRST(C_2), which is pushed on top of C_1 on the stack. (Not accounted for in $\phi(C_2)$. Computed by (3.3).)

4. C_2 is created. ($\phi(C_2)$)

5. C_1 and C_2 are adjoined by an action t (ADJOIN-LEFT or ADJOIN-RIGHT). (Computed by (3.4).) The history H' is a history of size *histsize* grown from H with C_2's head added on.

$$\phi(C) = \phi(C_1, H) \times P(\text{SHIFT}|H) \times P(\text{FIRST}(C_2)|H, \text{SHIFT}) \times \phi(C_2, H') \times P(t|H') \tag{4.3}$$

The ϕ value, then, stores part of a constituent's contribution to the probability of a word parse. It contains all of the word and parse actions which take place *within* the con-

1. We shall assume that all histories stored in the parser are of length *histsize*. If a history is longer than is required for a specific probability computation, then we shall assume that the history is automatically truncated to the appropriate length before probabilities are computed.

stituent, but none which take place outside (except for the case in which the constituent starts at 0.) In the case of constituents starting at 0, there is only one history H, and ϕ(C, H) is the probability of the word parse consisting of the single constituent C. We next determine how to calculate P(W) from the ϕ values.

4.3.2 Getting P(W) from ϕ

We shall compute P(W) by summing all P(W, T) for all parse stacks. We cannot determine P(W, T) directly from the ϕ values as the ϕ values do not contain the adjacency probabilities of items on the parse stack. The chart parser's representation makes it most efficient for us to sum up P(W, T) for *classes* of parses rather than individual parses. Since we are interested most in P(W), and not P(W, T), there is no harm in computing word parse probabilities in bunches.

At each time step i, for all histories H which include the heads of constituents ending at i, we shall memoize the value $P_{i,H}$. This is the probability of all word parses ending with history H at position i. We shall also define $\mathbf{C}(i, j, H)$ to be the set of all constituents spanning from i to j with a ϕ value defined for H. We shall compute the P values as follows:

$$P_{1,H} = \left(\sum_{C \in C(0,1,H)} \phi(C,H) \right) \times P(\text{SHIFT}|H) \tag{4.4}$$

$$P_{n+1,H} = \left(\sum_{C \in C(0,n+1,H)} \phi(C) + \sum_{i=1}^{n} \sum_{H'} \left(P_{i,H'} \sum_{C \in C(i,n+1,H)} P(\text{FIRST}(C)|H', \text{SHIFT}) \phi(C,H) \right) \right) \times \tag{4.5}$$

$$P(\text{SHIFT}|H)$$

$$= \left(\sum_{C \in C(0,n+1,H)} \phi(C) + \sum_{i=1}^{n} \sum_{H'} \left(P_{i,H'} \cdot P(w_{i+1}|H', \text{SHIFT}) \sum_{C \in C(i,n+1,H)} \phi(C,H) \right) \right) \times \tag{4.6}$$

$$P(\text{SHIFT}|H)$$

Equation (4.4) computes P for position 1 of a sentence. The equation is trivial as there is only one possible word-parse from 0 to 1, a constituent from 0 to 1. The computation of P values for subsequent positions in the sentence cannot rely on ϕ values alone. As (4.5)

and (4.6) show, we look at all constituents ending at position n+1 when computing P values at position n+1. For such constituents starting at 0, we need only include their ϕ values. For each constituent C not starting at 0, we must account for all word parses containing C, where C starts at position i. For each history H' leading up to C's head, we have $P_{i,\,H'}$, the sum of all word parses *before* C with ending history H'. We multiply this value by word prediction probability of C's first word, and then we multiply by C's ϕ value to include probability computations internal to C. Notice that all of the above equations include P(SHIFT | H). Recall that all time steps are completed by SHIFT actions.

To obtain P(W) for a n word sentence, we simply sum up $P_{n,\,H}$ for all H which can end at position n.

4.3.3 The ρ Computation

Each time a constituent C is added to the chart, we shall increment all appropriate P counters. According to (4.6), we also must add the prefix and adjacency probabilities for C for *each* history H' *preceding* the start of C. Furthermore, we repeat this computation for all constituents which start at C's position. Much of this computation is unnecessary. All constituents which start at position i are preceded by the same set of histories. We can, then, make the computation in (4.6) more efficient by memoizing prefix and adjacency probabilities.

We shall denote a history H's set of predecessors ending at position i as PRED(H, i). We can rewrite (4.6)

$$P_{n+1,\,H} = \left(\sum_{C \in C(0,\,n+1,\,H)} \phi(C) + \sum_{i=1}^{n} \sum_{C \in C(i,\,n+1,\,H)} \phi(C, H) \cdot \rho(w_{i+1}, i, \mathrm{PRED}(H, i)) \right) \times P(\mathrm{SHIFT} \mid H) \tag{4.7}$$

where

$$\rho(w, i, \mathbf{H}) = \sum_{H \in \mathbf{H}} P_{i,\,H} \cdot P(w \mid H, \mathrm{SHIFT})$$

The ρ value needs to be computed only once for a given set of parameters. Subsequent computations can be looked up in constant time. One concern, though, is the complexity of computing PRED(H, i). Actually, this set can be uniquely represented by the pair (H_{-1}, i) where H_{-1} is H with its most recent item chopped off. Therefore, PRED(H, i) can be computed in constant time.

4.4 Dynamic Programming

Since our probabilistic model requires only exposed heads, and not a complete parse action history, we can consolidate different constituents with the same exposed heads. If two constituents have the same start position, the same end position, and the same head symbol, then they are identical with respect to our probabilistic model. Once identical constituents are added to the chart, then further identical constituents are created from ADJOIN-LEFT and ADJOIN-RIGHT actions. We identify identical constituents and consolidate them in the agenda before they are added to the chart.

We shall represent our agenda as a heap. Constituents with shorter spans are at the top of the heap. Constituents of the same length are sorted with respect to their head word. Once added to the chart, a constituent C will produce other constituents for addition to the agenda, but all of these constituents must be longer than C. By keeping shorter constituents at the top, we ensure that all constituents identical to C_{top}, the top item of the heap, are already in the agenda.

We consolidate constituents when we extract an item from the agenda.

1.0 Let C_{top} be the top item of the agenda. Pop the agenda.

2.0 While the top item C' of the agenda has the same head, start position, and end position as C_{top},

  2.1 For all H of C_{top}, increase $\phi(C_{top}, H)$ by $\phi(C', H)$.[1]

2.2 Pop the agenda.

3.0 Add C_{top} to the chart.

4.5 Efficiency Considerations

Because of backoff, all possible parses of a particular word sequence have nonzero probabillity. Even with dynamic programming, a sentence of length n would generate $\Theta(n^3)$ constituents. Each constituent would have $\Theta(n^{histsize})$ histories associated with it. It is essential to eliminate extremely unlikely parses from consideration.

In our parser, we prune in three stages. First, we prune at the agenda level when constituents are being requested from the agenda. Second, we undertake an additional pruning pass at the end of each time step. Finally, we eliminate low-probability histories associated with each constituent. The first step prevents the generation of several constituents which we would eventually prune out in the second step. We perform the second step to keep constant the number of new constituents added at each time step. For these two steps, we prune based on P(C), the probability of all word parses whose last item is constituent C. The computation of this value is discussed in Appendix A. We perform the final pruning step so that the running time will be independent of *histsize*.

4.5.1 Agenda Pruning

1. Since C' and C_{top} have the same head word and the same start position, they are guaranteed to have the same set of histories.

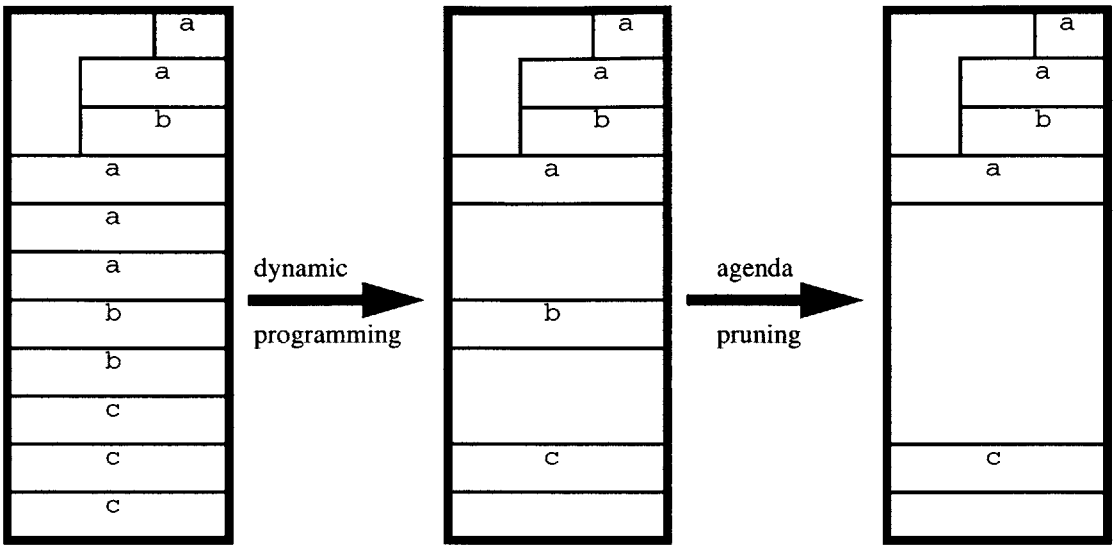


Figure 4.2: Dynamic programming and agenda pruning on constituents added at a particular time step.

Figure 4.2 illustrates the relationship between dynamic programming and agenda pruning. Dynamic programming ensures that there is only one constituent with a particular head over a particular span. Agenda pruning eliminates low probability constituents (such as a `b` constituent in the figure) which survive dynamic programming.

We perform pruning separately on constituents of different lengths. Recall that the agenda is a heap in which shorter constituents are at the top. All constituents in the agenda end at the current time step. For each constituent length l, we select from all constituents C of length l the one which maximizes P(C). We call this constituent C_l^*. Finally, we prune for each length l according to two parameters:

1. $\varepsilon_{\text{agenda}}$: We keep C only if $\frac{P(C)}{P(C_l^*)} \geq \varepsilon_{\text{agenda}}$
2. $\text{MAX}_{\text{agenda}}$: At most this number of constituents of length l are kept.

4.5.2 Chart Pruning

Once all constituents are added to the chart for a particular time step, we perform another pruning pass over all constituents ending at that time step. Our two parameters are similar to those we used for agenda pruning.

1. $\varepsilon_{\text{chart}}$: We keep C only if $\frac{P(C)}{P(C^*)} \geq \varepsilon_{\text{chart}}$ where C^* is the constituent which maximizes P(C) over all C ending at the current time step.

2. $\text{MAX}_{\text{chart}}$: At most this number of constituents are kept.

4.5.3 History Pruning

We keep at most MAX_{hist} histories for each constituent. We keep the histories H for which the value P(C, H) is the highest, where P(C, H) is the probability of all word parses ending in constituent C with history H. Computing P(C, H) for a constituent starting at position i is straightforward:

$$P(C, H) = \rho(w_{i+1}, i, \text{PRED}(H)) \times \phi(C, H) \tag{4.8}$$

4.5.4 Normalization

We normalize probability values after all pruning steps at time step i by multiplying all ϕ values by $\frac{P_{total}}{P_i}$ where P_i is computed *after* pruning. P_{total} is the probability of all word parses at time i *before* pruning.

4.6 Software Validation

Our complicated computations and pruning strategy made the parser extremely difficult to validate. The chart data structure was monolithic. It was not possible to break it into testable components. We had to test the whole chart parser as a unit. It was possible to design simple controlled experiments for the chart. The pruning strategy, however, only operated on more complicated examples, and so systematic validation was not possible.

We validated our parser's ability to handle flat parses by passing in n-gram grammars (100% SHIFT probability) and comparing them to known results. We tested deeper parses with simple test cases. To test pruning, we compared probability values from unpruned

and pruned parses, and verified that the differences were reasonable. Finally, we added runtime assertions as sanity checks to warn of unsound probability values. These assertions do not assure program correctness. No validation strategy does. By placing them in parts of the code which are most error prone, and by allowing them to run on several thousand sentences through many reestimation iterations, we are confident that we eliminated the critical bugs.

4.7 Conclusion

In this chapter, we have outlined a chart parser to implement our probabilistic model. Because the model is based on a shift-reduce framework, we have to do a significant amount of book keeping to make it work in a chart parser. The chart parser is much more complex, but its potential to store a greater number of parses and to handle word lattices may make up for this complexity.

Chapter 5

Reestimation

This chapter discusses our use of the expectation maximization (EM) method for reestimating probabilities. The counts initially used to compute our maximum likelihood probabilities are reestimated based on all parses which survive pruning. The manner of computing probabilities from the counts does not change. Likewise, backoff is unchanged. We must, however, add new machinery to obtain reestimated counts from the parser.

We begin with an overview of EM reestimation and then proceed to discuss an implementation of EM with our chart parser.

5.1 EM Overview

The E part of EM stands for "expectation." For each sentence in the reestimation set, the counter for an event is incremented by the expected number of times that event occurs over all parses for the event. That is, for each time an event A occurs in parse T of a word sequence W, C(A) is incremented by P(T | W).

The M part stands for "maximization." We are attempting to maximize the expectations of observed events. We can do so by continuing to estimate pre-backoff probabilities by maximum likelihood.

5.2 EM Implementation

Implementing EM with a shift-reduce parser is straightforward. For a chart parser, reestimation is more complicated. In a chart parser based EM reestimation, we store counts for the following events for each constituent C:

1. The adjacency of C to its left context. For all left histories *H*, C(SHIFT, *H*) and C(FIRST(C), *H*, SHIFT) must be incremented.

2. The occurrence of the parse actions internal to C. The counts for all adjoin actions and all stack histories much also be updated.

3. The adjacency of C to its right context. This part is handled when a constituent in C's right context is processed by the reestimation algorithm.

All of these events must be weighted by $\frac{P(\text{all word-parses containing } C \text{ and } H)}{P(W)}$ for all H. We must calculate this value efficiently. We described in the previous chapter how to calculate P(W). We obtain the probability of all word parses containing C and H with the following equation:

$$P(\text{all word parses containing } C \text{ and } H) = P(\text{all word-parse actions preceding C, ending in H}) \times \phi(C, H) \times P(\text{all word-parse actions after C and H}) \quad (5.1)$$

The P computations help us determine the probability of all word parses preceding C, and the probability of C's adjacency to those word parses. The ϕ computation provides the probability of all word parse actions internal to C. What's currently missing is the probability of all word parse actions which can follow C.

5.2.1 The R Computation

We shall define $R_{i,H}$ to be the probability of all word parse actions beginning at position i and preceded by history H. The following algorithm calculates R after all the parser has finished running on a particular input sequence.

1.0 For all constituents C, in descending order of start position.

1.1 All R are initially zero.

1.2 Let *start* be the start position of C, and *end* be the end position.

1.3 If no constituents begin at *end*,

1.3.1 For all H *immediately preceding* C, let H' be the corresponding history for C, and increment $R_{start,H}$ by $P(\text{FIRST}(C)|H) \cdot \phi(C, H')$.

1.4 If there are constituents beginning at *end*,

1.4.1 For all *H immediately preceding* C, let *H'* be the corresponding history for C, and increment $R_{start, H}$ by $P(\text{FIRST}(C)|H) \cdot \phi(C, H') \cdot R_{end, H'}$.

5.2.2 Updating Counts

The R value allows us easily to update all counts used for probability estimation. To do so, we introduce the following helper functions:

• INC_STACK_COUNT(*H*, *weight*): Increases by *weight* the number of times history *H* is on the top of the stack.

• INC_WORD_COUNT(*w*, *H*, *weight*): Increases by *weight* the number of times the word *w* is shifted from the front of the input sequence to the top of the stack when *H* is on top of the stack.

• INC_PARSE_COUNT(*t*, *H*, *weight*): Increases by *weight* the number of times parse action *t* occurs when *H* is on top of the stack.

When we say "*H* is on top of the stack," we mean that the top *histsize* elements in the stack correspond to the items in *H*.

The algorithm for computing reestimation counts is below. For each constituent C in the chart after a word sequence *W* has been parsed,

1.0 Let *start* be C's start position, and *end* be C's end position.

2.0 For all histories *H* which precede *start*,

2.1 Let *H'* be the corresponding history of C.

2.2 If C starts at 0, let *prev* be 1.

2.3 If C does not start at 0, let *prev* be $P_{start, H} \cdot P(\text{FIRST}(C)|H)$

2.4 If no constituents start at *end*, let *weight* be $\frac{prev \times \phi(C, H')}{P(W)}$.

2.5 If constituents do start at *end*, let *weight* be

$$\frac{prev \times \phi(C, H') \times P(\text{SHIFT} \mid H') \times R_{end, H'}}{P(W)}.$$

2.6 INC_PARSE_COUNT(SHIFT, *H*, *weight*)

2.7 INC_WORD_COUNT(FIRST(C), *H*, *weight*)

2.8 INTERNAL_INC_PROBS(C, *H*, *weight*)

Above, steps 2.6 and 2.7 account for C's being adjacent to its left context. The internal event counts, including counts for all stack events, is left to the procedure INTERNAL_INC_PROBS, listed below.

INTERNAL_INC_PROBS(C, *H*, *weight*)

1.0 Once again, let *H'* be the corresponding history of C.

2.0 INC_STACK_COUNT(*H'*, *weight*)

3.0 If C is a leaf constituent, return. Otherwise, continue.

4.0 Let C_L be C's left child, and C_R be C's right child.

5.0 INTERNAL_INC_PROBS(C_L, *H*, *weight*)

6.0 Let H_L be the history of C_L corresponding to *H*. INC_PARSE_COUNT(SHIFT, H_L, *weight*).

7.0 INC_WORD_COUNT(FIRST(C_R), H_L, *weight*)

8.0 INTERNAL_INC_PROBS(C_R, H_L, *weight*)

9.0 Let H_R be the history of C_R corresponding to H_L. INC_PARSE_COUNT(*t*, H_R, *weight*). Here, *t* is either ADJOIN-LEFT or ADJOIN-RIGHT, whichever was used to form C.

5.3 Conclusion

EM reestimation with the chart parser utilizes the chart parser's data structures and computations. We only add the R computation. As with parsing, reestimation is also significantly more complicated with a chart parser than with a shift/reduce parser.

Chapter 6

Results

6.1 Head-Dependency Model Results

6.1.1 Perplexity Results

Our shift-reduce reestimation algorithm produces perplexity results which give slight improvement over the bigram. Interpolation with a bigram model yields a 1.1% improvement in perplexity over the bigram. Reestimating with the chart parser, however, causes no decrease in perplexity, but interpolation gives a small improvement. Our results are in Table 6.1.

Table 6.1: Perplexity results.

Language Model	Training Set	Test set	Test interpolated
Bigram	16.96	19.99	19.99
Initial HD Model (chart)	29.59	22.53	19.99
Initial HD Model (S/R)	19.47	22.56	19.99
Chart Reestimated (5) HD (chart parsed)	17.09	20.04	19.83
Chart Reestimated (5) HD (S/R parsed)	17.09	20.05	19.84
S/R Reestimated (6) HD (chart parsed)	16.85	19.89	19.78
S/R Reestimated HD (6) (S/R parsed)	16.85	19.90	19.79

Notice that shift-reduce parsing and chart parsing of a particular model yield very similar results. In each case, the perplexity from the shift-reduce parser is very slightly higher than that of the chart parser. Reestimating with a shift-reduce parser, however, causes a much greater improvement in perplexity. Perplexity numbers from chart-based reestimation appear to converge to a value slightly higher than the bigram perplexity. Shift-based reestimation results in a perplexity drop of 0.09, or 0.45%. Interpolation causes an overall

perplexity drop of 0.21, or 1.1% with a shift-reduce reestimated model. Interpolating the bigram with the chart-reestimated model gives a 0.8% improvement over the bigram.

Time constraints limited the number of tests we could run. Ideally, we would have experimented with a two item history model for comparison to a trigram model. Trigrams are more commonly used in speech recognition than bigrams, and the Chelba/Jelinek model was trigram based. We also did not have time to experiment with the "ladder" within-chunk training scheme described in the first chapter. We employed a flat within-chunk scheme instead. Finally, we were unable to test our results on different corpora.

6.1.2 Word Accuracy Results

With interpolation and shift-reduce reestimation, we achieved 0.3% improvement in word accuracy over the bigram.

6.2 Comparison to Chelba/Jelinek Results

Our perplexity improvements were not as strong as those of Chelba and Jelinek. On an annotated version of the Switchboard corpus[1], they demonstrated 4.7% perplexity improvement and a 0.8% word accuracy improvement.

6.3 Analysis

Our chart parser produces lower perplexity numbers on a given model than our shift-reduce parser, but the latter produces better results through reestimation. We believe that the chart parser is more likely to converge to a local maximum through reestimation than the shift-reduce parser. In this case, the local maximum appears to be the bigram model. The chart reestimated models are very similar to the bigram model, and so they demonstrate a smaller improvement through interpolation. The shift-reduce models are less similar to the bigram, and so perplexity results improve with and without interpolation.

---

1. Switchboard is an unconstrained speech corpus.

Why does the chart parser converge to the bigram? N-gram parses are the flattest possible parses. They have the fewest parse actions, and so they have higher parse probabilities. A good pruning strategy will have to ensure that the overall probability of a sentence does not get overrun by the probabilities of the flattest parses. The shift-reduce parser appears to differ to a greater degree from the n-gram model. We believe that the shift-reduce parser's pruning strategy is more successful at preserving deeper parses than the chart parser's.

Chapter 7

Conclusion

7.1 Automatic Grammar Acquisition

Through the Riccardi/Bangalore automatic grammar acquisition scheme and the methods described in this thesis, we demonstrated a small improvement in perplexity and word accuracy over n-gram models. We do not achieve the results of the supervised algorithm in [2]. Our information theoretic measures, however, do indicate that we should achieve a significant improvement over n-grams. Although our own results do not exhibit a large gain over bigram models, the information theoretic measures indicate that a refinement of our approach may produce more significant improvements.

At this time, then, it is not possible to draw a firm conclusion on the suitability of Riccardi/Bangalore head-dependencies for language modeling. While our own work reveals a very small improvement, there is still hope that our work may be tuned to yield better results.

7.2 Chart Parsing in a Head-Dependency Language Model

While we do demonstrate that it is possible to implement the Chelba/Jelinek model efficiently with a chart parser, our results prevent our drawing a conclusion on the chart parser's suitability for language modeling. While the chart parser did not yield results as good as those of the shift-reduce parser, the former did show some improvement with a new pruning strategy which has yet to be fine tuned.

Why did the shift-reduce parser outperform the chart parser? In particular, why did the latter converge towards flatter parses. The two primary differences between the parsers are the presence of dynamic programming and the pruning strategy. One would expect dynamic programming to help deeper parses relative to flatter ones. Without dynamic pro-

gramming, the contribution of a particular deep parse may be entirely ignored. The pruning strategies of the two models were significantly different, because imposing the pruning scheme from [2] on a chart parser would have been extraordinarily difficult. Because of time constraints, we did not have time to study different pruning strategies systematically.

A potentially discouraging feature of the chart parser is its extraordinary complexity relative to that of a shift-reduce parser. Bugs were much more frequent and harder to find in the chart parser, and validation was much more difficult. More time and greater care are needed to implement a chart parser.

At this time, it is too early to conclude whether a chart parser is suitable for language modeling. Considerably more work needs to be done if we are to develop a chart parser for language modeling. At this time, we can only say that there is still potential for the chart parser to improve through further research.

7.3 Future Work

As mentioned above, one major difference between our chart parser and our shift-reduce parser is the pruning strategy. Further exploration of pruning strategies both for the shift-reduce and chart parsers could produce better results more in line with the information theoretic measures in Chapter 2. In particular, any pruning strategy will have to combat the natural tendency to select flatter parses over deeper ones. As the chart parser is more complicated and is achieving poorer results currently, a more systematic exploration of chart parser pruning strategies would be particularly enlightening.

We would also like to have explored whether one can use a probabilistic head-dependency chart parser to process word lattices efficiently. Our own probabilistic model, pruning strategy, and chart parsing code base may or may not be appropriate for word lattices. Due to time constraints, we abandoned work in this area.

7.4 Summary

In summary, we conclude that automatically acquired head-dependency language models demonstrate minor improvements over n-grams. Our models also have the potential to demonstrate further improvements. Chart parsing may or may not be a viable option for language modeling, but they do have potential. Further research needs to be done in pruning strategies, and perhaps also in parsing word lattices.

Appendix A

Computing P(C)

P(C) is the probability of all word-parses starting at 0 and ending with constituent C. If C starts at position 0, then we can obtain P(C) directly from C's ϕ values. (Since C's prior history is just the START symbol, it actually has only one history.)

$$P(C) = \sum_{H \text{ for } C} \phi(C, H) \tag{A.1}$$

If C starts at nonzero position i, it is straightforward to derive P(C) from the ρ computation:

$$P(C) = \sum_{H \text{ for } C} \phi(C, H) \cdot \rho(w_{i+1}, i, \text{PRED}(H, i)) \tag{A.2}$$

References

[1] A. V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA,1988.

[2] C. Chelba and F. Jelinek, "Refinement of a structured language model," ICAPR, 1998.

[3] M. J. Collins, "A new statistical parser based on bigram lexical dependencies," *Proceedings of the 34th Annual Meeting of the Association for Computation Linguistics*, pages 184-191, Santa Cruz, California, 1996.

[4] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, Wiley, 1991.

[5] A. L. Gorin, G. Riccardi, and J. G. Wright, "How May I Help You," *Speech Communication*, pages 113-127, 23, 1997.

[6] R. Hudson, *English Word Grammar*, Blackwell, Cambridge, MA, 1991.

[7] C. de Marcken, "On the unsupervised induction of phrase-structure grammars," *3rd Workshop on Very Large Corpora*, 1995.

[8] K. Inui, V. Sornlertlamvanich, H. Tanaka, and T. Tokunaga, "A new formalization of probabilistic GLR parsing," *Proceedings of International Workshop on Parsing Technologies*, 1997.

[9] F. Jelinek, *Statistical Methods For Speech Recognition*, MIT Press, Cambridge, MA, 1997.

[10] G. Riccardi, S. Bangalore and P. D. Sarin, "Learning Head-Dependency Relations from Unannotated Corpora," 1999.

[11] G. Riccardi and S. Bangalore, phrase-grammar paper.

[12] P. D. Sarin, unpublished, 1998.

[13] Sleator and Temperly, link grammar paper, cmu.

[14] M. Tomita, editor, *Generalized LR Parsing*, Kluwer Academic Publishers, 1991.

[15] D. Wu, Z. Jun, and S. Zhifang, "An information-theoretic empirical analysis of dependency-based feature types for word prediction models," EMNLPVLC-99, 1999.