Parsing Inside-Out

A thesis presented

by

Joshua T. Goodman

to

The Division of Engineering and Applied Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

May 1998

Abstract

Probabilistic Context-Free Grammars (PCFGs) and variations on them have recently become some of the most common formalisms for parsing. It is common with PCFGs to compute the inside and outside probabilities. When these probabilities are multiplied together and normalized, they produce the probability that any given non-terminal covers any piece of the input sentence. The traditional use of these probabilities is to improve the probabilities of grammar rules. In this thesis we show that these values are useful for solving many other problems in Statistical Natural Language Processing.

We give a framework for describing parsers. The framework generalizes the inside and outside values to semirings. It makes it easy to describe parsers that compute a wide variety of interesting quantities, including the inside and outside probabilities, as well as related quantities such as Viterbi probabilities and n-best lists. We also present three novel uses for the inside and outside probabilities. The first novel use is an algorithm that gets improved performance by optimizing metrics other than the exact match rate. The next novel use is a similar algorithm that, in combination with other techniques, speeds Data-Oriented Parsing, by a factor of 500. The third use is to speed parsing for PCFGs using thresholding techniques that approximate the inside-outside product; the thresholding techniques lead to a 30 times speedup at the same accuracy level as conventional methods. At the time this research was done, no state of the art grammar formalism could be used to compute inside and outside probabilities. We present the Probabilistic Feature Grammar formalism, which achieves state of the art accuracy, and can compute these probabilities.

Acknowledgements

To Erica
my love, my help

There are too many people for me to thank[1], and for too many things. Most of all, this thesis is for Erica, who I met a month after I started graduate school, and who I will marry a month after I finish.

Next, I want to thank my committee. Stuart Shieber is an amazing advisor, who gave me wonderful support, and just the right amount of guidance. Barbara Grosz has been a great acting advisor in my last year, putting her foot down on important things, but allowing me to occassionally split infinitives when it did not matter. Fernando Pereira has given me very good advice and is a terrific source of knowledge. Thanks also to Leslie Valiant for the almost thankless task of serving on my committee.

For my first year or two, I shared an office with Stan Chen and Andy Kehler. I'd rather share a small office with them than have a large office to myself – they taught me as much as anyone here and made grad school fun. They, and the other members of the AI-Research group read endless drafts of my papers, attended no end of nearly-identical practice talks, and listened to hundreds of bad ideas. Wheeler Ruml in particular has done countless small favors. Lillian Lee has been a source of guidance. I'm happy to call Rebecca Hwa my friend. Others – Luke Hunsberger, Ellie Baker, Kathy Ryall, Jon Christenson, Christine Nakatani, Nadia Shalaby, Greg Galperin, and David Magerman (for poker, and rejecting my ACL paper) – have all helped me through and made my time here better.

A year after getting here, I developed tendonitis in my hands. I needed more help than usual to graduate, and I got it, both from the Division of Engineering and Applied Sciences, and from many of the people I have named so far.

A Ph.D. is built on a deep foundation. My parents and family have always pushed me and supported me, and continue to do so. My father encouraged me from a ridiculously early age (what kind of person gives a 5 year old Scientific American?) while my mother, through what she called gentle teasing, gave me her own special kind of encouragement. I want to thank Edward Siegfried, my mentor for nine years, who taught me so much about computers. As an undergraduate, I had many helpful professors, among whom Harry Lewis stands out. After college, I worked at Dragon Systems, where Dean Sturtevant, Larry

[1] In the AI Research Group, acknowledgments are traditionally funny. This encourages people to read the acknowledgements, and skip the thesis. I have therefore written rather dry acknowledgments, and put one joke in this thesis, to encourage the reading of this work, in its entirety. The joke is not very funny, so you will have to read the whole thing to be sure you have found it. I will send $5 to the first person each year to find the joke without help. Some restrictions apply.

Gillick, and Bob Roth initiated me into the black art of speech recognition, and taught me how to be a good programmer.

I'd also like to thank the National Science Foundation for providing most of my funding with Grant IRI-9350192, Grant IRI 9712068, and an NSF Graduate Student Fellowship.

Contents

List of Figures

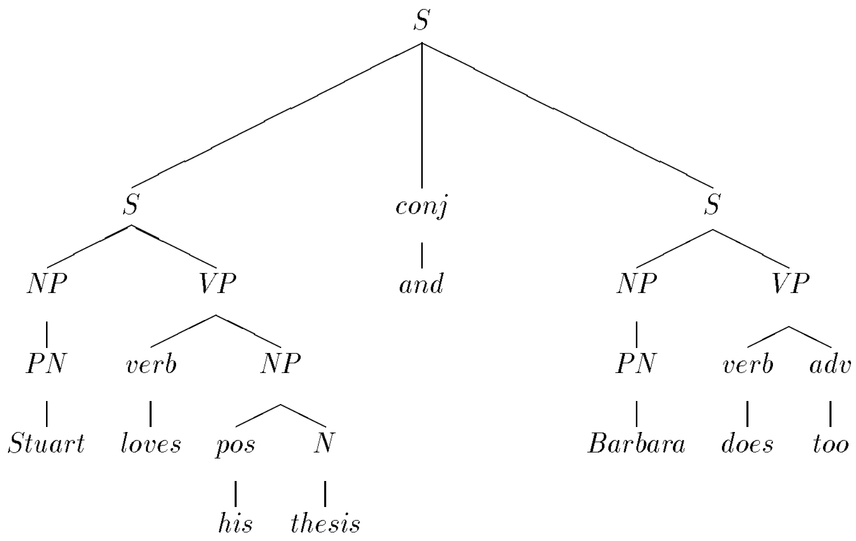
List of Tables

Chapter 1

Introduction

1.1 Background

Consider the sentence "Stuart loves his thesis, and Barbara does too." If a human being, or a computer, attempts to understand this sentence, what will the steps be? One might imagine that the first step would be a gross analysis of the sentence, a syntactic parsing to determine its structure, breaking the sentence into a conjunction and two sentential clauses, breaking each sentential clause into a noun phrase (*NP*) and a verb phrase (*VP*), and so on, recursively to the words.



After this first syntactic step, a semantic step would follow. The sentence would be converted into a logical representation, ruling out incorrect interpretations such as "Stuart loves someone's thesis, and Barbara loves someone else's thesis," and allowing only inter-

Figure 1.1: Inside probability example

pretations such as "Stuart loves his own thesis, and Barbara loves Stuart's thesis too", and "Stuart loves someone's thesis, and Barbara loves that thesis too." After this semantic step, a pragmatic step would be necessary to disambiguate between these interpretations. Given that Stuart is notoriously hard to please, we would probably decide that it is Stuart's own thesis which is the intended antecedent, rather than some more recent one.

The eventual goal of Natural Language Processing (NLP) research is to understand the meaning of sentences of human language. This is a difficult goal, and one we are still a long way from achieving. Rather than attacking the full problem all at once, it is prudent to attack subproblems separately, such as syntax, semantics, or pragmatics. We will be solely concerned with the first problem: syntax. There are two broad approaches to syntax: the rule-based and the statistical. The statistical approach has become possible only recently with the advent of bodies of text (corpora) which have had their structures hand annotated, called tree banks. Each sentence in the corpus has been assigned a parse tree – a representation of its structure – by a human being. The majority of these sentences can be used to train a statistical model, and another portion can be used to test the accuracy of the model.

In this thesis, we will be particularly concerned with two special quantities in statistical NLP, the inside and outside probabilities. For each span of terminal symbols (words), and for each nonterminal symbol (i.e., a phrase type, such as a noun phrase), the inside probability is the probability that that nonterminal would consist of exactly those terminals. For instance, if one out of every ten thousand noun phrases is "his thesis," then

$$inside(\text{"his thesis"}, NP) = .0001$$

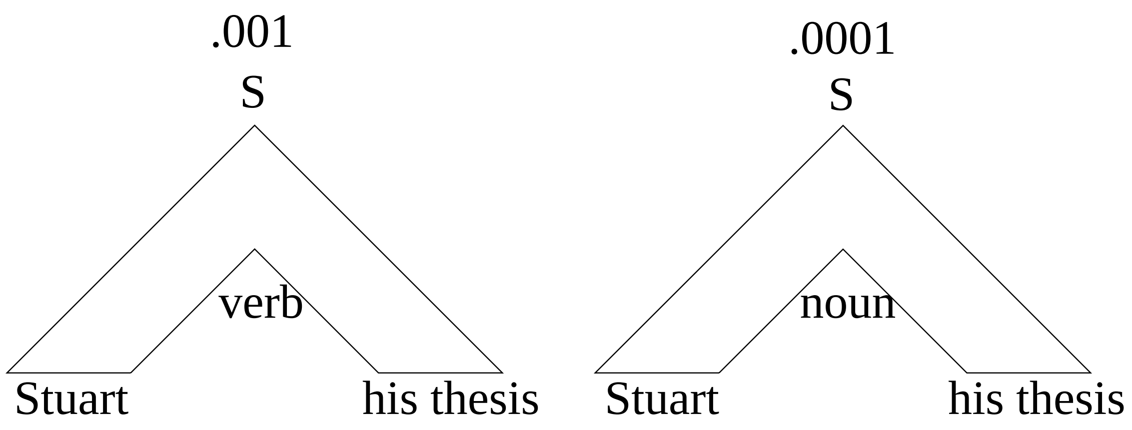


Figure 1.2: Outside probability example

While a phrase like "his thesis" can be interpreted only as a noun phrase, natural language is replete with ambiguities. For instance, the word "loves" can be either a verb or a noun. Let us assume that there is a one in ten chance that a given verb will be loves (as seems true in the literature of computational linguistics examples), and a one in one hundred chance that a given noun will be loves. Then

$$inside(\text{“loves”}, verb) = .1$$

$$inside(\text{“loves”}, noun) = .01$$

We might denote this graphically as in Figure 1.1.

The outside probabilities can be thought of as the probability of everything surrounding a phrase. For instance, if the probability of a sentence of the form "Stuart *verb* his thesis" is say one in one thousand, and one of the form "Stuart *noun* his thesis" is one in ten thousand, then

$$outside(\text{“Stuart \_\_\_\_\_\_ his thesis”}, verb) = .001$$

$$outside(\text{“Stuart \_\_\_\_\_\_ his thesis”}, noun) = .0001$$

This is illustrated in Figure 1.2.

Now, we can multiply the inside probabilities by the outside probabilities, as illustrated in Figure 1.3. The product of the inside and outside probabilities gives us the overall probability of the sentence having the indicated structure. For instance, there is a $.1 \times .001 =$

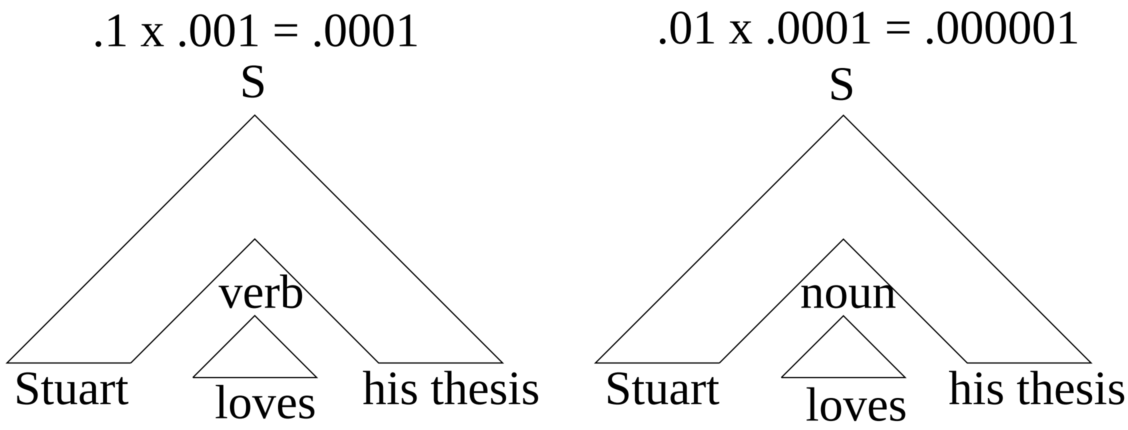


Figure 1.3: Inside-Outside probability example

.0001 chance of the sentence "Stuart loves his thesis", with "loves" being a verb, and a $.01 \times .0001 = .000001$ chance of the sentence with "loves" being a noun. Since these are the only two possible parts of speech for "loves," the overall probability of the sentence is $.0001 + .000001 = .000101$. We can divide by this overall probability to get the conditional probabilities. That is, $\frac{.0001}{.000101} \approx 0.99$ is the conditional probability that "loves" is a verb, given the sentence as a whole, while $\frac{.000001}{.000101} \approx 0.01$ is the conditional probability that "loves" is a noun.

Traditionally, the inside-outside probabilities have been used as a tool to learn the parameters of a statistical model. In particular, the inside-outside probabilities of a training set using one model can be used to find the parameters of another model, with typically improved performance. This procedure can be iterated, and is guaranteed to converge to a local optimum.

While learning the probabilities of a statistical grammar is the traditional use for the inside and outside probabilities, the goal of this thesis is different. Our goal is to show that the inside-outside values are useful for solving many other problems in statistical parsing, and to provide useful tools for finding these values.

1.2 Introduction to Statistical NLP

In this section, we will give a very brief introduction to statistical NLP, describing context-free grammars (CFGs), probabilistic context-free grammars (PCFGs), and algorithms for

```
boolean chart[1..n, 1..|N|, 1..n+1] := FALSE;
for each start s
    for each rule A → w_s
        chart[s, A, s+1] := TRUE;
for each length l, shortest to longest
    for each start s
        for each split length t
            for each rule A → BC ∈ R
                /* extra TRUE for expository purposes */
                chart[s, A, s+l] := chart[s, A, s+l] ∨
                        chart[s, B, s+t] ∧ chart[s+t, C, s+l] ∧ TRUE;

return chart[1, S, n+1];
```

Figure 1.4: CKY algorithm

parsing CFGs and PCFGs.

1.2.1 Context-Free Grammars

We begin by quickly reviewing Context-Free Grammars (CFGs), less with the intention of aiding the novice reader, than of clarifying the relationship to PCFGs, in the next subsection.

A CFG G is a 4-tuple $\langle N, \Sigma, R, S\rangle$ where N is the set of nonterminals including the start symbol S, Σ is the set of terminal symbols, and R is the set of rules (we use R rather than the more conventional P to avoid confusion with probabilities, which will be introduced later). Let V, the vocabulary of the grammar, be the set $N \cup \Sigma$. We will use lowercase letters $a, b, c, ...$ to represent terminal symbols, uppercase letters $A, B, C, ...$ for nonterminals, and Greek symbols, α, β, γ,... to represent a string of zero or more terminals and nonterminals. We will use the special symbol ϵ to represent a string of zero symbols. Rules in the grammar are all of the form $A \rightarrow \alpha$.

Given a string $\alpha A \beta$ and a grammar rule $A \rightarrow \gamma \in R$, we write

$$\alpha A \beta \Rightarrow \alpha \gamma \beta$$

to indicate that the first string produces the second string by substituting γ for A. A sequence of zero or more such substitutions, called a derivation, is indicated by $\stackrel{*}{\Rightarrow}$. For instance, if we have an input sentence $w_1...w_n$, and a sequence of substitutions starting with the start symbol S derives the sentence, then we write $S \stackrel{*}{\Rightarrow} w_1...w_n$.

There are several well known algorithms for determining whether for a given input sentence $w_1...w_n$ such a sequence of substitutions exists. The two best known algorithms are the CKY algorithm (Kasami, 1965; Younger, 1967) and Earley's algorithm (Earley, 1970). The CKY algorithm makes the simplifying assumption that the grammar is in a special form, Chomsky Normal Form (CNF), in which all productions are of the form $A \rightarrow BC$ or $A \rightarrow a$. The CKY algorithm is given in Figure 1.4.

Most of the parsing algorithms we use in this thesis will strongly resemble the CKY algorithm, so it is important that the reader understand this algorithm. Briefly, the algorithm can be described as follows. The central data structure in the CKY algorithm is a boolean three dimensional array, the chart. An entry $chart[i, A, j]$ contains *TRUE* if $A \stackrel{*}{\Rightarrow} w_i...w_{j-1}$, *FALSE* otherwise. The key line in the algorithm,

$$\begin{aligned}
&\textit{/* extra TRUE for expository purposes */}\\
&chart[s, A, s{+}l] := chart[s, A, s{+}l] \vee\\
&\qquad\qquad chart[s, B, s{+}t] \wedge chart[s{+}t, C, s{+}l] \wedge \mathit{TRUE};
\end{aligned}$$

says that if $A \rightarrow BC$ and $B \stackrel{*}{\Rightarrow} w_s...w_{s+t-1}$ and $C \stackrel{*}{\Rightarrow} w_{s+t}...w_{s+l-1}$, then $A \stackrel{*}{\Rightarrow} w_s...w_{s+l-1}$. Notice that once all spans of length one have been examined, which occurs in the first double set of loops, we can then proceed to examine all spans of length two, and from there all spans of length three (which must be formed only from shorter spans), and so on. Thus the outermost loop of the main set of loops is a loop over lengths, from shortest to longest. The next three loops examine all possible combinations of start positions, split lengths, and rules, so that the main inner statement examines all possibilities. Because array elements covering shorter spans are filled in first, this style of parser is also called a bottom-up chart parser.

1.2.2 Probabilistic Context-Free Grammars

In this thesis, we will primarily be concerned with a variation on context-free grammars, probabilistic context-free grammars (PCFGs). A PCFG is simply a CFG augmented with probabilities. We denote the probability of a rule $A \rightarrow \alpha$ by $P(A \rightarrow \alpha)$. We can also discuss the probability of a particular derivation or of all possible derivations of one string from

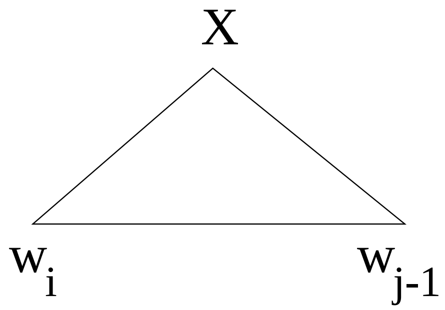


Figure 1.5: Inside probabilities

another. In order to make sure that equivalent derivations are not counted twice, we need the concept of a *leftmost derivation.* A leftmost derivation is one in which the leftmost nonterminal symbol is the one that is substituted for. We will write $\alpha \overset{A\rightarrow\beta}{\Rightarrow} \gamma$ to indicate a leftmost derivation using a substitution of β for A. Now, we define the probability of a one step derivation to be

$$P(\alpha \overset{A\rightarrow\beta}{\Rightarrow} \gamma) = P(A \rightarrow \beta)$$

We can define the probability of a string of substitutions

$$P(\alpha \overset{A_1\rightarrow\beta_1}{\Rightarrow} \gamma_1 \overset{A_2\rightarrow\beta_2}{\Rightarrow} \gamma_2 \overset{A_3\rightarrow\beta_3}{\Rightarrow} ... \overset{A_k\rightarrow\beta_k}{\Rightarrow} \gamma_k) = \prod_{i=1}^{k} P(A_i \rightarrow \beta_i)$$

Finally, we can define the probability of all of the leftmost derivations of some string δ from some initial string α. In particular, we define

$$P(\alpha \overset{*}{\Rightarrow} \delta) = \sum_{k,A_1,\beta_1,\gamma_1,...,A_k,\beta_k,\gamma_k \text{ s.t. } \alpha \overset{A_1\rightarrow\beta_1}{\Rightarrow} \gamma_1 ... \overset{A_k\rightarrow\beta_k}{\Rightarrow} \gamma_k \wedge \gamma_k = \delta} P(\alpha \overset{A_1\rightarrow\beta_1}{\Rightarrow} \gamma_1 \overset{A_2\rightarrow\beta_2}{\Rightarrow} ... \overset{A_k\rightarrow\beta_k}{\Rightarrow} \gamma_k)$$

Several probabilities of this form are of special interest. In particular, we say that the probability of a sentence $w_1...w_n$ is $P(S \overset{*}{\Rightarrow} w_1...w_n)$, the sum of the probabilities of all (leftmost) derivations of the sentence. In general, we will be interested in probabilities of nonterminals deriving sections of the sentence, $P(A \overset{*}{\Rightarrow} w_i...w_{j-1})$. We will call this probability the *inside probability* of A over the span i to j, and will write it as $inside(i, A, j)$.

Note that in general, a derivation of the form $A \overset{*}{\Rightarrow} \alpha$ can be drawn as a parse tree, in which the internal branches of the parse tree represent the nonterminals where substitutions

$\mathit{float}\ \mathit{chart}[1..n, 1..|N|, 1..n+1] := 0;$
for each start s
    for each rule $A \rightarrow w_s$
        $\mathit{inside}[s, A, s+1] := P(A \rightarrow w_s);$
for each length l, shortest to longest
    for each start s
        for each split length t
            for each rule $A \rightarrow BC \ \in R$
                $\mathit{inside}[s, A, s+l] := \mathit{inside}[s, A, s+l] +$
                        $\mathit{inside}[s, B, s+t] \times \mathit{inside}[s+t, C, s+l] \times P(A \rightarrow BC);$
return $\mathit{inside}[1, S, n+1];$

Figure 1.6: Inside algorithm

occured; for each substitution $A \rightarrow B_1 ... B_k$ there will be a node with parent A and children $B_1 ... B_k$. The leaves of the tree when concatenated form the string α. There is a one-to-one correspondence between leftmost derivations and parse trees, so we will use the concepts interchangeably.

When we consider inside probabilities, we are summing over the probabilities of all possible parse trees with a given root node covering a given span. Since the internal structure is summed over for an inside probability, we graphically represent the inside probability of a nonterminal X covering words $w_i ... w_{j-1}$ as shown in Figure 1.5. The inside algorithm, shown in Figure 1.6, computes these probabilities. Notice that the inside algorithm is extremely similar to the CKY algorithm of Figure 1.4. The inside algorithm was created by Baker (1979), and Lari and Young (1990) have written a good tutorial explaining it.

In many practical applications, we are not interested only in the sum of probabilities of all derivations, but also in the most probable derivation. The inside algorithm of Figure 1.6 can be easily modified to return the probability of the most probable derivation (which corresponds uniquely to a most probable parse tree) instead of the sum of the probabilities of all derivations. We simply change the inner loop of the inside algorithm to read:

$$\mathit{Viterbi}[s, A, s+l] := \max(\mathit{Viterbi}[s, A, s+l], \mathit{Viterbi}[s, B, s+t] \times \mathit{Viterbi}[s+t, C, s+l] \times P(A \rightarrow BC));$$

These probabilities are known as the Viterbi probabilities, by analogy to the Viterbi probabilities for Hidden Markov Models (HMMs) (Rabiner, 1989; Viterbi, 1967). With slightly

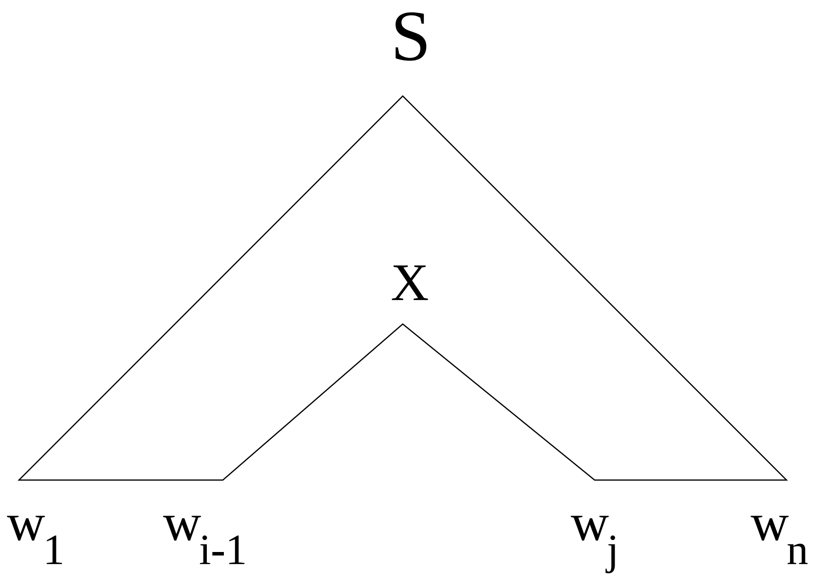


Figure 1.7: Outside probabilities

for each length l, longest **downto** shortest
  for each start s
    for each split length t
      for each rule $A \rightarrow BC \in R$
        $outside[s, B, s+t] := outside[s, B, s+t] +$
          $outside[s, A, s+l] \times inside[s+t, C, s+l] \times P(A \rightarrow BC);$
        $outside[s+t, C, s+l] := outside[s+t, C, s+l] +$
          $outside[s, A, s+l] \times inside[s, B, s+t] \times P(A \rightarrow BC);$

Figure 1.8: Outside algorithm

larger changes, the algorithm can record the actual productions used to create this most probable parse tree.

Another useful probability is the *outside probability*. The outside probability of a non-terminal X covering w_i to w_{j-1} is

$$P(S \stackrel{*}{\Rightarrow} w_1 ... w_{i-1} X w_j ... w_n)$$

This probability is illustrated in Figure 1.7. The outside probability can be computed using the outside algorithm, as given in Figure 1.8 (Baker, 1979; Lari and Young, 1990).

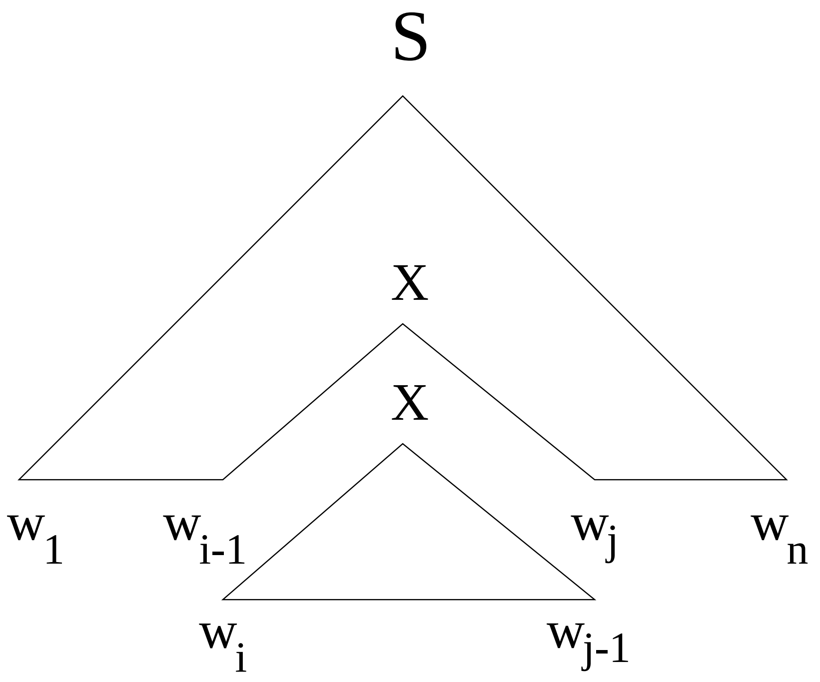


Figure 1.9: Inside-outside probabilities

Notice that if we multiply an inside probability by the corresponding outside probability, we get

$$\begin{aligned} inside(i,X,j) \times outside(i,X,j) &= P(X \stackrel{*}{\Rightarrow} w_i...w_{j-1}) \times P(S \stackrel{*}{\Rightarrow} w_1...w_{i-1} X w_j ... w_n) \\ &= P(S \stackrel{*}{\Rightarrow} w_1...w_{i-1} X w_j ... w_n \stackrel{*}{\Rightarrow} w_1...w_n) \end{aligned}$$

which is the probability that a derivation of the whole sentence uses a constituent X covering w_i to w_{j-1}. This combination is illustrated in Figure 1.9. Now, the probability of the sentence as a whole is just

$$inside(1,S,n+1) = P(S \stackrel{*}{\Rightarrow} w_1...w_n)$$

If we normalize by dividing by the probability of the sentence, we get the conditional probability that X covers $w_i...w_{j-1}$ given the sentence:

$$\frac{inside(i,X,j) \times outside(i,X,j)}{inside(1,S,n+1)} = \frac{P(X \stackrel{*}{\Rightarrow} w_i...w_{j-1}) \times P(S \stackrel{*}{\Rightarrow} w_1...w_{i-1} X w_j ... w_n)}{P(S \stackrel{*}{\Rightarrow} w_1...w_n)}$$

for each iteration i until the probabilities have converged
  for each rule $A \to \alpha$
    $C[A \to \alpha] := 0;$
  for $j := 1$ **to** number of sentences
    compute inside probabilities of sentence j using P_i;
    compute outside probabilities of sentence j using P_i;
    for each length l, shortest to longest
      for each start s
        for each rule $A \to \alpha$
          $C[A \to \alpha] := C[A \to \alpha]+$
        $P_i(S \overset{*}{\Rightarrow} w_1 ... w_{i-1} A w_j ... w_n \overset{A \to \alpha}{\Rightarrow} w_1 ... w_{i-1} \alpha w_j ... w_n \overset{*}{\Rightarrow} w_1 ... w_n | S \overset{*}{\Rightarrow} w_1 ... w_n);$
  for each rule $A \to \alpha$
    $P_{i+1}(A \to \alpha) := \frac{C[A \to \alpha]}{\sum_\beta C[A \to \beta]};$

Figure 1.10: Inside-outside algorithm

$$= \quad P(S \overset{*}{\Rightarrow} w_1 ... w_{i-1} X w_j ... w_n \overset{*}{\Rightarrow} w_1 ... w_n | S \overset{*}{\Rightarrow} w_1 ... w_n)$$

We might also want the probability that a particular rule was used to cover a particular span, given the sentence

$$\begin{aligned} & \frac{\sum_k \mathit{inside}(i, B, k) \times \mathit{inside}(k, C, j) \times \mathit{outside}(i, A, j) \times P(A \to \alpha)}{\mathit{inside}(1, S, n+1)} \\ = \quad & P(S \overset{*}{\Rightarrow} w_1 ... w_{i-1} A w_j ... w_n \overset{A \to \alpha}{\Rightarrow} w_1 ... w_{i-1} \alpha w_j ... w_n \overset{*}{\Rightarrow} w_1 ... w_n | S \overset{*}{\Rightarrow} w_1 ... w_n) \end{aligned}$$

This conditional probability is extremely useful. Traditionally, it has been used for estimating the probabilities of a PCFG from training sentences, using the *inside-outside algorithm*, shown in Figure 1.10. In this algorithm, we start with some initial probability estimate, $P_1(A \to \alpha)$. Then, for each sentence of training data, we determine the inside and outside probabilities to compute, for each production, how likely it is that that production was used as part of the derivation of that sentence. This gives us a number of counts for each production for each sentence. Summing these counts across sentences gives us an estimate of the total number of times each production was used to produce the sentences in the training corpus. Dividing by the total counts of productions used for each nonterminal A gives us a new estimate of the probability of the production. It is an important theorem that this new estimate will assign a higher probability to the training data than the old estimate, and that when this algorithm is run repeatedly, the rule probabilities converge towards locally optimum values (Baker, 1979), in terms of maximizing the probability of the training data.

1.3 Overview

While the traditional use for the inside-outside probabilities is to estimate the parameters of a PCFG, our goal is different: our goal is to demonstrate that the inside-outside probabilities are useful for solving many other problems in statistical parsing, and to provide useful tools for finding these values. In the remaining chapters of this thesis, we will first provide a general framework for specifying parsers that makes it easy to compute inside and outside probabilities. Next, we will show three novel uses: improved parser performance on specific criteria; faster parsing for the Data-Oriented Parsing (DOP) model; and faster parsing more generally by using thresholding. Finally, we describe a state-of-the-art parsing formalism that can compute inside and outside probabilities.

In Chapter 2, we develop a novel framework for specifying parsers that makes it easy to compute the inside and outside probabilities, and others. The CKY algorithm of Figure 1.4 is very similar to the inside algorithm of Figure 1.6, and the inside algorithm in some ways resembles the outside algorithm of Figure 1.8. For a simple parsing technique, such as CKY parsing, it is not too much work to derive each of these algorithms separately, ignoring their commonalities, but for more complicated algorithms and formalisms, the duplicated effort is significant. In particular, for sophisticated formalisms or techniques, the outside formula can be especially complicated to derive. Also, for parsing algorithms that can handle loops, like those that result from a grammar rule like $A \rightarrow A$, the inside and outside algorithms may become yet more complicated, because of the infinite summations that result. We develop a framework that allows a single parsing description to be used to compute recognition, inside, outside, and Viterbi values, among others. The framework uses a description language that is independent of the values being derived, and thus allows the complicated manipulations required to handle infinite summations to be separated out from the construction of the parsing algorithms. Using this framework, we show how to easily compute many interesting values, including the set of all parses of a grammar, the top n parses of a sentence, the most probable completion of a sentence, and many others. With this format, it will be simple to specify the thresholding and parsing algorithms of the following chapters, although we will also use traditional pseudocode as well, in an effort to keep the chapters self-contained.

In Chapter 3, we present our first novel use for the inside-outside probabilities, tailoring parsing algorithms to various metrics. Most probabilistic parsing algorithms are similar to the Viterbi algorithm. They attempt to maximize a single metric, the probability that the guessed parse tree is exactly correct. If the score that the parser receives is given by the number of exactly correct guessed trees, then this approach is correct. However, in practice, many other metrics are typically used, such as precision and recall, or crossing brackets. These metrics measure, in one way or another, how many pieces of the sentence are correct, rather than whether the whole sentence is exactly correct. Because the inside-outside product is proportional to the probability that a given constituent is correct, we can use it maximize correct pieces rather than the whole. We give various algorithms using the inside-outside probabilities for maximizing performance on these piecewise metrics. We also show that surprisingly, a similar problem, maximizing performance on the well known zero crossing brackets rate, is NP-Complete. Finally, we give an algorithm that, using the inside and outside probabilities, allows the tradeoff of precision versus recall. These algorithms can be easily specified in the format of Chapter 2.

Next, we show another use for the inside-outside probabilities: Data-Oriented Parsing. When we began this research, DOP was one of the most promising techniques available, with reported results an order of magnitude better than other parsers. However, the only algorithms available for parsing using the DOP model required exponentially large grammars. Furthermore, these algorithms were randomized algorithms with some chance of failure. We show how to use the inside-outside techniques of Chapter 3 to parse the DOP model in $O(n^3)$ time, deterministically, without sampling at all. We also show a grammar construction technique that is linear in sentence length, rather than exponential. Using these techniques, we are able to parse 500 times faster than previous algorithms. Our results are not as good as the previously published results, and we give an analysis of the data that shows that these previous results are probably due to a fortuitous split of the data into test and training sections, or to easy data.

The third novel use we give for the inside-outside probabilities is to speed parsing, which we discuss in Chapter 5. Parsers can be sped up using *thresholding*, a technique in which some low probability hypotheses are discarded, speeding later parsing. Since the normalized inside-outside probability gives the probability that any constituent is correct, it is the mathematically ideal probability to use to determine which hypotheses to discard. However, since the outside probability cannot be determined until after parsing is complete, we instead use three approximations to the inside-outside probabilities. These include a variation on beam search in which, rather than thresholding based only on the inside probability, we also include a very simple approximation to the outside probability; another thresholding technique that uses a more complicated approximation to the inside-outside probability which takes into account the whole sentence; and a multiple pass technique that uses the inside-outside probability from one pass to threshold later passes. All three of these algorithms lead to significantly improved speed. In order to maximize performance using all of these algorithms at once, it is necessary to maximize many parameters simultaneously. We give a novel algorithm for maximizing the parameters of multiple thresholding algorithms. Rather than directly maximizing accuracy, this algorithm maximizes the inside probability, which turns out to be much more efficient. Combining all of the thresholding algorithms together leads to about a factor of 30 speedup over traditional thresholding algorithms at the same error rates. All of the thresholding algorithms can be succinctly described using the item-based descriptions of Chapter 2.

Despite the usefulness of the inside and outside probabilities, as shown in the previous chapters, most state of the art parsing formalisms cannot be used to compute either the inside or the outside scores. In Chapter 6, we introduce a grammar formalism, Probabilistic Feature Grammar (PFG), that combines the best properties of most of the previous existing formalisms, but more elegantly. PFGs can be used to compute both inside and outside probabilities, meaning that they can be used with the previously introduced algorithms. PFGs achieve state of the art performance on parsing tasks.

This thesis shows that using the inside-outside probabilities is a powerful, general technique. We have simplified the process of computing inside and outside probabilities for new parsing algorithms. We have shown how to use these probabilities to improve performance by matching parsing algorithms to metrics; to quickly parse DOP grammars;

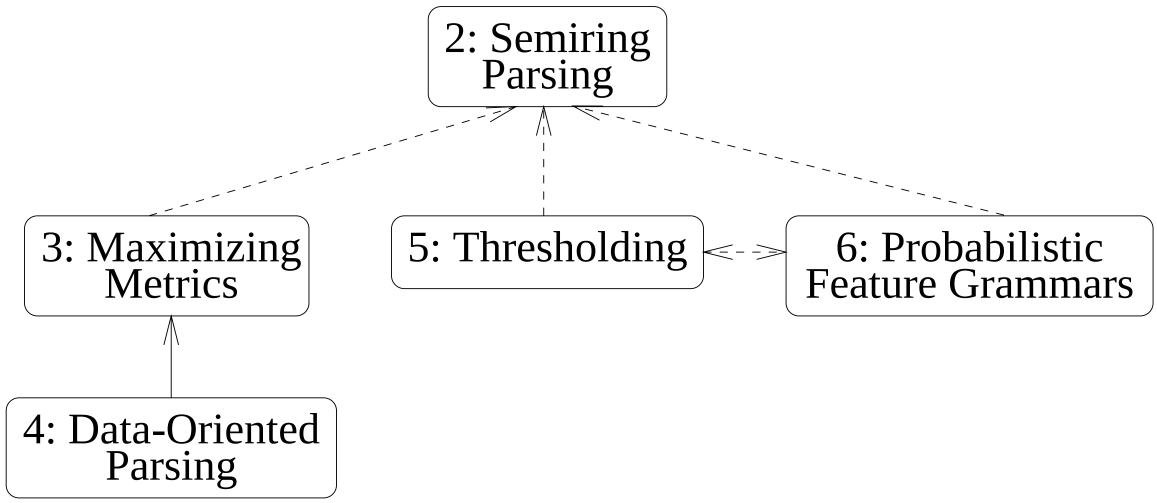


Figure 1.11: Dependencies in the thesis

and to quickly parse Probabilistic Context-Free Grammars (PCFGs) and PFGs with novel thresholding techniques. Finally, we have introduced a novel formalism, PFG, for which the inside and outside probabilities can be easily computed, and that achieves state of the art performance.

In writing this thesis, we have taken into account that it will be a rare person who wishes to read the thesis in its entirety. Thus, whenever it is reasonable to make a chapter self-contained, or to isolate interdependencies, we have sought to do so.

Figure 1.11 shows the organization and dependencies of the content chapters of the thesis. The chapters on maximizing metrics, thresholding, and probabilistic feature grammars all depend somewhat on the semiring parsing chapter, in that algorithms in these chapters are given using the format and theory of semiring parsing. However, to keep the chapters self-contained, all algorithms in these chapters are also presented in traditional pseudocode. The most important dependency is that of the Data-Oriented Parsing chapter, which uses algorithms and ideas from the chapter on maximizing metrics, and should probably not be read on its own. The thresholding and probabilistic feature grammar chapters each depend somewhat on the other, and can best be appreciated as a pair, although each can be read separately.

Chapter 2

Semiring Parsing

In this chapter, we present a system for describing parsers that allows a single simple representation to be used for describing parsers that compute inside and outside probabilities, as well as many other values, including recognition, derivation forests, and Viterbi values. This representation will be used throughout the thesis to describe the parsing algorithms we develop.

2.1 Introduction

For a given grammar and string, there are many interesting quantities we can compute. We can determine whether the string is generated by the grammar; we can enumerate all of the derivations of the string; if the grammar is probabilistic, we can compute the inside and outside probabilities of components of the string. Traditionally, a different parser description has been needed to compute each of these values. For some parsers, such as CKY parsers, all of these algorithms (except for the outside parser) strongly resemble each other. For other parsers, such as Earley parsers, the algorithms for computing each value are somewhat different, and a fair amount of work can be required to construct each one. We present a formalism for describing parsers such that a single simple description can be used to generate parsers that compute all of these quantities and others. This will be especially useful for finding parsers for outside values, and for parsers that can handle general grammars, like Earley-style parsers.

We will compare the CKY algorithm (Kasami, 1965; Younger, 1967) to the inside al-

```
boolean chart[1..n, 1..|N|, 1..n+1] := FALSE;
for each start s
    for each rule A → w_s
        chart[s, A, s+1] := TRUE;
for each length l, shortest to longest
    for each start s
        for each split length t
            for each rule A → BC ∈ R
                /* extra TRUE for expository purposes */
                chart[s, A, s+l] := chart[s, A, s+l] ∨
                        chart[s, B, s+t] ∧ chart[s+t, C, s+l] ∧ TRUE;

return chart[1, S, n+1];
```

Figure 2.1: CKY Recognition Algorithm

```
float chart[1..n, 1..|N|, 1..n+1] := 0;
for each start s
    for each rule A → w_s
        chart[s, A, s+1] := P(A → w_s);
for each length l, shortest to longest
    for each start s
        for each split length t
            for each rule A → BC ∈ R
                chart[s, A, s+l] := chart[s, A, s+l] +
                        chart[s, B, s+t] × chart[s+t, C, s+l] × P(A → BC);
return chart[1, S, n+1];
```

Figure 2.2: CKY Inside Algorithm

gorithm (Baker, 1979; Lari and Young, 1990) to illustrate the similarity of parsers for computing different values. Both of these parsers were described in in Section 1.2.1; we repeat the code here in Figures 2.1 and 2.2. Notice how similar the inside algorithm is to the recognition algorithm: essentially, all that has been done is to substitute $+$ for $\vee$, $\times$ for $\wedge$, and $P(A \rightarrow w_s)$ and $P(A \rightarrow BC)$ for *TRUE*. For many parsing algorithms, this, or a similarly simple modification, is all that is needed to create a probabilistic version of the algorithm. On the other hand, a simple substitution is not always sufficient. To give a trivial example, if in the CKY recognition algorithm we had written

$$\mathit{chart}[s, A, s{+}l] := \mathit{chart}[s, A, s{+}l] \;\vee\; \mathit{chart}[s, B, s{+}t] \wedge \mathit{chart}[s{+}t, C, s{+}l];$$

instead of the less natural

$$\mathit{chart}[s, A, s{+}l] := \mathit{chart}[s, A, s{+}l] \;\vee\; \mathit{chart}[s, B, s{+}t] \wedge \mathit{chart}[s{+}t, C, s{+}l] \wedge \mathit{TRUE};$$

larger changes would be necessary to create the inside algorithm.

Besides recognition, there are four other quantities that are commonly computed by parsing algorithms: derivation forests, Viterbi scores, number of parses, and outside probabilities. The first quantity, a derivation forest, is a data structure that allows one to efficiently compute the set of legal derivations of the input string. The derivation forest is typically found by modifying the recognition algorithm to keep track of "back pointers" for each cell of how it was produced. The second quantity often computed is the Viterbi score, the probability of the most probable derivation of the sentence. This can typically be computed by substituting $\times$ for $\wedge$ and max for $\vee$. Less commonly computed is the total number of parses of the sentence, which like the inside values, can be computed using multiplication and addition; unlike for the inside values, the probabilities of the rules are not multiplied into the scores. One last commonly computed quantity, the outside probability, cannot be found with modifications as simple as the others. We will discuss how to compute outside quantities later, in Section 2.4.

One of the key ideas of this chapter is that all five of these commonly computed quantities can be described as elements of *complete semirings* (Kuich, 1997). A complete semiring is a set of values over which a multiplicative operator and a commutative additive operator have been defined, and for which infinite summations are defined. For parsing algorithms satisfying certain conditions, the multiplicative and additive operations of any complete semiring can be used in place of $\wedge$ and $\vee$, and correct values will be returned. We will give a simple normal form for describing parsers, then precisely define complete semirings, the conditions for correctness, and a simple normal form for describing parsers.

We now describe our normal form for parsers, which is very similar to that used by Shieber *et al.* (1993) and by Sikkel (1993). In most parsers, there is at least one chart of some form. In our normal form, we will use a corresponding concept, *items*. Rather than, for instance, a chart element $chart[i, A, j]$, we will use an item $[i, A, j]$. Conceptually, chart elements and items are equivalent. Furthermore, rather than use explicit, procedural descriptions, such as

$$chart[s, A, s{+}l] := chart[s, A, s{+}l] \;\vee\; chart[s, B, s{+}t] \wedge chart[s{+}t, C, s{+}l] \wedge \mathit{TRUE}$$

we will use *inference rules* such as

$$\frac{R(A \rightarrow BC) \qquad [i, B, k] \qquad [k, C, j]}{[i, A, j]}$$

The meaning of an inference rule is that if the top line is all true, then we can conclude the bottom line. For instance, this example inference rule can be read as saying that if $A \rightarrow BC$ and $B \stackrel{*}{\Rightarrow} w_i ... w_{k-1}$ and $C \stackrel{*}{\Rightarrow} w_k ... w_{j-1}$, then $A \stackrel{*}{\Rightarrow} w_1 ... w_{j-1}$.

The general form for an inference rule will be

$$\frac{A_1 \cdots A_k}{B}$$

where if the conditions $A_1 ... A_k$ are all true, then we infer that B is also true. The A_i can be either items, or (in an extension to the usual convention for inference rules), can be rules, such as $R(A \rightarrow BC)$. We write $R(A \rightarrow BC)$ rather than $A \rightarrow BC$ to indicate that we could be interested in a value associated with the rule, such as the probability of the rule if we were computing inside probabilities. If an A_i is in the form $R(...)$, we call it a *rule*. All of the A_i must be rules or items; when we wish to refer to both rules and items, we use the word *terms*.

We now give an example of an item-based description, and its semantics. Figure 2.3

Item form:

$[i, A, j]$

Goal:

$[1, S, n{+}1]$

Rules:

$$\frac{R(A \to w_i)}{[i, A, i{+}1]} \qquad \text{Unary}$$

$$\frac{R(A \to BC) \qquad [i, B, k] \qquad [k, C, j]}{[i, A, j]} \qquad \text{Binary}$$

Figure 2.3: Item-based description of a CKY parser

gives a description of a CKY style parser. For this example, we will use the inside semiring, whose additive operator is addition and whose multiplicative operator is multiplication. We use the input string xxx to the following grammar:

$$\begin{array}{lcll} S & \to & XX & 1.0 \\ X & \to & XX & 0.2 \\ X & \to & x & 0.8 \end{array} \tag{2.1}$$

Our first step is to use the unary rule,

$$\frac{R(A \to w_i)}{[i, A, i{+}1]}$$

The effect of the unary rule will exactly parallel the first set of loops in the CKY inside algorithm. We will instantiate the free variables of the unary rule in every possible way. For instance, we instantiate the free variable i with the value 1, and the free variable A with the nonterminal X. Since $w_1 = x$, the instantiated rule is then

$$\frac{R(X \to x)}{[1, X, 2]}$$

Because the value of the top line of the instantiated unary rule, $R(X \to x)$, has value 0.8, we deduce that the bottom line, $[1, X, 2]$, has value 0.8. We instantiate the rule in two other ways, and compute the following chart values:

$$\begin{array}{rcl}[1, X, 2] & = & 0.8 \\ [2, X, 3] & = & 0.8 \\ [3, X, 4] & = & 0.8\end{array}$$

Next, we will use the binary rule,

$$\frac{R(A \to BC) \qquad [i, B, k] \qquad [k, C, j]}{[i, A, j]}$$

The effect of the binary rule will parallel the second set of loops for the CKY inside algorithm. Consider the instantiation $i = 1$, $k = 2$, $j = 3$, $A = X$, $B = X$, $C = X$,

$$\frac{R(X \to XX) \qquad [1, X, 2] \qquad [2, X, 3]}{[1, X, 3]}$$

We use the multiplicative operator of the semiring of interest to multiply together the values of the top line. In the inside semiring, the multiplicative operator is just multiplication, so we get: $0.2 \times 0.8 \times 0.8 = 0.128$, and deduce that $[1, X, 3]$ has value 0.128. We can do the same thing for the instantiation $i = 2$, $k = 3$, $j = 4$, $A = X$, $B = X$, $C = X$, getting the following item values:

$$\begin{array}{rcl}[1, X, 3] & = & 0.128 \\ [2, X, 4] & = & 0.128\end{array}$$

We can also deduce that

$$\begin{array}{rcl}[1, S, 3] & = & 0.128 \\ [2, S, 4] & = & 0.128\end{array}$$

There are two more ways to instantiate the conditions of the binary rule:

$$\frac{R(S \to XX) \qquad [1, X, 2] \qquad [2, X, 4]}{[1, S, 4]}$$

$$\frac{R(S \to XX) \qquad [1, X, 3] \qquad [3, X, 4]}{[1, S, 4]}$$

The first has the value $1 \times 0.8 \times 0.128 = 0.1024$, and the second also has the value 0.1024. When there is more than one way to derive a value for an item, we use the additive operator of the semiring to sum them up. In the inside semiring, the additive operator is just addition, so the value of this item is $[1, S, 4] = 0.2048$. Notice that the goal item for the CKY parser is $[1, S, 4]$. Thus we now know that the inside value for xxx is 0.2048. The goal item exactly parallels the return statement of the CKY inside algorithm.

Besides the fact that this item-based description is simpler than the explicit looping description, there will be other reasons we wish to use it. Unlike the other quantities we wish to compute, the outside probabilities cannot be computed by simply substituting a different semiring into either an iterative or item-based description. Instead, we will show how to compute the outside probabilities using a modified interpreter of the same item-based description used for computing the inside probabilities.

2.1.1 Earley Parsing

Many parsers are much more complicated than the CKY parser. This will make our description of semiring parsing a bit longer, but will also explain why our format is so useful: these complexities occur in many different parsers, and the ability of semiring parsing to handle them automatically will prove to be its main attraction.

Most of the interesting complexities we wish to discuss are exhibited by Earley parsing (Earley, 1970). Earley's parser is often described as a bottom-up parser with top-down filtering. In a probabilistic framework, the bottom-up and top-down aspects are very different; the bottom-up sections compute probabilities, while the top-down filtering non-probabilistically removes items that cannot be derived. In order to capture these differences, we expand our notation for deduction rules, to the following form:

$$\frac{A_1 \cdots A_k}{B} \, C_1 \cdots C_l$$

$C_1 \cdots C_l$ are *side conditions*, interpreted non-probabilistically, while $A_1 \cdots A_k$ are *main conditions* with values in whichever semiring we are using. While the values of all main conditions are multiplied together to yield the value for the item under the line, the side conditions are interpreted in a boolean manner: either they all have non-zero value or not. The rule can only be used if all of the side conditions have non-zero value, but other than that, their values are ignored.

Figure 2.4 gives an item-based description of Earley's parser. We assume the addition of a distinguished nonterminal S' with a single rule $S' \rightarrow S$. An item of the form $[i, A \rightarrow \alpha \bullet \beta, j]$ asserts that $A \Rightarrow \alpha\beta \stackrel{*}{\Rightarrow} w_i \ldots w_{j-1}\beta$.

Item form:

$$[i, A \to \alpha \bullet \beta, j]$$

Goal:

$$[1, S' \to S \bullet, n{+}1]$$

Rules:

$$\frac{}{[1, S' \to \bullet S, 1]} \qquad \text{Initialization}$$

$$\frac{[i, A \to \alpha \bullet w_j \beta, j]}{[i, A \to \alpha w_j \bullet \beta, j{+}1]} \qquad \text{Scanning}$$

$$\frac{R(B \to \gamma)}{[j, B \to \bullet \gamma, j]}\, [i, A \to \alpha \bullet B\beta, j] \qquad \text{Prediction}$$

$$\frac{[i, A \to \alpha \bullet B\beta, k] \qquad [k, B \to \gamma \bullet, j]}{[i, A \to \alpha B \bullet \beta, j]} \qquad \text{Completion}$$

Figure 2.4: Earley Parsing

The prediction rule includes a side condition, making it a good example. The rule is:

$$\frac{R(B \to \gamma)}{[j, B \to \bullet \gamma, j]}\, [i, A \to \alpha \bullet B\beta, j]$$

Through the prediction rule, Earley's algorithm guarantees that an item of the form $[j, B \to \bullet \gamma, j]$ can only be produced if $S \stackrel{*}{\Rightarrow} w_1 ... w_{j-1} B \delta$ for some δ; this top down filtering leads to significantly more efficient parsing for some grammars than the CKY algorithm. The prediction rule combines side and main conditions. The side condition

$$[i, A \to \alpha \bullet B\beta, j]$$

provides the top-down filtering, ensuring that only items that might be used later by the completion rule can be predicted, while the main condition,

$$R(B \to \gamma)$$

provides the probability of the relevant rule. The side condition is interpreted in a boolean fashion, while the main condition's actual probability is used.

Unlike the CKY algorithm, Earley's algorithm can handle grammars with epsilon (ϵ), unary, and n-ary branching rules. In some cases, this can significantly complicate parsing. For instance, given unary rules $A \rightarrow B$ and $B \rightarrow A$, a cycle exists. This kind of cycle may allow an infinite number of different derivations, requiring an infinite summation to compute the inside probabilities. The ability of item-based parsers to handle these infinite loops with ease is a major attraction.

2.1.2 Overview

This chapter will simplify the development of new parsers in several ways. First, it will simplify specification of parsers: the item-based description is simpler than a procedural description. Second, it will make it easier to generalize parsers to other tasks: a single item-based description can be used to compute values in a variety of semirings, and outside values as well. This will be especially advantageous for parsers that can handle loops resulting from rules like $A \rightarrow A$ and computations resulting from ϵ productions, both of which typically lead to infinite sums. In these cases, the procedure for computing an infinite sum differs from semiring to semiring, and the fact that we can specify that a parser computes an infinite sum separately from its method of computing that sum will be very helpful.

In the next section, we describe the basics of semiring parsing. In the following sections, we derive formulae for computing the values of items in semiring parsers, and then describe an algorithm for interpreting an item-based description. Next, we discuss using this same formalism for performing grammar transformations. At the end of the chapter, we give examples of using semiring parsers to solve a variety of problems.

2.2 Semiring Parsing

In this section we first describe the inputs to a semiring parser: a semiring, an item-based description, and a grammar. Next, we give the conditions under which a semiring parser gives correct results. At the end of this section we discuss three especially complicated and interesting semirings.

2.2.1 Semiring

In this subsection, we define and discuss semirings. The best introduction to semirings that we know of, and the one we follow here, is that of Kuich (1997), who also gives more formal definitions than those given in this chapter.

A semiring has two operations, $\oplus$ and $\otimes$, that intuitively have most (but not necessarily all) of the properties of the conventional $+$ and $\times$ operations on the positive integers. In particular, we require the following properties: $\oplus$ is associative and commutative; $\otimes$ is associative and distributes over $\oplus$. If $\otimes$ is commutative, we will say that the semiring is commutative. We assume an additive identity element, which we write as 0, and a multiplicative identity element, which we write as 1. Both addition and multiplication can be defined over finite sets of elements; if the set is empty, then the value is the respective identity element, 0 or 1. We also assume that $x \otimes 0 = 0 \otimes x = 0$ for all x. In other words, a semiring is just like a ring, except that the additive operator need not have an inverse. We will write

$$\langle \mathbb{A}, \oplus, \otimes, 0, 1 \rangle$$

to indicate a semiring over the set $\mathbb{A}$ with additive operator $\oplus$, multiplicative operator $\otimes$, additive identity 0, and multiplicative identity 1.

For parsers with loops, i.e. those in which an item can be used to derive itself, we will also require that sums of an infinite number of elements be well defined. In particular, we will require that the semirings be *complete* (Kuich, 1997, p. 611). This means that sums of an infinite number of elements should be associative, commutative, and distributive just like finite sums. All of the semirings we will deal with in this chapter are complete. Completeness is a somewhat stronger condition than we really need; we could, instead, require that limits be appropriately defined for those infinite sums that occur while parsing, but this weaker condition is more complicated to describe precisely.

Certain semirings are *naturally ordered*, meaning that we can define a partial ordering, $\sqsubseteq$, such that $x \sqsubseteq y$ if and only if there exists z such that $x + z = y$. We will call a naturally ordered complete semiring ω-continuous (Kuich, 1997, p. 612) if for any sequence $x_1, x_2, ...$ and for any constant y, if for all n, $\bigoplus_{0 \leq i \leq n} x_i \sqsubseteq y$, then $\bigoplus_i x_i \sqsubseteq y$. That is, if every partial sum is less than or equal to y, then the infinite sum is also less than or equal to y. This important property makes it easy to compute, or at least approximate, infinite sums. All

boolean	$\langle\{TRUE, FALSE\}, \vee, \wedge, FALSE, TRUE\rangle$
inside	$\langle\mathbb{R}_0^\infty, +, \times, 0, 1\rangle$
Viterbi	$\langle\mathbb{R}_0^1, \max, \times, 0, 1\rangle$
counting	$\langle\mathbb{N}_0^\infty, +, \times, 0, 1\rangle$
tropical semiring	$\langle\mathbb{R}_0^\infty, \min, +, \infty, 0\rangle$
arctic semiring	$\langle\mathbb{R}\cup\{-\infty\}, \max, +, -\infty, 0\rangle$
derivation forest	$\langle 2^{\mathbb{E}}, \cup, \cdot, \emptyset, \{\langle\rangle\}\rangle$
Viterbi-derivation	$\langle\mathbb{R}_0^1 \times 2^{\mathbb{E}}, \max_{Vit}, \times_{Vit}, \langle 0, \emptyset\rangle, \langle 1, \{\langle\rangle\}\rangle\rangle$
Viterbi-n-best	$\langle\{topn(X) \mid X \in 2^{\mathbb{R}_0^1\times\mathbb{E}}\}, \max_{Vit\text{-}n}, \times_{Vit\text{-}n}, \emptyset, \{\langle 1, \{\langle\rangle\}\rangle\}\rangle$

Figure 2.5: Semirings Used: $\langle A, \oplus, \otimes, 0, 1\rangle$

of the semirings we discuss here will be both naturally ordered and ω-continuous.

There will be several especially useful semirings in this chapter, which are defined in Figure 2.5. We will write $\mathbb{R}_a^b$ to indicate the set of real numbers from a to b inclusive, with similar notation for the natural numbers, $\mathbb{N}$. We will write $\mathbb{E}$ to indicate the set of all derivations, where a derivation is an ordered list of grammar rules. We will write $2^{\mathbb{E}}$ to indicate the set of all sets of derivations. There are three derivation semirings: the derivation forest semiring, the Viterbi-derivation semiring, and the Viterbi-n-best semiring. The operators used in the derivation semirings ($\cdot$, $\max_{Vit}$, $\times_{Vit}$, $\max_{Vit\text{-}n}$, and $\times_{Vit\text{-}n}$) will be described later, in Section 2.2.5.

The inside semiring includes all non-negative real numbers, to be closed under addition, and includes infinity to be closed under infinite sums, while the Viterbi semiring contains only numbers up to 1, since that is all that is required to be closed under max.

There are two additional semirings, the tropical semiring (which usually is restricted to natural numbers, but is extended to real numbers here), and another semiring, which we have named the arctic semiring, since it is the opposite of the tropical semiring, taking maxima rather than minima.

The three derivation forest semirings can be used to find especially important values: the derivation forest semiring computes all derivations of a sentence; the Viterbi-derivation semiring computes the most probable derivation; and the Viterbi-n-best semiring computes the n best derivations. A derivation is simply a list of rules from the grammar. From a derivation, a parse tree can be derived, so the derivation forest semiring is analogous to conventional parse forests. Unlike the other semirings, all three of these semirings are non-commutative. The additive operation of these semirings is essentially union or maximum, while the multiplicative operation is essentially concatenation. These semirings are relatively complicated, and are described in more detail in Section 2.2.5.

2.2.2 Item-Based Description

A semiring parser requires an item-based description, $\mathcal{D}$, of the parsing algorithm, in the form given earlier. So far, we have skipped one important detail of semiring parsing. In a simple recognition system, as used in deduction systems, all that matters is whether an item can be deduced or not. Thus, in these simple systems, the order of processing items is relatively unimportant, as long as some simple constraints are met. On the other hand, for a semiring such as the inside semiring, there are important ordering constraints: for instance, we cannot compute the inside value of a CKY-style chart element until the inside values of all of its children have been computed.

Thus, we need to impose an ordering on the items, in such a way that no item precedes any item on which it depends. We will associate with each item x a "bucket" B and write $bucket(x) = B$. We order the buckets in such a way that if item y depends on item x, then $bucket(x) \leq bucket(y)$. We will write *first*, *last*, $next(B)$, and $previous(B)$ for the first, last, next and previous buckets respectively. For some pairs of items, it may be that both depend, directly or indirectly, on each other; we associate these item with special "looping" buckets, whose values may require infinite sums to compute. We will also call a bucket looping if an item in it depends on itself. The predicate $loop(B)$ will be true for looping buckets B.

One way to achieve a bucketing with the required ordering constraints is to create a graph of the dependencies, with a node for each item, and an edge from each item x to each item b that depends on it. We then separate the graph into its strongly connected components, and perform a topological sort. Items forming singleton strongly connected components are in their own buckets; items forming non-singleton strongly connected components are together in looping buckets.

An actual example may help here. Consider an example grammar, such as

$$\begin{array}{rcl} S & \rightarrow & CAC \\ C & \rightarrow & c \\ B & \rightarrow & A \\ A & \rightarrow & B \\ A & \rightarrow & a \end{array}$$

and an input sentence *cac*, parsed with the Earley parser of Figure 2.4. It will be possible to derive items such as $[1, C \rightarrow \bullet c, 1]$ through prediction, $[1, C \rightarrow c \bullet, 2]$ through scanning, and $[1, S \rightarrow C \bullet AC, 2]$ through completion. Each of these items would form a singleton strongly connected component, and could be put into its own bucket. Now, using completion, we see that

$$\frac{[2, B \rightarrow \ \bullet A, 2] \qquad [2, A \rightarrow a \bullet, 3]}{[2, B \rightarrow A \bullet, 3]}$$

Next comes the looping part. Notice that

$$\frac{[2, A \rightarrow \ \bullet B, 2] \qquad [2, B \rightarrow A \bullet, 3]}{[2, A \rightarrow B \bullet, 3]}$$

and

$$\frac{[2, B \rightarrow \ \bullet A, 2] \qquad [2, A \rightarrow B \bullet, 3]}{[2, B \rightarrow A \bullet, 3]}$$

Thus, the items $[2, B \rightarrow A \bullet, 3]$ and $[2, A \rightarrow B \bullet, 3]$ can each be derived from the other, and since they depend on each other, will be placed together in a looping bucket.

A topological sort is not the only way to bucket the items. In particular, for items such that neither depends on the other, it is possible to place them into a bucket together with no loss of efficiency. For some descriptions, this could be used to produce faster, simpler parsing algorithms. For instance, in a CKY style parser, we could simply place all items of the same length in the same bucket, ordering buckets from shortest to longest, avoiding the need to perform a topological sort.

Later, when we discuss algorithms for interpreting an item-based description, we will need another concept. Of all the items associated with a bucket B, we will be able to find derivations for only a subset. If we can derive an item x associated with bucket B, we write $x \in B$, and say that item x is in bucket B. For example, the goal item of a parser will almost always be *associated* with the last bucket; if the sentence is grammatical, the goal item will be *in* the last bucket, and if it is not grammatical, it won't be.

It will be useful to assume that there is a single, variable free goal item, and that this goal item does not occur as a condition for any rules. We can always add a new goal item $[goal]$ and a rule $\dfrac{[old\text{-}goal]}{[goal]}$ where $[old\text{-}goal]$ is the goal in the original description. We will assume in general that this transformation has been made, or is not necessary.

2.2.3 The Grammar

A semiring parser also requires a grammar as input. We will need a list of rules in the grammar, and a function that gives the value for each rule in the grammar. This latter function will be semiring specific. For instance, for computing the inside and Viterbi probabilities, the value of a grammar rule is just the conditional probability of that rule, or 0 if it is not in the grammar. For the boolean semiring, the value is *TRUE* if the rule is in the grammar, *FALSE* otherwise. For the counting semiring, the value is 1 if the rule is in the grammar, 0 otherwise. We call this function $R(rule)$. This function replaces the set of rules R of a conventional grammar description; a *rule* is in the grammar if $R(rule)$ is not the zero element of the semiring.

2.2.4 Conditions for Correct Processing

We will say that a semiring parser works correctly if for any grammar, input and semiring, the value of the input according to the grammar equals the value of the input using the parser. In this subsection, we will define the value of an input according to the grammar; the value of an input using the parser; and give a sufficient condition for a semiring parser to work correctly.

From this point onwards, unless we specifically mention otherwise, we will assume that some fixed semiring, item-based description, and grammar have been given, without specifically mentioning which ones.

Value according to grammar

Under certain conditions, a semiring parser will work correctly for any grammar, input, and semiring. First, we must define what we mean by working correctly. Essentially, we mean that the value of a sentence according to the grammar equals the value of the sentence using the parser.

Consider a derivation E, consisting of grammar rules $e_1, e_2, ..., e_l$. We define the value of the derivation to be simply the product (in the semiring) of the values of the rules used in E:

$$V_G(E) = \bigotimes_{i=1}^{l} R(e_i)$$

Then we can define the value of a sentence that can be derived using grammar derivations $E^1, E^2, ..., E^k$ to be:

$$V_G = \bigoplus_{j=1}^{k} V_G(E^j)$$

where k is potentially infinite. In other words, the value of the sentence according to the grammar is the sum of the values of all derivations. We will assume that in each grammar formalism there is some way to define derivations uniquely; for instance, in CFGs, one way would be using left-most derivations. For simplicity, we will simply refer to derivations, rather than e.g. left-most derivations, since we are never interested in non-unique derivations.

Example of value according to grammar

A short example will help clarify. We consider the following grammar:

$$\begin{array}{lcll} S & \rightarrow & AA & R(S \rightarrow AA) \\ A & \rightarrow & AA & R(A \rightarrow AA) \\ A & \rightarrow & a & R(A \rightarrow a) \end{array} \tag{2.2}$$

and the input string aaa. There are two grammar derivations, the first of which is

$$S \overset{S \rightarrow AA}{\Rightarrow} AA \overset{A \rightarrow AA}{\Rightarrow} AAA \overset{A \rightarrow a}{\Rightarrow} aAA \overset{A \rightarrow a}{\Rightarrow} aaA \overset{A \rightarrow a}{\Rightarrow} aaa$$

which has value

$$R(S \rightarrow AA) \otimes R(A \rightarrow AA) \otimes R(A \rightarrow a) \otimes R(A \rightarrow a) \otimes R(A \rightarrow a)$$

Notice that the rules in the value are the same rules in the same order as in the derivation. The other grammar derivation is

$$S \overset{S \to AA}{\Rightarrow} AA \overset{A \to a}{\Rightarrow} aA \overset{A \to AA}{\Rightarrow} aAA \overset{A \to a}{\Rightarrow} aaA \overset{A \to a}{\Rightarrow} aaa$$

which has value

$$R(S \to AA) \otimes R(A \to a) \otimes R(A \to AA) \otimes R(A \to a) \otimes R(A \to a)$$

We note that for commutative semirings, the value of the two grammar derivations are equal, but for non-commutative semirings, they differ.

The value of the sentence is the sum of the values of the two derivations,

$$\begin{array}{l} R(S \to AA) \otimes R(A \to AA) \otimes R(A \to a) \otimes R(A \to a) \otimes R(A \to a) \\ \oplus \\ R(S \to AA) \otimes R(A \to a) \otimes R(A \to AA) \otimes R(A \to a) \otimes R(A \to a) \end{array}$$

Item derivations

Next, we must define item derivations, i.e. derivations using the item-based description of the parser. We will define item derivation in such a way that for a correct parser description, there will be exactly one item derivation for each grammar derivation. The value of a sentence using the parser is the sum of the value of all item derivations of the goal item.

We say that $\frac{a_1 ... a_k}{b}\ c_1 ... c_l$ is an *instantiation* of deduction rule $\frac{A_1 ... A_k}{B}\ C_1 ... C_l$ whenever the first expression is a variable-free instance of the second; that is, the first expression is the result of consistently substituting constant terms for each variable in the second. Now, we can define an *item derivation tree*. Intuitively, an item derivation tree for x just gives a way of deducing x from *ground* items (items that don't depend on other items, i.e. items that can be deduced using rules that have no items in the A_i.) We define an item derivation tree recursively. The base case is rules of the grammar: $\langle r \rangle$ is an item derivation tree, where r is a rule of the grammar. Also, if $D_{a_1}, ..., D_{a_k}, D_{c_1}, ..., D_{c_l}$ are derivation trees headed by $a_1 ... a_k, c_1 ... c_l$ respectively, and if $\frac{a_1 ... a_k}{b}\ c_1 ... c_l$ is the instantiation of a deduction rule, then $\langle b : D_{a_1}, ..., D_{a_k} \rangle$ is also a derivation tree. Notice that the $D_{c_1} ... D_{c_l}$ do not occur in this tree: they are side conditions, and although their existence is required to prove that $c_1 ... c_l$ could be derived, they do not contribute to the value of the tree. We will write

$\frac{a_1 ... a_k}{b}$ to indicate that there is an item derivation tree of the form $\langle b : D_{a_1}, ..., D_{a_k} \rangle$. As mentioned in Section 2.2.2, we will write $x \in B$ if $bucket(x) = B$ and there is an item derivation tree for x.

Example of item derivation

We can continue the example of parsing aaa, now using the item based CKY parser of Figure 2.3. There are two item derivation trees for the goal item; we give the first as an example, displaying it as a tree, rather than with angle bracket notation, for simplicity. Figure 2.6 shows this tree and the corresponding grammar derivation.

Notice that an item derivation is a tree, not a directed graph. Thus, an item subderivation could occur multiple times in a given item derivation. This means that we can have a one-to-one correspondence between item derivations and grammar derivations; loops in the grammar lead to an infinite number of grammar derivations, and an infinite number of corresponding item derivations.

A grammar including rules such as

$$\begin{array}{rcl} S & \rightarrow & AAA \\ A & \rightarrow & B \\ A & \rightarrow & a \\ B & \rightarrow & A \\ B & \rightarrow & \epsilon \end{array}$$

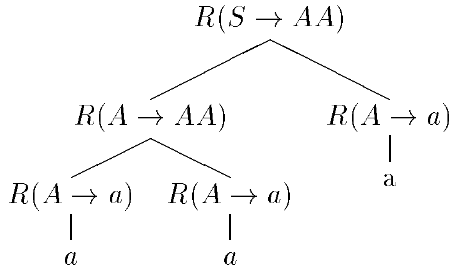
would allow derivations such as $S \Rightarrow AAA \Rightarrow BAA \Rightarrow AA \Rightarrow BA \Rightarrow A \Rightarrow B \Rightarrow \epsilon$. Depending on the parser, we might include the exact same item derivation showing $A \Rightarrow B \Rightarrow \epsilon$ three times. Similarly, for a derivation such as $A \Rightarrow B \Rightarrow A \Rightarrow B \Rightarrow A \Rightarrow a$, we would have a corresponding item derivation tree that included multiple uses of the $A \rightarrow B$ and $B \rightarrow A$ rules.

Value of item derivation

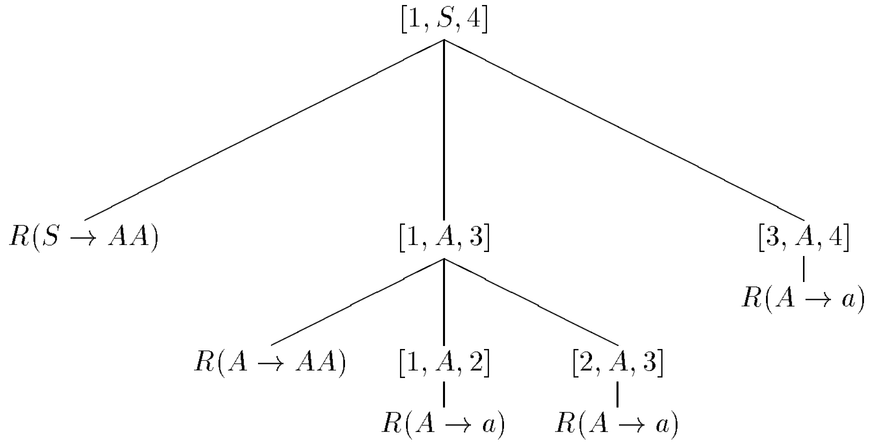
The value of an item derivation D, $V(D)$, is the product of the value of its rules, $R(r)$, in the same order that they appear in the item derivation tree. Since rules occur only in the leaves of item derivation trees, there is no ambiguity as to order in this definition. For an

$$S \overset{S \to AA}{\Rightarrow} AA \overset{A \to AA}{\Rightarrow} AAA \overset{A \to a}{\Rightarrow} aAA \overset{A \to a}{\Rightarrow} aaA \overset{A \to a}{\Rightarrow} aaa$$

Grammar Derivation



Grammar Derivation Tree



Item Derivation Tree

$$R(S \to AA) \otimes R(A \to AA) \otimes R(A \to a) \otimes R(A \to a) \otimes R(A \to a)$$

Derivation Value

Figure 2.6: Grammar derivation tree; item derivation tree; value

item derivation tree D with rule values $d_1, d_2, ..., d_d$ as its leaves,

$$V(D) = \bigotimes_{i=1}^{d} R(d_i) \tag{2.3}$$

Alternatively, we can write this equation recursively as

$$V(D) = \begin{cases} R(D) \text{ } \mathit{if} \text{ } D \text{ is a rule} \\ \bigotimes_{i=1}^{k} V(D_i) \text{ } \mathit{if} \text{ } D = \langle b : D_1, ..., D_k \rangle \end{cases} \tag{2.4}$$

Continuing our example, the value of the item derivation tree of Figure 2.6 is

$$R(S \rightarrow AA) \otimes R(A \rightarrow a) \otimes R(A \rightarrow AA) \otimes R(A \rightarrow a) \otimes R(A \rightarrow a)$$

the same as the value of the first grammar derivation.

Notice that Equation 2.4 is just a recursive expression for the product of the rule values appearing in the leaves of the tree. Thus,

Let $\mathit{inner}(x)$ represent the set of all item derivation trees headed by an item x. Then the value of x is the sum of all the values of all item derivation trees headed by x. Formally,

$$V(x) = \bigoplus_{D \in \mathit{inner}(x)} V(D)$$

The value of a sentence is just the value of the goal item, $V(\mathit{goal})$.

Iso-valued derivations

In certain cases, a particular grammar derivation and a particular item-derivation will have the same value for any semiring and any rule value function R. In particular, if the same rules occur in the same order in both the grammar derivation and the item derivation, then their values will be the same no matter what. If a grammar derivation and an item derivation meet this condition, then we define a new term to describe them, *iso-valued*. In Figure 2.6, the grammar derivation and item derivation both have the rules $R(S \rightarrow AA), R(A \rightarrow AA), R(A \rightarrow a), R(A \rightarrow a), R(A \rightarrow a)$, and so they are iso-valued.

In some cases, a grammar derivation and an item-derivation will have the same value for any commutative semiring and any rule value function. If the same rules occur the same number of times in both the grammar derivation and the item derivation, then they will have the same value in any commutative semiring. We say that a grammar derivation and an item derivation meeting this condition are *commutatively iso-valued.*

Iso-valued and commutatively iso-valued derivations will be important when we discuss conditions for correctness.

Example value of item derivation

Finishing our example, the value of the goal item given our example sentence is just the sum of the values of the two item-based derivations,

$$\begin{array}{l} R(S \to AA) \otimes R(A \to AA) \otimes R(A \to a) \otimes R(A \to a) \otimes R(A \to a) \\ \oplus \\ R(S \to AA) \otimes R(A \to a) \otimes R(A \to AA) \otimes R(A \to a) \otimes R(A \to a) \end{array}$$

This value is the same as the value of the sentence according to the grammar.

Conditions for correctness

We can now specify the conditions for an item-based description to be correct.

Theorem 2.1
Given an item-based description $\mathcal{D}$, if for every grammar G, there exists a one-to-one correspondence between the item derivations using $\mathcal{D}$ and the grammar derivations, and the corresponding derivations are iso-valued, then for every complete semiring, the value of a given input $w_1...w_n$ is the same according to the grammar as the value of the goal item. If the semiring is commutative, then the corresponding derivations need only be commutatively iso-valued.

Proof The proof is very simple; essentially, each term in each sum occurs in the other. We separate the proof into two cases. First is the non-commutative case. In this case, by hypothesis, for a given input there are grammar derivations $E_1...E_k$ (for $0 \leq k \leq \infty$) and corresponding iso-valued item derivation trees $D_1...D_k$ of the goal item. Since corresponding items are iso-valued, for all i, $V(E_i) = V(D_i)$. Now, since the value of the string according to the grammar is just $\sum_i V(E_i) = \sum_i V(D_i)$, and the value of the goal item is $\sum_i V(D_i)$, the value of the string according to the grammar equals the value of the goal item.

The second case, the commutative case, follows analogously. $\diamond$

There is one additional condition for an item-based description to be usable in practice, which is that there be only a finite number of derivable items for a given input sentence; there may, however, be an infinite number of derivations of any item.

2.2.5 The derivation semirings

All of the semirings we use should be familiar, except for the derivation semirings, which we now describe. These semirings, unlike the other semirings described in Figure 2.5, are not commutative under their multiplicative operator, concatenation.

In many parsers, it is conventional to compute parse forests: compact representations of the set of trees consistent with the input. We will use a related concept, derivation forests, a compact representation of the set of derivations consistent with the input, which corresponds to the parse forest for CNF grammars, but is easily extended to other formalisms. Although the terminology we use is different, the representation of derivation forests is similar to that used by Billot and Lang (1989).

Often, we will not be interested in the set of all derivations, but only in the most probable derivation. The Viterbi-derivation semiring computes this value. Alternatively, we might want the n best derivations, which would be useful if the output of the parser were passed to another stage, such as semantic disambiguation; this value is computed by the Viterbi-n-best derivation semiring.

Notice that each of the derivation semirings can also be used to create transducers. That is, we simply associate strings rather than grammar rules with each rule value. Instead of grammar rule concatenation, we perform string concatenation. The derivation semiring then corresponds to nondeterministic transductions; the Viterbi semiring corresponds to a weighted or probabilistic transducer; and the inside semiring could be used to, for instance, perform re-estimation of probabilistic transducers.

Derivation Forest

The derivation forest semiring consists of sets of derivations, where a derivation is a list of rules of the grammar. In the CFG case, these rules would form, for instance, a left-most derivation. The additive operator $\cup$ produces a union of derivations, and the multiplicative operator $\cdot$ produces the concatenation, one derivation concatenated with the next. The concatenation operation $(\cdot)$ is defined on both derivations and sets of derivations; when applied to a set of derivations, it produces the set of pairwise concatenations. The simplest derivations are simply rules of the grammar, such as $X \to YZ$ for a CFG. Sets containing one rule, such as $\{\langle X \to YZ\rangle\}$ constitute the primitive elements of the semiring.[1]

A few examples may help. The additive identity, the zero element, is simply the empty set, $\emptyset$: union with the empty set is an identity operation. The multiplicative identity is the set containing the empty derivation, $\{\langle\rangle\}$: concatenation with the empty derivation is an identity operation. Derivations need not be complete. For instance, assuming we are using left-most derivations with CFGs, $\{\langle X \to YZ,\ Y \to y\rangle\}$ is a valid element, as is $\{\langle Y \to y,\ X \to x\rangle\}$. In fact, $\{\langle X \to A,\ B \to b\rangle\}$ is a valid element of the semiring, even though it could not occur in a valid grammar derivation; this value should never occur in a correctly functioning parser.

The obvious implementation of derivation forests, as actual sets of derivations, would be extremely inefficient. In the worst case, when we allow infinite unions, a case we will wish to consider, the obvious implementation does not work at all. However, in Section 2.3.2, we will show how to use pointers to efficiently implement infinite unions of derivation forests, in a manner analogous to the traditional implementation of parse forests.

We can now describe a simple, efficient implementation of the derivation forest semiring. We will assume that four operations are desired: concatenation, union, primitive creation, and extraction. Primitive creation is used to create the basic elements of the semiring, sets containing a single rule. Extraction non-deterministically extracts a single derivation from the derivation forest. Code for these four functions is given in Figure 2.7. All of the code is straightforward. We assume a function *choose*, needed to handle unions in the extract function, that nondeterministically chooses between its inputs.

A straightforward implementation of this algorithm would work fine, but slight variations are required for good efficiency. The problem comes from the concatenation operation. Typically, in LISP-like languages, concatenation is implemented as a copy operation. If we were to build up a derivation of length n one rule at a time, then the run time would be $O(n^2)$, since we would first copy one element, then two, then three, etc., resulting in $O(n^2)$

[1] The derivation forest semiring is equivalent to a semiring well known to mathematicians, the polynomials over non-commuting variables. Such a polynomial is a sum of terms, each of which is an ordered product of variables. If these variables correspond to the basic elements of this semiring, then each term in the polynomial corresponds to a derivation.

Function *Concatenate*$(f,\ g)$
    return $\langle$"Concatenate" $f\ g\rangle$;

Function *Union*$(f,\ g)$
    return $\langle$"Union" $f\ g\rangle$;

Function *Create*$(rule)$
    return $\langle$"Create" $rule\rangle$;

Function *Extract*(f)
    switch f_1:
        "Concatenate":
            return *Concat!*$(Extract(f_2),\ Extract(f_3))$;
        "Union":
            return *choose*$(Extract(f_2),\ Extract(f_3))$;
        "Create":
            return $\langle f_2\rangle)$;

Figure 2.7: Derivation Forest Implementation

rules being copied. To get good efficiency, we need to implement concatenation destructively; we assume that lists, indicated with angle brackets, are implemented with linked lists, with a pointer to the last element. (For those skilled in Prolog, this implementation of linked lists is essentially equivalent to difference lists.) List creation can be performed efficiently using non-destructive operations. Destructive concatenation can operate in constant time. Given this destructive operation, any interpreter of this nondeterministic algorithm would need to keep a list of destructive changes to undo during backtracking, as is done in many implementations of unification grammars, or of unification in Prolog.

This modified implementation is efficient, in the following senses. First, concatenation and union are constant time operations. Second, if we were to use *Extract* with a nondeterministic interpreter to generate all derivations in the derivation forest, the time used would be at worst proportional to the total size of all trees generated.

Billot and Lang (1989) show how to create grammars to represent parse forests. Readers familiar with their work will recognize the similarity between their representation and ours. Where we write

$$e := Concatenate(f, g)$$

Billot *et al.* write

$$e \to fg$$

Where we write

$$e := Union(f, g)$$

they write

$$\begin{aligned} e &\to f \\ e &\to g \end{aligned}$$

Viterbi-derivation Semiring

The Viterbi-derivation semiring computes the most probable derivation of the sentence, given a probabilistic grammar. Elements of this semiring are a pair of a real number v and a derivation forest E, i.e. the set of derivations with score v. We define $\max\limits_{Vit}$, the additive operator, as

$$\max_{Vit}(\langle v, E\rangle, \langle w, D\rangle) = \begin{cases} \langle v, E\rangle & \textit{if } v > w \\ \langle w, D\rangle & \textit{if } v < w \\ \langle v, E \cup D\rangle & \textit{if } v = w \end{cases}$$

In typical practical Viterbi parsers, when two derivations have the same value, one of the derivations is arbitrarily chosen. In practice, this is usually a fine solution, and one that could be used in a real-world implementation of the ideas in this chapter, but from a theoretical viewpoint, the arbitrary choice destroys the associative property of the additive operator, $\max\limits_{Vit}$. To preserve associativity, we keep derivation forests of all elements that tie for best. An alternate technique for preserving associativity would be to choose between derivations using some ordering, but the derivation forest solution simplifies the discussion in Section 2–C.1

The definition for $\max\limits_{Vit\text{-}n}$ is only defined for two elements. Since the operator is associative, it is clear how to define $\max\limits_{Vit\text{-}n}$ for any finite number of elements, but we also need infinite summations to be defined. We require an operator, sup, the *supremum*, for this definition. The supremum of a set is the smallest value at least as large as all elements of the set; that is, it is a maximum that is defined in the infinite case.

We can now define $\max\limits_{Vit\text{-}n}$ for the case of infinite sums. First, let

$$w = \sup_{\langle v, E\rangle \in X} v$$

Then, let

$$D = \{E | \langle w, E\rangle \in X\}$$

Then $\max_{Vit} X = \langle w, D\rangle$. In the finite case, this is equivalent to our original definition. In the infinite case, D is potentially empty, but this causes us no problems in theory, and infinite sums with an empty D will not appear in practice.

We define the multiplicative operator, $\underset{Vit}{\times}$, as

$$\langle v, E\rangle \times \langle w, D\rangle = \langle v \times w, E \cdot D\rangle$$

where $E \cdot D$ represents the concatenation of the two derivation forests.

Viterbi-n-best semiring

The last kind of derivation semiring is the Viterbi-n-best semiring, which is used for constructing n-best lists. Intuitively, the value of a string using this semiring will be the n most likely derivations of that string (unless there are fewer than n total derivations.) Furthermore, in a practical implementation, this is actually how a Viterbi-n-best semiring would typically be implemented. From a theoretical viewpoint, however, this implementation is inadequate, since we must also define infinite sums and be sure that the distributive property holds. Thus, we introduce two complications. First, when not only are there more than n total derivations, but there is a tie for the n'th most likely, there will be more than n entries, which we can represent efficiently with a derivation forest. Second, in order to make infinite sums well defined, it will be useful to have an additional value, ∞, counted as a legal derivation. The value ∞ will arise due to infinite sums of elements approaching a supremum. Thus, we will want to consider ∞ to represent an infinite number of values approaching the derivation value.

The best way to define the Viterbi-n-best semiring is as a homomorphism from a simpler semiring, the Viterbi-all semiring. The Viterbi-all semiring keeps all derivations and their values. The additive operator is set union, and the multiplicative operator is $\star$, defined as

$$X \star Y = \{\langle vw, d \cdot e\rangle | \langle v, d\rangle \in X \wedge \langle w, e\rangle \in Y\}$$

Then, the Viterbi-all semiring is

$$\langle 2^{\mathbb{R}_0^1 \times \mathbb{E}}, \cup, \star, \emptyset, \{\langle 1, \langle\rangle\rangle\}\rangle$$

Now, we can define a helper function we will need for the homomorphism to the Viterbi-n-best semiring. We define *simpletopn*, which returns the n highest valued elements. Ties for last are kept; this property will make the additive operator commutative and associative.

$$\mathit{simpletopn}(X) = \{\langle v, d\rangle \in X | \text{there are at most } n-1 \text{ items } \langle w, e\rangle \in X \text{ s.t. } w < v\}$$

We can now define a function, *topn*, which will provide the homomorphism. Like *simpletopn*, *topn* returns the n highest valued elements, keeping ties. In addition, if there is an infinite number of elements approaching a supremum, *topn* returns a special element whose value is the supremum, and whose derivation is the symbol ∞.

$$\mathit{topn}(X) = \mathit{simpletopn}(X) \cup \begin{cases} \emptyset & \mathit{if}\ |\mathit{simpletopn}(X)| \geq n \\ \emptyset & \mathit{if}\ X = \mathit{simpletopn}(X) \\ \{\langle \sup_{v|\langle v,d\rangle \in X - \mathit{simpletopn}(X)} v, \infty\rangle\} & \mathit{otherwise} \end{cases}$$

We can now define the Viterbi-n-best semiring as a homomorphism from the Viterbi-all semiring. In particular, we define the elements of the semiring to be $\{\mathit{topn}(X) | X \in 2^{\mathbb{R}_0^1 \times \mathbb{E}}\}$. Because of this definition, for every A, B in the Viterbi-n-best semiring, there is some X, Y such that $\mathit{topn}(X) = A$ and $\mathit{topn}(Y) = B$. We can then define

$$\max_{\mathit{Vit\text{-}n}} A, B = C$$

where $C = \mathit{topn}(X \cup Y)$, for some, X, Y such that $\mathit{topn}(X) = A$ and $\mathit{topn}(Y) = B$. In appendix 2–A.1, we prove that C is uniquely defined by this relationship. Similarly, we define the multiplicative operator $\underset{\mathit{Vit\text{-}n}}{\times}$ to be

$$A \underset{\mathit{Vit\text{-}n}}{\times} B = C$$

where $C = \mathit{topn}(X \star Y)$, for some, X, Y such that $\mathit{topn}(X) = A$ and $\mathit{topn}(Y) = B$, and again prove the uniqueness of the relationship in the appendix. Also in the appendix, we prove that these operators do indeed form an ω-continuous semiring.

2.3 Efficient Computation of Item Values

Recall that the value of an item x is just $V(x) = \bigoplus_{D \in inner(x)} V(D)$. This definition may require summing over exponentially many or even infinitely many terms. In this section, we give relatively efficient formulas for computing the values of items. There are three cases that must be handled. First is the base case: if x is a rule, then

$$V(x) = \bigoplus_{D \in inner(x)} V(D) = \bigoplus_{D \in \{\langle x \rangle\}} V(D) = V(\langle x \rangle) = R(x)$$

The second and third cases occur when x is an item. Recall that each item is associated with a bucket, and that the buckets are ordered. Each item x is either associated with a non-looping bucket, in which case its value depends only on the values of items in earlier buckets; or with a looping bucket, in which case its value depends potentially on the values of other items in the same bucket. In the second case, when the item is associated with a non-looping bucket, and if we compute items in the same order as their buckets, we can assume that the values of items $a_1 ... a_k$ contributing to the value of item b are known. We give a formula for computing the value of item b that depends only on the values of items in earlier buckets.

For the third case, in which x is associated with a looping bucket, infinite loops may occur, when the value of two items in the same bucket are mutually dependent, or an item depends on its own value. These infinite loops may require computation of infinite sums. Still, we can express these infinite sums in a relatively simple form, allowing them to be efficiently computed or approximated.

2.3.1 Item Value Formula

Theorem 2.2

If an item x is not in a looping bucket, then

$$V(x) = \bigoplus_{a_1 ... a_k \text{ s.t. } \frac{a_1 ... a_k}{x}} \bigotimes_{i=1}^{k} V(a_i) \tag{2.5}$$

Proof Let us expand our notion of inner to include deduction rules: $inner(\frac{a_1...a_k}{b})$ is the set of all derivation trees of the form $\langle b : \langle a_1...\rangle\langle a_2...\rangle...\langle a_k...\rangle\rangle$. For any item derivation tree that is not a simple rule, there is some $a_1...a_k, b$ such that $D \in inner(\frac{a_1...a_k}{b})$. Thus, for any item x,

$$\begin{aligned} V(x) &= \bigoplus_{D\in inner(x)} V(D) \\ &= \bigoplus_{a_1...a_k} \bigoplus_{D\in inner(\frac{a_1...a_k}{x})} V(D) \end{aligned} \tag{2.6}$$

Consider item derivation trees $D_{a_1}...D_{a_k}$ headed by items $a_1...a_k$ such that $\frac{a_1...a_k}{x}$. Recall that $\langle x : D_{a_1}, ..., D_{a_k}\rangle$ is the item derivation tree formed by combining each of these trees into a full tree, and notice that

$$\bigcup_{\substack{D_{a_1}\in inner(a_1),...,\\ D_{a_k}\in inner(a_k)}} \langle x : D_{a_1}, ..., D_{a_k}\rangle = inner(\frac{a_1...a_k}{x})$$

Therefore

$$\bigoplus_{D\in inner(\frac{a_1...a_k}{x})} V(D) = \bigoplus_{\substack{D_{a_1}\in inner(a_1),...,\\ D_{a_k}\in inner(a_k)}} V(\langle x : D_{a_1}, ..., D_{a_k}\rangle)$$

Also notice that $V(\langle x : D_{a_1}, ..., D_{a_k}\rangle) = \bigotimes_{i=1}^{k} V(D_{a_i})$. Thus,

$$\bigoplus_{\substack{D_{a_1}\in inner(a_1),...,\\ D_{a_k}\in inner(a_k)}} V(\langle x : D_{a_1}, ..., D_{a_k}\rangle) = \bigoplus_{\substack{D_{a_1}\in inner(a_1),...,\\ D_{a_k}\in inner(a_k)}} \bigotimes_{i=1}^{k} V(D_{a_i})$$

Since for all semirings, both operations are associative, and multiplication distributes over addition, we can rearrange summations and products:

$$\bigoplus_{\substack{D_{a_1}\in inner(a_1),...,\\ D_{a_k}\in inner(a_k)}} \bigotimes_{i=1}^{k} V(D_{a_i}) = \bigotimes_{i=1}^{k} \bigoplus_{D_{a_i}\in inner(a_i)} V(D_{a_i})$$

$$= \bigotimes_{i=1}^{k} V(a_i)$$

Substituting this back into Equation 2.6, we get

$$V(x) = \bigoplus_{a_1 \ldots a_k \text{ s.t. } \frac{a_1 \ldots a_k}{x}} \bigotimes_{i=1}^{k} V(a_i)$$

completing the proof. $\diamond$

Now, we address the case in which x is an item in a looping bucket. This case requires computation of an infinite sum. We will write out this infinite sum, and discuss how to compute it exactly in all cases, except for one, where we approximate it.

Consider the derivable items $x_1 \ldots x_m$ in some looping bucket B. If we build up derivation trees incrementally, when we begin processing bucket B, only those trees with no items from bucket B will be available, what we will call 0th generation derivation trees. We can put these 0th generation trees together to form first generation trees, headed by elements in B. We can combine these first generation trees with each other and with 0th generation trees to form second generation trees, and so on. Formally, we define the *generation* of a derivation tree headed by x in bucket B to be the largest number of items in B we can encounter on a path from the root to a leaf. It will be convenient to define the generation of the derivation of any item x in a bucket preceding B to be generation 0; generation 0 will not contain derivations of any items in bucket B.

Consider the set of all trees of generation at most g headed by x. Call this set $inner_{\leq g}(x, B)$. We can define the *$\leq g$ generation value* of an item x in bucket B, $V_{\leq g}(x, B)$:

$$V_{\leq g}(x, B) = \bigoplus_{D \in inner_{\leq g}(x,B)} V(x)$$

Intuitively, as g increases, for $x \in B$, $inner_{\leq g}(x, B)$ becomes closer and closer to $inner(x)$. Thus, intuitively, the finite sum of values in the latter approaches the infinite sum of values in the former. For ω-continuous semirings (which includes all of the semirings considered in this chapter), an infinite sum is equal to the supremum of the partial sums (Kuich, 1997,

p. 613). Thus,

$$V(x) = \bigoplus_{D \in inner(x,B)} V(x) = \sup_g V_{\leq g}(x, B)$$

It will be easier to compute the supremum if we find a simple formula for $V_{\leq g}(x, B)$.

Notice that for items x in buckets preceding B, since these items are all included in generation 0,

$$V_{\leq g}(x, B) = V(x) \tag{2.7}$$

Also notice that for items $x \in B$, there will be no generation 0 derivations, so

$$V_{\leq 0}(x, B) = 0$$

Thus, generation 0 makes a trivial base for a recursive formula. Now, we can consider the general case:

Theorem 2.3

For x an item in a looping bucket B, and for $g \geq 1$,

$$V_{\leq g}(x, B) = \bigoplus_{a_1 \ldots a_k \text{ s.t. } \frac{a_1 \ldots a_k}{x}} \bigotimes_{i=1..k} \begin{cases} V(a_i) \ \textit{if} \ a_i \notin B \\ V_{\leq g-1}(a_i, B) \ \textit{if} \ a_i \in B \end{cases} \tag{2.8}$$

The proof parallels that of Theorem 2.2, and is given in Appendix 2–A.

2.3.2 Solving the Infinite Summation

A formula for $V_{\leq g}(x, B)$ is useful, but what we really need is specific techniques for computing the supremum, $V(x) = \sup_g V_{\leq g}(x, B)$.

For all ω-continuous semirings, the supremum of iteratively approximating the value of a set of polynomial equations, as we are essentially doing in Equation 2.8, is equal to the smallest solution to the equations (Kuich, 1997, p. 622). In particular, consider the equations:

$$V_{\leq \infty}(x, B) = \bigoplus_{a_1 \ldots a_k \text{ s.t. } \frac{a_1 \ldots a_k}{x}} \bigotimes_{i=1..k} \begin{cases} V(a_i) \ \textit{if} \ a_i \notin B \\ V_{\leq \infty}(a_i, B) \ \textit{if} \ a_i \in B \end{cases} \tag{2.9}$$

where $V_{\leq\infty}(x, B)$ can be thought of as indicating $|B|$ different variables, one for each item x in the looping bucket B. Equation 2.8 represents the iterative approximation of this equation, and therefore the smallest solution to this equation represents the supremum of that one.

One fact will be useful for several semirings: whenever the values of all items $x \in B$ at generation $g+1$ are the same as the values of all items in the preceding generation, g, they will be the same at all succeeding generations, as well. Thus, the value at generation g will be the value of the supremum. The proof of this fact is trivial: we substitute the value of $V_{\leq g}(x, B)$ for $V_{\leq g+1}(x, B)$ to show that $V_{\leq g+2}(x, B) = V_{\leq g+1}(x, B)$.

$$\begin{aligned}
V_{\leq g+2}(x, B) &= \bigoplus_{a_1 \ldots a_k \text{ s.t. } \frac{a_1 \ldots a_k}{x}} \quad \bigotimes_{i=1..k} \begin{cases} V(a_i) \text{ if } a_i \notin B \\ V_{\leq g+1}(a_i, B) \text{ if } a_i \in B \end{cases} \\
&= \bigoplus_{a_1 \ldots a_k \text{ s.t. } \frac{a_1 \ldots a_k}{x}} \quad \bigotimes_{i=1..k} \begin{cases} V(a_i) \text{ if } a_i \notin B \\ V_{\leq g}(a_i, B) \text{ if } a_i \in B \end{cases} \\
&= V_{\leq g+1}(x, B)
\end{aligned}$$

Now, we can consider various semiring specific algorithms for computing the supremum. We first examine the simplest case, the boolean semiring (booleans under $\vee$ and $\wedge$). Notice that whenever a particular item has value *TRUE* at generation g, it must also have value *TRUE* at generation $g+1$, since if the item can be derived in at most g generations then it can certainly be derived in at most $g+1$ generations. Thus, since the number of *TRUE* valued items is non-decreasing, and is at most $|B|$, eventually the values of all items must not change from one generation to the next. Therefore, for the boolean semiring, a simple algorithm suffices: keep computing successive generations, until no change is detected in some generation; the result is the supremum. We can perform this computation efficiently if we keep track of items that change value in generation g and only examine items that depend on them in generation $g+1$. This algorithm is then similar to the algorithm of Shieber *et al.* (1993).

For the next three semirings – the counting semiring, the Viterbi semiring, and the derivation forest semiring – we need the concept of a *derivation subgraph.* In Section 2.2.2 we considered the strongly connected components of the dependency graph, consisting of items that for some sentence could possibly depend on each other, and we put these possibly interdependent items together in looping buckets. For a given sentence and grammar, not all items will have derivations. We will find the subgraph of the dependency graph of items with derivations, and compute the strongly connected components of this subgraph. The strongly connected components of this subgraph correspond to loops that actually occur given the sentence and the grammar, as opposed to loops that might occur for some sentence and grammar, given the parser alone. We call this subgraph the derivation subgraph, and we will say that items in a strongly connected component of the derivation subgraph are part of a loop.

Now, we can discuss the counting semiring (integers under $+$ and $\times$.) In the counting semiring, for each item, there are three cases: the item can be in a loop; the item can depend (directly or indirectly) on an item in a loop; or the item does not depend on loops. If the item is in a loop or depends on a loop, its value is infinite. If the item does not depend on a loop in the current bucket, then its value becomes fixed after some generation. We can now give the algorithm: first, compute successive generations until the set of items in B does not change from one generation to the next. Next, compute the derivation subgraph, and its strongly connected components. Items in a strongly connected component (a loop) have an infinite number of derivations, and thus an infinite value. Compute items that depend directly or indirectly on items in loops: these items also have infinite value. Any other items can only be derived in finitely many ways using items in the current bucket, so compute successive generations until the values of these items do not change.

The arctic semiring,$\langle \mathbb{R} \cup \{-\infty\}, \max, +, -\infty, 0\rangle$, is analogous to the counting semiring: loops lead to infinite values. Thus, we can use the same algorithm.

The method for solving the infinite summation for the derivation forest semiring depends on the implementation of derivation forests. Essentially, that representation will use pointers to efficiently represent derivation forests. Pointers, in various forms, allow one to efficiently represent infinite circular references, either directly (Goodman, 1998a), or indirectly (Goodman, 1998b).

The algorithm we use is to compute the derivation subgraph, and then create pointers analogous to the directed edges in the derivation subgraph, including pointers in loops whenever there is a loop in the derivation subgraph (corresponding to an infinite number of derivations). For the representation described in Section 2.2.5 we perform two steps. First, for each item $x \in B$, we will set

$$V(x) = \langle\text{“Union”}\ x_1\ \langle\text{“Union”}\ x_2\ \langle\text{“Union”}\ x_3\ \langle\cdots\rangle\rangle\rangle\rangle$$

where there is one x_i for each instantiation of a rule $\frac{a_1 \ldots a_k}{x}$. Next, for each derivation of x, $\frac{a_1 \ldots a_k}{x}$, we will create a value

$$\langle\text{“Concatenate”}\ V(a_1)\ \langle\text{“Concatenate”}\ V(a_2)\ \langle\cdots\langle\text{“Concatenate”}\ V(a_{k-1})\ V(a_k)\rangle\cdots\rangle\rangle\rangle$$

and destructively change an appropriate x_i to this value. In a LISP-like language, this will create the appropriate circular pointers. As in the finite case, this representation is equivalent to that of Billot and Lang (1989).

For the Viterbi semiring, the algorithm is analogous to the boolean case. Derivations using loops in these semi-rings will always have values lower than derivations not using loops, since the value with the loop will be the same as some value without the loop, multiplied by some set of rule probabilities that are at most 1. Thus, loops do not change values. Therefore, we can simply compute successive generations until values fail to change from one iteration to the next. The tropical semiring is analogous: again, loops do not change values, so we can compute successive generations until values don't change.

Now, consider implementations of the Viterbi-derivation semiring in practice, in which we keep only a representative derivation, rather than the whole derivation forest. In this case, loops do not change values, and we use the same algorithm as for the Viterbi semiring. On the other hand, for a theoretically correct implementation, loops with value 1 can lead to an infinite number of derivations, in the same way they did for the derivation forest semiring. Thus, for a theoretically correct implementation, we must use the same techniques we used for the derivation semiring. In an implementation of the Viterbi-n-best semiring, in practice, loops can change values, but at most n times, so the same algorithm used for the Viterbi semiring still works. Again, in a theoretical implementation, we need to use the same mechanism as in the derivation forest semiring.

The last semiring we consider is the inside semiring. This semiring is the most difficult. There are two cases of interest, one of which we can solve exactly, and the other of which

for each length l, longest **downto** shortest
  for each start s
    for each split length t
      for each rule $A \to BC \ \in R$

$$
\begin{aligned}
&\mathit{outside}[s, B, s{+}t] := \mathit{outside}[s, B, s{+}t] + \\
&\qquad \mathit{outside}[s, A, s{+}l] \times \mathit{inside}[s{+}t, C, s{+}l] \times P(A \to BC); \\
&\mathit{outside}[s{+}t, C, s{+}l] := \mathit{outside}[s{+}t, C, s{+}l] + \\
&\qquad \mathit{outside}[s, A, s{+}l] \times \mathit{inside}[s, B, s{+}t] \times P(A \to BC);
\end{aligned}
$$

Figure 2.8: Outside algorithm

requires approximations. In many cases involving looping buckets, all deduction rules will be of the form $\frac{a_1 x}{b}$, where a_1 and b are items in the looping bucket, and x is either a rule, or an item in a previously computed bucket. This case corresponds to the items used for deducing singleton productions, such as Earley's algorithm uses for rules such as $A \to B$ and $B \to A$. In this case, Equation 2.9 forms a set of linear equations that can be solved by matrix inversion. In the more general case, as is likely to happen with epsilon rules, we get a set of nonlinear equations, and must solve them by approximation techniques, such as simply computing successive generations for many iterations.[2] Stolcke (1993) provides an excellent discussion of these cases, including a discussion of sparse matrix inversion, useful for speeding up some computations.

2.4 Reverse Values

The previous section showed how to compute several of the most commonly used values for parsers, including boolean, inside, Viterbi, counting, and derivation forest values, among others. Noticeably absent from the list are the outside probabilities. In general, computing outside probabilities is significantly more complicated than computing inside probabilities.

In this section, we show how to compute outside probabilities from the same item-based descriptions used for computing inside values. Outside probabilities have many uses,

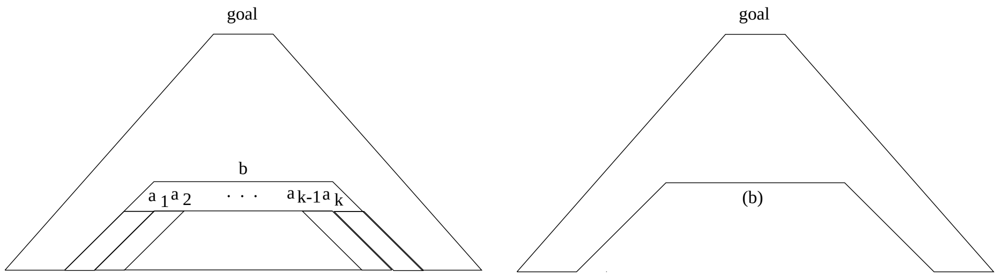
[2]Note that even in the case where we can only use approximation techniques, this algorithm is relatively efficient. By assumption, in this case, there is at least one deduction rule with two items in the current generation; thus, the number of deduction trees over which we are summing grows exponentially with the number of generations: a linear amount of computation yields the sum of the values of exponentially many trees.

including for reestimating grammar probabilities (Baker, 1979), for improving parser performance on some criteria (Chapter 3), for speeding parsing in some formalisms, such as Data-Oriented Parsing (Chapter 4), and for good thresholding algorithms (Chapter 5).

We will show that by substituting other semirings, we can get values analogous to the outside probabilities for any commutative semiring, and in Sections 2.4.1 and 2–C that we can get similar values for many non-commutative semirings as well. We will refer to these analogous quantities as *reverse* values. For instance, the quantity analogous to the outside value for the Viterbi semiring will be called the reverse Viterbi value. Notice that the inside semiring values of a Hidden Markov Model (HMM) correspond to the forward values of HMMs, and the reverse inside values of an HMM correspond to the backwards values.

Compare the outside algorithm(Baker, 1979; Lari and Young, 1990; Lari and Young, 1991), given in Figure 2.8, to the inside algorithm of Figure 2.2. Notice that while the inside and recognition algorithms were very similar, the outside algorithm is quite a bit different. In particular, while the inside and recognition algorithms looped over items from shortest to longest, the outside algorithm loops over items in the reverse order, from longest to shortest. Also, compare the inside algorithm's main loop formula to the outside algorithm's main loop formula. While there is clearly a relationship between the two equations, the exact pattern of the relationship is not obvious. Notice that the outside formula is about twice as complicated as the inside formula. This doubled complexity is typical of outside formulas, and partially explains why the item-based description format is so useful: descriptions for the simpler inside values can be developed with relative ease, and then automatically transformed used to compute the twice as complicated outside values.

For a context-free grammar, using the CKY parser of Figure 2.3, recall that the inside probability for an item $[i, A, j]$ is $P(A \rightarrow w_i...w_{j-1})$. The outside probability for the same item is $P(S \stackrel{*}{\Rightarrow} w_1...w_{i-1}Aw_j...w_n)$. Thus, the outside probability has the property that when multiplied by the inside probability, it gives the probability that the start symbol generates the sentence using the given item, $P(S \stackrel{*}{\Rightarrow} w_1...w_{i-1}Aw_j...w_n \stackrel{*}{\Rightarrow} w_1...w_n)$. This probability equals the sum of the probabilities of all derivations using the given item. Formally, letting $P(D)$ represent the probability of a particular derivation, and $C(D, [i, X, j])$ represent the number of occurrences of item $[i, X, j]$ in derivation D (which for some parsers



Derivation of [goal] Outer tree of $[b]$

Figure 2.9: Goal tree, outer tree

could be more than one if X were part of a loop),

$$\mathit{inside}(i, X, j) \times \mathit{outside}(i, X, j) = \sum_{D \text{ a derivation}} P(D)\ C(D, [i, X, j])$$

The reverse values in general have an analogous meaning. Let $C(D, x)$ represent the number of occurrences (the count) of item x in item derivation tree D. Then, for an item x, the reverse value $Z(x)$ should have the property

$$V(x) \otimes Z(x) = \bigoplus_{D \text{ a derivation}} V(D) C(D, x) \tag{2.10}$$

Notice that we have multiplied an element of the semiring, $V(D)$, by an integer, $C(D, x)$. This multiplication is meant to indicate repeated addition, using the additive operator of the semiring. Thus, for instance, in the Viterbi semiring, multiplying by a count other than 0 has no effect, since $x \oplus x = \max(x, x) = x$, while in the inside semiring, it corresponds to actual multiplication. This value represents the sum of the values of all derivation trees that the item x occurs in; if an item x occurs more than once in a derivation tree D, then the value of D is counted more than once.

To formally define the reverse value of an item x, we must first define the *outer* trees $\mathit{outer}(x)$. Consider an item derivation tree of the goal item, containing one or more instances of item x. Remove one of these instances of x, and its children too, leaving a gap in its place. This tree is an outer tree of x. Figure 2.9 shows an item derivation tree of the goal item, including a subderivation of an item b, derived from terms $a_1, ..., a_k$. It also shows an outer tree of b, with b and its children removed; the spot b was removed from is labelled by (b).

For an outer tree $D \in outer(x)$, we define its value, $Z(D)$, to be the product of the value of all rules in D, $\bigotimes_{r \in D} R(r)$. Then, the reverse value of an item can be formally defined as

$$Z(x) = \bigoplus_{D \in outer(x)} Z(D) \tag{2.11}$$

That is, the reverse value of x is the sum of the values of each outer tree of x.

Now, we show that this definition of reverse values has the property described by Equation 2.10.[3]

Theorem 2.4

$$V(x) \otimes Z(x) = \bigoplus_{D \text{ a derivation}} V(D)C(D, x)$$

Proof

$$\begin{aligned} V(x) \otimes Z(x) &= V(x) \otimes \bigoplus_{O \in outer(x)} Z(O) \\ &= \left(\bigoplus_{I \in inner(x)} V(I) \right) \otimes \bigoplus_{O \in outer(x)} Z(O) \\ &= \bigoplus_{I \in inner(x)} \bigoplus_{O \in outer(x)} V(I) \otimes Z(O) \end{aligned} \tag{2.12}$$

Next, we argue that this last expression equals the expression on the right hand side of Equation 2.10, $\bigoplus_D V(D)C(D, x)$. For an item x, any outer part of an item derivation tree for x can be combined with any inner part to form a complete item derivation tree. That is, any $O \in outer(x)$ and any $I \in inner(x)$ can be combined to form an item derivation tree D containing x, and any item derivation tree D containing x can be decomposed into such outer and inner trees. Thus, the list of all combinations of outer and inner trees corresponds exactly to the list of all item derivation trees containing x. In fact, for an item derivation tree D containing $C(D, x)$ instances of x, there are $C(D, x)$ ways to form D from

[3]We note that satisfying Equation 2.10 is a useful but not sufficient condition for using reverse inside values for grammar re-estimation. While this definition will typically provide the necessary values for the E step of an E-M algorithm, additional work will typically be required to prove this fact; Equation 2.10 should be useful in such a proof.

combinations of outer and inner trees. Also, notice that for D combined from O and I

$$V(I) \otimes Z(O) = \bigotimes_{r \in I} R(r) \otimes \bigotimes_{r \in O} R(r) = \bigotimes_{r \in D} R(r) = V(D)$$

Thus,

$$\bigoplus_{I \in inner(x)} \bigoplus_{O \in outer(x)} V(I) \otimes Z(O) = \bigoplus_{D} V(D)C(D,x) \tag{2.13}$$

Combining Equation 2.12 with Equation 2.13, we see that

$$V(x) \otimes Z(x) = \bigoplus_{D \text{ a derivation}} V(D)C(D,x)$$

completing the proof. $\diamond$

There is a simple, recursive formula for efficiently computing reverse values. Recall that the basic equation for computing forward values not involved in loops was

$$V(x) = \bigoplus_{a_1 \ldots a_k \text{ s.t. } \frac{a_1 \ldots a_k}{x}} \bigotimes_{i=1}^{k} V(a_i)$$

At this point, for conciseness, we introduce a nonstandard notation. We will soon be using many sequences of the form $1, 2, \ldots, j-2, j-1, j+1, j+2, \ldots, k-1, k$. We indicate such sequences by $1, \overset{-j}{\ldots}, k$, significantly simplifying some expressions. By extension, we will also write $f(1), \overset{-j}{\ldots}, f(k)$ to indicate a sequence of the form $f(1), f(2), \ldots, f(j-2), f(j-1), f(j+1), f(j+2), \ldots, f(k-1), f(k)$.

Now, we can give a simple formula for computing reverse values $Z(x)$ not involved in loops:

Theorem 2.5

For items $x \in B$ where B is non-looping,

$$Z(x) = \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} Z(b) \bigotimes_{i=1, \overset{-j}{\ldots}, k} V(a_i) \tag{2.14}$$

unless x is the goal item, in which case $Z(x) = 1$, the multiplicative identity of the semiring.

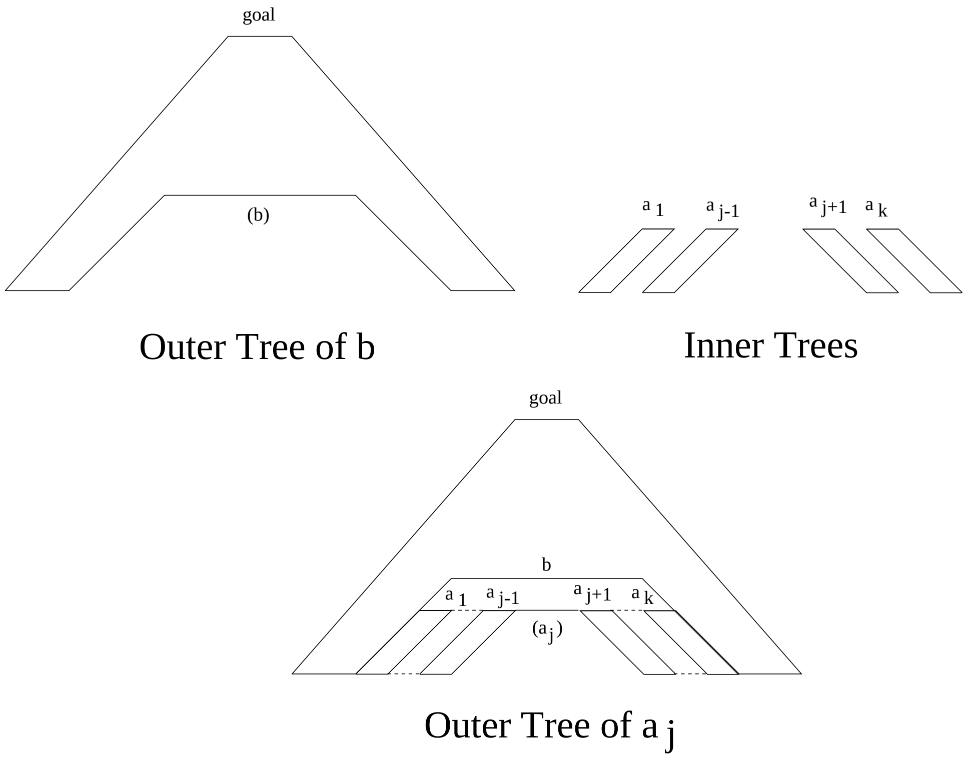


Figure 2.10: Combining an outer tree with inner trees to form an outer tree

Proof The simple case is when x is the goal item. Since an outer tree of the goal item is a derivation of the goal item, with the goal item and its children removed, and since we assumed in Section 2.2.2 that the goal item can only appear in the root of a derivation tree, the outer trees of the goal item are all empty. Thus,

$$Z(goal) = \bigoplus_{D \in outer(goal)} Z(D) = Z(\{\langle\rangle\}) = \bigotimes_{r \in \{\langle\rangle\}} R(r) = 1$$

As mentioned in Section 2.2.1, the product of zero elements is the multiplicative identity.

Now, we consider the general case. We need to expand our concept of *outer* to include deduction rules, where $outer\left(j, \frac{a_1 \ldots a_k}{b}\right)$ is an item derivation tree of the goal item with one sub-tree removed, a sub-tree headed by a_j whose parent is b and whose siblings are headed by $a_1, \overset{\neg j}{\ldots}, a_k$. Notice that for every outer tree $D \in outer(x)$, there is exactly one $j, a_1, \ldots, a_k$, and b such that $x = a_j$ and $D \in outer\left(j, \frac{a_1 \ldots a_k}{b}\right)$: this corresponds to the deduction rule used at the spot in the tree where the sub-tree headed by x was deleted.

Figure 2.10 illustrates the idea of putting together an outer tree of b with inner trees for $a_1, \overset{\neg j}{\ldots}, a_k$ to form an outer tree of $x = a_j$. Using this observation,

$$\begin{aligned} Z(x) &= \bigoplus_{D \in outer(x)} Z(D) \\ &= \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} \; \bigoplus_{D \in outer\left(j, \frac{a_1 \ldots a_k}{b}\right)} Z(D) \end{aligned} \tag{2.15}$$

Now, consider all of the outer trees $outer\left(j, \frac{a_1 \ldots a_k}{b}\right)$. For each item derivation tree $D_{a_1} \in inner(a_1), \overset{\neg j}{\ldots}, D_{a_k} \in inner(a_k)$ and for each outer tree $D_b \in outer(b)$, there will be one outer tree in the set $outer\left(j, \frac{a_1 \ldots a_k}{b}\right)$. Similarly, each tree in $outer\left(j, \frac{a_1 \ldots a_k}{b}\right)$ can be decomposed into an outer tree in $outer(b)$ and derivation trees for $a_1, \overset{\neg j}{\ldots}, a_k$. Then,

$$\begin{aligned} & \bigoplus_{D \in outer\left(j, \frac{a_1 \ldots a_k}{b}\right)} Z(D) \\ = & \bigoplus_{\substack{D_b \in outer(b), \\ D_{a_1} \in inner(a_1), \overset{\neg j}{\ldots}, \\ D_{a_k} \in inner(a_k)}} Z(D_b) \otimes V(D_{a_1}) \otimes \overset{\neg j}{\cdots} \otimes V(D_{a_k}) \\ = & \left(\bigoplus_{D_b \in outer(b)} Z(D_b) \right) \otimes \left(\bigoplus_{D_{a_1} \in inner(a_1)} V(D_{a_1}) \right) \otimes \overset{\neg j}{\cdots} \otimes \left(\bigoplus_{D_{a_k} \in inner(a_k)} V(D_{a_k}) \right) \\ = & \; Z(b) \otimes V(a_1) \otimes \overset{\neg j}{\cdots} \otimes V(a_k) \\ = & \; Z(b) \otimes \bigotimes_{i=1, \overset{\neg j}{\ldots}, k} V(a_i) \end{aligned} \tag{2.16}$$

Substituting equation 2.16 into equation 2.15, we conclude that

$$Z(x) = \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} Z(b) \otimes \bigotimes_{i=1, \overset{\neg j}{\ldots}, k} V(a_i)$$

completing the general case. $\diamond$

Computing the reverse values for loops is somewhat more complicated. As in the forward case, it requires an infinite sum. Computation of this infinite sum will be semiring specific. Also as in the forward case, we use the concept of generation. Let us define the generation g of an outer tree D of item x in bucket B to be the number of items in bucket B on the path between the root and the removal point, inclusive. Thus, outer trees of items in buckets following B will be in generation 0. Let $outer_{\leq g}(x, B)$ represent the set of outer trees of x with generation at most g. It should be clear that as g approaches ∞, $outer_{\leq g}(x, B)$ approaches $outer(x)$. Now, we can define the $\leq g$ generation reverse value of an item x in bucket B, $Z_{\leq g}(x, B)$:

$$Z_{\leq g}(x, B) = \bigoplus_{D \in outer_{\leq g}(x,B)} Z(D)$$

For ω-continuous semirings, an infinite sum is equal to the supremum of the partial sums:

$$\bigoplus_{D \in outer(x,B)} Z(D) = Z_{\leq \infty}(x, B) = \sup_g Z_{\leq g}(x, B)$$

Thus, we wish to find a simple formula for $Z_{\leq g}(x, B)$.

Notice that for $x \in C$, where C is a bucket following B, by the inclusion of derivations of these items in generation 0,

$$Z_{\leq g}(x, B) = Z(x) \tag{2.17}$$

Also notice that for $g = 0$ and $x \in B$,

$$Z_{\leq 0}(x, B) = \bigoplus_{D \in outer_{\leq g}(x,B)} Z(D) = \bigoplus_{D \in \emptyset} Z(D) = 0$$

Thus, generation 0 makes a simple base for a recursive formula for outer values. Now, we can consider the general case, for $g \geq 1$,

Theorem 2.6

For items $x \in B$ and $g \geq 1$,

$$Z_{\leq g}(x, B) = \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} \left(\bigotimes_{i=1, \neg j, k} V(a_i) \right) \otimes \begin{cases} Z_{\leq g-1}(b, B) & \mathit{if}\ b \in B \\ Z(b) & \mathit{if}\ b \notin B \end{cases} \tag{2.18}$$

The proof parallels that of Theorem 2.5, and is given in Appendix 2–A.

2.4.1 Reverse Values in Non-commutative Semirings

Equations 2.14 and 2.18 apply only to commutative semirings, since their derivation makes use of the commutativity of the multiplicative operator. We might wish to compute reverse values in non-commutative semirings as well. For instance, the reverse values in the derivation semiring could be used to compute the set of all parses that include a particular constituent. It turns out that there is no equation in the non-commutative semirings corresponding directly to Equations 2.14 and 2.18; in fact, in general, there are no values in non-commutative semirings corresponding directly to the reverse values.

When we compute reverse values, we need them to be such that the product of the forward and the reverse values gives the sum of the values over all derivation trees using the item. Consider a case in which the forward value of the item is b and there are derivation trees with the values abc and dbe. The product of the forward and reverse values should thus be $abc + dbe$, but in a non-commutative semiring there will not be any reverse value x such that $xb = abc + dbe$. Instead, one must create what we call Pair semirings, corresponding to multi-sets of pairs of values in the base semiring. In this example, the reverse value would be the multiset $\{\langle a, c\rangle, \langle d, e\rangle\}$ and a special operator would combine b with this set to yield $abc + dbe$. Using Pair semirings, one can find equations directly analogous to Equations 2.14 and 2.18. A complete explication of reverse values for non-commutative semirings and the derivation of the equations are given in Appendix 2–C.

2.5 Semiring Parser Execution

2.5.1 Bucketing

Executing a semiring parser is fairly simple. There is, however, one issue that must be dealt with before we can actually begin parsing. A semiring parser computes the values of items in the order of the buckets they fall into. Thus, before we can begin parsing, we need to know which items fall into which buckets, and the ordering of those buckets. There are three approaches to determining the buckets and ordering that we will discuss in this section. The first approach is a simple, brute force enumeration of all items, derivable or not, followed by a topological sort. This approach will have suboptimal time and space complexity for some item-based descriptions. The second approach is to use an agenda parser in the boolean semiring to determine the derivable items and their dependencies, and to then perform a topological sort. This approach has optimal time complexity, but typically suboptimal space complexity. The final approach is to use bucketing code specific to the item-based interpreter. This achieves optimal performance for additional programming effort.

The simplest way to determine the bucketing is to simply enumerate all possible items for the given item-based description, grammar and input sentence. Then, we compute the strongly connected components and a partial ordering; both steps can be done in time proportional to the number of items plus the number of dependencies (Cormen *et al.*, 1990, ch. 23). For some parsers, this technique has optimal time complexity, although poor space complexity. In particular, for the CKY algorithm, the time complexity is optimal, but since it requires computing and storing all possible $O(n^3)$ dependencies between the items, it takes significantly more space than the $O(n^2)$ space required in the best implementation. In general, the brute force technique raises the space complexity to be the same as the time complexity. Furthermore, for some algorithms, such as Earley's algorithm, there could be a significant time complexity added as well. In particular, Earley's algorithm may not need to examine all possible items. For certain grammars, Earley's algorithm examines only a linear number of items and a linear number of dependencies, even though there are $O(n^2)$ possible items, and $O(n^3)$ possible dependencies. Thus the brute force approach would require $O(n^3)$ time and space instead of $O(n)$ time and space.

The next approach to finding the bucketing solves the time complexity problem. In this approach, we first parse in the boolean semiring, using the agenda parser described by Shieber *et al.* (1993), and then we perform a topological sort. Shieber *et al.* use an interpreter which, after an item is derived, determines all items that could *trigger* off of that item. For instance, in an Earley-style parser, such as that of Figure 2.4, if an item $[i, A \rightarrow \alpha \bullet B\beta, j]$ is processed, and there is a rule $B \rightarrow \gamma$, then, by the prediction rule, $[j, B \rightarrow \ \bullet \gamma, j]$ can be derived. Thus, immediately after $[i, A \rightarrow \alpha \bullet B\beta, j]$ is processed, $[j, B \rightarrow \ \bullet \gamma, j]$ is added to an agenda; new items to be processed are taken off of the agenda. This approach works fine for the boolean semiring, where items only have value *TRUE* or *FALSE*, but cannot be used directly for other semirings. For other semirings, we need to make sure that the values of items are not computed until after the values of all items they depend on are computed. However, we can use the algorithm of Shieber *et al.* to compute all of the items that are derivable, and to store all of the dependencies between the items. Then we perform a topological sort on the items. The time complexity of both the agenda parser and the topological sort will be proportional to the number of dependencies, which will be proportional to the optimal time complexity. Unfortunately, we still have the space complexity problem, since again, the space used will be proportional to the number of dependencies, rather than to the number of items.

The third approach to bucketing is to create algorithm-specific bucketing code; this results in parsers with both optimal time and optimal space complexity. For instance, in a CKY style parser, we can simply create one bucket for each length, and place each item into the bucket for its length. For some algorithms, such as Earley's algorithm, special-purpose code for bucketing might have to be combined with code for triggering, just as in the algorithm of Shieber *et al.* , in order to achieve optimal performance.

2.5.2 Interpreter

Once we have the bucketing, the parsing step is fairly simple. The basic algorithm appears in Figure 2.11. We simply loop over each item in each bucket. There are two types of buckets: looping buckets, and non-looping buckets. If the current bucket is a looping bucket, we compute the infinite sum needed to determine the bucket's values; in a working system, we substitute semiring specific code for this section, as described in Section 2.3.2. If the bucket is not a looping bucket, we simply compute all of the possible instantiations that could contribute to the values of items in that bucket. Finally, we return the value of the goal item.

The reverse semiring parser interpreter is very similar to the forward semiring parser interpreter. The differences are that in the reverse semiring parser interpreter, we traverse the buckets in reverse order, and we use the formula for the reverse values, rather than the forward values.

Both interpreters are closely based on formulas derived earlier. The forward semiring parser interpreter uses the code

$$\textbf{for each } x \in \mathit{current}, a_1 ... a_k \text{ s.t. } \frac{a_1 ... a_k}{x}$$

$current := first;$
do
 if $loop(current)$
  /* *replace with semiring specific code* */
  for $x \in current$
   $V[x, 0] = 0;$
  for $g := 1$ **to** ∞
   for each $x \in current, a_1 ... a_k$ s.t. $\frac{a_1 ... a_k}{x}$
    $V[x, g] := V[x, g] \oplus \bigotimes_{i=1}^{k} \begin{cases} V[a_i] & a_i \notin current \\ V[a_i, g-1] & a_i \in current \end{cases}$
  for each $x \in current$
   $V[x] := V[x, \infty];$
 else
  for each $x \in current, a_1 ... a_k$ s.t. $\frac{a_1 ... a_k}{x}$
   $V[x] := V[x] \oplus \bigotimes_{i=1}^{k} V[a_i];$
 $oldCurrrent := current;$
 $current := next(current);$
while $oldCurrent \neq last$
return $V[goal];$

Figure 2.11: Forward Semiring Parser Interpreter

$current := last;$
do
    if $loop(current)$
        / replace with semiring specific code */*
        for $x \in current$
            $Z[x, 0] = 0;$
        for $g := 1$ **to** ∞
            for each $j, a_1 ... a_k, x$ s.t. $\frac{a_1 ... a_k}{x}$ and $a_j \in current$

$$Z[a_j, g] := Z[a_j, g] \oplus \bigotimes_{i=1,\overline{\dots j},k} V[a_i] \otimes \begin{cases} Z[x] & x \notin current \\ Z[x, g-1] & x \in current \end{cases}$$

        for each $x \in current$
            $Z[x] := Z[x, \infty];$
    else
        for each $j, a_1 ... a_k, x$ s.t. $\frac{a_1 ... a_k}{x}$ and $a_j \in current$
            $Z[a_j] := Z[a_j] \oplus Z[x] \otimes \bigotimes_{j=1,\overline{\dots j},k} V[a_j];$
    $oldCurrrent := current;$
    $current := previous(current);$
while $oldCurrent \neq first$

Figure 2.12: Reverse Semiring Parser Interpreter

$$V[x,g] := V[x,g] \oplus \bigotimes_{i=1}^{k} \begin{cases} V[a_i] & a_i \notin \mathit{current} \\ V[a_i, g-1] & a_i \in \mathit{current} \end{cases}$$

to implement Equation 2.8 for computing the values of looping buckets.

$$V_{\leq g}(x, B) = \bigoplus_{a_1 \ldots a_k \text{ s.t. } \frac{a_1 \ldots a_k}{x}} \bigotimes_{i=1..k} \begin{cases} V(a_i) \text{ if } a_i \notin B \\ V_{\leq g-1}(a_i, B) \text{ if } a_i \in B \end{cases}$$

For computing the values of non-looping buckets, the interpreter uses the code

$$\textbf{for each } x \in \mathit{current}, a_1 \ldots a_k \text{ s.t. } \frac{a_1 \ldots a_k}{x}$$
$$V[x] := V[x] \oplus \bigotimes_{i=1}^{k} V[a_i] \ ;$$

which is simply an implementation of Equation 2.5

$$V(x) = \bigoplus_{a_1 \ldots a_k \text{ s.t. } \frac{a_1 \ldots a_k}{x}} \bigotimes_{i=1}^{k} V(a_i)$$

The corresponding lines in the reverse interpreter correspond to Equations 2.18 and 2.14, respectively.

Using these equations, a simple inductive proof shows that the semiring parser interpreter is correct, and an analogous theorem holds for the reverse semiring parser interpreter.

Theorem 2.7

The forward semiring parser interpreter correctly computes the value of all items.

A sketch of the proof is given in Appendix 2–A.

There are two other implementation issues. First, for some parsers, it will be possible to discard some items. That is, some items serve the role of temporary variables, and can be discarded after they are no longer needed, especially if only the forward values are going to be computed. Also, some items do not depend on the input string, but only on the rule value function of the grammar. The values of these items can be precomputed, using the forward semiring parser interpreter.

2.6 Grammar Transformations

We can apply the same techniques to grammar transformations that we have so far applied to parsing. Consider a grammar transformation, such as the Chomsky Normal Form (CNF) grammar transformation, which takes a grammar with epsilon, unary, and n-ary branching productions, and converts it into one in which all productions are of the form $A \rightarrow BC$ or $A \rightarrow a$. For any sentence $w_1...w_n$ its value under the original grammar in the boolean semiring (*TRUE* if the sentence can be generated by the grammar, *FALSE* otherwise) is the same as its value under a transformed grammar. Therefore, we say that this grammar transformation is *value preserving* under the boolean semiring. We can generalize this concept of value preserving to other semirings. For instance, if properly specified, the CNF transformation also preserves value under any complete commutative semiring, so that the value of any sentence in the transformed grammar is the same as the value of the sentence in the original grammar. Thus, for instance, we could start with a grammar with rule probabilities, transform it using the CNF transformation, and find Viterbi values using a CKY parser and the transformed grammar; the Viterbi values for any sentence would be the values using the original grammar.

We now show how to specify grammar transformations using almost the same item-based descriptions we used for parsing. We give a value preserving transformation to CNF in this section, and in Appendix 2–B.3, we give a value preserving transformation to Greibach Normal Form (GNF). While item-based descriptions have been used to specify parsers by Shieber *et al.* (1993) and Sikkel (1993), we do not know of previous uses for specifying grammar transformations.

The concept of value preserving grammar transformation is known in the intersection of formal language theory and algebra. Kuich (1997; 1986) shows how to perform transformations to both CNF and GNF, with a value-preserving formula. Teitelbaum (1973) shows how to convert to CNF for a subclass of ω-continuous semirings. The contribution, then, of this section is to show that these value preserving transformations can be fairly simply given as item-based descriptions, allowing the same computational machinery to be used for grammar transformations as is used for parsing, and to some extent showing the relationship between certain grammar transformations and certain parsers, such as that of Graham *et al.* (1980), discussed in Section 2.7.5 and Appendix 2–B.1. While the relation-

Item form:
$[A \to \alpha\beta]$

Rule Goal
$R_2(A \to \alpha)$

Rules:

$$\frac{R_1(A \to BC\alpha)}{[A \to BC\alpha]} \qquad \text{N-ary}$$

$$\frac{R_1(A \to a)}{[A \to a]} \qquad \text{Unary}$$

$$\frac{R_1(A \to B) \qquad [B \to \alpha]}{[A \to \alpha]} \qquad \text{Extension}$$

$$\frac{[A \to \alpha]}{R_2(A \to \alpha)} \qquad \text{Output}$$

Figure 2.13: Removal of Unary Productions

ship between grammar transformations and parsers is already known in the literature on covering grammars (Nijholt, 1980; Leermakers, 1989), our treatment is clearer, because we use the same machinery for specifying both the transformations and the parsers, allowing commonalities to be expressed in the same way in both cases.

There are three steps to the CNF transformation: removal of epsilon productions; removal of unary productions; and, finally, splitting of n-ary productions. Of these three, the best one for expository purposes is the removal of unary productions, shown in Figure 2.13, so we will explicate this transformation first, even though logically it belongs second. This transformation assumes that the grammar contains no epsilon rules. It will be convenient to allow variables A, B, C to represent either nonterminals or terminals throughout this section.

To distinguish rules in the old grammar from rules in the new grammar, we have numbered the rule functions, R_1 in this transformation for the original grammar, and R_2 for the new grammar. The only difference between an item-based parser description and an item-based grammar transformation description is the goals section; instead of a single goal item, there is a rule goal with variables, such as $R_2(A \to \alpha)$, giving the value of items in the new grammar.

An item of the form $[A \rightarrow BC\alpha]$ can derived if and only there is a derivation of the form

$$A \Rightarrow D \Rightarrow E \Rightarrow \cdots \Rightarrow F \Rightarrow BC\alpha$$

An item in this form is simply derived using the extension rule several times, combined with the N-ary rule. Similarly, an item of the form $[A \rightarrow a]$ can be derived if and only if there is a derivation of the form

$$A \Rightarrow D \Rightarrow E \Rightarrow \cdots \Rightarrow F \Rightarrow a$$

Items of the form $[A \rightarrow B]$ where B is a nonterminal cannot be derived.

A short example will help illustrate how this grammar transformation works. Consider the following grammar, with values in the inside semiring:

$$\begin{array}{lcll} S & \rightarrow & Aa & (1.0) \\ A & \rightarrow & a & (0.5) \\ A & \rightarrow & A & (0.5) \end{array} \tag{2.19}$$

There are an infinite number of derivations of the item $[A \rightarrow a]$. It can be derived using just the unary rule, with value 0.5; it can be derived using the unary rule and the extension rule, with value 0.25; it can be derived using the unary rule and using the extension rule twice, with value 0.125; and so on. The total of all derivations is 1.0. There is just one derivation for the item $[S \rightarrow Aa]$, which has value 1.0 also. Thus, the resulting grammar is:

$$\begin{array}{lcll} S & \rightarrow & Aa & (1.0) \\ A & \rightarrow & a & (1.0) \end{array} \tag{2.20}$$

which has no nonterminal unary rules.

Now, with an example finished, we can discuss conditions for correctness for a grammar transformation.

Consider a grammar derivation D (as always, left-most) in the original grammar, and a grammar derivation E in the new grammar, using item derivations I to derive the rules used in E. Roughly, if there is a one-to-one pairing between old-grammar derivations D and pairs (E, I), then, the transformation is value preserving. Formally,

Theorem 2.8

Consider a derivation D in the original grammar, using rules $D_1...D_d$. Consider also a derivation in the new grammar E using rules $E_1...E_e$. For each new grammar rule E_i, there is some set of item-based derivations of that rule, $I_i^1...I_i^{j_i}$. We can consider sequences of such rule derivations, $I_1^{k_1}, I_2^{k_2}, ..., I_e^{k_e}$, selecting one rule derivation $I_i^{k_i}$ for each new grammar rule E_i. A grammar transformation will be value preserving for a commutative semiring if there is a one-to-one pairing between derivations $D_1...D_d$ and pairs $(E_1...E_e, I_1^{k_1}...I_e^{k_e})$ and if all rule values (from the original grammar) occur the same number of times in $I_1^{k_1}...I_e^{k_e}$ as they do in $D_1...D_d$.

Proof The proof is obvious, since each term in the sum over derivations in the original grammar has a term in the sum over derivations in the new grammar. The details essentially follow Theorem 2.1. $\diamond$

There are two important caveats to note about this form of grammar transformation. The first is that the grammar transformation is semiring specific. Consider the probabilistic grammar example, 2.19, transformed with the unary productions removal transformation. If we transform it using the inside semiring, we get Grammar 2.20. On the other hand, if we transform the grammar using the Viterbi semiring, we get

$$\begin{array}{lcll} S & \rightarrow & Aa & (1.0) \\ A & \rightarrow & a & (0.5) \end{array}$$

which is also correct: the Viterbi semiring value of the string aa in the original grammar is 0.5, just as it is in the transformed grammar. Notice that the values differ; it is important to remember that grammar transformations are semiring specific. Furthermore, notice that the probabilities do not sum to 1 in the transformed grammar in the Viterbi semiring. While for this example, and the unary productions removal transformation in general, transformations using the inside ring do preserve summation to 1, this is not always true, as we will show during our discussion of the epsilon removal transformation.

Next, we consider the epsilon removal transformation. This transformation is fairly simple; it is derived from Earley's algorithm. This transformation assumes S does not occur on the right hand side of any productions. There is one item form, $[A \rightarrow \alpha \bullet \beta]$ that can be derived if and only if there is a derivation of the form $A \rightarrow \gamma\beta \stackrel{*}{\Rightarrow} \alpha\beta$, using only substitutions of the form $B \stackrel{*}{\Rightarrow} \epsilon$, and where each symbol C in α can derive some string of terminals.

Item form:
$[A \to \alpha \bullet \beta]$

Rule Goal
$R_1(A \to \alpha)$

Rules:

$$\frac{R_0(A \to \alpha)}{[A \to \bullet \alpha]} \qquad \text{Prediction}$$

$$\frac{[A \to \alpha \bullet B\beta] \qquad [B \to \bullet]}{[A \to \alpha \bullet \beta]} \qquad \text{Epsilon Completion}$$

$$\frac{[A \to \alpha \bullet B\beta]}{[A \to \alpha B \bullet \beta]} [B \to C\gamma \bullet] \qquad \text{Non-epsilon Completion}$$

$$\frac{[A \to \alpha \bullet a\beta]}{[A \to \alpha a \bullet \beta]} \qquad \text{Scanning}$$

$$\frac{[A \to B\alpha \bullet]}{R_1(A \to B\alpha)} \qquad \text{N-ary Output}$$

$$\frac{[S \to \bullet]}{R_1(S \to \epsilon)} \qquad \text{Epsilon Output}$$

Figure 2.14: Removal of Epsilon Productions

There are six rules. The first, prediction, simply makes sure there is one initial item, $[A \to \bullet\alpha]$ for each rule of the form $A \to \alpha$. The next rule, epsilon completion, allows deletion of symbols B that derive epsilon from rules of the form $A \to \alpha B\beta$, while the following rule, non-epsilon completion, simply moves the dot over symbols B that can derive terminal strings. The fourth rule, scanning, moves the dot over terminals. The last two rules, n-ary output and epsilon output, derive the output values.

For simplicity, the removal of unary productions transformation and the removal of n-ary productions transformations do not handle the rule $S \to \epsilon$ produced by the epsilon output rule, but could easily be modified to do so.

While the epsilon removal transformation preserves the value of any string using the inside semiring, it does not in general produce a grammar with probabilities that sum to 1. Consider the probabilistic grammar

$$\begin{array}{lcll} S & \to & aB & (1.0) \\ B & \to & \epsilon & (0.7) \\ B & \to & b & (0.3) \end{array}$$

The grammar with epsilons removed using the inside semiring will be

$$\begin{array}{lcll} S & \to & aB & (1.0) \\ S & \to & a & (0.7) \\ B & \to & b & (0.3) \end{array}$$

which generates the same strings with the same probabilities: value is preserved; notice however that probabilities of individual nonterminals sum to both more and less than one and that simply normalizing probabilities by dividing through by the total for each left hand side leads to a grammar with different string probabilities. The lack of summation to one could potentially make this grammar less useful in a system that needed, for instance, intermediate probabilities for thresholding.

We can correctly renormalize using a modified item-based description of Earley's algorithm, in Figure 2.15. This parser is just like Earley's parser, except that the indices have been removed, and the scanning rule does not make reference to the words of the sentence. There is one valid derivation in this parser for each derivation in the grammar. Computing

$$\frac{V([A \to \alpha \bullet]) \times Z([A \to \alpha \bullet])}{\sum_{\beta} V([A \to \beta \bullet]) \times Z([A \to \beta \bullet])}$$

Item form:
$[A \rightarrow \alpha \bullet \beta]$

Goal:
$[S' \rightarrow S \bullet]$

Rules:

$$\frac{}{[S' \rightarrow \ \bullet S]} \qquad \text{Initialization}$$

$$\frac{[A \rightarrow \alpha \bullet a\beta]}{[A \rightarrow \alpha a \bullet \beta]} \qquad \text{Scanning}$$

$$\frac{R(B \rightarrow \gamma)}{[B \rightarrow \ \bullet \gamma]} [A \rightarrow \alpha \bullet B\beta] \qquad \text{Prediction}$$

$$\frac{[A \rightarrow \alpha \bullet B\beta] \qquad [B \rightarrow \gamma \bullet]}{[A \rightarrow \alpha B \bullet \beta]} \qquad \text{Completion}$$

Figure 2.15: Renormalization Parsing

gives the normalized probability $P(A \rightarrow \alpha)$. The intuition behind this formula is simply that it is the usual formula for inside-outside re-estimation, and inside-outside re-estimation, when in a local minimum, stays in the same place. Since a grammar is a local minimum of itself, we should get essentially the same grammar, but with normalized rule probabilities. A stronger argument is given in Appendix 2–A, Theorem 2.9.

The renormalization parser has some other useful properties. For instance, in the counting semiring, $V([S' \rightarrow S \bullet])$ gives the total number of parses in the language; in the Viterbi-n-best semiring, it gives the probabilities and parses of the n most probable parses in the language; and in the parse-forest semiring, it gives a derivation forest for the entire language. In the inside semiring, it should always be 1.0, assuming a proper grammar as input.

One last interesting property of the renormalization parser is that it can be used to remove useless rules. The forward boolean value times the reverse boolean value of an item $[A \rightarrow \alpha \bullet]$ will be *TRUE* if and only if the rule $A \rightarrow \alpha$ is useful; that is, if it can appear in the derivation of some string. Useless rules can be eliminated.

For completeness, there are two more steps in the CNF transformation: conversion from n-ary branching rules ($n \geq 2$) to binary branching rules, and conversion from binary

Item form:

Rule Goal
$R_3(\alpha \rightarrow \beta)$

Rules:

$$\frac{R_2(A \rightarrow B\ C)}{R_3(A \rightarrow B\ C)}$$

$$\frac{R_2(A \rightarrow B\ C\ D\ \alpha)}{R_3(A \rightarrow B\ \langle CD\alpha\rangle)}$$

$$\frac{}{R_3(\langle CD\rangle \rightarrow C\ D)}\ R_2(A \rightarrow B\alpha CD)$$

$$\frac{}{R_3(\langle CDE\alpha\rangle \rightarrow C\ \langle DE\alpha\rangle)}\ R_2(A \rightarrow B\beta CDE\alpha)$$

Figure 2.16: Removal of n-ary Productions

branching rules with terminal symbols to those with nonterminals. We show how to convert from n-ary to binary branching rules in Figure 2.16. This transformation assumes that all unary and ϵ rules have been removed. We construct many new nonterminals in this transformation, each of the form $\langle CD\alpha\rangle$. This step is fairly simple, so we won't explicate it. The remaining step, removal of terminal symbols from binary branching rules, is trivial and we do not present it.

We should note that not all transformations have a value-preserving version. For instance, in the general case, the transformation of a non-deterministic finite state automaton (NFA) into a deterministic finite state automaton (DFA) cannot be made value preserving: the problem is that in the DFA, there is exactly one derivation for a given input string, so it is not possible to get a one-to-one correspondence between derivations in the NFA and the DFA, meaning that these techniques cannot be used. (Mohri (1997) discusses the conditions under which some NFAs in certain semirings can be made deterministic.)

2.7 Examples

In this section, we give examples of several parsing algorithms, expressed as item-based descriptions.

2.7.1 Finite State Automata and Hidden Markov Models

NFAs and HMMs can both be expressed using a single item-based description, shown in Figure 2.17. We will express transitions from state A to state B emitting symbol a as $A \rightarrow a, B$. As usual, we will let $R(A \rightarrow a, B)$ have different values depending on the semiring used.

Boolean	*TRUE* if there is a transition from A to B emitting a, *FALSE* otherwise
Counting	1 if there is a transition from A to B emitting a, 0 otherwise
Derivation	$\{\langle A \rightarrow a, B\rangle\}$ if there is a transition from A to B emitting a, $\emptyset$ otherwise
Viterbi	Probability of a transition from A to B emitting a
Inside	Probability of a transition from A to B emitting a

We assume there is a single start state S and a single final state F, and we allow ϵ transitions.

For HMMs, notice that the forward algorithm is obtained simply by using the inside semiring; the backwards algorithm is obtained using the reverse values of the inside semiring; and the Viterbi algorithm is obtained using the Viterbi semiring. For NFAs, we can use the boolean semiring to determine whether a string is in the language of an NFA; we can use the counting semiring to determine how many state sequences there are in the NFA for a given string; and we can use the derivation forest semiring to get a compact representation of all state sequences in an NFA for an input string.

2.7.2 Prefix Values

For language modeling, it may be useful to compute the prefix probability of a string. That is, given a string $w_1...w_n$, we may wish to know the total probability of all sentences beginning with that string,

$$\sum_{k \geq 0, x_1, ... x_k} P(S \rightarrow w_1...w_n x_1...x_k)$$

Item form:
$[A, i]$

Goal
$[F, n+1]$

Rules:

$$\frac{}{[S,1]} \quad \text{Start Axiom}$$

$$\frac{[A,i] \qquad R(A \rightarrow w_i, B)}{[B, i+1]} \quad \text{Scanning}$$

$$\frac{[A,i] \qquad R(A \rightarrow \epsilon, B)}{[B, i]} \quad \text{Epsilon Scanning}$$

Figure 2.17: NFA/HMM parser

Jelinek and Lafferty (1991) and Stolcke (1993) both give algorithms for computing the prefix probabilities. However, the derivations are somewhat complex, requiring five pages of Jelinek and Lafferty's nine page paper.

In contrast, we give a fairly simple item-based description in Figure 2.18, which we will explicate in detail below. We will call a *prefix derivation* $X \stackrel{pre}{\Rightarrow} w_i...w_j$ a derivation in which $X \stackrel{*}{\Rightarrow} w_i...w_j x_1 x_2 ... x_k$.

The prefix value of a string is the sum of the products of the values used in the prefix derivation. A brief example will help: consider the prefix a and the grammar

$$\begin{array}{lclc} S & \rightarrow & sA & (1) \\ A & \rightarrow & a & (0.3) \\ A & \rightarrow & b & (0.7) \end{array}$$

There are two prefix derivations of s of the form: $S \stackrel{*}{\Rightarrow} sA \stackrel{*}{\Rightarrow} sa$ (0.3) and $S \stackrel{*}{\Rightarrow} sA \stackrel{*}{\Rightarrow} sb$ (0.7). The inside score is the sum, 1, while the Viterbi score is the max, 0.7.

Figure 2.18 gives an item-based description for finding prefix values for CNF grammars; it would be straightforward to modify this algorithm for an Earley-style parser, allowing it to handle any CFG, as shown by Stolcke (1993).

There are three item types in this description. The first item type, $[i, A, j]$, called *In*, can be derived only if $A \stackrel{*}{\Rightarrow} w_i...w_{j-1}$. The second type, $[i, A]$, called *Between*, can be

Item form:

$[i, A, j]$
$[i, A]$
$[A]$

Goal

$[1, S]$

Rules:

$$\frac{R(A \to w_i)}{[i, A, i{+}1]} \quad \text{Unary In}$$

$$\frac{R(A \to BC) \qquad [i, B, k] \qquad [k, C, j]}{[i, A, j]} \quad \text{In In}$$

$$\frac{R(A \to a)}{[A]} \quad \text{Unary Out}$$

$$\frac{R(A \to BC) \qquad [B] \qquad [C]}{[A]} \quad \text{Out Out}$$

$$\frac{R(A \to w_n)}{[n, A]} \quad \text{Unary Between}$$

$$\frac{R(A \to BC) \qquad [i, B, j] \qquad [j, C]}{[i, A]} \quad \text{In Between}$$

$$\frac{R(A \to BC) \qquad [i, B] \qquad [C]}{[i, A]} \quad \text{Between Out}$$

Figure 2.18: Prefix Derivation Rules

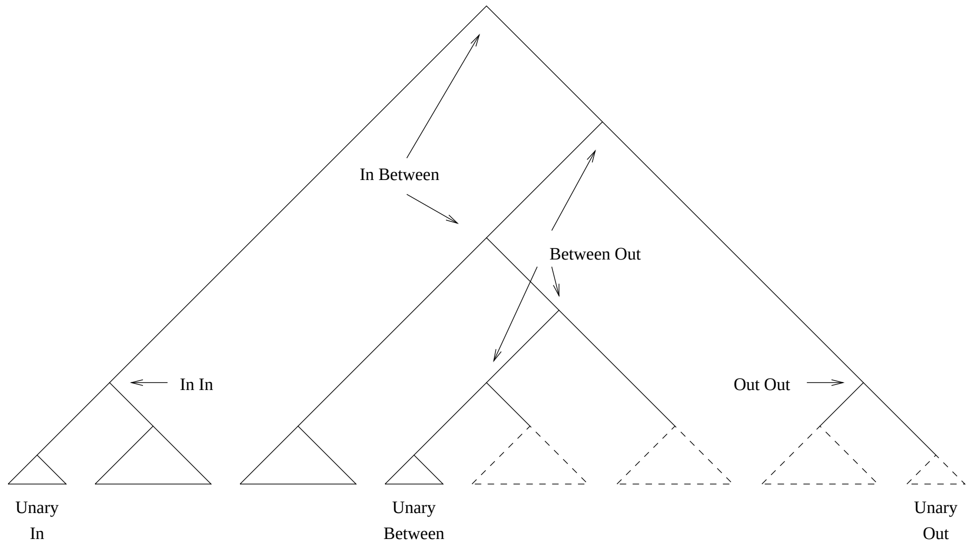


Figure 2.19: Prederivation Illustration

derived only if $A \stackrel{*}{\Rightarrow} w_i...w_n x_1...x_k$. The final type, $[A]$, called *Out*, can be derived only if $A \stackrel{*}{\Rightarrow} x_1...x_k$. The Unary In and In In rules correspond to the usual unary and binary rules. The Unary Out and Out Out rules correspond to the usual unary and binary rules, but for symbols after the prefix. For instance, the Out Out rule says that if $A \rightarrow BC$ and $B \stackrel{*}{\Rightarrow} x_1...x_k$ and $C \stackrel{*}{\Rightarrow} y_1...y_l$ then $A \stackrel{*}{\Rightarrow} x_1...x_k y_1...y_l$. The last three rules deal with Between items. Unary Between says that if $A \rightarrow w_n$ then $A \stackrel{*}{\Rightarrow} w_n$. In Between says that if $A \rightarrow BC$ and $B \stackrel{*}{\Rightarrow} w_i...w_{j-1}$ and $C \stackrel{*}{\Rightarrow} w_j...w_n x_1...x_k$ then $A \stackrel{*}{\Rightarrow} w_i...w_n x_1...x_k$. Finally, Between Out says that if $A \rightarrow BC$ and $B \stackrel{*}{\Rightarrow} w_i...w_n x_1...x_k$ and $C \stackrel{*}{\Rightarrow} y_1...y_l$ then $A \stackrel{*}{\Rightarrow} w_i...w_n x_1...y_l$. Figure 2.19 illustrates the seven different rules. Words in the prefix are indicated by solid triangles, and words that could follow the prefix are indicated by dashed ones.

There is one problem with the item-based description of Figure 2.18. Both the Out and the Between items are all associated with looping buckets. Since the Between items can only be computed on line (i.e. only once we know the input sentence), this means that we must perform time consuming infinite sums on line. It turns out that with a few modifications, the values of all items associated with looping buckets can be computed off-line, once per grammar, significantly speeding up the on-line part of the computation.

Figure 2.20 gives a faster item-based description. The fast version contains a new item type, $[A \stackrel{pre}{\Rightarrow} B]$ that can be derived only if there is a derivation of the form $A \stackrel{*}{\Rightarrow} Bx_1...x_k$. It turns out that for the fast description, we need to compute right-most derivations rather

Item form:

$[i, A, j]$

$[i, A]$

$[A]$

$[A \overset{pre}{\Rightarrow} B]$

Goal

$[1, S]$

Rules:

$$\frac{R(A \rightarrow w_i)}{[i, A, i{+}1]} \qquad \text{Unary In}$$

$$\frac{R(A \rightarrow BC) \qquad [k, C, j] \qquad [i, B, k]}{[i, A, j]} \qquad \text{In In}$$

$$\frac{R(A \rightarrow a)}{[A]} \qquad \text{Unary Out}$$

$$\frac{R(A \rightarrow BC) \qquad [C] \qquad [B]}{[A]} \qquad \text{Out Out}$$

$$\frac{}{[A \overset{pre}{\Rightarrow} A]} \qquad \text{Prederivation Axiom}$$

$$\frac{R(A \rightarrow BC) \qquad [C] \qquad [B \overset{pre}{\Rightarrow} D]}{[A \overset{pre}{\Rightarrow} D]} \qquad \text{Prederivation Completion}$$

$$\frac{[A \overset{pre}{\Rightarrow} B] \qquad R(B \rightarrow w_n)}{[n, A]} \qquad \text{Between Initialization}$$

$$\frac{[A \overset{pre}{\Rightarrow} B] \qquad R(B \rightarrow CD) \qquad [j, D] \qquad [i, C, j]}{[i, A]} \qquad \text{Between Continuation}$$

Figure 2.20: Fast Prefix Derivation Rules

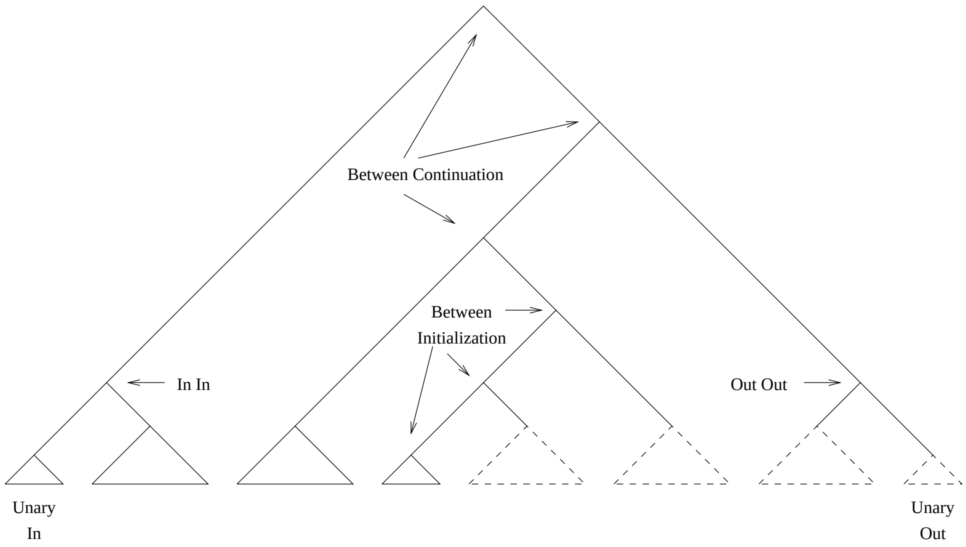


Figure 2.21: Fast Prederivation Illustration

than left-most derivations. There are eight deduction rules, the first four of which are the same as before, modified for right-most derivations. The next two rules, the Prederivation Axiom, and Prederivation Completion, compute all items of the form $[A \overset{pre}{\Rightarrow} B]$. The last rule, Between Continuation is the most complicated. The idea behind this rule is the following. In our previous implementation, there would be an In Between rule followed by a string of Between Out rules. It was because of the Between Out rules that the Between items were associated with looping buckets. In the fast version, we collapse the In Between followed by a string of many Between Outs into a single Between Continuation rule. Between Continuation is the same as In Between, except with $[A \overset{pre}{\Rightarrow} B]$ prepended; the $[A \overset{pre}{\Rightarrow} B]$ is essentially equivalent to a string of zero or more Between Out rules. It is prepended rather than appended because we are now finding right-most derivations. We use a similar trick for Unary Between; in the original description, there could be a Unary Between rule followed by a string of Between Out rules. Again, we collapse the string of Between Out rules using a single item of the form $[A \overset{pre}{\Rightarrow} B]$. Figure 2.21 shows a schematic tree with examples of the rules.

We can give a more formal justification for each of the last four rules. The Prederivation Axiom says that for all A, $A \overset{*}{\Rightarrow} A$. The Prederivation Completion rule says that if $A \rightarrow BC$ and $B \overset{*}{\Rightarrow} Dx_1...x_k$ and $C \overset{*}{\Rightarrow} y_1...y_l$ then $A \overset{*}{\Rightarrow} Dx_1...x_k y_1...y_l$. The Between Initialization rule says that if $A \overset{*}{\Rightarrow} Bx_1...x_k$ and $B \rightarrow w_n$ then $A \overset{*}{\Rightarrow} w_n x_1...x_k$. Finally, Between

Continuation says that if $A \stackrel{*}{\Rightarrow} Bx_1...x_k$ and $B \rightarrow CD$ and and $C \stackrel{*}{\Rightarrow} w_i...w_{j-1}$ and $D \stackrel{*}{\Rightarrow} w_j...w_ny_1...y_l$ then $A \stackrel{*}{\Rightarrow} w_i...w_ny_1...y_lx_1...x_k$.

The careful reader can verify that there is a one-to-one correspondence between item derivations in this system, and prefix derivations of the input string. Notice that the only items associated with a looping bucket are of the form $[A \stackrel{pre}{\Rightarrow} B]$ and $[A]$; these can all be precomputed, independent of the input string. This algorithm is essentially the same as that of Jelinek and Lafferty (1991).

As usual, there are advantages to the item based description, besides its simplicity. For instance, we can use the same item-based description with the Viterbi semiring to find the Viterbi prefix probabilities: the probability of the most likely prefix derivation; with the boolean semiring to compute the valid prefix property: whether $S \stackrel{pre}{\Rightarrow} w_1...w_n$, etc.

Prefix derivation values are potentially useful for starting interpretation of sentences before they are completed. For instance, a travel agent program, on hearing "Show me flights on April 21 so I can" could compute the Viterbi-derivation prefix parse so that it could begin processing the transaction before the user was finished speaking. Similar values, such as the prefix derivation-forest value, would yield the set of all possible derivations that could complete the sentence.

We note that in the inside semiring,

$$\frac{V([i,A,j]) \times Z([i,A,j])}{V([1,S])} = \sum_{k,x_1,..,x_k} P(S \stackrel{*}{\Rightarrow} w_1...w_{i-1}Aw_j...w_nx_1...x_k \stackrel{*}{\Rightarrow} w_1...w_nx_1...x_k | w_1...w_n)$$

which is the probability that symbol A covers terminals i to $j-1$ given all input symbols seen so far. From an information content point of view, this formula is the optimal one to use for thresholding, although the resulting algorithm would be $O(n^4)$, since the outside values, which require time $O(n^3)$ to compute, would need to be recomputed after each new input symbol. Thus, although thresholding with this formula probably would not provide a speedup, it could be useful for incremental interpretation, or for analyzing garden path sentences.

2.7.3 Beyond Context-Free

There has been quite a bit of previous work on the intersection of formal language theory and algebra, as described by Kuich (1997), among others. This previous work has made heavy use of the fact that there is a strong correspondence between algebraic equations in certain non-commutative semirings, and CFGs. This correspondence has made it possible to manipulate algebraic systems, rather than grammar systems, simplifying many operations.

On the other hand, there is an inherent limit to such an approach, namely a limit to context-free systems. It is then perhaps slightly surprising that we can avoid these limitations, and create item-based descriptions of parsers for weakly context-sensitive grammars, such as Tree Adjoining Grammars (TAGs). We avoid the limitations of previous approaches using two techniques. One technique is, rather than computing parse trees for TAGs, we compute derivation trees. Computing derivation trees for TAGs is significantly easier than computing parse trees, since the derivation trees are context-free. The other trick we use is that while earlier formulations created one set of equations for each grammar, our parsing approach can be thought of as creating a set of equations for each grammar and string length. Because the number of equations grows with the string length, we can recognize strings in weakly context-sensitive languages.

A further explication of this subject, including an item-based description for a simple TAG parser is given in the appendix, in Section 2–B.2.

2.7.4 Tomita Parsing

Our goal in this section has been to show that item-based descriptions can be used to simply describe almost all parsers of interest. One parsing algorithm that would seem particularly difficult to describe is Tomita's graph-structured-stack LR parsing algorithm. This algorithm at first glance bears little resemblance to other parsing algorithms, and worse, uses pointers extensively. Since there is no obvious way to emulate pointers in an item-based description, it would appear that this parser has no simple item-based description. However, Sikkel (1993) gives an item-based description for a Tomita-style parser for the boolean semiring, which is also more efficient than Tomita's algorithm. Sikkel's format is similar enough to ours that his description can be easily converted to our format, where it can be used for ω-continuous semirings in general.

2.7.5 Graham Harrison Ruzzo Parsing

Graham *et al.* (1980) describe a parser similar to Earley's, but with several speedups that lead to significant improvements. Essentially, there are three improvements in the GHR parser. First, epsilon productions are precomputed. Second, unary productions are precomputed; and, finally, completion is separated into two steps, allowing better dynamic programming.

In Appendix 2–B.1, we give a full item-based description of a GHR parser. The forward values of many of the items in our parser related to unary and epsilon productions can be computed off-line, once per grammar. This idea of precomputing values off-line in a probabilistic GHR-style parser is due to Stolcke (1993). Since reverse values require entire strings, the reverse values of these items cannot be computed until the input string is known. Because we use a single item-based description for precomputed items and non-precomputed items, and for forward and reverse values, this combination of off-line and on-line computation is easily and compactly specified.

2.8 Previous Work

The previous work in this area is extensive, including work in deductive parsing, work in statistical parsing, and work in the combination of formal language theory and algebra. This chapter can be thought of as synthetic, combining the work in all three areas, although in the course of synthesis, several general formulas have been found, most notably the general formula for reverse values. A comprehensive examination of all three areas is beyond the scope of this chapter, but we can touch on a few significant areas of each.

First, there is the work in deductive parsing. This work in some sense dates back to Earley (1970), in which the use of items in parsers is introduced. More recent work (Pereira and Warren, 1983; Pereira and Shieber, 1987) demonstrates how to use deduction engines for parsing. Finally, both Shieber *et al.* (1993) and Sikkel (1993) have shown how to specify parsers in a simple, interpretable, item-based format. This format is roughly the format we have used here, although there are differences due to the fact that their work was strictly in the boolean semiring.

Work in statistical parsing has also greatly influenced this work. We can trace this work back to research in HMMs by Baum and his colleagues (Baum and Eagon, 1967; Baum, 1972). A good introduction to this work (and its practical application to speech recognition) was written by Rabiner (1989). In particular, the work of Baum developed the concept of backward probabilities (in the inside semiring), as well as many of the techniques for computing in the inside semiring. Viterbi (1967) developed corresponding algorithms for computing in the Viterbi semiring. Baker (1979) extended the work of Baum *et al.* to PCFGs, including to computation of the outside values (or reverse inside values in our terminology.) Baker's work is described by Lari and Young (1990; 1991). Baker's work was only for PCFGs in Chomsky Normal Form, avoiding the need to compute infinite summations. Jelinek and Lafferty (1991) showed how to compute some of the infinite summations in the inside semiring, those needed to compute the prefix probabilities of PCFGs in CNF. Stolcke (1993) showed how to use the same techniques to compute inside probabilities for Earley parsing, dealing with the difficult problems of unary transitions, and the more difficult problems of epsilon transitions. He thus solved all of the important problems encountered in using an item-based parser to compute the inside and outside values (forward and reverse inside values); he also showed how to compute the forward Viterbi values.

The final area of work is in formal language theory and algebra. Although it is not widely known, there has been quite a bit of work showing how to use formal power series to elegantly derive results in formal language theory. In particular, the major classic results can be derived in this framework, but with the added benefit that they typically apply to all ω-continuous semirings, or at least to all commutative ω-continuous semirings. The most accessible introduction to this literature we have found is by Kuich (1997). There are also books by Salomaa and Soittola (1978) and Kuich and Salomaa (1986), as well as a book concentrating primarily on the finite state case by Berstel and Reutenauer (1988). This work dates back to to Chomsky and Schützenberger (1963). Kuich (1997) gives a much more complete bibliography than we give here.

One piece of work deserves special mention. Teitelbaum (1973) showed that any semiring could be used in the CKY algorithm. He further showed that a subset of complete semirings could be used in value-preserving transformations to CNF. Thus, he laid the foundation for much of the work that followed.

In summary, this chapter synthesizes work from several different related fields, including deductive parsing, statistical parsing, and formal language theory; we emulate and expand on the earlier synthesis of Teitelbaum. The synthesis here is powerful: by generalizing and integrating many results, we make the computation of a much wider variety of values possible.

2.8.1 Recent similar work

There has also been recent similar work by Tendeau (1997b; 1997a). Tendeau (1997b) gives an Earley-like algorithm that can be adapted to work with complete semirings satisfying certain conditions. Unlike our version of Earley's algorithm, Tendeau's version requires time $O(n^{L+1})$ where L is the length of the longest right hand side, as opposed to $O(n^3)$ for the classic version, and for our description. There is also not much detail about how the algorithm handles looping productions.

Tendeau (1997a) also shows how to compute values in an abstract semiring with a CKY style algorithm, in a manner similar to Teitelbaum (1973). More importantly, he gives a generic description for dynamic programming algorithms. His description is very similar to our item-based descriptions with two caveats. First, he includes a single rule term, on the left, rather than a set of rule values intermixed with item values. For commutative semirings this limitation is fine, but for non-commutative semirings, it may lead to inelegancies. Second, he does not have an equivalent to our side conditions. This means that algorithms such as Earley's algorithm which rely on side conditions for efficiency cannot be described in this formalism in a way that captures those efficiency considerations.

Tendeau (1997b; 1997a) introduces a parse forest semiring, similar to our derivation forest semiring, in that it encodes a parse forest succinctly. Tendeau's parse forest has an advantage over ours in that it is commutative. However, it has two disadvantages. First, it is partially because of the encoding of this parse forest that Tendeau's version of Earley's algorithm has its poor time complexity. Second, to implement this semiring, Tendeau's version of rule value functions take as their input not only a nonterminal, but also the span that it covers; this is somewhat less elegant than our version. Tendeau (1997b) also shows that Definite Clause Grammars (DCGs) can be described as semirings, with some caveats, including the same problems as the parse forest semiring.

2.9 Conclusion

In this chapter, we have shown that a simple item-based description format can be used to describe a very wide variety of parsers. These parsers include the CKY algorithm, Earley's algorithm, prefix probability computation, a TAG parsing algorithm, Graham, Harrison, Ruzzo parsing, and HMM computations. We have shown that this description format makes it easy to find parsers that compute values in any ω-continuous semiring. The same description can be used to find reverse values in commutative ω-continuous semirings, and in many non-commutative ones as well. We have also shown that this description format can be used to describe grammar transformations, including transformations to CNF and GNF, which preserve values in any commutative ω-continuous semiring.

While theoretical in nature, this chapter is of some practical value. There are three reasons the results of this chapter would be used in practice: first, these techniques make computation of the outside values simple and mechanical; second, these techniques make it easy to show that a parser will work in any ω-continuous semiring; and third, these techniques isolate computation of infinite sums in a given semiring from the parser specification process.

Probably the way in which these results will be used most is to find formulas for outside values. For parsers such as CKY parsers, finding outside formulas is not particularly burdensome, but for complicated parsers such as TAG parsers, Graham, Harrison, Ruzzo parsers, and others, it can require a fair amount of thought to find these equations through conventional reasoning. With these techniques, the formulas can be found in a simple mechanical way.

The second advantage comes from clarifying the conditions under which a parser can be converted from computing values in the boolean semiring (a recognizer) to computing values in any ω-continuous semiring. We should note that because in the boolean semiring, infinite summations can be computed trivially ($TRUE$ if any element is $TRUE$) and because repeatedly adding a term does not change results, it is not uncommon for parsers that work in the boolean semiring to require significant modification for other semirings. For parsers like CKY parsers, verifying that the parser will work in any semiring is trivial, but for other parsers the conditions are more complex. With the techniques in this chapter, all that is necessary is to show that there is a one-to-one correspondence between item derivations and grammar derivations. Once that correspondence has been shown, Theorem 2.1 states that any ω-continuous semiring can be used.

The third use of this chapter is to separate the computation of infinite sums from the main parsing process. Infinite sums can come from several different phenomena, such as loops from productions like $A \to A$; productions involving ϵ; and sometimes left recursion. In traditional procedural specifications, the solution to these difficult problems is intermixed with the parser specification, and makes the parser specific to semirings using the same techniques for solving the summations.

It is important to notice that the algorithms for solving these infinite summations vary fairly widely, depending on the semiring. On the one hand, boolean infinite summations are nearly trivial to compute. For other semirings, such as the counting semiring, or derivation forest semiring, more complicated computations are required, including the detection of loops. Finally, for the inside semiring, in most cases only approximate techniques can be used, although in some cases, matrix inversion can be used. Thus, the actual parsing algorithm, if specified procedurally, can vary quite a bit depending on the semiring.

On the other hand, using our techniques makes infinite sums easier to deal with in two ways. First, these difficult problems are separated out, relegated conceptually to the parser interpreter, where they can be ignored by the constructor of parsing algorithms. Second, because they are separated out, they can be solved once, rather than again and again. Both of these advantages make it significantly easier to construct parsers.

In summary, the techniques of this chapter will make it easier to compute outside values, easier to construct parsers that work for any ω-continuous semiring, and easier to compute infinite sums in those semirings. In 1973, Teitelbaum wrote:

> We have pointed out the relevance of the theory of algebraic power series in non-commuting variables in order to minimize further piecemeal rediscovery.

Many of the techniques needed to parse in specific semirings continue to be rediscovered, and outside formulas are derived without observation of the basic formula given here. We hope this chapter will bring about Teitelbaum's wish.

Appendix

2–A Additional Proofs

In this appendix, we prove theorems given earlier. It will be helpful to refer back to the original statements of the theorems for context.

Theorem 2.3

For x an item in a looping bucket B, and for $g \geq 1$,

$$V_{\leq g}(x, B) = \bigoplus_{a_1...a_k \text{ s.t. } \frac{a_1...a_k}{x}} \bigotimes_{i=1..k} \begin{cases} V(a_i) \text{ if } a_i \notin B \\ V_{\leq g-1}(a_i, B) \text{ if } a_i \in B \end{cases}$$

Proof Observe that for any item x in a bucket preceding B, $inner_{\leq g}(x, B) = inner(x)$. Then

$$\begin{aligned}
V_{\leq g}(x, B) &= \bigoplus_{D \in inner_{\leq g}(x,B)} V(x) \\
&= \bigoplus_{a_1...a_k \text{ s.t. } \frac{a_1...a_k}{x}} \bigoplus_{\substack{D_{a_1} \in inner_{\leq g-1}(a_1,B),..., \\ D_{a_k} \in inner_{\leq g-1}(a_k,B)}} V(\langle x : (D_{a_1}, ..., D_{a_k}\rangle) \\
&= \bigoplus_{a_1...a_k \text{ s.t. } \frac{a_1...a_k}{x}} \bigoplus_{\substack{D_{a_1} \in inner_{\leq g-1}(a_1,B),..., \\ D_{a_k} \in inner_{\leq g-1}(a_k,B)}} \bigotimes_{i=1..k} V(D_{a_i}) \\
&= \bigoplus_{a_1...a_k \text{ s.t. } \frac{a_1...a_k}{x}} \bigotimes_{i=1..k} \bigoplus_{D_{a_i} \in inner_{\leq g-1}(a_i,B)} V(D_{a_i}) \\
&= \bigoplus_{a_1...a_k \text{ s.t. } \frac{a_1...a_k}{x}} \bigotimes_{i=1..k} V_{\leq g-1}(a_i, B)
\end{aligned} \tag{2.21}$$

Now, for elements $x \notin B$, we can substitute Equation 2.7 into Equation 2.21 to yield

$$V_{\leq g}(x, B) = \bigoplus_{a_1...a_k \text{ s.t. } \frac{a_1...a_k}{x}} \bigotimes_{i=1..k} \begin{cases} V(a_i) \text{ if } a_i \notin B \\ V_{\leq g-1}(a_i, B) \text{ if } a_i \in B \end{cases}$$

completing the proof. $\diamond$

Theorem 2.6

For items $x \in B$ and $g \geq 1$,

$$Z_{\leq g}(x, B) = \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} \left(\bigotimes_{i=1, \overset{-j}{\ldots}, k} V(a_i) \right) \otimes \begin{cases} Z_{\leq g-1}(b, B) & \textit{if } b \in B \\ Z(b) & \textit{if } b \notin B \end{cases}$$

Proof Define $\mathit{MakeOuter}(j, D_{a_1}, \overset{-j}{\ldots}, D_{a_k}, D_b)$ to be a function that puts together the specified trees to form an outer tree for a_j. Then,

$$\begin{aligned}
& Z_{\leq g}(x, B) \\
&= \bigoplus_{D \in \mathit{outer}_{\leq g-1}(x, B)} Z(D) \\
&= \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} \; \bigoplus_{\substack{D_{a_1} \in \mathit{inner}(a_1), \overset{-j}{\ldots}, \\ D_{a_k} \in \mathit{inner}(a_k), \\ D_b \in \mathit{outer}_{\leq g-1}(b, B)}} Z(\mathit{MakeOuter}(j, D_{a_1}, \overset{-j}{\ldots}, D_{a_k}, D_b)) \\
&= \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} \; \bigoplus_{\substack{D_{a_1} \in \mathit{inner}(a_1), \overset{-j}{\ldots}, \\ D_{a_k} \in \mathit{inner}(a_k), \\ D_b \in \mathit{outer}_{\leq g-1}(b, B)}} Z(D_b) \otimes \bigotimes_{i=1, \overset{-j}{\ldots}, k} V(D_{a_i}) \\
&= \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} \; \bigoplus_{\substack{D_{a_1} \in \mathit{inner}(a_1), \overset{-j}{\ldots}, \\ D_{a_k} \in \mathit{inner}(a_k)}} \; \bigoplus_{D_b \in \mathit{outer}_{\leq g-1}(b, B)} Z(D_b) \otimes \bigotimes_{i=1, \overset{-j}{\ldots}, k} V(D_{a_i}) \\
&= \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} \left(\bigoplus_{\substack{D_{a_1} \in \mathit{inner}(a_1), \overset{-j}{\ldots}, \\ D_{a_k} \in \mathit{inner}(a_k)}} \; \bigotimes_{i=1, \overset{-j}{\ldots}, k} V(D_{a_i}) \right) \otimes \bigoplus_{D_b \in \mathit{outer}_{\leq g-1}(b, B)} Z(D_b) \\
&= \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} \left(\bigotimes_{i=1, \overset{-j}{\ldots}, k} \; \bigoplus_{D_{a_i} \in \mathit{inner}(a_i, B)} V(D_{a_i}) \right) \otimes Z_{\leq g-1}(b, B) \\
&= \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} \left(\bigotimes_{i=1, \overset{-j}{\ldots}, k} V(a_i) \right) \otimes Z_{\leq g-1}(b, B)
\end{aligned} \tag{2.22}$$

Now, substituting Equation 2.17 into Equation 2.22, we get

$$Z_{\leq g}(x, B) = \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} \left(\bigotimes_{i=1,\ldots,\neg j,k} V(a_i) \right) \otimes \begin{cases} Z_{\leq g-1}(b, B) & \textit{if } b \in B \\ Z(b) & \textit{if } b \notin B \end{cases}$$

$\diamond$

Theorem 2.7

The forward semiring parser interpreter correctly computes the value of all items.

Sketch of proof We show that every item in each bucket has its value correctly computed. The proof is by induction on the buckets. The base case is the first bucket. Since items in each bucket depend only on the values of items in preceding buckets, items in the first bucket depend on no previous buckets. Now, if the first bucket is a looping bucket, the interpreter uses an implementation of Equation 2.8, previously shown correct in Theorem 2.3. If the first bucket is a non-looping bucket, the interpreter uses an implementation of Equation 2.5, previously shown correct in Theorem 2.5. Since these equations refer to items in the first bucket, and such items do not depend on values outside the first bucket, other than rule values, the first bucket is correctly computed.

For the inductive step, consider the current bucket of the loop. Assume that the values of all items in all previous buckets have been correctly computed. Then, depending on whether the current bucket is a looping bucket or not, either an implementation of Equation 2.8 or 2.5 is used. These equations only depend on other values in the current bucket and on values in previously correctly computed buckets, so the values in the current bucket are correctly computed.

Thus, by induction, the values of all items in all buckets are correctly computed. $\diamond$

We now discuss the renormalization parser of Figure 2.15. We claim that using the equation

$$P_{new}(A \rightarrow \alpha) = \frac{V([A \rightarrow \alpha \bullet]) \times Z([A \rightarrow \alpha \bullet])}{\sum_\beta V([A \rightarrow \beta \bullet]) \times Z([A \rightarrow \beta \bullet])} \tag{2.23}$$

yields new probabilities that produce the same trees with the same probabilities as the original grammar and that clearly, these probabilities sum to 1 for each nonterminal (which they may not have done in the original grammar). The intuition behind this statement is that Equation 2.23 is simply the inside-outside reestimation formula. We have applied it here to the probability distribution of all trees produced by the original grammar, but this should be very similar to applying the inside-outside formula to a very large (approaching infinite) sample of trees from that distribution. Since the inside-outside probabilities approach a local optimum, and since a grammar producing the same trees with the same probabilities will be optimal, we expect to achieve that grammar.

We will now give a formal proof that the resulting trees have the same probabilities as the original trees, subject to one caveat. In particular, we do not know how to show that the resulting probability distribution is tight, that is, that the sum of the probabilities of all of the trees equals 1. (Not all grammars are tight. For instance, $S \rightarrow SS(.9), S \rightarrow a(.1)$ is not tight.) Chi and Geman (1998) showed that CNF grammars produced by the inside-outside reestimation formula are tight, but it is not clear whether is has ever been shown for the case of grammars with unary productions.

Theorem 2.9
Assuming that the inside-outside reestimation formula produces tight grammars, and that the original grammar is tight, the grammar produced using the renormalization parser of Figure 2.15, and Equation 2.23 produces a grammar that produces an identical probability distribution over trees as the original grammar, but with each rule having a probability between 0 and 1.

Proof We will show by induction that for each left-most derivation D of length at most k, the value of all derivations starting with D is the same in the new grammar as the old.

The base case follows from our assumption that both the new grammar and the old grammar are tight, since the sum of all derivations starting with the empty derivation (k=0) is the sum of the probability of all trees, or, by assumption, 1 in both cases.

Next, for the inductive step, assume the theorem for $k-1$. Consider a left-most derivation D of length k. We can split D into a left-most derivation E of length $k-1$, followed by the single step $A \rightarrow \alpha$: $S \stackrel{E}{\Rightarrow} w_1 ... w_j A\gamma \Rightarrow w_1 ... w_j \alpha\gamma$. We show that the sum of the values of all derivations that begin this way is the same in both old and new grammars. Since the new grammar is, by assumption, tight, and has rule probabilities that sum to one, the probability of all derivations starting with D is just $P(D)$:

$$\begin{aligned} P_{new}(S \overset{D}{\Rightarrow} w_1...w_j\alpha\gamma) &= P_{new}(S \overset{E}{\Rightarrow} w_1...w_jA\gamma \Rightarrow w_1...w_j\alpha\gamma) \\ &= P_{new}(S \overset{E}{\Rightarrow} w_1...w_jA\gamma) \times P_{new}(A \rightarrow \alpha) \\ &= P_{new}(S \overset{E}{\Rightarrow} w_1...w_jA\gamma) \times \frac{V([A \rightarrow \alpha\bullet])}{\sum_\beta V([A \rightarrow \beta\bullet])} \end{aligned} \tag{2.24}$$

Now, we consider values in the original grammar. Let the value of A, $V(A)$, be $\sum_\beta V([A \rightarrow \beta \bullet])$. Let the nonterminal symbols in β be denoted by $\beta_1...\beta_{|\beta|}$ and let $V(\beta) = \prod_{i=1}^{|\beta|} V(\beta_i)$. Notice that given a derivation of the form $S \overset{D}{\Rightarrow} w_1...w_j\alpha\gamma$, the sum of the values of all derivations starting with D is the value of D times the product of the value of all of the nonterminals in $\alpha\gamma$:

$$\begin{aligned} &V(S \overset{D}{\Rightarrow} w_1...w_j\alpha\gamma) \times V(\alpha) \times V(\gamma) \\ &= V(S \overset{E}{\Rightarrow} w_1...w_jA\gamma \Rightarrow w_1...w_j\alpha\gamma) \times V(\alpha) \times V(\gamma) \\ &= V(S \overset{E}{\Rightarrow} w_1...w_jA\gamma) \times P_{orig}(A \rightarrow \alpha) \times V(\alpha) \times V(\gamma) \\ &= V(S \overset{E}{\Rightarrow} w_1...w_jA\gamma) \times V([A \rightarrow \alpha \bullet]) \times V(\gamma) \end{aligned} \tag{2.25}$$

Next, by the inductive assumption,

$$V(S \overset{E}{\Rightarrow} w_1...w_jA\gamma) \times V(A) \times V(\gamma) = P_{new}(S \overset{E}{\Rightarrow} w_1...w_jA\gamma)$$

Dividing both sides by $V(A)$,

$$V(S \overset{E}{\Rightarrow} w_1...w_jA\gamma) \times V(\gamma) = \frac{P_{new}(S \overset{E}{\Rightarrow} w_1...w_jA\gamma)}{V(A)}$$

Substituting into Equation 2.25, we get

$$\begin{aligned} V(S \overset{D}{\Rightarrow} w_1...w_j\alpha\gamma) \times V(\alpha) \times V(\gamma) &= P_{new}(S \overset{E}{\Rightarrow} w_1...w_jA\gamma) \times \frac{V([A \rightarrow \alpha \bullet])}{V(A)} \\ &= P_{new}(S \overset{E}{\Rightarrow} w_1...w_jA\gamma) \times \frac{V([A \rightarrow \alpha \bullet])}{\sum_\beta V([A \rightarrow \beta \bullet])} \end{aligned}$$

which equals Expression 2.24, completing the inductive step. $\diamond$

2–A.1 Viterbi-n-best is a semiring

In this section we show that the Viterbi-n-best semiring, as described in Section 2, has all the required properties of a semiring, and is ω-continuous. Recall that the Viterbi-n-best semiring is a homomorphism from the Viterbi-all semiring. We first show that the Viterbi-all semiring is ω-continuous.

We begin by arguing that the Viterbi-all semiring has all of the properties of an ω-continuous semiring, saving the most complicated property, the distributive property, for last. It should be clear that $\cup$ and $\star$ are associative. To show that the semiring is ω-continuous, we first note that the proper convergence of infinite sums follows directly from the properties of union, as does the associativity of infinite sums (unions). The only complicated property is that $\star$ distributes over $\cup$, even in the infinite case. We sketch this proof quickly. We need to show that for any set of Y_i, and any I,

$$X \star (\bigcup_{i \in I} Y_i) = \bigcup_{i \in I} (X \star Y_i)$$

Recall the definition of $\star$:

$$X \star Y = \{\langle vw, d \cdot e \rangle | \langle v, d \rangle \in X \land \langle w, e \rangle \in Y\}$$

First, consider an element $\langle vw, d \cdot e \rangle \in X \star (\bigcup_{i \in I} Y_i)$. It must be the case that $v \in X$ and $w \in Y_i$ for some v, w, i. So then, $\langle vw, d \cdot e \rangle \in \bigcup_{i \in I} X \star Y_i$. A reverse argument also holds, proving equality. Technically, we also need to show distributivity in the opposite direction, that $(\bigcup_{i \in I} Y_i) \star X = \bigcup_{i \in I} (Y_i \star X)$, but this follows from an exactly analogous argument.

Now, we will show that our definitions of operations in the Viterbi-n-best semiring are well defined, and then show that *topn* really does define a homomorphism. Recall our definitions: $\max\limits_{Vit\text{-}n} A, B = C$ if and only if there is some X, Y in the Viterbi-all semiring such that such that $topn(X) = A$, $topn(Y) = B$, and $topn(X \cup Y) = C$. We need to show that this defines a one to one relationship – that there is exactly one C which satisfies this relationship for each A and B. The existence of at least one such C follows from the definition of the Viterbi-n-best semiring. There must be at least one X, Y such that $topn(X) = A$ and $topn(Y) = B$ and thus at least one $C = topn(X \cup Y)$.

Now, consider any X, X' such that $topn(X) = topn(X')$ and Y, Y' such that $topn(Y) =$

$topn(Y')$. To show that C is unique, we need to show that $topn(X \cup Y) = topn(X' \cup Y')$. We will say that an element of a set Z is *simple* if there are at most a finite number of larger elements in Z, and *complex* otherwise. We first show equality of the simple elements in the two sets. Consider a simple element z in $topn(X \cup Y)$. There are at most $n - 1$ larger elements in $X \cup Y$. z must have been an element of either X or Y or both. Assume without loss of generality that it was in X. Then, since there are at most $n - 1$ larger elements in $X \cup Y$, there are at most $n - 1$ larger elements in X, and thus $z \in topn(X)$. Since $topn(X) = topn(X')$, $z \in topn(X')$, and therefore $z \in X'$, and $z \in X' \cup Y'$. By analogous reasoning, for any simple element $z' \in topn(X' \cup Y')$, $z' \in X \cup Y$. Now, it must be the case that $z \in topn(X' \cup Y')$, since each element of $topn(X' \cup Y')$ is in $X \cup Y$ and if there were n elements larger than z in $X' \cup Y'$, then each of these elements would be in $topn(X' \cup Y')$ and thus in $X \cup Y$, which would prevent z from being in $topn(X \cup Y)$. By analogous reasoning, every simple element $z' \in topn(X' \cup Y')$ is also in $topn(X \cup Y)$. Thus, the simple elements of the two sets are equal.

Now, let us consider the infinite elements, $\langle v, \infty \rangle$, of $topn(X \cup Y)$. There are several cases to consider, involving the possibilities where zero, one, or both of X, Y have an infinite element. We only consider the most complicated case, in which both do. Notice that if there are n or more simple elements in X and Y, then $topn(X \cup Y)$ will not have an infinite element, so we need only consider the case where there are at most $n - 1$ simple elements in $X \cup Y$. Then the infinite element of $topn(X \cup Y)$ will just be the supremum of all of the complex elements of $X \cup Y$, which will equal the supremum of the complex elements of X union the complex elements of Y, which will equal the maximum of the supremums of the complex elements of X and the complex elements of Y, which will equal the maximum of the infinite elements of $topn(X)$ and $topn(Y)$. The same reasoning applies to any X', Y', and, since if $topn(X) = topn(X')$, and $topn(Y) = topn(Y')$, their infinite elements must be the same, the maximum of their infinite elements must be the same, and thus the infinite elements of $topn(X \cup Y)$ and $topn(X' \cup Y')$ must be the same. Since we have already shown equality of the finite elements, this shows equality overall, and thus shows the uniqueness of C in our definition of additive operation $\max_{Vit\text{-}n}$. Therefore, $\max_{Vit\text{-}n}$ is well defined.

Next, we must go through very similar reasoning for $\star$, showing that the multiplicative operator, $\underset{Vit\text{-}n}{\times}$, is well defined. Recall that we defined $A \underset{Vit\text{-}n}{\times} B = C$ if and only if there is some X, Y in the Viterbi-all semiring such that such that $topn(X) = A$, $topn(Y) = B$, and $topn(X \star Y) = C$. There is at least one such C (equal to $topn(X \star Y)$) and now we need to prove its uniqueness. If $x = \langle v, d \rangle$ and $y = \langle w, e \rangle$, then we will write xy to indicate the product $\langle vw, de \rangle$. We examine first the simple elements of X and Y. Consider a simple element $z \in topn(X \star Y)$. $z = xy$, for some $x \in X, y \in Y$. Now, $x \in topn(X)$: otherwise, the n larger elements of X multiplied by y would result in n larger elements in $X \star Y$, meaning that $z \notin topn(X \star Y)$. Similarly, $y \in topn(Y)$. Thus, $z \in X' \star Y'$. Similarly, each $z' \in topn(X' \star Y')$ is in $X \star Y$. Following the same reasoning as before, the simple elements of the two sets are equal.

Now, consider the infinite elements. Again, we will consider only the case where both $topn(X)$ and $topn(Y)$ have an infinite element. If there is an infinite element in $topn(X \star Y)$, it must be formed in one of three ways: by taking the top element of x and multiplying it by the elements approaching a supremum in Y; by taking the top element of y multiplied by the elements approaching a supremum in X; or if there is no top element in X or Y, by taking the elements approaching the supremum in X and multiplying them by the elements approaching the supremum in Y. Thus, the infinite element of $topn(X \star Y)$ is the maximum of these three quantities, which will be same for any X' such that $topn(X') = topn(X)$ and Y' such that $topn(Y') = topn(Y)$.

Now that we have shown that both $\max\limits_{Vit\text{-}n}$ and $\underset{Vit\text{-}n}{\times}$ are well-defined, the associative and distributive properties in the Viterbi-n-best semiring follow automatically. We simply map the expression on the left side backwards from the Viterbi-n-best semiring into expressions in the Viterbi-all semiring; map the expression on the right side into the Viterbi-all semiring; and then use the appropriate property in the Viterbi-all semiring to show equality. We show how to use this technique for the associativity of $\max\limits_{Vit\text{-}n}$; the other properties follow analogously. We simply note that since our homomorphism is onto, for any A, B, C in the Viterbi-n-best semiring, there must be some X, Y, Z in the Viterbi-all semiring such that $A = topn(X)$, $B = topn(Y)$, and $C = topn(Z)$. Consider

$$\max_{Vit\text{-}n} (\max_{Vit\text{-}n} A, B), C$$

There must be some $D = \max\limits_{Vit\text{-}n} A, B = topn(X \cup Y)$. Then $\max\limits_{Vit\text{-}n} (\max\limits_{Vit\text{-}n} A, B), C = \max\limits_{Vit\text{-}n} D, C = topn((X \cup Y) \cup Z)$. By analogous reasoning, $\max\limits_{Vit\text{-}n} A, (\max\limits_{Vit\text{-}n} B, C) = topn(X \cup (Y \cup Z))$, and thus by the associativity of $\cup$, the associativity of $\max\limits_{Vit\text{-}n}$ is proved. Using the same reasoning, we can show associativity of $\underset{Vit\text{-}n}{\times}$ and distributivity.

Now, we must show that the Viterbi-n-best semiring is complete (has the associative and distributive property for infinite sums) and is ω-continuous. To do this, we need to show that our homomorphism works even for infinite sums. Then, by the same mapping argument we just made, we can show associativity, distributivity, and ω-continuity for infinite sums as well.

Consider $X_1 ... X_\infty$ and $X' ... X'_\infty$ such that $topn(X_i) = topn(X'_i)$. We need to show that $topn(\bigcup_i X_i) = topn(\bigcup_i X'_i)$. Let $X = \bigcup_i X_i$ and let $X' = \bigcup_i X'_i$. Consider a simple element z in $topn(X)$. Our reasoning will be nearly identical to the finite case, and is included here for completeness. There are at most $n-1$ larger elements in X. For some i, z must have been an element of X_i. Then, since there are at most $n-1$ larger elements in X, there are at most $n-1$ larger elements in $topn(X_i)$ and thus $z \in topn(X_i)$. Thus, $z \in topn(X'_i)$, and therefore $z \in X'_i$ and $z \in X'$. By analogous reasoning, for any simple element $z' \in topn(X')$, $z' \in X$. Now, it must be the case that $z \in topn(X')$, since each element of $topn(X')$ is in X and if there were n elements larger than z in X', then each of these elements would be in $topn(X')$ and thus in X, which would prevent z from being in $topn(X)$. By analogous reasoning, every simple element $z' \in topn(X')$ is also in $topn(X)$. Thus, the simple elements of the two sets are equal.

Now consider the case where there is an infinite element in $topn(X)$; this case is somewhat more complex. Let

$$w = \sup_{v|\langle v,d\rangle \in X - simpletopn(X)}$$

and

$$w' = \sup_{v|\langle v,d\rangle \in X' - simpletopn(X')}$$

Take an infinite sequence of elements in $X - simpletopn(X)$ approaching w. Consider an element x in this sequence. x must have come from some $X_i - simpletopn(X)$. We will show that there is an element of $X'_i - simpletopn(X')$ which is at least as close to w as x is, and thus that w' is at least as large as w.

Here are the cases to consider.

a X_i has at most $n-1$ elements.

b X_i has at most $n-1$ simple elements and an infinite number of complex elements.

c X_i has at least n simple elements.

In case a, since $topn(X_i) = topn(X_i')$, we deduce $X_i = X_i'$ and thus that $x \in X_i'$. In case b, either x is a simple element, in which case this reduces to case a, or x is a complex element. Now, since $topn(X_i) = topn(X_i')$, the supremum of the complex elements is the same in both cases, and thus there is an element at least as large as x in X_i'. In case c, we notice that not all n simple elements can be in $simpletopn(X)$, since otherwise there would be n simple elements in X, and we would not be concerned with the supremum of the complex elements. In particular, at least the nth largest element cannot be in $simpletopn(X)$. Now, if x is one of the n largest elements of X_i, then $x \in X_i'$, since $topn(X_i) = topn(X_i')$. And if x is not as large as the nth largest element, then the nth largest element of X_i' is an element of X_i' which is larger than x and in $X_i' - simpletopn(X')$.

This shows that even in the case of infinite sums, the homomorphism works. Following the same reasoning as before, it should be clear how to prove associativity and distributivity for infinite sums, and ω-continuity.

Appendix

2–B Additional Examples

2–B.1 Graham, Harrison, and Ruzzo (GHR) Parsing

In this section, we give a detailed description of a parser similar to that of Graham *et al.* (1980), as introduced in Section 2.7.5. The GHR parser is in many ways similar to Earley's parser, but with several improvements, including that epsilon and unary chains are both precomputed, and that completion is separated into two steps, allowing better dynamic programming.

Sikkel (1993, p. 122) gives a deduction system for GHR parsing. Note, however, that Sikkel's parser appears to have more than one derivation for a given parse; in particular, chains of unary productions can be broken down in several ways. In a boolean parser, such as Sikkel's, this repetition leads to no problems, but in a general, semiring parser, this is not acceptable. Another issue is derivation rules using precomputed chains, such as a rule using a condition like $A \stackrel{*}{\Rightarrow} \epsilon$. In a boolean parser, we can simply have a side condition, $A \stackrel{*}{\Rightarrow} \epsilon$. However, in a semiring parser, we will need to multiply in the value of the derivation, as well. Thus, we will need to explicitly compute items such as $[A \stackrel{*}{\Rightarrow} \epsilon]$, recording the value of the derivations. We will need similar items for chain rules.

Figure 2.22 gives an item-based GHR parser description. We assume that the start symbol, S', does not occur on the right hand side of any rule. We note that this parser does not work for non-commutative semirings. However, since there will still be a one-to-one correspondence between item derivations and grammar derivations, with simple modifications, it would be possible to map from the item values derived here for the non-commutative derivation semirings to the grammar values, as a post-processing step.

There are five different item types for this parser. The first item type, $[A \stackrel{*}{\Rightarrow} \alpha \circ \beta]$ is used for two purposes: determining which elements have derivations of the form $A \stackrel{*}{\Rightarrow} \epsilon$ and for determining which items have derivations of the form $A \rightarrow \alpha B \beta$ such that $\alpha \stackrel{*}{\Rightarrow} \epsilon$ and $\beta \stackrel{*}{\Rightarrow} \epsilon$; these are the rules which form a single step in a chain of unary productions. We can actually only derive two sub-types of this item: $[A \stackrel{*}{\Rightarrow} \circ \beta]$, which can be derived only if there is a derivation of the form $A \Rightarrow \alpha\beta \stackrel{*}{\Rightarrow} \beta$; and $[A \stackrel{*}{\Rightarrow} A \circ \gamma]$ which can be derived only if there is a derivation of the form $A \Rightarrow \alpha A \beta \gamma \stackrel{*}{\Rightarrow} A\gamma$. Items of the type $[A \stackrel{*}{\Rightarrow} \alpha \circ \beta]$ are derived using three different rules: the initial axiom, initial epsilon scanning, and initial

Item form:

$$[A \stackrel{*}{\Rightarrow} \alpha \circ \beta]$$
$$[A \stackrel{*}{\Rightarrow} B]$$
$$[i, A \to \alpha \bullet \beta, j, u]$$
$$[i, \mathit{finished}(A), j]$$
$$[i, \mathit{extendedFinished}(A), j]$$

Goal:

$$[1, \mathit{extendedFinished}(S'), n{+}1]$$

Rules:

$$\frac{R(A \to \alpha)}{[A \stackrel{*}{\Rightarrow} \circ\alpha]} \qquad \text{Initial Axiom}$$

$$\frac{[A \stackrel{*}{\Rightarrow} \alpha \circ B\beta] \qquad [B \stackrel{*}{\Rightarrow} \circ]}{[A \stackrel{*}{\Rightarrow} \alpha \circ \beta]} \qquad \text{Initial Epsilon Scanning}$$

$$\frac{[A \stackrel{*}{\Rightarrow} \circ B\beta]}{[A \stackrel{*}{\Rightarrow} B \circ \beta]} \qquad \text{Initial Unary Scanning}$$

$$\frac{}{[A \stackrel{*}{\Rightarrow} A]} \qquad \text{Unary Axiom}$$

$$\frac{[A \stackrel{*}{\Rightarrow} B] \qquad [B \stackrel{*}{\Rightarrow} C\circ]}{[A \stackrel{*}{\Rightarrow} C]} \qquad \text{Unary Completion}$$

$$\frac{R(S' \to \alpha)}{[1, S' \to \ \bullet \alpha, 1, 0]} \qquad \text{Initialization}$$

$$\frac{[i, A \to \alpha \bullet B\beta, j, u] \qquad [B \stackrel{*}{\Rightarrow} \circ]}{[i, A \to \alpha \bullet \beta, j{+}1, u]} \qquad \text{Epsilon Scanning}$$

$$\frac{R(B \to \gamma)}{[j, B \to \ \bullet \gamma, j, 0]}\, [i, A \to \alpha \bullet B\beta, j, u] \qquad \text{Prediction}$$

$$\frac{[i, A \to \alpha \bullet B\beta, k, u] \qquad [k, \mathit{extendedFinished}(B), j]}{[i, A \to \alpha B \bullet \beta, j, \min(u{+}1, 2)]}\, i < k \qquad \text{Completion}$$

$$\frac{}{[i, \mathit{finished}(a), i{+}1]}\, w_i = a \qquad \text{Terminal Finishing}$$

$$\frac{[i, A \to \alpha \bullet, j, u]}{[i, \mathit{finished}(A), j]}\, u = 2 \vee A = S' \wedge u = 0 \qquad \text{Finishing}$$

$$\frac{[i, \mathit{finished}(A), j] \qquad [B \stackrel{*}{\Rightarrow} A]}{[i, \mathit{extendedFinished}(B), j]} \qquad \text{Extended Finishing}$$

Figure 2.22: Graham Harrison Ruzzo

unary scanning. The first rule, the initial axiom, simply says that if $A \rightarrow \alpha$ then $A \stackrel{*}{\Rightarrow} \alpha$. The next rule, initial epsilon scanning, says that if $A \stackrel{*}{\Rightarrow} \alpha B \beta$ and $B \stackrel{*}{\Rightarrow} \epsilon$, then $A \stackrel{*}{\Rightarrow} \alpha\beta$. And finally, initial unary scanning is a technical rule, which simply advances the circle over a single nonterminal. It is because of this rule that we cannot use non-commutative semirings. Consider a grammar

$$\begin{array}{lcll} A & \rightarrow & EBE & R(A \rightarrow EBE) \\ E & \rightarrow & \epsilon & R(E \rightarrow \epsilon) \end{array}$$

Then the value of $[A \stackrel{*}{\Rightarrow} B\circ]$ will be $R(A \rightarrow EBE) \otimes R(E \rightarrow \epsilon) \otimes R(E \rightarrow \epsilon)$. For a commutative semiring, this grammar will work fine, but for a non-commutative semiring, the value is incorrect, since we would need to put the (yet to be determined) value of B's derivation between the two $R(E \rightarrow \epsilon)$'s. Essentially, this is the same reason that our grammar transformations only work for commutative semirings. In some sense, then, the GHR parser performs a grammar transformation to CNF, and then parses the transformed grammar. Of course, for many non-commutative semirings such as the derivation semirings, there will still be a one-to-one correspondence between item derivations and grammar derivations; with slight modifications we could easily map from the transposed item derivation to the corresponding correct grammar derivation.

The next item type, $[A \stackrel{*}{\Rightarrow} B]$, can be derived only if there is a derivation of the form $A \stackrel{*}{\Rightarrow} B$. This item is derived with two rules. The unary axiom simply states that $A \stackrel{*}{\Rightarrow} A$. Unary completion states that if $A \stackrel{*}{\Rightarrow} B$ and $B \stackrel{*}{\Rightarrow} C$, then $A \stackrel{*}{\Rightarrow} C$. It is important that unary completion uses items of the form $[B \stackrel{*}{\Rightarrow} C\circ]$, rather than items $[B \stackrel{*}{\Rightarrow} C]$, since this makes sure that there is a one-to-one correspondence between item derivations and grammar derivations.

The next item type is $[i, A \rightarrow \alpha \bullet \beta, j, u]$. This is almost exactly the usual Earley style item: it can be derived only if $S \rightarrow w_1 ... w_{i-1} A \gamma$ and $A \stackrel{*}{\Rightarrow} \alpha\beta \stackrel{*}{\Rightarrow} w_i ... w_{j-1} \beta$. There is one additional condition however. We wish to precompute ϵ and unary derivations. Therefore, in order to avoid duplicate derivations, we need to make sure that we do not recompute ϵ or unary derivations during the body of the computation. Thus, we keep track of how many non-ϵ derivations have been used to compute $[i, A \rightarrow \alpha \bullet \beta, j, u]$, and encode this in u. If u is zero, only ϵ rules have been used; if one, then one non-ϵ rule has been used, and if two, then at least two non-ϵ rules have been used. u can only take the values 0,1, or 2.

We wish to collapse unary derivations. We can do this using two more item types. Roughly, we can deduce $[i, \mathit{finished}(A), j]$ if there is an item of the form $[i, A \rightarrow \alpha \bullet, j, 2]$, i.e. when there is a derivation of the form $A \stackrel{*}{\Rightarrow} w_i...w_{j-1}$ using two non-epsilon rules. We require the last element to be 2, to avoid recomputing unary chains or epsilon chains. We will also derive such an item if $A = S'$ and the last element is 0; this allows us to recognize sentences of the form $S' \stackrel{*}{\Rightarrow} \epsilon$. The second item type, $[i, \mathit{extendedFinished}(A), j]$, can be derived only if there is a derivation of the form $A \stackrel{*}{\Rightarrow} w_i...w_{j-1}$. The difference between this item type and the previous one is that there are now no restrictions on unary branches and non-epsilon rules; this item type captures unary branching extensions.

These items are derived using many rules. The initialization rule is just the initialization rule of Earley parsing; the epsilon scanning rule is new: it allows us to skip over nonterminals A such that $A \stackrel{*}{\Rightarrow} \epsilon$. The prediction rule is the same as the Earley prediction rule. We thus implement prediction incrementally, without collapsing unary chains; Graham *et al.* (1980, p. 436) suggest this as one possible implementation of prediction. With a few more deduction rules, we could implement prediction more efficiently, in a manner analogous to the finishing rules.

Completion is the same as Earley completion, with a few caveats. First, we use the extended finishing items, rather than the items used in Earley's algorithm. This automatically takes into account unary chains. Also, we increment the non-epsilon count, not letting it go above two.

We use the terminal finishing rule to handle terminal symbols, rather than the scanning rule of Earley's algorithm. This lets us take into account unary chains using the terminal symbols. The finishing rule simply computes items of the form $[i, \mathit{finished}(A)]$, as previously described, and the extended finishing rule takes the finished rules, and extends them with precomputed unary chains.

Notice that the value of items of the form $[A \stackrel{*}{\Rightarrow} \alpha \circ \beta]$ and $[A \stackrel{*}{\Rightarrow} B]$, which are the only items in looping buckets, can be computed without the input sentence; thus the value of these items can be computed off-line, from the grammar alone. Notice also that the reverse values of these items require the input sentence. The item-based description format makes it easy to specify parsers that use off-line computation for some values, and on-line computation for other similar values.

2–B.2 Beyond Context-Free

In this section, we consider the problem of formalisms that are more powerful than CFGs. We will show that these formalisms pose a slight problem for the conventional algebraic treatment of formal language theory, but that we can solve this problem without much trouble. In particular, in the conventional view of formal language theory combined with algebra, a language is described as a formal power series. The terms of the formal power series are strings of the language; the coefficients give the values of the strings. These formal power series are most readily described as sets of algebraic equations; the formal power series represents a solution to the equations. Sets of algebraic equations used in this way can only describe context-free languages. On the other hand, it is straightforward to describe a Tree Adjoining Grammar (TAG) parser, which also can be transformed into a set of algebraic equations. Since Tree Adjoining Grammars are more powerful than context-free languages, this is a useful result.

Formal Language Theory Background

At this point, we need to discuss some results from algebra/formal language theory. One of the primary results is that there is a one-to-one correspondence between CFGs and certain sets of algebraic equations.

We must begin by defining a *formal power series.* A formal power series $\mathbb{A}\langle\langle\Sigma^*\rangle\rangle$ is a semiring using elements from a semiring $\mathbb{A}$ and an alphabet Σ. Elements in $\mathbb{A}\langle\langle\Sigma^*\rangle\rangle$ map from strings α in Σ^* to elements of the semiring $\mathbb{A}$. If s is an element of $\mathbb{A}\langle\langle\Sigma^*\rangle\rangle$, we will write (s, α) to indicate the value s maps to α. Essentially, s can be thought of as a language; $\mathbb{A}$ is the semiring used to assign values to strings of the language. Let $\mathbb{B}$ represent the booleans; then formal power series $s \in \mathbb{B}\langle\langle\Sigma^*\rangle\rangle$ represent formal languages; strings α such that $(s, \alpha) = \mathit{TRUE}$ correspond to the elements in the language; if $\mathbb{A}$ were the inside semiring, then (s, α) would equal $P(\alpha)$, the probability of the string.

We can define sums $s + t$ in $\mathbb{A}\langle\langle\Sigma^*\rangle\rangle$ as

$$(u, \alpha) = (s, \alpha) \oplus (t, \alpha)$$

We can defines products $s \times t$ in $\mathbb{A}\langle\langle\Sigma^*\rangle\rangle$ as

$$(u, \alpha) = \bigoplus_{\beta,\gamma|\alpha=\beta\gamma} (s, \beta) \otimes (t, \gamma)$$

The product and sum definitions are analogous to the way products are defined for polynomials in variables that do not commute, which is why $\mathbb{A}\langle\langle\Sigma^*\rangle\rangle$ is called a formal power series; this is, however, probably not the best way to think of $\mathbb{A}\langle\langle\Sigma^*\rangle\rangle$. We can define multiplication by a constant $t \in \mathbb{A}$, $t \times s$ as

$$(u, \alpha) = t \otimes (s, \alpha)$$

Now, a formal power series is called *algebraic* if it is the solution to a set of algebraic equations. Consider the formal power series

$$z + xzy + xxzyy + xxxzyyy + xxxxzyyyy + \cdots$$

in $\mathbb{B}\langle\langle\Sigma^*\rangle\rangle$. This formal power series is a solution to the following algebraic equations in $\mathbb{B}$:

$$\begin{aligned} S &= xSy + Z \\ Z &= z \end{aligned}$$

which is very similar to the CFG

$$\begin{array}{lcl} S & \rightarrow & xSy \\ S & \rightarrow & Z \\ Z & \rightarrow & z \end{array}$$

The preceding example is in the boolean semiring, but could be extended to any ω-continuous semiring, by adding constants to the equations. In general, given a CFG and a rule value function R, we can form a set of algebraic equations. For instance, for a nonterminal A with rules such as

$$\begin{array}{lcll} A & \rightarrow & \alpha & R(A \rightarrow \alpha) \\ A & \rightarrow & \beta & R(A \rightarrow \beta) \\ A & \rightarrow & \gamma & R(A \rightarrow \gamma) \\ & \vdots & & \end{array}$$

there is a corresponding algebraic equation

$$A = R(A \rightarrow \alpha)\alpha + R(A \rightarrow \beta)\beta + R(A \rightarrow \gamma)\gamma ...$$

with analogous equations for the other nonterminals in the grammar. Each nonterminal symbol represents a variable; each terminal symbol is a member of the alphabet, Σ.

It is an important theorem from the intersection of formal language theory and algebra, that for each CFG there is a set of algebraic equations in $\mathbb{B}\langle\langle\Sigma^*\rangle\rangle$, the solutions to which represent the strings of the language; for each algebraic equation in $\mathbb{B}\langle\langle\Sigma^*\rangle\rangle$, there is a CFG, whose language corresponds to the solutions of the equations.

Tree Adjoining Grammars

In this section, we address TAGs. TAGs pose an interesting challenge for semiring parsers, since the tree adjoining languages are weakly context sensitive. We develop an item-based description for a TAG parser, essentially using the description of Shieber *et al.* (1993), modified for our format, and including a few extra rule values, to ensure that there is a one-to-one correspondence between item and grammar derivations which can easily be recovered. The reader is strongly encouraged to refer to that work for background on TAGs and explication of the parser.

In the TAG formalism, there are two trees that correspond to a parse of a sentence. One tree is the parse tree, which is a conventional parse of the sentence. The other tree is the derivation tree; a traversal of the derivation tree gives the rules that would actually be used in a derivation to produce the parse tree, in the order they would be used. While the parse trees of TAGs cannot be produced by a CFG, the derivation trees can. It is important to note that the derivations produced by the parser of Figure 2.23 correspond to derivation trees, not parse trees. This is, in part, what allows us to parse with a formalism more powerful than CFGs.

The other reason that we are able to parse with a more powerful formalism is that while the algebraic formulation of formal language theory specifies a language as a single set of algebraic equations, we specify parsers as an input-string specific set of algebraic equations. Recall that the way we find the values of items in a looping bucket is to create a set of algebraic equations for the bucket. Recall also that we can always place all items into a

Item form:

$[\nu^\bullet, i, j, k, l]$
$[\nu_\bullet, i, j, k, l]$
$[goal]$

Goal

$[goal]$

Rules:

$$\frac{R(start(\alpha)) \qquad [\alpha@\epsilon^\bullet, 0, \_, \_, n]}{[goal]} \qquad \text{Find Start}$$

$$\frac{}{[\nu^\bullet, i, \_, \_, i+1]}\ label(\nu) = w_{i+1} \qquad \text{Terminal Axiom}$$

$$\frac{}{[\nu^\bullet, i, \_, \_, i]}\ label(\nu) = \epsilon \qquad \text{Empty String Axiom}$$

$$\frac{}{[\beta@foot(\beta)_\bullet, p, p, q, q]}\ \beta \in A \qquad \text{Foot Axiom}$$

$$\frac{[\alpha@(p \cdot 1)^\bullet, i, j, k, l]}{[\alpha@p_\bullet, i, j, k, l]}\ \alpha@(p \cdot 2) \text{ undefined} \qquad \text{Complete Unary}$$

$$\frac{[\alpha@(p \cdot 1)^\bullet, i, j, k, l] \qquad [\alpha@(p \cdot 2)^\bullet, l, j', k', m]}{[\alpha@p_\bullet, i, j \cup j', k \cup k', m]} \qquad \text{Complete Binary}$$

$$\frac{R(noadjoin(\nu)) \qquad [\nu_\bullet, i, j, k, l]}{[\nu^\bullet, i, j, k, l]} \qquad \text{No Adjoin}$$

$$\frac{R(adjoin(\beta, \nu)) \qquad [\beta@\epsilon^\bullet, i, p, q, l] \qquad [\nu_\bullet, p, j, k, q]}{[\nu^\bullet, i, j, k, l]} \qquad \text{Adjoin}$$

Figure 2.23: TAG parser item-based description

single large looping bucket. Thus, we can solve the parsing problem for any specific input string by solving a set of algebraic equations. However the algebraic equations we solve are specific to the input string, and the number of equations will almost always grow with the length of the input string. This variability of the equations is the other factor that allows us to parse formalisms more powerful than CFGs.

2–B.3 Greibach Normal Form

As we discussed in Section 2.6, item-based descriptions can be used to specify grammar transformations. In Section 2.6, we showed how to use item-based descriptions to convert to Chomsky Normal Form. In this section, we show how to use these descriptions to convert to Greibach Normal Form (GNF). In GNF, every rule is of the form $A \to a\alpha$. We give here a value-preserving transformation to GNF, following Hopcroft and Ullman (1979). While value preserving GNF transformations have been given before (Kuich and Salomaa, 1986), this is the first item-based description of such a transformation.

Figure 2.24 gives an item-based description for a value-preserving GNF transformation. There is a sequence of steps in GNF transformation, so we will use items of the form $[A \to \alpha]_j$, where j indicates the step number. The first step in a GNF transformation is to put the grammar in Chomsky Normal Form, which we have previously shown how to do, in Section 2.6. We will assume for this subsection that our nonterminal symbols are A_1 to A_m. The next step is conversion to "ascending" form,

$$\begin{array}{ll} A_i \to A_j\alpha & (j \geq i) \\ A_i \to a\alpha & \end{array}$$

This will be accomplished by first putting all rules with A_1 into this form, then A_2, etc. To transform rules into this form, we substitute the right hand sides of A_j for A_j in any rule of the form $A_i \to A_j\alpha$ $(j < i)$, a process that must terminate after at most $i - 1$ steps. This step is done using three rules, the Rule Axiom, which creates one item of the form $[A_i \to \alpha]_0$ for every rule of the form $A_i \to \alpha$, i.e. for every rule in the grammar. We use two different item forms: $[A_i \to A_j\alpha]_0$, $j \leq i$, for rules not yet in ascending form; and $[A_i \to A_j\alpha]_1$, $j \geq i$, for rules in ascending form. The Ascent Substitution rule substitutes the right hand side of a rule in ascending form into the first nonterminal of a rule not in ascending form. The Ascent Completion rule detects that an item is in ascending form, and

Item form:

$[A_i \to \alpha]_j$

$[B_i \to \alpha]_j$

Rule Goal

$R_1(X \to \alpha)$

Rules:

$\dfrac{R_0(A_i \to A\alpha)}{[A_i \to \alpha]_0}$	Rule Axiom
$\dfrac{[A_i \to A_j\alpha]_0 \qquad [A_j \to \beta]_1}{[A_i \to \beta\alpha]_0}\ j < i$	Ascent Substitution
$\dfrac{[A_i \to X\alpha]_0}{[A_i \to X\alpha]_1}\ X = a \vee (X = A_j \wedge j \geq i)$	Ascent Completion
$\dfrac{[A_i \to X\beta]_1}{[A_i \to X\beta B_i]_2}\ X = a \vee (X = A_j \wedge j > i)$	Right Beginning
$\dfrac{[A_i \to X\beta]_1}{[A_i \to X\beta]_2}\ X = a \vee (X = A_j \wedge j > i)$	Right Single Step
$\dfrac{[A_i \to A_i\alpha]_1}{[B_i \to \alpha]_2}\ j > i$	Right Termination
$\dfrac{[A_i \to A_i\alpha]_1}{[B_i \to \alpha B_i]_2}$	Right Continuation
$\dfrac{[A_i \to A_j\alpha]_2 \qquad [A_j \to a\beta]_3}{[A_i \to a\beta\alpha]_3}\ j > i$	Descent Substitution
$\dfrac{[A_i \to a\alpha]_2}{[A_i \to a\alpha]_3}$	Descent Non-substitution
$\dfrac{[B_i \to A_j\alpha]_2 \qquad [A_j \to a\beta]_3}{[B_i \to a\beta\alpha]_3}$	Auxiliary Substitution
$\dfrac{[B_i \to a\alpha]_2}{[B_i \to a\alpha]_3}$	Auxiliary Non-substitution
$\dfrac{[X \to \alpha]_3}{R_1(X \to \alpha)}$	Output

Figure 2.24: Greibach Normal Form Transformation

promotes it, creating an item with subscript 1.

The next step is removal of all left branching rules of the form $A_i \rightarrow A_i\alpha$ by conversion to the form

$$\begin{array}{l} A_i \rightarrow A_j\alpha \quad (j > i) \\ A_i \rightarrow a\alpha \\ B_i \rightarrow \alpha \end{array}$$

There are four rules that perform this operation: right beginning, right single step, right termination, and right continuation, using items in four different forms. The first item form is $[A_i \rightarrow A_j\beta]_1$, which is essentially the input to this step. The other three item forms are $[A_i \rightarrow A_j\alpha]_2$, $j > i$, $[A_i \rightarrow a\alpha]_2$, and $[B_i \rightarrow \alpha]_2$, which are essentially the outputs of this step. Consider a left-branching derivation of the form

$$A_i \overset{A_i \rightarrow \alpha}{\Rightarrow} A_i\alpha \overset{A_i \rightarrow \beta}{\Rightarrow} A_i\beta\alpha \overset{A_i \rightarrow A_j\gamma}{\Rightarrow} A_j\gamma\beta\alpha$$

The right branching equivalent will be

$$A_i \overset{A_i \rightarrow A_j\gamma B_i}{\Rightarrow} A_j\gamma B_i \overset{B_i \rightarrow \beta B_i}{\Rightarrow} A_j\gamma\beta B_i \overset{B_i \rightarrow \alpha}{\Rightarrow} A_j\gamma\beta\alpha$$

The item generating the rule used in the first substitution in the right branching form will be derived using Right Beginning; the item for the second substitution will be derived using Right Continuation; and the item for the final substitution will be derived using Right Termination. The rule $A_i \rightarrow A_j\gamma$ is valid in both forms; Right Single Step derives the needed item.

Now, we are ready for the next and penultimate step. Notice that since all items of the form $[A_i \rightarrow A_j\alpha]_2$ have $j > i$, the last nonterminal, A_m, must have rules only of the form $A_m \rightarrow a\alpha$, the target form. We can substitute A_m's productions into any production of the form $A_{m-1} \rightarrow A_m\alpha$, putting all A_{m-1} productions into the target form. This recursive substitution can be repeated all the way down to terminal A_1. Formally, we substitute A_j into any rule of the form $A_i \rightarrow A_j\alpha$ $(j > i)$, yielding rules of the form

$$A_i \rightarrow a\alpha$$

We call this penultimate step Descent Substitution. Descent Substitution is done using two rules: Descent Substitution and Descent Non-substitution. The inputs to this step are items in the form $[A_i \to A_j\alpha]_2$ and the outputs are in the form $[A_i \to a\beta]_3$. Descent Non-substitution simply detects when an item already has a terminal as its first symbol, and promotes it to the output form.

Finally, we substitute the A_i into the B_j to yield rules of the form:

$$A_i \to a\alpha$$
$$B_i \to a\alpha$$

This step is accomplished with the Auxiliary Substitution and Auxiliary Non-substitution rules. Hopcroft and Ullman (1979) shows that all items $[B_i \to \alpha]_2$ are actually in the form $[B_i \to A_j\alpha]_2$ or $[B_i \to a\alpha]_2$. The Auxiliary Substitution rule performs substitution into items of the first form, and the Auxiliary Non-substitution rule promotes items of the second form.

The output rule trivially states that items of the form $[X \to \alpha]_3$ correspond to the rules of the transformed grammar.

Proving that this transformation is value-preserving is somewhat tedious, and essentially follows the proof that grammars can be converted to GNF given by Hopcroft and Ullman (1979).

Appendix

2–C Reverse Value of Non-commutative Semirings

In this appendix, we consider the problem of finding reverse values in non-commutative semirings; the situation is somewhat more complex than in the commutative case. The problem is, for non-commutative semirings, there is no obvious equivalent to the reverse values. Consider the following grammar:

$$\begin{array}{lcl} S & \rightarrow & SA \\ S & \rightarrow & B \\ A & \rightarrow & \epsilon \\ B & \rightarrow & b \end{array}$$

Now, consider the derivation forest semiring. Without making reference to a specific parser, let us call the item deriving the terminal symbol $[B]$. We will denote derivations using just the nonterminals on the left hand side, for conciseness. So, for instance, a derivation

$$S \overset{S\rightarrow SA}{\Rightarrow} SA \overset{S\rightarrow SA}{\Rightarrow} SAA \overset{S\rightarrow B}{\Rightarrow} BAA \overset{B\rightarrow b}{\Rightarrow} bAA \overset{A\rightarrow \epsilon}{\Rightarrow} bA \overset{A\rightarrow \epsilon}{\Rightarrow} b$$

will be written as simply $SSSBAA$. The inside value of $[B]$ will just be

$$V([B]) = \{B\}$$

The value of the sentence is the union of all derivations, namely:

$$\{SB, SSBA, SSSBAA, SSSSBAAA, SSSSSBAAAA, ...\}$$

Now, since all derivations use the item $[B]$, it should be the case that the forward value times the reverse value of $[B]$ should just be the value of the sentence:

$$V([B]) \cdot Z([B]) = \{SB, SSBA, SSSBAA, SSSSBAAA, SSSSSBAAAA, ...\}$$

Since $V([B]) = \{B\}$, we get

$$\{B\} \cdot Z([B]) = \{SB, SSBA, SSSBAA, SSSSBAAA, SSSSSBAAAA, ...\}$$

for some $Z([B])$. But it should be clear that there is no such value for $Z([B])$ in a non-commutative semiring – the problem, intuitively, is that we need to get $V([B])$ into the center of the product, and there is no way to do that. One might think that what we need is two reverse values, a left outside value $Z_L([B])$, and a right outside value $Z_R([B])$, so that we can find values such that

$$Z_L([B]) \cdot V([B]) \cdot Z_R([B]) = \{SB, SSBA, SSSBAA, SSSSBAAA, SSSSSBAAAA, ...\}$$

but some thought will show that even this is not possible.

The solution, then, is to keep track of pairs of values. That is, we let $Z([B])$ be a set of pairs of values, such as

$$Z([B]) = \{\langle\{S\},\{\}\rangle, \langle\{SS\},\{A\}\rangle, \langle\{SSS\},\{AA\}\rangle, \langle\{SSSS\},\{AAA\}\rangle, ...\}$$

and then define an appropriate way to combine these pairs of values with inside values. This lets us define reverse values in the non-commutative case. We can use this same technique of using pairs of values to find an analog to the reverse values for most non-commutative semirings.

2–C.1 Pair Semirings

There are many technical details. In particular, we will want to compute infinite sums using these pairs of values, in ways similar to what we have already done. Thus, it will be convenient to define things so that we can use the mathematical machinery of semirings. Therefore, given a semiring $\mathbb{A}$, we will define a new semiring $\mathcal{P}(\mathbb{A})$. Later, we will describe a property, preserving pair order, and show that if $\mathbb{A}$ is ω-continuous and preserves pair order, then $\mathcal{P}(\mathbb{A})$ is ω-continuous, allowing us to compute infinite sums just as before. We will call $\mathcal{P}(\mathbb{A})$ the pair semiring of $\mathbb{A}$.

Intuitively, we simply want pairs of values, but in practice it will be more convenient to define addition if we allow pairs to occur multiple times. Therefore, we will define elements of $\mathcal{P}(\mathbb{A})$ as a mapping function from pairs a, a' to $\mathbb{N}^\infty$. Thus, an element of the semiring will be $r : \mathbb{A} \times \mathbb{A} \to \mathbb{N}^\infty$.[4]

[4]In retrospect, it might have been simpler to consider the semiring of functions from $\mathbb{A}$ to $\mathbb{A}$, where the

Continuing our example, if $r = Z([B])$, then

$$r(E_1, E_2) = \begin{cases} 1 & \textit{if } E_1 = \{S^k\} \wedge E_2 = \{A^{k-1}\} \\ 0 & \textit{otherwise} \end{cases}$$

Next, we need to define combinations of an element $r \in \mathcal{P}(\mathbb{A})$ with an element $a \in \mathbb{A}$, which we will write as $r.a$, to evoke function application. We simply multiply a between each of the pairs. Formally,

$$r.a = \bigoplus_{b,b' \in \mathbb{A}} r(b, b')bab'$$

where the premultiplication by the integral value $r(b, b')$ indicates repeated addition in $\mathbb{A}$.

Furthermore, we define two elements, r and s to be equivalent if whenever we combine them with any element of $\mathbb{A}$, they yield the same value. Formally,

$$r = s \text{ iff } \forall a \; r.a = s.a$$

It will be convenient to denote certain single pairs of elements of $\mathbb{A}$ concisely. Let $\lfloor a, a' \rfloor$ indicate

$$\lfloor a, a' \rfloor(b, b') = \begin{cases} 1 & \textit{if } a = b \wedge a' = b' \\ 0 & \textit{otherwise} \end{cases}$$

Intuitively, this value is the set containing the single pair a, a'.

We will need to make frequent reference to pairs of values. We will therefore let $\overline{a}$ indicate the pair a, a', and interchange these notations freely.

Addition can be simply defined pairwise: if $t = r \oplus s$, then

$$t(\overline{a}) = r(\overline{a}) + s(\overline{a})$$

Commutativity of addition follows trivially. We denote the zero element by $\lfloor 0, 0 \rfloor$; notice that for all a, $\lfloor 0, 0 \rfloor.a = 0$

We now show that application distributes over addition.

$$(r + s).a \quad = \quad \sum_{b,b'} (r + s)(b, b')bab'$$

multiplicative operator is composition. On the other hand, the Pair semiring will simplify the discussion in Section 2–C.2, where the fact that all elements of the semiring have the form of multi-sets of pairs will be useful.

$$
\begin{aligned}
&= \sum_{b,b'}(r(b,b')+s(b,b'))bab' \\
&= \sum_{b,b'} r(b,b')bab' + s(b,b')bab' \\
&= \sum_{b,b'} r(b,b')bab' + \sum_{b,b'} s(b,b')bab' \\
&= r.a + s.a
\end{aligned}
$$

A similar argument shows that application also distributes over infinite sums.

Next, we show that for any elements r, s, t, our definition of equality is consistent with our definition of addition, i.e. if $r = s$ then $r + t = s + t$. We simply notice that, for all $a \in \mathbb{A}$, $(r+t).a = r.a + t.a = s.a + t.a = (s+t).a$.

Now, when we define multiplication, we want the property that

$$\lfloor a, a' \rfloor \otimes \lfloor b, b' \rfloor = \lfloor ab, b'a' \rfloor$$

multiplying the first element on the right, and the second element on the left.

More generally, we define multiplication as, for $t = r \otimes s$,

$$t(c, c') = \sum_{a,a'} \sum_{b,b' \text{ s.t. } ab=c \wedge b'a'=c'} r(a,a')s(b,b')$$

This definition of multiplication has the desired property. We will also write $\overline{a} \otimes \overline{b}$ to indicate the pair $\lfloor ab, b'a' \rfloor$; this allows us to write the multiplicative formula as:

$$t(\overline{c}) = \sum_{\overline{a}} \sum_{\overline{b} \text{ s.t. } \overline{a} \otimes \overline{b} = \overline{c}} r(\overline{a})s(\overline{b})$$

It should be clear that $\lfloor 1, 1 \rfloor$ is a multiplicative identity.

We now show that for any elements r, s, t, our definition of equality is consistent with our definition of multiplication, i.e. if $r = s$, then $r \otimes t = s \otimes t$ and $t \otimes r = t \otimes s$. We first show that for all r, s, $(r \otimes s).a = r.(s.a)$:

$$
\begin{aligned}
(r \otimes s).a &= \sum_{b,b'} (r \otimes s)(b,b')bab' \\
&= \sum_{b,b'} \sum_{c,c'} \sum_{d,d' \text{ s.t. } cd=b \wedge d'c'=b'} r(c,c')s(d,d')bab'
\end{aligned}
$$

$$
\begin{aligned}
&= \sum_{c,c'} \sum_{d,d'} r(c,c')s(d,d')cdad'c' \\
&= \sum_{c,c'} r(c,c')c \left(\sum_{d,d'} s(d,d')dad' \right) c' \\
&= \sum_{c,c'} r(c,c')c(s.a)c' \\
&= r.(s.a) \qquad (2.26)
\end{aligned}
$$

Now, if $r = s$ then for all a,

$$(r \otimes t).a = r.(t.a) = s.(t.a) = (s \otimes t).a$$

Also, if $r = s$ then

$$(t \otimes r).a = t.(r.a) = t.(s.a) = (t \otimes s).a$$

This shows that our definition of equality is consistent with our definition of multiplication.

Multiplication is not commutative, but is associative, meaning that $(r \otimes s) \otimes t = r \otimes (s \otimes t)$. We show this now, first computing a simple form for $(r \otimes s) \otimes t$.

$$
\begin{aligned}
((r \otimes s) \otimes t)(\overline{e}) &= \sum_{\overline{c}} \sum_{\overline{d} | \overline{c} \otimes \overline{d} = \overline{e}} (r \otimes s)(\overline{c}) t(\overline{d}) \\
&= \sum_{\overline{c}} \sum_{\overline{d} | \overline{c} \otimes \overline{d} = \overline{e}} \sum_{\overline{a}} \sum_{\overline{b} | \overline{a} \otimes \overline{b} = \overline{c}} r(\overline{a}) s(\overline{b}) t(\overline{d}) \\
&= \sum_{\overline{c}} \sum_{\overline{a}} \sum_{\overline{b} | \overline{a} \otimes \overline{b} = \overline{c}} \sum_{\overline{d} | \overline{c} \otimes \overline{d} = \overline{e}} r(\overline{a}) s(\overline{b}) t(\overline{d}) \\
&= \sum_{\overline{a}} \sum_{\overline{b}} \sum_{\overline{c} | \overline{a} \otimes \overline{b} = \overline{c}} \sum_{\overline{d} | \overline{c} \otimes \overline{d} = \overline{e}} r(\overline{a}) s(\overline{b}) t(\overline{d}) \\
&= \sum_{\overline{a}} \sum_{\overline{b}} \sum_{\overline{d} | \overline{a} \otimes \overline{b} \otimes \overline{d} = \overline{e}} r(\overline{a}) s(\overline{b}) t(\overline{d})
\end{aligned}
$$

A similar rearrangement yields the same formula for $r \otimes (s \otimes t)$, showing associativity.

Now, we need to show that addition distributes over multiplication, both left and right. We do only the right case, i.e. $r \otimes (s \oplus t) = r \otimes s \oplus r \otimes t$. The left case is symmetric.

$$(r \otimes (s \oplus t))(\overline{a}) = \sum_{\overline{b}} \sum_{\overline{c} \text{ s.t. } \overline{b}\overline{c} = \overline{a}} r(\overline{b})(s \oplus t)(\overline{c})$$

$$
\begin{aligned}
&= \sum_{\overline{b}} \sum_{\overline{c}\text{ s.t. }\overline{b}\overline{c}=\overline{a}} r(\overline{b})(s(\overline{c})+t(\overline{c})) \\
&= \sum_{\overline{b}} \sum_{\overline{c}\text{ s.t. }\overline{b}\overline{c}=\overline{a}} r(\overline{b})s(\overline{c}) + \sum_{\overline{b}} \sum_{\overline{c}\text{ s.t. }\overline{b}\overline{c}=\overline{a}} r(\overline{b})t(\overline{c}) \\
&= (r\otimes s\oplus r\otimes t)(\overline{a})
\end{aligned}
$$

We have now shown all of the non-trivial properties of a semiring (and a few of the trivial ones, as well.)

There is one more property we must show: that $\mathcal{P}(\mathbb{A})$ is ω-continuous, assuming that $\mathbb{A}$ is ω-continuous, and assuming an additional quality, preserving pair order, which we define below. This involves several steps. The first is to show that $\mathcal{P}(\mathbb{A})$ is naturally ordered, meaning that we can define an ordering $\sqsubseteq$, such that $r \sqsubseteq s$ if and only if there exists t such that $r \oplus t = s$. To show that this ordering defines a true partial ordering, we must show that if $r \sqsubseteq s$ and $s \sqsubseteq r$ then $r = s$. (The other properties of a partial ordering, reflexivity and transitivity, follow immediately from the properties of addition.) From the fact that $r \sqsubseteq s$, we know that for some t, $r + t = s$. Therefore, we know that for all $a \in \mathbb{A}$, $(r+t).a = s.a$. Thus, $r.a + t.a = s.a$. Now, since $\mathbb{A}$ is also ω-continuous, it is also naturally ordered, implying that $r.a \sqsubseteq s.a$. Similarly, from the fact that $s \sqsubseteq r$, we conclude that $s.a \sqsubseteq r.a$. Thus, $r.a = s.a$, which shows that $r = s$.

The next step in showing ω-continuity is to show that $\mathcal{P}(\mathbb{A})$ is complete, meaning that we must show that infinite sums are commutative and satisfy the distributive law. Associativity of infinite sums means that given an index set J, and disjoint index sets I_j for $j \in J$,

$$\bigoplus_{j\in J}\bigoplus_{i\in I_j} r_i = \bigoplus_{i\in\bigcup_j I_j} r_i$$

This property follows directly from the fact that $\mathbb{N}^\infty$ is complete. We also must show that multiplication distributes (both left and right) over infinite sums. We show one case; the other is symmetric. We use a simple rearrangement of sums, which again relies on the fact that $\mathbb{N}^\infty$ is complete.

$$
\begin{aligned}
\left(\bigoplus_{i\in I} r\otimes s_i\right)(\overline{c}) &= \sum_{i\in I}\sum_{\overline{a}}\sum_{\overline{b}|\overline{a}\overline{b}=\overline{c}} r(\overline{a})s_i(\overline{b}) \\
&= \sum_{\overline{a}}\sum_{\overline{b}|\overline{a}\overline{b}=\overline{c}} r(\overline{a}) \sum_{i\in I} s_i(\overline{b})
\end{aligned}
$$

$$= \left(r \otimes \bigoplus_{i \in I} s_i\right)(\overline{c})$$

The final step in showing ω-continuity is to show that for all s, for all sequences r_i, if $\bigoplus_{0 \leq i \leq n} r_i \sqsubseteq s$ for all $n \in \mathbb{N}$, then $\bigoplus_{i \in \mathbb{N}} r_i \sqsubseteq s$. We do not know how to show this for ω-continuous semirings $\mathbb{A}$ in general, although ω-continuity of $\mathbb{A}$ is certainly a requirement. We therefore make an additional assumption which is true of all semirings discussed in this chapter: we assume that if for all $a \in \mathbb{A}, r.a \sqsubseteq s.a$ then $r \sqsubseteq s$. If this assumption is true for a semiring $\mathbb{A}$, we shall say that $\mathbb{A}$ *preserves pair order*. Now, assuming $\mathbb{A}$ preserves pair order, it is straightforward to show that $\mathcal{P}(\mathbb{A})$ is ω-continuous. Given a sequence r_i, let $r = \bigoplus_{0 \leq i} r_i$, and let $r_{\leq n} = \bigoplus_{0 \leq i \leq n} r_i$. Now, for all n, $r_{\leq n} \sqsubseteq r$, since $r_{\leq n} \oplus \bigoplus_{i > n} r_i = r$. Notice that for any t, $t \sqsubseteq s$ implies that for all a, $t.a \sqsubseteq s.a$. Therefore, since for all n and all a, $r_{\leq n} \sqsubseteq s$, we conclude that

$$r_{\leq n}.a \sqsubseteq s.a \tag{2.27}$$

Now, it is a property of ω-continuous semirings that (Kuich, 1997, p. 613)

$$\sup_n \bigoplus_{0 \leq i \leq n} a_i = \bigoplus_{0 \leq i} a_i$$

Notice that

$$r_{\leq n}.a = \left(\bigoplus_{0 \leq i \leq n} r_i\right).a = \bigoplus_{0 \leq i \leq n} (r_i.a)$$

and

$$r.a = \left(\bigoplus_{0 \leq i} r_i\right).a = \bigoplus_{0 \leq i} (r_i.a)$$

Thus,

$$\sup_n r_{\leq n}.a = r.a \tag{2.28}$$

From Equation 2.27 and Equation 2.28, we conclude that for all a, $r.a \sqsubseteq s.a$, and from this and our assumption that $\mathbb{A}$ preserves pair order, we conclude that $r \sqsubseteq s$, which shows that $\mathcal{P}(\mathbb{A})$ is an ω-continuous semiring.

2–C.2 Specific Pair Semirings

Now, we can discuss specific semirings, showing that they preserve pair order, and showing how to efficiently implement $\mathcal{P}(\mathbb{A})$. We first note that for any commutative semiring $\mathbb{A}$, $\mathcal{P}(\mathbb{A})$ is isomorphic to $\mathbb{A}$; the equivalence classes of $\mathcal{P}(\mathbb{A})$ are in direct correspondence with the elements of $\mathbb{A}$; application $r.a$ is equivalent to multiplication. It is thus straightforward to show that $\mathbb{A}$ preserves pair order. Thus, the formulae we will give in the sequel for paired semirings hold equally well for all commutative semirings.

Next, we notice that for all of the non-commutative semirings discussed in this chapter, the three derivation semirings, if $a \sqsubseteq b$ then $a \oplus b = b$. Now, for such a semiring, if for all a, $r.a \sqsubseteq s.a$ then for all a, $r.a + s.a = s.a$ and thus, $r + s = s$ which implies that $r \sqsubseteq s$. Thus all of the non-commutative semirings discussed in this chapter preserve pair order.

Implementations of the paired versions of the three derivation semirings are straightforward. To implement the Pair-derivation semiring, we simply keep sets of pairs of derivation forests. Recall that for the Viterbi-derivation semiring, in theory we keep a derivation forest of all top scoring values, but in practical implementations, typically keep just an arbitrary element of this forest. In a theoretically correct implementation of the Pair-Viterbi-derivation semiring, we simply maintain a top scoring element v, and the set of pairs of derivations with value v, using the same implementation as for the Pair-derivation semiring. If in practice we are only interested in a representative top scoring derivation, rather than the set of all top scoring derivations, then we can simply maintain a pair, d, e, of a left and a right derivation, along with the value v. Similar considerations are true for both theoretical and practical implementations of the Pair-Viterbi-n-best semiring.

2–C.3 Derivation of Non-Commutative Reverse Value Formulas

Now, we can rederive the reverse formulas, but using non-commutative semirings, and the corresponding pair semirings. These derivations almost exactly follow the corresponding derivations in commutative semirings.

For non-commutative semirings $\mathbb{A}$, $Z(x)$ will represent a value in $\mathcal{P}(\mathbb{A})$. We will construct reverse values in such a way that

$$Z(x).V(x) = \bigoplus_{D \text{ a derivation}} V(D)C(D, x) \tag{2.29}$$

which is directly analogous to Equation 2.10.

Recall that an outer tree O has a hole in it, from where some inner tree headed by x was deleted. We now define the left and right reverse values of an outer tree O, $Z_L(O)$ and $Z_R(O)$, to be the product of the values of the rules to the left and to the right of the hole, respectively.

$$Z_L(O) = \bigotimes_{r \in D \text{ to left}} R(r)$$

$$Z_R(O) = \bigotimes_{r \in D \text{ to right}} R(r)$$

Notice that these are values in $\mathbb{A}$. Now, we will define the value of an outer tree. We will use the same notation, $Z(O)$ as we used for non-commutative semirings for consistency, which we hope will not lead to confusion. If these formulas were applied to commutative semirings, the values would be the same as with the original formulas, so this redefinition should not be problematic.

We define the value of $Z(O)$ to be the pair of values of the rules to the left and right of the hole.

$$Z(O) = \lfloor Z_L(O), Z_R(O) \rfloor$$

which is a value in $\mathcal{P}(\mathbb{A})$. Then, the reverse value of an item can be defined, just as before in Equation 2.11.

$$Z(x) = \bigoplus_{D \in outer(x)} Z(D)$$

That is, the reverse value of x is the sum of the values of each outer tree of x, this time in the pair semiring.

Next, we show that this new definition of reverse values has the property described by Equation 2.29, following almost exactly the proof of Theorem 2.4.

Theorem 2.10

$$Z(x).V(x) = \bigoplus_{D \text{ a derivation}} V(D)C(D, x)$$

Proof

$$
\begin{aligned}
Z(x).V(x) &= \left(\bigoplus_{O \in outer(x)} Z(O) \right) .V(x) \\
&= \left(\bigoplus_{O \in outer(x)} Z(O) \right) \cdot \left(\bigoplus_{I \in inner(x)} V(I) \right) \\
&= \left(\bigoplus_{O \in outer(x)} \lfloor Z_L(O), Z_R(O) \rfloor \right) \cdot \left(\bigoplus_{I \in inner(x)} V(I) \right) \\
&= \bigoplus_{O \in outer(x)} \left(\lfloor Z_L(O), Z_R(O) \rfloor . \bigoplus_{I \in inner(x)} V(I) \right) \\
&= \bigoplus_{O \in outer(x)} Z_L(O) \left(\bigoplus_{I \in inner(x)} V(I) \right) Z_R(O) \\
&= \bigoplus_{I \in inner(x)} \bigoplus_{O \in outer(x)} Z_L(O) V(I) Z_R(O)
\end{aligned}
$$

Now, using the same reasoning as in Theorem 2.4, it should be clear that this last expression equals the expression on the right hand side of Equation 2.29, $\bigoplus_D V(D)C(D,x)$, completing the proof. $\diamond$

There is a simple, recursive formula for efficiently computing reverse values, analogous to that shown in Theorem 2.5, and using an analogous proof. Recall that the basic equation for computing forward values not involved in loops was

$$
V(x) = \bigoplus_{a_1 \ldots a_k \text{ s.t. } \frac{a_1 \ldots a_k}{x}} \bigotimes_{i=1}^{k} V(a_i)
$$

Earlier, we introduced the notation $1, \overset{-j}{\ldots}, k$ to indicate the sequence $1, \ldots, j-1, j+1, \ldots, k$. Now, we introduce additional notation for constructing elements of the pair semiring. Let

$$
\overline{\bigotimes_{i=1, \overset{-j}{\ldots}, k}} a_i = \lfloor \bigotimes_{i=1}^{j-1} a_i, \bigotimes_{i=j+1}^{k} a_i \rfloor
$$

We need to show

Lemma 2.11

$\overline{\bigotimes}_{i=1, \overset{-j}{\ldots}, k}$ distributes over $\bigoplus$.

Proof Let $H_1, \overset{\neg j}{\ldots}, H_k$ be index sets such that for $h_i \in H_i$, x_{h_i} is a value in $\mathbb{A}$. Then

$$
\begin{aligned}
\overline{\bigotimes_{i=1,\overset{\neg j}{\ldots},k}} \bigoplus_{h_i \in H_i} x_{h_i} &= \left\lfloor \bigotimes_{i=1,\ldots,j-1} \bigoplus_{h_i \in H_i} x_{h_i}, \bigotimes_{i=j+1,\ldots,k} \bigoplus_{h_i \in H_i} x_{h_i} \right\rfloor \\
&= \left\lfloor \bigotimes_{i=1,\ldots,j-1} \bigoplus_{h_i \in H_i} x_{h_i},\ 1 \right\rfloor \otimes \left\lfloor 1, \bigotimes_{i=j+1,\ldots,k} \bigoplus_{h_i \in H_i} x_{h_i} \right\rfloor \\
&= \bigotimes_{i=1,\ldots,j-1} \left\lfloor \bigoplus_{h_i \in H_i} x_{h_i},\ 1 \right\rfloor \otimes \bigotimes_{i=k,\ldots,j+1} \left\lfloor 1, \bigoplus_{h_i \in H_i} x_{h_i} \right\rfloor \\
&= \bigotimes_{i=1,\ldots,j-1} \bigoplus_{h_i \in H_i} \lfloor x_{h_i}, 1 \rfloor \otimes \bigotimes_{i=k,\ldots,j+1} \bigoplus_{h_i \in H_i} \lfloor 1, x_{h_i} \rfloor \\
&= \bigoplus_{h_1 \in H_1,\ldots,h_{j-1} \in H_{j-1}} \bigotimes_{i=1\ldots j-1} \lfloor x_{h_i}, 1 \rfloor \otimes \bigoplus_{h_{j+1} \in H_{j+1},\ldots,h_k \in H_k} \bigotimes_{i=k,\ldots,j+1} \lfloor 1, x_{h_i} \rfloor \\
&= \bigoplus_{h_1 \in H_1,\overset{\neg j}{\ldots},h_k \in H_k} \left(\bigotimes_{i=1,\ldots,j-1} \lfloor x_{h_i}, 1 \rfloor \otimes \bigotimes_{i=k,\ldots,j+1} \lfloor 1, x_{h_i} \rfloor \right) \\
&= \bigoplus_{h_1 \in H_1,\overset{\neg j}{\ldots},h_k \in H_k} \left\lfloor \bigotimes_{i=1,\ldots,j-1} x_{h_i}, \bigotimes_{i=j+1,\ldots,k} x_{h_i} \right\rfloor \\
&= \bigoplus_{h_1 \in H_1,\overset{\neg j}{\ldots},h_k \in H_k} \overline{\bigotimes_{i=1,\overset{\neg j}{\ldots},k}} x_{h_i}
\end{aligned}
$$

$\diamond$

Now, we can give a simple formula for computing reverse values $Z(x)$ not involved in loops:

Theorem 2.12

For items $x \in B$ where B is non-looping,

$$Z(x) = \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} Z(b) \otimes \overline{\bigotimes_{i=1,\overset{\neg j}{\ldots},k}} V(a_i)$$

unless x is the goal item, in which case $Z(x) = \lfloor 1, 1 \rfloor$, the multiplicative identity of the pair semiring.

Proof We begin with the goal item. As before, the outer trees of the goal item are all empty. Thus,

$$\begin{aligned}
Z(goal) &= \bigoplus_{D \in outer(goal)} \lfloor Z_L(D), Z_R(D) \rfloor \\
&= \lfloor Z_L(\{\langle\rangle\}), Z_R(\{\langle\rangle\}) \rfloor \\
&= \lfloor \bigotimes_{r \in \{\langle\rangle\}} R(r), \bigotimes_{r \in \{\langle\rangle\}} R(r) \rfloor \\
&= \lfloor 1, 1 \rfloor
\end{aligned}$$

Using the same observation as in Theorem 2.5, that every outer tree of a_j, D_{a_j}, can be described as a combination of the surrounding outer tree of b, D_b and inner trees of $a_1, \overset{\neg j}{\ldots}, a_k$ where $\frac{a_1 \ldots a_k}{b}$, and observing that the value of such an outer tree is $Z(D_b) \otimes \overline{\bigotimes_{i=1,\overset{\neg j}{\ldots},k}} V(D_{a_i})$,

$$\begin{aligned}
Z(x) &= \bigoplus_{D \in outer(x)} Z(D) \\
&= \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} \; \bigoplus_{D \in outer\left(j, \frac{a_1 \ldots a_k}{b}\right)} Z(D) \\
&= \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} \; \bigoplus_{\substack{D_{a_1} \in inner(a_1), \overset{\neg j}{\ldots}, \\ D_{a_k} \in inner(a_k), \\ D_b \in outer(b)}} Z(D_b) \otimes \overline{\bigotimes_{i=1,\overset{\neg j}{\ldots},k}} V(D_{a_i}) \\
&= \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} \; \bigoplus_{D_b \in outer(b)} Z(D_b) \otimes \bigoplus_{\substack{D_{a_1} \in inner(a_1), \overset{\neg j}{\ldots}, \\ D_{a_k} \in inner(a_k)}} \overline{\bigotimes_{i=1,\overset{\neg j}{\ldots},k}} V(D_{a_i}) \\
&= \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} Z(b) \otimes \overline{\bigotimes_{i=1,\overset{\neg j}{\ldots},k}} \bigoplus_{D_{a_i} \in inner(a_i)} V(D_{a_i}) \\
&= \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} Z(b) \otimes \overline{\bigotimes_{i=1,\overset{\neg j}{\ldots},k}} V(a_i)
\end{aligned}$$

completing the general case. $\diamond$

The case for looping buckets, is, as usual, somewhat more complicated, but follows Theorem 2.6 very closely. As in the commutative case, it requires an infinite sum. This infinite sum is why we so carefully made sure that the pair semiring is ω-continuous: we wanted to make sure that we could easily handle the infinite case, by using the properties of ω-continuous semirings.

Recall that $outer_{\leq g}(x, B)$ represents the set of outer trees of x with generation at most g. Now, we can define the left and right $\leq g$ generation reverse value of an item x in bucket B

$$Z_{\leq g}(x, B) = \bigoplus_{D \in outer_{\leq g}(x,B)} \lfloor Z_L(D), Z_R(D) \rfloor$$

Since $\mathcal{P}(\mathbb{A})$ is an ω-continuous semiring, an infinite sum is equal to the supremum of the partial sums:

$$\bigoplus_{D \in outer(x,B)} Z(D) = Z_{\leq \infty}(x, B) = \sup_g Z_{\leq g}(x, B)$$

Thus, as before, we wish to find a simple formula for $Z_{\leq g}(x, B)$.

Recall that for x in a bucket following B, $Z_{\leq g}(x, B) = Z(x)$ and that for $x \in B$, $Z_{\leq 0}(x, B) = 0$, providing a base case. We can then address the general case, for $g \geq 1$:

Theorem 2.13

For $x \in B$ and $g \geq 1$,

$$Z_{\leq g}(x, B) = \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} \left(\begin{cases} Z_{\leq g-1}(b, B) & \mathit{if}\ b \in B \\ Z(b) & \mathit{if}\ b \notin B \end{cases} \right) \otimes \overline{\bigotimes_{i=1, \stackrel{-j}{\ldots}, k} V(a_i)}$$

Proof Recall that $\mathit{MakeOuter}(j, D_{a_1}, \stackrel{-j}{\ldots}, D_{a_k}, D_b)$ is a function that puts together the specified trees to form an outer tree for a_j. Then,

$$\begin{aligned} & Z_{\leq g}(x, B) \\ &= \bigoplus_{D \in outer_{\leq g}(x,B)} Z(D) = \\ &= \bigoplus_{j, a_1 \ldots a_k, b \text{ s.t. } \frac{a_1 \ldots a_k}{b} \wedge x = a_j} \quad \bigoplus_{\substack{D_b \in outer_{\leq g-1}(b,B), \\ D_{a_1} \in inner(a_1), \stackrel{-j}{\ldots}, \\ D_{a_k} \in inner(a_k)}} Z(\mathit{MakeOuter}(j, D_{a_1}, \stackrel{-j}{\ldots}, D_{a_k}, D_b)) = \end{aligned}$$

$$
\begin{aligned}
&= \bigoplus_{j,a_1\ldots a_k,b \text{ s.t. } \frac{a_1\ldots a_k}{b} \wedge x=a_j} \;\; \bigoplus_{\substack{D_b \in outer_{\leq g-1}(b,B),\\ D_{a_1}\in inner(a_1),\overset{\neg j}{\ldots},\\ D_{a_k}\in inner(a_k B)}} Z(D_b) \otimes \overline{\bigotimes_{i=1,\overset{\neg j}{\ldots},k}} V(D_{a_i}) = \\
&= \bigoplus_{j,a_1\ldots a_k,b \text{ s.t. } \frac{a_1\ldots a_k}{b} \wedge x=a_j} \;\; \bigoplus_{D_b \in outer_{\leq g-1}(b,B)} Z(D_b) \otimes \bigoplus_{\substack{D_{a_1}\in inner(a_1),\overset{\neg j}{\ldots},\\ D_{a_k}\in inner(a_k)}} \overline{\bigotimes_{i=1,\overset{\neg j}{\ldots},k}} V(D_{a_i}) = \\
&= \bigoplus_{j,a_1\ldots a_k,b \text{ s.t. } \frac{a_1\ldots a_k}{b} \wedge x=a_j} Z_{\leq g-1}(b,B) \otimes \overline{\bigotimes_{i=1,\overset{\neg j}{\ldots},k}} \;\; \bigoplus_{D_{a_i}\in inner(a_i,B)} V(D_{a_i}) = \\
&= \bigoplus_{j,a_1\ldots a_k,b \text{ s.t. } \frac{a_1\ldots a_k}{b} \wedge x=a_j} Z_{\leq g-1}(b,B) \otimes \overline{\bigotimes_{i=1,\overset{\neg j}{\ldots},k}} V(a_i) \qquad (2.30)
\end{aligned}
$$

Substituting Equation 2.17 into Equation 2.30, we get

$$
Z_{\leq g}(x,B) = \bigoplus_{j,a_1\ldots a_k,b \text{ s.t. } \frac{a_1\ldots a_k}{b} \wedge x=a_j} \left(\begin{cases} Z_{\leq g-1}(b,B) & \textit{if } b \in B \\ Z(b) & \textit{if } b \notin B \end{cases} \right) \otimes \overline{\bigotimes_{i=1,\overset{\neg j}{\ldots},k}} V(a_i)
$$

◇

Chapter 3

Maximizing Metrics

It is well known how to find the parse with the highest probability of being exactly right. In this chapter, we show how to use the inside-outside probabilities to find parses that are the best in other senses, such as maximizing the expected number of correct constituents (Goodman, 1996b). We will use these algorithms in the next chapter to parse the DOP model efficiently.

3.1 Introduction

In corpus-based approaches to parsing, researchers are given a treebank (a collection of text annotated with the "correct" parse tree). The researchers then attempt to find algorithms that, given unlabelled text from the treebank, produce as similar a parse as possible to the one in the treebank.

Various methods can be used for finding these parses. Some of the most common involve inducing Probabilistic Context-Free Grammars (PCFGs), and then using an algorithm, such as the Labelled Tree (Viterbi) algorithm, which maximizes the probability that the output of the parser (the "guessed" tree) is the one that the PCFG produced.

There are many different ways to evaluate the output parses. In Section 3.2, we will define them, using for consistency our own terminology. We will often include the conventional term in parentheses for those measures with standard names. The most common evaluation metrics include the Labelled Tree rate (also called the Viterbi Criterion or Exact Match rate), Consistent Brackets Recall rate (also called the Crossing Brackets rate), Consistent Brackets Tree rate (also called the Zero Crossing Brackets rate), and Precision and Recall. Despite the variety of evaluation metrics, nearly all researchers use algorithms that maximize performance on the Labelled Tree rate, even when they evaluate performance using other criteria.

We propose that by creating algorithms that optimize the evaluation criterion, rather than some general criterion, improved performance can be achieved.

There are two commonly used kinds of parse trees. In one kind, binary branching parse trees, the trees are constrained to have exactly two branches for each nonterminal node. In the other kind, n-ary branching parse trees, each nonterminal node may have any number of branches; however, in this chapter we will only be considering n-ary trees with at least two branches.

In the first part of this chapter, we discuss the binary branching case. In Section 3.2, we define most of the evaluation metrics used in this chapter and discuss previous approaches. Then, in Section 3.3, we discuss the Labelled Recall algorithm, a new algorithm that maximizes performance on the Labelled Recall rate. In Section 3.4, we discuss another new algorithm, the Bracketed Recall algorithm, that maximizes performance on the Bracketed Recall rate, closely related to the Consistent Brackets Recall (Crossing Brackets) rate. In Section 3.5, we present experimental results using these two algorithms on appropriate tasks, and compare them to the Viterbi algorithm. Next, in Section 3.6, we show that optimizing a similar criterion, the Bracketed Tree rate, which resembles the Consistent Brackets Tree rate, is NP-Complete.

In the second part of the chapter, we extend these results in two ways. First, in Section 3.7, we show that the algorithms of the first part are both special cases of a more general algorithm, and give some potential applications for this more general algorithm. Second, in Section 3.8, we generalize the algorithm further, so that it can handle n-ary branching parses. In particular, we give definitions for Recall, Precision, and a new, related measure, Mistakes. We then show how to maximize the weighted difference of Recall minus Mistakes, which we call the Combined rate. Finally, we give experimental results for the n-ary branching case.

3.2 Evaluation Metrics

In this section, we first define some of the basic terms and symbols, including *parse tree*, and then specify the different kinds of errors parsing algorithms can make. Next, we define the different metrics used in evaluation. Finally, we discuss the relationship of these metrics to parsing algorithms.

In this chapter, we spend quite a bit of time on the topic of parsing metrics, and the use of a consistent naming convention will simplify the discussion. However, this consistent naming convention is not standard, and so will only be used in this chapter. A glossary of these terms is provided in Appendix 3–B.

3.2.1 Basic Definitions

In this chapter, we are primarily concerned with scoring parse trees and finding parse trees that optimize certain scores. Parse trees are typically scored by the number of their constituents that are correct, according to some measure, so it will be convenient in this chapter to define parse trees as sets of constituents. As we have done throughout this thesis, we will let $w_1...w_n$ denote the sequence of terminals (words) in the sentence under discussion. Then we define a parse tree T as a set of constituents $\langle i, X, j\rangle$. A triple $\langle i, X, j\rangle$ indicates that $w_i w_{i+1}...w_{j-1}$ can be parsed as a terminal or nonterminal X. The triples must meet the following requirements, which enforce binary branching constraints and consistency:

- The sentence was generated by the start symbol, S. Formally, $\langle 1, S, n+1\rangle \in T$ and for all $X \neq S$, $\langle 1, X, n+1\rangle \notin T$.
- The tree is binary branching and consistent. Formally, for every $\langle i, X, j\rangle$ in T, $i \neq j-1$, there is exactly one k, Y, and Z such that $i < k < j$ and $\langle i, Y, k\rangle \in T$ and $\langle k, Z, j\rangle \in T$.

Let T_C denote the "correct" parse (the one in the tree bank) and let T_G denote the "guessed" parse (the one output by the parsing algorithm). Let N_C denote $|T_C|$, the number of vocabulary symbols in the correct parse tree, and let N_G denote $|T_G|$, the number of vocabulary symbols in the guessed parse tree.

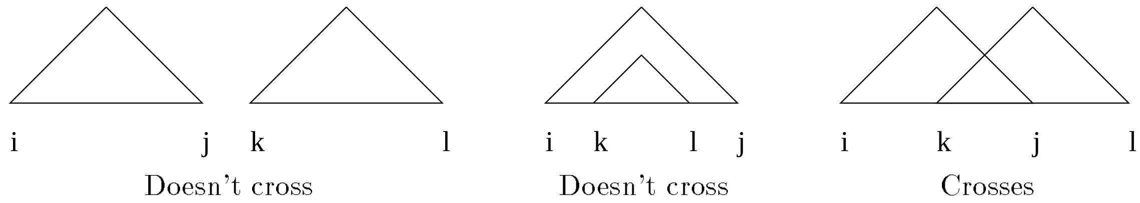


Figure 3.1: Non-crossing and crossing constituents

3.2.2 Evaluation Metrics

There are various levels of strictness for determining whether a constituent (element of T_G) is "correct." The strictest of these is *Labelled Match.* A constituent $\langle i, X, j\rangle \in T_G$ is correct according to Labelled Match if and only if $\langle i, X, j\rangle \in T_C$. In other words, a constituent in the guessed parse tree is correct if and only if it occurs in the correct parse tree.

The next level of strictness is *Bracketed Match.* Bracketed match is like Labelled Match, except that the nonterminal label is ignored. Formally, a constituent $\langle i, X, j\rangle \in T_G$ is correct according to Bracketed Match if and only if there exists a Y such that $\langle i, Y, j\rangle \in T_C$.

The least strict level is *Consistent Brackets* (traditionally misnamed Crossing Brackets). Consistent Brackets is like Bracketed Match in that the label is ignored. It is even less strict in that the observed $\langle i, X, j\rangle$ need not be in T_C—it must simply not be ruled out by any $\langle k, Y, l\rangle \in T_C$. A particular triple $\langle k, Y, l\rangle$ rules out $\langle i, X, j\rangle$ if there is no way that $\langle i, X, j\rangle$ and $\langle k, Y, l\rangle$ could both be in the same parse tree. Figure 3.1 shows examples of non-crossing and crossing constituents. In particular, if the interval $\langle i, j\rangle$ crosses the interval $\langle k, l\rangle$, then $\langle i, X, j\rangle$ is ruled out and counted as an error. Formally, we say that $\langle i, j\rangle$ crosses $\langle k, l\rangle$ if and only if $i < k < j < l$ or $j < i < k < l$.[1]

If T_C is binary branching, then Consistent Brackets and Bracketed Match are identical (a proof of this fact is given in Appendix 3–A, immediately following this chapter). The following symbols denote the number of constituents that match according to each of these criteria.

> $L = |\{T_C \cap T_G\}|$: the number of constituents in T_G that are correct according to Labelled Match.

[1] This follows Pereira and Schabes (1992), except that in our notation spans include the first element but not the last.

$B = |\{\langle i, X, j\rangle : \langle i, X, j\rangle \in T_G$ and for some $Y, \langle i, Y, j\rangle \in T_C\}|$: the number of constituents in T_G that are correct according to Bracketed Match.

$C = |\{\langle i, X, j\rangle : \langle i, X, j\rangle \in T_G$ and there is no $\langle k, Y, l\rangle \in T_C$ crossing $\langle i, X, j\rangle\}|$: the number of constituents in T_G correct according to Consistent Brackets.

Following are the definitions of the six metrics used in this chapter for evaluating binary branching trees:

- *Labelled Recall Rate* $= L/N_C$
- *Labelled Tree Rate* $= \begin{cases} 1 & \textit{if } L = N_C \\ 0 & \textit{otherwise} \end{cases}$
  This metric is also called the Viterbi criterion or the Exact Match rate.
- *Bracketed Recall Rate* $= B/N_C$
- *Bracketed Tree Rate* $= \begin{cases} 1 & \textit{if } B = N_C \\ 0 & \textit{otherwise} \end{cases}$
- *Consistent Brackets Recall Rate* $= C/N_G$
  This metric is often called the Crossing Brackets rate. In the case where the parses are binary branching, this criterion is the same as the Bracketed Recall rate.
- *Consistent Brackets Tree rate* $= \begin{cases} 1 & \textit{if } C = N_G \\ 0 & \textit{otherwise} \end{cases}$
  This metric is closely related to the Bracketed Tree rate. In the case where the parses are binary branching, the two metrics are the same. This criterion is also called the Zero Crossing Brackets rate.

The preceding six metrics each correspond to cells in the following table:

	Recall	Tree
Consistent Brackets	C/N_G	$\begin{cases} 1 & \textit{if } C = N_G \\ 0 & \textit{otherwise} \end{cases}$
Bracketed	B/N_C	$\begin{cases} 1 & \textit{if } B = N_C \\ 0 & \textit{otherwise} \end{cases}$
Labelled	L/N_C	$\begin{cases} 1 & \textit{if } L = N_C \\ 0 & \textit{otherwise} \end{cases}$

We will define metrics for n-ary branching trees, including Precision and Recall, in Section 3.8.

3.2.3 Maximizing Metrics

Although several metrics are available for evaluating parsers, there is only one metric most parsing algorithms attempt to maximize, namely the Labelled Tree rate. That is, most parsing algorithms attempt to solve the following problem:

$$\arg\max_{T_G} E\left(\left\{\begin{array}{ll} 1 & \textit{if } L = N_C \\ 0 & \textit{otherwise} \end{array}\right. \middle| \mathcal{M}\right) \tag{3.1}$$

Here, E is the expected value operator, and $\mathcal{M}$ is the model under consideration, typically a probabilistic grammar. In words, this maximization finds that tree T_G which maximizes the expected Labelled Tree score, assuming that the sentence was generated by the model. Equation 3.1 is equivalent to maximizing the probability that T_G is exactly right. The assumption that the sentence was generated by the model, or at least that the model is a good approximation, is key to any parsing algorithm. The Viterbi algorithm, the prefix probability estimate of Jelinek and Lafferty (1991), language modeling using PCFGs, and n-best parsing algorithms all implicitly make this approximation: without it, very little can be determined. Whether or not the approximation is a good one is an empirical question. Later, we will show experiments, using grammars obtained in two different ways, that demonstrate that the algorithms we derive using this approximation do indeed work well. Since essentially every equation in this chapter requires this approximation, we will implicitly assume the conditioning on $\mathcal{M}$ from now on.

The maximization of Equation 3.1 is used by most parsing algorithms, including the Labelled Tree (Viterbi) algorithm and stochastic versions of Earley's algorithm (Stolcke, 1993), and variations such as those used in Picky parsing (Magerman and Weir, 1992), and in current state-of-the-art systems, such as those of Charniak (1997) and Collins (1997). Even in probabilistic models not closely related to PCFGs, such as Spatter parsing (Magerman, 1994), Expression 3.1 is still computed. One notable exception is Brill's Transformation-Based Error Driven system (Brill, 1993), which induces a set of transformations designed to maximize the Consistent Brackets Recall (Crossing Brackets) rate. However, Brill's system does not induce a PCFG, and the techniques Brill introduced are not as powerful as modern probabilistic techniques. We will show that by matching the parsing algorithm to the evaluation criteria, better performance can be achieved for the more commonly used PCFG formalism, and variations.

Ideally, one might try to directly maximize the most commonly used evaluation criteria, such as the Consistent Brackets Recall (Crossing Brackets) rate or Consistent Brackets Tree (Zero Crossing Brackets) rate. However, these criteria are relatively difficult to maximize, since it is time-consuming to compute the probability that a particular constituent crosses some constituent in the correct parse. On the other hand, the Bracketed Recall and Bracketed Tree rates are easier to handle, since computing the probability that a bracket matches one in the correct parse is not too difficult. It is plausible that algorithms which optimize these closely related criteria will do well on the analogous Consistent Brackets criteria.

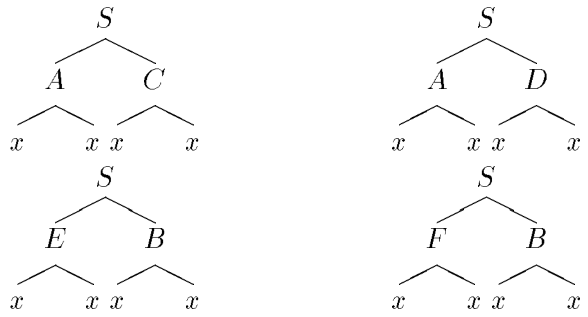
3.2.4 Which Metrics to Use

When building a system, one should use the metric most appropriate for the target problem. For instance, if one were creating a database query system, such as an automated travel agent, then the Labelled Tree (Viterbi) metric would be most appropriate. This is because a single error in the syntactic representation of a query will likely result in an error in the semantic representation, and therefore in an incorrect database query, leading to an incorrect result. For instance, if the user request "Find me all flights on Tuesday" is misparsed with the prepositional phrase attached to the verb, then the system might wait until Tuesday before responding: a single error leads to completely incorrect behavior.

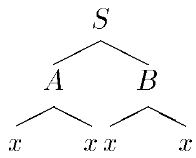
On the other hand, for a machine-assisted translation system, in which the system provides translations, and then a human fluent in the target language manually edits them, Labelled Recall is more appropriate. If the system is given the foreign language equivalent of "His credentials are nothing which should be laughed at," and makes the single mistake of attaching the relative clause at the sentential level, it might translate the sentence as "His credentials are nothing, which should make you laugh." With this translation, the human translator must make some changes, but certainly needs to do less editing than if the sentence were completely misparsed. The more errors there are, the more editing the human translator needs to do. Thus, a criterion such as Labelled Recall is appropriate for this task, where the number of incorrect constituents correlates to application performance.

$$\begin{array}{rcll} S & \rightarrow & A\ C & 0.25 \\ S & \rightarrow & A\ D & 0.25 \\ S & \rightarrow & E\ B & 0.25 \\ S & \rightarrow & F\ B & 0.25 \\ A, B, C, D, E, F & \rightarrow & x\,x & 1.0 \end{array}$$

Sample grammar illustrating Labelled Recall



Four trees in the sample grammar



Labelled Recall tree

Figure 3.2: Four trees of sample grammar, and Labelled Recall tree

3.3 Labelled Recall Parsing

The Labelled Recall parsing algorithm finds that tree T_G that has the highest expected value for the Labelled Recall rate, L/N_C (where L is the number of correctly labelled constituents, and N_C is the number of nodes in the correct parse). Formally, the algorithm finds

$$T_G = \arg\max_T E(L/N_C) \tag{3.2}$$

The difference between the Labelled Recall maximization of Expression 3.2 and the Labelled Tree maximization of Expression 3.1 may be seen from the following example. Figure 3.2 gives an example grammar that generates four trees with equal probability. For the top left tree in Figure 3.2, the probabilities of being correct are S: 100%; A:50%; and C: 25%. Similar counting holds for the other three. Thus, the expected value of L for any of these trees is 1.75.

In contrast, the optimal Labelled Recall parse is shown in the bottom of Figure 3.2. This tree has 0 probability according to the grammar, and thus is non-optimal according to the Labelled Tree rate criterion. However, for this tree the probabilities of each node being correct are S: 100%; A: 50%; and B: 50%. The expected value of L is 2.0, the highest of any tree. This tree therefore optimizes the Labelled Recall rate.

3.3.1 Formulas

We now derive an algorithm for finding the parse that maximizes the expected Labelled Recall rate. We do this by expanding Expression 3.2 out into a probabilistic form, converting this into a recursive equation, and finally creating an equivalent dynamic programming algorithm.

We begin by rewriting Expression 3.2, expanding out the expected value operator, and removing the $\frac{1}{N_C}$, which is the same for all T_G, and so plays no role in the maximization.

$$\arg\max_{T_G} \sum_{T_C} P(T_C \mid w_1...w_n)\ |T_G \cap T_C| \tag{3.3}$$

This can be further expanded to

$$\arg\max_{T_G} \sum_{T_C} P(T_C \mid w_1...w_n) \sum_{\langle i,X,j\rangle \in T_G} \begin{cases} 1 & \textit{if}\ \langle i,X,j\rangle \in T_C \\ 0 & \textit{otherwise} \end{cases} \tag{3.4}$$

Now, given a Probabilistic Context-Free Grammar G with start symbol S, the following equality holds:

$$P(S \stackrel{*}{\Rightarrow} w_1...w_{i-1} X w_j...w_n | w_1...w_n) = \sum_{T_C} P(T_C | w_1...w_n) \times \begin{cases} 1 & \textit{if}\ \langle i,X,j\rangle \in T_C \\ 0 & \textit{otherwise} \end{cases}$$

By rearranging the summation in Expression 3.4 and then substituting this equality, we get

$$\arg\max_{T_G} \sum_{\langle i,X,j\rangle \in T_G} P(S \stackrel{*}{\Rightarrow} w_1...w_{i-1} X w_j...w_n | w_1...w_n)$$

At this point, it is useful to recall the inside and outside probabilities, introduced in Section 1.2. Recall that the inside probability is defined as $inside(i, X, j) = P(X \stackrel{*}{\Rightarrow} w_i...w_{j-1})$ and the outside probability is $outside(i, X, j) = P(S \stackrel{*}{\Rightarrow} w_1...w_{i-1} X w_j...w_n)$.

Let us define a new symbol, $g(i, X, j)$.

$$\begin{aligned} g(i, X, j) &= P(S \stackrel{*}{\Rightarrow} w_1...w_{i-1} X w_j...w_n | w_1...w_n) \\ &= \frac{P(S \stackrel{*}{\Rightarrow} w_1...w_{i-1} X w_j...w_n) \times P(X \stackrel{*}{\Rightarrow} w_i...w_{j-1})}{P(S \stackrel{*}{\Rightarrow} w_1...w_n)} \\ &= \frac{outside(i, X, j) \times inside(i, X, j)}{inside(1, S, n+1)} \end{aligned}$$

Now, the definition of a Labelled Recall Parse can be rewritten as

$$\arg\max_{T_G} \sum_{\langle i,X,j\rangle \in T_G} g(i, X, j)$$

3.3.2 Pseudocode Algorithm

Given the values of $g(i, X, j)$, it is a simple matter of dynamic programming to determine the parse that maximizes the Labelled Recall rate. Define

$$MAXC(i, j) = \max_X g(i, X, j) + \begin{cases} \max_{k \text{ s.t. } i \leq k < j} MAXC(i, k) + MAXC(k, j) & \textit{if } i \neq j-1 \\ 0 & \textit{if } i = j-1 \end{cases} \tag{3.5}$$

We now show that this recursive equation is correct, with a simple proof. We will write $T_{i,j}$ to represent a subtree covering $w_i...w_{j-1}$ and we will write $L(T_{i,j})$ to represent a function giving the expected number of correctly labelled constituents in $T_{i,j}$.

Theorem 3.1

$$MAXC(i, j) = \max_{T_{i,j}} L(T_{i,j})$$

Proof The proof is by induction over the size of the parse tree, $j - i$. The base case is parse trees of size 1, when $i = j - 1$. In this case, there is a single constituent in the possible parse trees. Thus,

$$\begin{aligned}
\max_{T_{i,j}} L(T_{i,j}) &= \max_{\{\langle i,X,j\rangle\}} L(\{\langle i,X,j\rangle\}) \\
&= \max_{\{\langle i,X,j\rangle\}} g(i,X,j) \\
&= \max_{X} g(i,X,j) \\
&= MAXC(i,j)
\end{aligned}$$

completing the base case.

In the inductive step, assume the theorem for lengths less than $j-i$. Then, we notice that for trees of size greater than one, we can always break down a maximal subtree

$$T'_{i,j} = \arg\max_{T_{i,j}} L(T_{i,j})$$

into a root node, $\langle i,X,j\rangle$ and two smaller maximal child trees:

$$T'_{i,k} = \arg\max_{T_{i,k}} L(T_{i,k})$$

$$T'_{k,j} = \arg\max_{T_{k,j}} L(T_{k,j})$$

Notice that if we were trying to maximize the Labelled Tree rate, the most probable child trees would depend on the parent nonterminal. However, when we maximize the Labelled Recall rate, the child trees do not depend on the parent, as was illustrated in Figure 3.2, where we showed that in fact, the joint probability of the parent nonterminal and child trees could even be zero. Thus,

$$\begin{aligned}
\max_{T_{i,j}} L(T_{i,j}) &= \max_{X} g(i,X,j) + \max_{k}\left(\max_{T_{i,k}} L(T_{i,k}) + \max_{T_{k,j}} L(T_{k,j})\right) \\
&= \max_{X} g(i,X,j) + \max_{k}\left(MAXC(i,\,k) + MAXC(k,\,j)\right) \\
&= MAXC(i,j) \ \mathit{if}\ i \neq j-1
\end{aligned}$$

which completes the inductive step. $\diamond$

Notice that $MAXC(1, n{+}1)$ contains the score of the best parse according to the Labelled Recall rate.

Equation 3.5 can be converted into a dynamic programming algorithm as shown in

float $maxC[1..n, 1..n+1] := 0$;
for $length := 1$ **to** n
    for $i := 1$ **to** $n - length + 1$
        $j := i + length$;
        $maxG := \max_X g(i, X, j)$
        if $length \neq 1$
            $bestSplit := \max_{k|i<k<j} maxC[i, k] + maxC[k, j]$
        else
            $bestSplit := 0$;
        $maxC[i, j] := maxG + bestSplit$;

Figure 3.3: Labelled Recall Algorithm

Figure 3.3. For a grammar with r rules and k nonterminals, the run time of this algorithm is $O(n^3 + kn^2)$ since there are two layers of outer loops, each with run time at most n, and an inner loop, over nonterminals and n. However, the overall runtime is dominated by the computation of the inside and outside probabilities, which takes time $O(rn^3)$.

By modifying the algorithm slightly to record the actual split used at each node, we can recover the best parse. The entry $maxC[1, n+1]$ contains the expected number of correct constituents, given the model.

3.3.3 Item-Based Description

For those not familiar with the notation of Chapter 2, this section and succeeding references to item-based descriptions may be skipped without loss of continuity.

It is also possible to specify the Labelled Recall algorithm using an item-based description, as in Chapter 2, although we will need to slightly extend the notation of item-based descriptions to do so. These same extensions will be necessary in Chapter 5 when we describe global thresholding and multiple-pass parsing. There are two extensions that will be required. First, we may have more than one goal item, leading to each item having a separate outside value for each goal item. Second, we will need to be able to make reference to the inside-outside value of an item. We will denote the forwards value of an item $[x]$ in the inside semiring by $V_{in}([x])$ and its reverse value with goal item $[goal]$ by $Z_{in}([x], [goal])$. We will abbreviate the quantity $\frac{V_{in}([x])Z_{in}([x],[goal])}{V_{in}[goal]}$ by $\frac{VZ}{V}_{in}([x], [goal])$.

Figure 3.4 gives the Labelled Recall item-based description. The description is similar

Item form:

$[i, A, j]$	inside semiring
$[i, A, j]^*$	arctic semiring, or similar

Primary Goal:

$[1, S, n+1]^*$

Secondary Goal:

$[1, S, n+1]$

Rules:

$$\frac{R(A \to w_i)}{[i, A, i+1]} \qquad \text{Unary}$$

$$\frac{R(A \to BC) \qquad [i, B, k] \qquad [k, C, j]}{[i, A, j]} \qquad \text{Binary}$$

$$\frac{R(A, w_i, \frac{VZ}{V}{}_{in}([i, A, i+1], [1, S, n+1])}{[i, A, i+1]^*} \qquad \text{Unary Labels}$$

$$\frac{R(A, B, C, \frac{VZ}{V}{}_{in}([i, A, j], [1, S, n+1])) \qquad [i, B, k]^* \qquad [k, C, j]^*}{[i, A, j]^*} \, xo \qquad \text{Binary Labels}$$

Figure 3.4: Labelled Recall Description

to the procedural version. The first step is to compute the inside-outside values. We thus have the usual items $[i, A, j]$ in the inside semiring, and the usual unary and binary rules for CKY parsing. The inside-outside values will simply be $\frac{VZ}{V}{}_{in}([i, A, j], [1, S, n+1])$, using the notation we just defined. Next, we need to find, for each i, A, j, the inside-outside value and the best sum of splits for the span. These will be stored in items of the form $[i, A, j]^*$. To get the inside-outside value for $\langle i, A, j\rangle$, we use a special rule value function,

$$R(A, B, C, \tfrac{VZ}{V}{}_{in}([i, A, j], [1, S, n+1]))$$

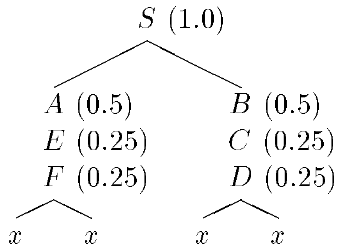
The value of this function is just the inside-outside value of the constituent $\langle i, A, j\rangle$, which is passed into the function as the last argument. We also need to find the best sum of splits for the span, i.e., a maximum over a sum. For this, we use a special semiring, the arctic semiring, whose operations are max, + (as defined in Section 2.2.1.) Notice that our two different item types use two different semirings: the inside semiring for $[i, A, j]$, and the arctic semiring for $[i, A, j]^*$. Using the arctic semiring and our special rule value function, the unary and binary labels rules compute the best nonterminal label and best sum of splits for each span. These rules are identical to the usual unary and binary rules, except that they use our special rule value function that gives the inside-outside value of the relevant constituent.

We must, of course, find the inside and outside values for the $[i, A, j]$ before we can compute the $[i, A, j]^*$ values; this means that the order of interpretation is changed as well. Normally, the order of interpretation is simply all of the forward values, in order, followed by all of the reverse values, in the reverse order. In this new version, we first have all of the forward values of items $[i, A, j]$, and then, in reverse order, all of the reverse values of these items; next, we have all of the forward values of $[i, A, j]^*$, and finally, optionally, in reverse order all of the reverse values of $[i, A, j]^*$.

If we use the arctic semiring, we get only the maximum expected number of correctly labelled constituents, but not the tree which gives this number. To get this tree, we would use the arctic-derivation semiring. As usual, there are advantages to describing the algorithm with item-based descriptions. For instance, we can easily compute n-best derivations, just by using the arctic-top-n semiring.

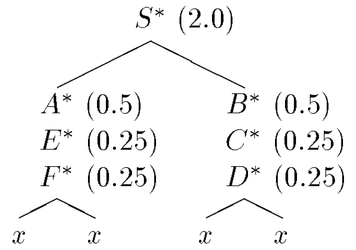
A short example may help clarify the algorithm. Consider the example grammar of

Figure 3.2. For this grammar, we have that the inside-outside values of the $[i, X, j]$, which equal $\frac{VZ}{V}_{in}([i, X, j], [1, S, n + 1]$, are as follows:



$$\begin{aligned} [1, S, 5] &= 1.0 \\ [1, A, 3] &= 0.5 \\ [1, E, 3] &= 0.25 \\ [1, F, 3] &= 0.25 \\ [3, B, 5] &= 0.5 \\ [3, C, 5] &= 0.25 \\ [3, D, 5] &= 0.25 \end{aligned}$$

We have that the forward arctic values of the $[i, X, j]^*$ are:



$$\begin{aligned} [1, S, 5]^* &= 2.0 \\ [1, A, 3]^* &= 0.5 \\ [1, E, 3]^* &= 0.25 \\ [1, F, 3]^* &= 0.25 \\ [3, B, 5]^* &= 0.5 \\ [3, C, 5]^* &= 0.25 \\ [3, D, 5]^* &= 0.25 \end{aligned}$$

The key element is $[1, S, 5]^*$, which is derived using the binary labels rule instantiated with:

$$\frac{R(S, B, C, 1.0)) \qquad [1, B, 3]^* \qquad [3, C, 5]^*}{[1, S, 5]^*}$$

which, computed in the arctic semiring where the multiplicative operator is addition, has the value $1.0 + 0.5 + 0.5 = 2.0$.

3.4 Bracketed Recall Parsing

The Labelled Recall algorithm maximizes the expected number of correct labelled constituents. However, many commonly used evaluation metrics, such as the Consistent Brackets Recall (Crossing Brackets) rate, ignore labels. Similarly, some grammar induction algorithms, such as those used by Pereira and Schabes (1992) do not produce meaningful labels. In particular, the Pereira and Schabes method induces a grammar from the brackets in the treebank, ignoring the labels in the treebank. While the grammar they induce has labels, these labels are not related to those in the treebank. Thus, while the Labelled Recall algorithm could be used with these grammars, perhaps maximizing a criterion that is more closely tied to the task will produce better results. Ideally, we would maximize the Consistent Brackets Recall rate directly. However, since it is time-consuming to deal with Consistent Brackets, as described in Section 3.2.3, we instead use the closely related Bracketed Recall rate.

For the Bracketed Recall algorithm, we find the parse that maximizes the expected Bracketed Recall rate, B/N_C. (Remember that B is the number of brackets that are correct, and N_C is the number of constituents in the correct parse.)

$$T_G = \arg\max_T E(B/N_C) \tag{3.6}$$

Following a derivation similar to that used for the Labelled Recall algorithm, we can rewrite Equation 3.6 as

$$T_G = \arg\max_T \sum_{\langle i,j\rangle \in T} \sum_X P(S \stackrel{*}{\Rightarrow} w_1...w_{i-1} X w_j...w_n | w_1...w_n)$$

The algorithm for Bracketed Recall parsing is almost identical to that for Labelled Recall parsing. The only required change is to sum, rather than maximize, over the symbols X to calculate $maxG$, substituting in the following line:

$$maxG := \sum_X g(i, X, j);$$

The Labelled Recall item-based description can be easily converted to a Bracketed Recall item-based description, with the addition of one new item type and one new rule. Figure 3.5 gives an item-based description for the Bracketed Recall algorithm. There are now 3 item types: $[i, A, j]$, which has the usual meaning; $[i, j]^\dagger$ which has the value of $\sum_A g(i, A, j)$; and $[i, j]^*$ which has the value of $MAXC(i, j)$. The summation rule sums over items of type $[i, A, j]$ to produce items of the type $[i, j]^\dagger$.

3.5 Experimental Results

We describe two experiments that tested these algorithms. The first uses a grammar without meaningful nonterminal symbols, and compares the Bracketed Recall algorithm to the traditional Labelled Tree (Viterbi) algorithm. The second uses a grammar with meaningful nonterminal symbols and performs a three-way comparison between the Labelled Recall,

Item form:

$[i, A, j]$	inside semiring
$[i, j]^{\dagger}$	inside semiring
$[i, j]^{*}$	arctic semiring, or similar

Primary Goal:

$[1, S, n+1]^{*}$

Secondary Goal:

$[1, S, n+1]$

Rules:

$$\frac{R(A \to w_i)}{[i, A, i+1]} \quad \text{Unary}$$

$$\frac{R(A \to BC) \qquad [i, B, k] \qquad [k, C, j]}{[i, A, j]} \quad \text{Binary}$$

$$\frac{\frac{VZ}{V}_{in}([i, A, j], [1, S, n+1])}{[i, j]^{\dagger}} \quad \text{Summation}$$

$$\frac{R(w_i, [i, i+1]^{\dagger})}{[i, i+1]^{*}} \quad \text{Unary Brackets}$$

$$\frac{R(B, C, [i, j]^{\dagger}) \qquad [i, k]^{*} \qquad [k, j]^{*}}{[i, j]^{*}} \quad \text{Binary Brackets}$$

Figure 3.5: Bracketed Recall Description

Bracketed Recall, and Labelled Tree algorithms. These experiments show that use of an algorithm matched appropriately to the evaluation criterion can lead to as much as a 10% reduction in error rate.

3.5.1 Grammar Induced by Pereira and Schabes method

We duplicated the experiment of Pereira and Schabes (1992). Pereira and Schabes trained a grammar from a bracketed form of the TI section of the ATIS corpus[2] using a modified form of the inside-outside algorithm. They then used the Labelled Tree (Viterbi) algorithm to select the best parse for sentences in held out test data. We repeated the experiment, inducing the grammar the same way. However, during the testing phase, we ran both the Labelled Tree and Labelled Recall algorithm for each sentence. In contrast to previous research, we repeated the experiment ten times, with different random splits of the data into training set and test set, and different random initial conditions each time. Note that in one detail our scoring for experiments differs from the theoretical discussion in the paper, so that we can more closely follow convention. In particular, constituents of length one are not counted in recall measures, since these are trivially correct for most criteria.

In three test sets there were sentences with terminals not present in the matching training set. The four sentences (out of 880) containing these terminals could not be parsed, and in the following analysis were assigned right branching, period high structure. This of course affects the Labelled Recall and Labelled Tree algorithms equally.

Table 3.1 shows the results of running this experiment, giving the minimum, maximum, mean, range, and standard deviation for three criteria, Consistent Brackets Recall, Consistent Brackets Tree, and Bracketed Recall. We also computed, for each split of the data, the difference between the Bracketed Recall algorithm and the Labelled Tree algorithm on each criterion. Notice that on each criterion the minimum of the differences is negative, meaning that on some data set the Labelled Tree algorithm worked better, and the maximum of the differences is positive, meaning that on some data set the Bracketed Recall algorithm

[2]Most researchers throw out a few sentences because of problems aligning the part of speech and parse files, or because of labellings of discontinuous constituents, which are not usable with the Crossing Brackets rate. The difficult data was cleaned up and used in these experiments, rather than thrown out. A diff file between the original ATIS data and the cleaned up version, in a form usable by the "ed" program, is available by anonymous FTP from `ftp://ftp.deas.harvard.edu/pub/goodman/atis-ed/` `ti_tb.par-ed` and `ti_tb.pos-ed`. The number of changes made was small: the diff files sum to 457 bytes, versus 269,339 bytes for the original files, or less than 0.2%.

Criteria	Min	Max	Range	Mean	StdDev
Labelled Tree Algorithm					
Cons Brack Rec	86.06%	93.27%	7.20%	90.13%	2.57%
Cons Brack Tree	51.14%	77.27%	26.14%	63.98%	7.96%
Brack Rec	71.38%	81.88%	10.50%	75.87%	3.18%
Bracketed Recall Algorithm					
Cons Brack Rec	88.02%	94.34%	6.33%	91.14%	2.22%
Cons Brack Tree	53.41%	76.14%	22.73%	63.64%	7.82%
Brack Rec	72.15%	80.69%	8.54%	76.03%	3.14%
Bracketed Recall - Labelled Tree					
Cons Brack Rec	-1.55%	2.45%	4.00%	1.01%	1.07%
Cons Brack Tree	-3.41%	3.41%	6.82%	-0.34%	2.34%
Brack Rec	-1.34%	2.02%	3.36%	0.17%	1.20%

Table 3.1: Labelled Tree (Viterbi) versus Bracketed Recall for P&S

worked better. The only criterion for which there was a statistically significant difference between the means of the two algorithms is the Consistent Brackets Recall rate, which was significant to the 2% significance level (paired t-test). Thus, use of the Bracketed Recall algorithm leads to a 10% reduction in error rate.

In addition, the performance of the Bracketed Recall algorithm was also qualitatively more appealing. Figure 3.6 shows typical results. Notice that the Bracketed Recall algorithm's Consistent Brackets rate (versus iteration) is smoother and more nearly monotonic than the Labelled Tree algorithm's. The Bracketed Recall algorithm also gets off to a much faster start, and is generally (although not always) above the Labelled Tree level. For the Labelled Tree rate, the two are usually very comparable.

3.5.2 Grammar Induced by Counting

The replication of the Pereira and Schabes experiment was useful for testing the Bracketed Recall algorithm. However, since that experiment induces a grammar with nonterminals not comparable to those in the training, a different experiment is needed to evaluate the Labelled Recall algorithm, one in which the nonterminals in the induced grammar are the same as the nonterminals in the test set.

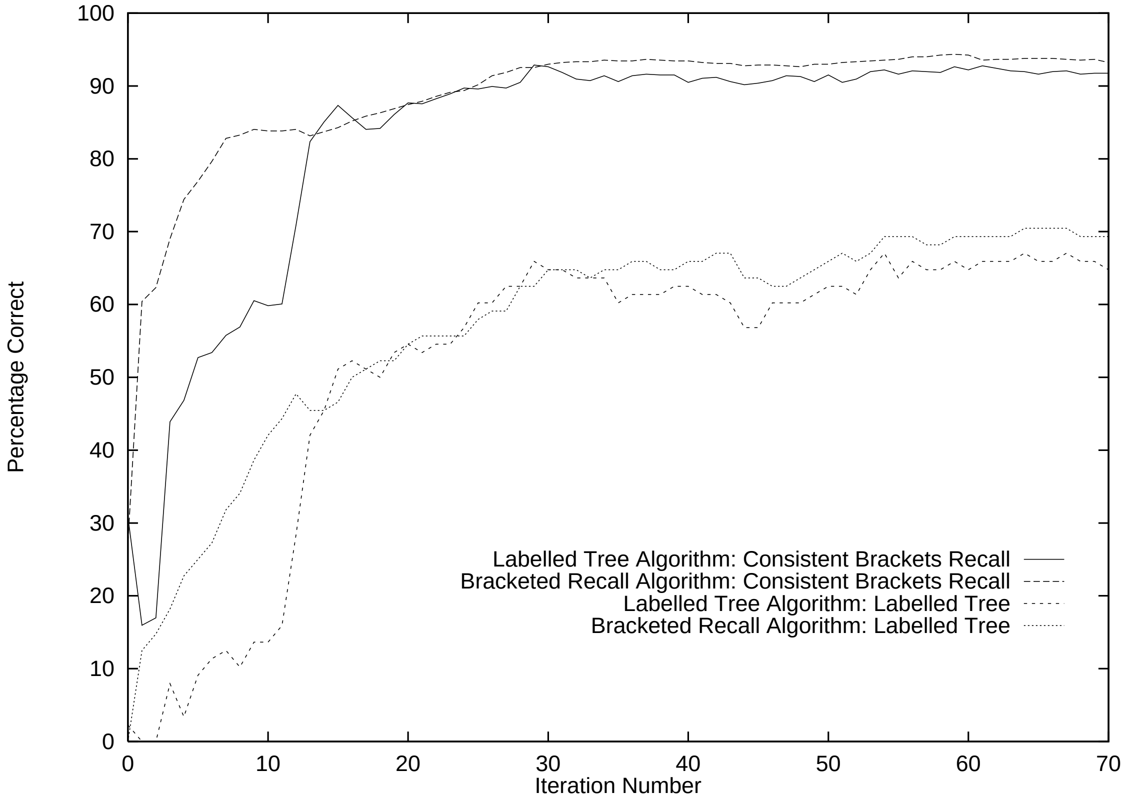


Figure 3.6: Labelled Tree versus Bracketed Recall in Pereira and Schabes Grammar

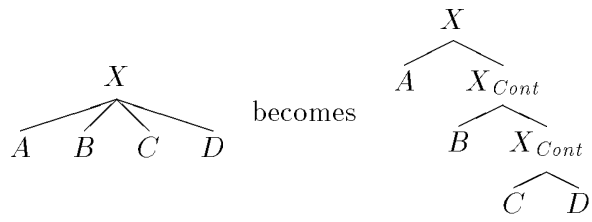


Figure 3.7: Conversion of Productions to Binary Branching

	Criterion				
Algorithm	Label Tree	Label Recall	Brack Recall	Cons Brack Recall	Cons Brack Tree
Label Tree	*4.54%*	48.60%	60.98%	66.35%	12.07%
Label Recall	3.71%	*49.66%*	61.34%	68.39%	11.63%
Bracket Recall			*61.63%*	68.17%	11.19%

Table 3.2: Grammar Induced by Counting: Three Algorithms Evaluated on Five Criteria

Grammar Induction by Counting

For this experiment, a very simple grammar was induced by counting, using the Penn Tree Bank, version 0.5. In particular, the trees were first made binary branching, by removing epsilon productions, collapsing singleton productions, and by converting n-ary productions ($n > 2$), as in Figure 3.7. The resulting trees were treated as the "Correct" trees in the evaluation.

A grammar was then induced in a straightforward way from these trees, simply by giving one count for each observed production. No smoothing was done. There were 1805 sentences and 38610 nonterminals in the test data. (The resulting grammar undergenerated somewhat, being unable to parse approximately 9% of the test data. The unparsable data were assigned a right branching structure with their right-most element attached high. Notice that the Labelled Tree (Viterbi), Labelled Recall, and Bracketed Recall algorithms all fail on exactly the same sentences (since the inside-outside calculation fails exactly when the Labelled Tree calculation fails). Thus, this default behavior affects all sentences equally.

Results

Table 3.2 shows the results of running all three algorithms, evaluating against five criteria. Notice that for each algorithm, for the criterion that it optimizes it is the best algorithm. That is, the Labelled Tree algorithm is the best for the Labelled Tree rate, the Labelled Recall algorithm is the best for the Labelled Recall rate, and the Bracketed Recall algorithm is the best for the Bracketed Recall rate.

3.6 NP-Completeness of Bracketed Tree Maximization

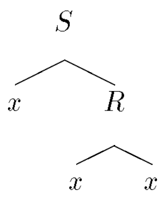
In this section, we show the NP completeness of the Bracketed Tree Maximization problem. Bracketed Tree Maximization is the problem of finding that bracketing which has the highest expected score on the Bracketed Tree criterion. In other words, it is that tree which has the highest probability of being exactly correct, according to the model, ignoring nonterminal labels. Formally, the Bracketed Tree Maximization problem is to compute

$$\arg\max_{T_G} E\left(\begin{cases} 1 & \textit{if } B = N_C \\ 0 & \textit{otherwise} \end{cases}\right)$$

The Bracketed Tree criterion can be distinguished from the Labelled Tree (Viterbi) criterion in the following example:

$$\begin{array}{rcll} S & \rightarrow & L\ x & 0.3 \\ S & \rightarrow & M\ x & 0.3 \\ S & \rightarrow & x\ R & 0.4 \\ L, M, R & \rightarrow & x\ x & 1.0 \end{array}$$

Here, given an input of xxx, the Labelled Tree parse is



and thus the Labelled Tree bracketing is $[x[xx]]$, with a 40% chance of being correct, assuming that the string was produced by the model. On the other hand, the bracketing $[[xx]x]$ has a 60% chance of being correct. If our goal is to maximize the probability that the bracketing is exactly correct, then we would like an algorithm that returns the second bracketing, rather than the first. (Our previous algorithm, the Bracketed Recall algorithm, may return bracketings with 0 probability: it does not solve this problem.) We will show that maximizing Bracketed Recall is NP-Complete. It will be easier to prove this if we first prove a related theorem about Hidden Markov Models (HMMs). The proof we give here is similar to one of Sima'an (1996a), that maximizing the Labelled Tree rate for a Stochastic Tree Substitution Grammar is NP-Complete.

We note that for binary branching trees, there is a crossing bracket in any tree that does not exactly match. Thus, for binary branching trees, Bracketed Tree Maximization is equivalent to Zero Crossing Brackets Rate Maximization. Thus we will implicitly also be showing that Zero Crossing Brackets Rate Maximization is NP-Complete.

3.6.1 NP-Completeness of HMM Most Likely String

HMM MOST LIKELY STRING (HMM-MLS)
INSTANCE: An HMM specification (with probabilities specified as fractions $\frac{x}{2^y}$), a length n in unary, and a probability p.
QUESTION: Is there some string of length n such that the specified HMM produces the string with probability at least p?

The proof that Bracketed Tree Maximization is NP-Complete is easier to understand if we first show the NP-Completeness of a simpler problem, namely finding whether the most likely string of a given length n output by a Hidden Markov Model has probability at least p. (A different problem, that of finding the most likely state sequence of an HMM, and the corresponding most likely output, can of course be solved easily. However, since the same string can be output by several state sequences, the most likely string problem is much harder.)

We note that a related problem, Most Likely String Without Length, in which the length of the string is not prespecified, can be shown NP-hard by a very similar proof. However, it cannot be shown NP-Complete by this proof, since the most likely string could be exponentially long, and thus the generate and test method cannot be used to show that the problem is in NP.

We show the NP-Completeness of HMM Most Likely String by a reduction of the NP-Complete problem 3-Sat. The problem 3-Sat is whether or not there is a way to satisfy a formula of the form $F_1 \wedge F_2 \wedge ... \wedge F_n$ where F_i is of the form $(\alpha_{i1} \vee \alpha_{i2} \vee \alpha_{i3})$. Here, the

α_{ij} represent literals, either x_k or $\neg x_k$. An example of 3-Sat is the question of whether the following formula is satisfiable:

$$(\neg x_1 \vee \neg x_2 \vee x_4) \wedge (x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_4 \vee x_3 \vee x_1)$$

We will show how to construct, for any 3-Sat formula, an HMM that has a high probability output if and only if the formula is satisfiable.

The most obvious way to pursue this reduction is to create a network whose output corresponds to satisfying assignments of the clauses, e.g. a string of T's and F's, one letter for each variable, denoting whether that variable is true or false. There will be one section of the network for each clause, and the probabilities will be assigned in such a way that for the output probability to sum sufficiently high, for every clause the maximal string must have a path through the corresponding subnetwork.

The problem with this technique is that such a string may be output by multiple paths through the subnetworks corresponding to some of the clauses, and zero paths through the parts of the network leading to other clauses. What we must do is to ensure that there is exactly one path through each clause's subnetwork. In order to do this, we augment the output with an initial sequence of the digits 1, 2, and 3. Each digit specifies which variable (the first, second, or third) of each clause satisfied that clause. This strategy allows construction of subnetworks for each clause with at most one path. Then, if there is a string whose probability indicates that it has n paths through the HMM, we know that the string satisfies every clause, and therefore satisfies the formula. Any string that does not correspond to a satisfying assignment will have fewer than n paths, and will not have a sufficiently high probability.

Theorem 3.2

The HMM Most Likely String problem is NP-Complete

Proof To show that HMM Most Likely String is in NP is trivial: guess a most likely string of the given length n; then compute its probability by well known polynomial-time dynamic programming techniques (the forward algorithm); then verify that the probability is at least p.

To show NP-Completeness, we show how to reduce a 3-Sat formula to an HMM Most Likely String problem. Let f represent the number of disjunctions and v represent the

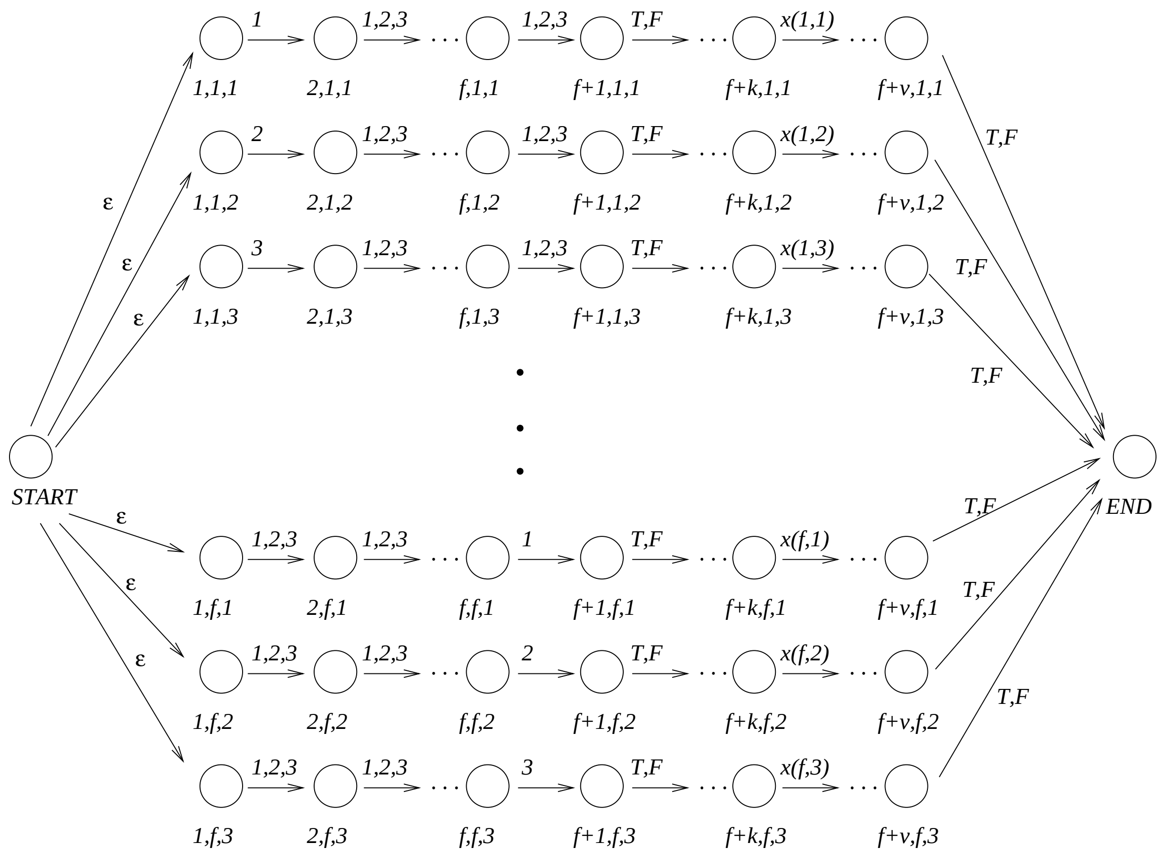


Figure 3.8: Portion of HMM corresponding to a 3-Sat formula

number of variables in the formula to satisfy. Create an HMM with $3f(f+v)+2$ states, as follows. The states will have the following names: $\underset{END}{\bigcirc}$, $\underset{START}{\bigcirc}$, and $\underset{h,i,j}{\bigcirc}$. h ranges from 1 to $f+v$; i ranges from 1 to f; and j ranges from 1 to 3. Two of the states are a start and an end state, with an equiprobable epsilon transition from the start state to the $3f$ states of the form $\underset{1,i,j}{\bigcirc}$.

The basic form of the network is

$$\underset{START}{\bigcirc} \xrightarrow{\epsilon} \underset{1,i,j}{\bigcirc} \xrightarrow{1,2,3} \cdots \underset{i,i,j}{\bigcirc} \xrightarrow{j} \cdots \underset{f,i,j}{\bigcirc} \xrightarrow{1,2,3} \underset{f+1,i,j}{\bigcirc} \xrightarrow{T,F} \cdots \underset{f+k,i,j}{\bigcirc} \xrightarrow{x(i,j)} \cdots \underset{f+v,i,j}{\bigcirc} \xrightarrow{T,F} \underset{END}{\bigcirc}$$

where $x(i,j)$ is T if α_{ij} is of the form x_k, and F if it is of the form $\neg x_k$. Figure 3.8 shows a section of the HMM, giving the parts that correspond to the first and last clause. The notation $\bigcirc \overset{1,2,3}{\rightarrow} \bigcirc$ indicates a state transition with equal probabilities of emitting the symbols 1, 2, or 3. Similarly $\bigcirc \overset{T,F}{\rightarrow} \bigcirc$ indicates a state transition with equal probabilities of emitting T or F.

We now show that if the given 3-Sat formula is satisfiable, then the constructed HMM has a string with probability $\frac{1}{3f} \times \frac{1}{2^{v-1}}$, whereas if it is not satisfiable, the most likely string will have lower probability. The proof is as follows. First, assume that there is some way to satisfy the 3-Sat formula. Let X_i denote T if the variable x_i takes on the value true, and let it denote F otherwise, under some assignment of values to variables that satisfies the formula. Also, let J_i denote the number 1, 2, or 3 respectively depending on whether for formula i, α_{i1}, α_{i2}, or α_{i3} respectively satisfies formula i. If J_i could take on more than one value by this definition (that is, if F_i is satisfied in more than one way) then arbitrarily, we assign it to the lowest of these three. Since we are assuming here that the formula *is* satisfied under the assignment of values to variables, at least one of α_{i1}, α_{i2}, or α_{i3} must satisfy each clause.

Then, the string

$$J_1 J_2 \cdots J_f X_1 X_2 \cdots X_v$$

will be a most probable string. There may be other most probable strings, but all will have the same probability and satisfy the formula. (Example: the string $131FTFT$ would be the most likely string output by an HMM corresponding to the sample formula.)

This string will have probability $\frac{1}{3f} \times \frac{1}{2^{v-1}}$ because there will be routes emitting that string through exactly f of the $3f$ subnetworks, and each route will have probability $\frac{1}{f} \times \frac{1}{3f} \times \frac{1}{2^{v-1}}$.

On the other hand, assume that the formula is not satisfiable. Now, the most likely string will have a lower probability. Obviously, the string could not have higher probability, since the best path through any subnetwork has probability $\frac{1}{3f-1} \times \frac{1}{2^{v-1}}$ and it is not possible to have routes through more than f of the $3f$ subnetworks. Therefore, the probability can be at most $\frac{1}{3f} \times \frac{1}{2^{v-1}}$. However, it cannot be this high, since if it were, then the string would have a path through f of the $3f$ networks, and would therefore correspond to a satisfaction of the formula. Thus, its probability must be lower. $\diamond$

3.6.2 Bracketed Tree Maximization is NP-Complete

Having shown the NP-Completeness of the HMM Most Likely String problem, we can show the NP-Completeness of Bracketed Tree Maximization. To phrase this as a language

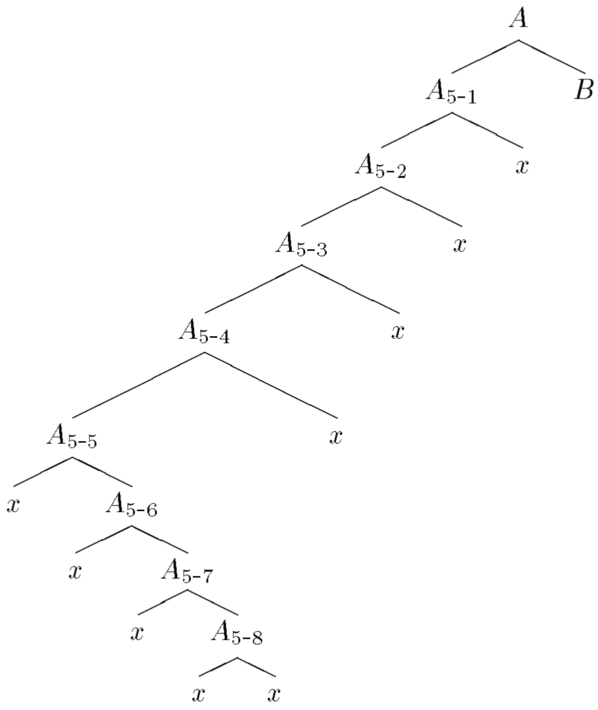


Figure 3.9: Tree corresponding to an output of symbol 5 from 8 symbol alphabet

problem, we rephrase it as the question of whether or not the best parse has an expected score of at least p, according to the Bracketed Tree rate criterion.

Bracketed Tree Maximization (BTM)

Instance: A Probabilistic Context-Free Grammar G, a string $w_1 ... w_n$, and a probability p.

Question: Is there a T_G such that $E\left(\left\{\begin{array}{ll} 1 & \textit{if } B = N_C \\ 0 & \textit{otherwise} \end{array}\right.\right) \geq p$?

Theorem 3.3

The Bracketed Tree Maximization problem is NP-Complete.

Proof Using the generate and test method, it is clear that this problem is in NP. To show completeness, we show that for every HMM, there is an equivalent PCFG that produces bracketings with the same probability that the HMM produces symbols. We do this by mapping states of the HMM to nonterminals of the grammar; we map each output symbol of the HMM to a small subtree with a unique bracket sequence.

The proof is as follows. Number the output symbols of the HMM 1 to k. Assign the start state of the HMM to the start symbol of the PCFG. For each state A with a transition

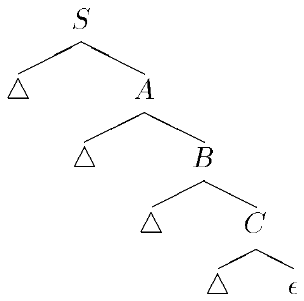


Figure 3.10: Tree corresponding to state sequence $SABC$

emitting symbol i to state B with probability p, include rules in the grammar that first have $k-i+2$ symbols on right branching nodes, followed by $i-1$ symbols on left branching nodes. For instance, for $i=5$ and $k=8$, the tree would look like the tree in Figure 3.9:

Formally, we can write this as.

$$A_i \quad \rightarrow \quad \overbrace{[[[\ldots[[}^{i} x \, \overbrace{[x[x\ldots[x}^{k-i} x \, \overbrace{]]\ldots]}^{k-i+1} \, \overbrace{x]x]\ldots x]}^{i-1} B \qquad (p)$$

Also, for each final state A, include a rule of the form

$$A \quad \rightarrow \quad \epsilon$$

If an HMM had a state sequence $SABC$, then the corresponding parse tree would look like Figure 3.10, where $\triangle$ represents a subtree corresponding to the HMM's output.

There will be a one-to-one correspondence between parse trees in this grammar, and state-sequence/outputs pairs in the HMM. If we throw away nonterminal information, leaving only brackets, there will be a one-to-one correspondence between bracketed parse trees and HMM outputs. The Bracketed Tree Maximization of a string in this grammar will correspond to the most likely output of the corresponding HMM. Thus, if we could solve the Bracketed Tree Maximization problem, we could also solve the HMM Most Likely String of length n problem. Therefore, Bracketed Tree Maximization is NP-Complete. $\diamond$

Corollary 3.4

Consistent Brackets Tree (Zero Crossing Brackets) Maximization for Binary Branching Trees is NP-Complete.

Proof Bracketed Tree is equivalent to Consistent Brackets Tree if both T_C and T_G are binary branching. $\diamond$

We note that for n-ary branching trees, discussed in Section 3.8, Consistent Brackets Tree maximization is typically achieved by simply placing parentheses at the top-most level, and nowhere else. Thus, it is the binary branching case that is of interest.

3.7 General Recall Algorithm

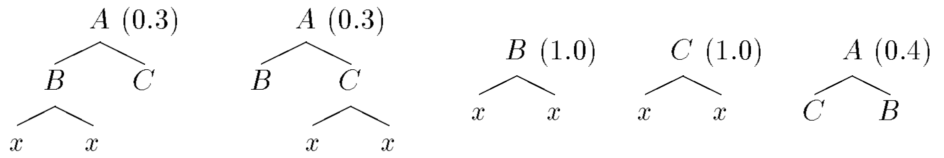
The Labelled Recall algorithm and Bracketed Recall algorithm are both special cases of a more general algorithm, called the General Recall algorithm. In fact, the General Recall algorithm was developed first, for parsing Stochastic Tree Substitution Grammars (STSGs). In this section, we first define STSGs, and then use them to motivate the General Recall algorithm. Next, we show how the other two algorithms reduce to the general algorithm, and finally give some examples of other cases in which one might wish to use the General Recall algorithm.

There are two ways to define a STSG: either as a Stochastic Tree Adjoining Grammar (Schabes, 1992) restricted to substitution operations, or as an extended PCFG in which entire trees may occur on the right hand side, instead of just strings of terminals and non-terminals. A PCFG then, is a special case of an STSG in which the trees are limited to depth 1. In the STSG formalism, for a given parse tree, there may be many possible derivations. Thus, variations on the Viterbi algorithm, which find the best *derivation* may not be the most appropriate. What is really desired is to find the best parse. But best according to what criterion? If the criterion used is the Labelled Tree rate, then the problem is NP-Complete (Sima'an, 1996a), and the only known approximation algorithm is a Monte Carlo algorithm, due to Bod (1993c).[3] On the other hand, if the criterion used is the Labelled Recall rate, then there is an algorithm, the General Recall algorithm.

In Figure 3.11 we give a sample STSG grammar, and two different parses of the string $xxxx$, demonstrating that the most likely derivation differs from the most likely parse.

To use the General Recall algorithm for parsing STSGs, we must first convert the STSG to a PCFG. This conversion is done in the usual way. That is, assign to every internal node

[3] We will show in Section 4.6 that this algorithm has a serious problem: either its accuracy decreases exponentially with sentence length, or its runtime increases exponentially.



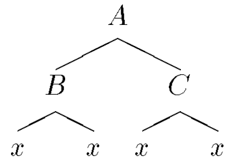
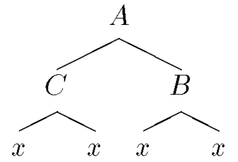
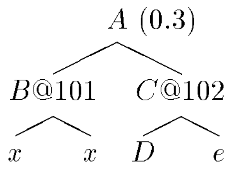
Parse Tree		
Derivation Probability	0.3	0.4
Parse Probability	0.6	0.4

Figure 3.11: Example Stochastic Tree Substitution Grammar and two parses



$$\begin{array}{rcll} A & \rightarrow & B_{101}C_{102} & (0.3) \\ B_{101} & \rightarrow & xx & (1.0) \\ C_{102} & \rightarrow & De & (1.0) \end{array}$$

Example STSG production	Corresponding PCFG productions

Figure 3.12: STSG to PCFG conversion example

of every subtree of the STSG a unique label, which we will indicate with an @ sign. Leave the root and leaf nodes unlabelled, or, equivalently, labelled with a null label. Now, for each root A of a subtree with probability p, and children $B@k$, $C@l$, create a PCFG rule

$$A \rightarrow B_k\, C_l \quad (p)$$

For each internal node $A@j$ of a subtree with children $B@k$, $C@l$, create a PCFG rule

$$A_j \rightarrow B_k\, C_l \quad (1.0)$$

Here, the notation X_j denotes a new nonterminal, created for this node. k or l may be null if the corresponding node is a leaf node. The number in parentheses indicates the probability associated with a given production. Figure 3.12 illustrates this conversion.

Now, if we were to apply the Viterbi algorithm directly to this PCFG, rather than returning the most likely parse tree, it would return a parse tree corresponding to the most likely derivation. Not only that, but the labels in this parse tree would not even be the same as the labels in the original grammar. Of course, we could solve this latter problem by creating a function MAP_STSG that would map from symbols in the PCFG to symbols in the STSG.

$$\begin{aligned} \text{MAP\_STSG}(X) &= X \\ \text{MAP\_STSG}(X_j) &= X \end{aligned}$$

In other words, MAP_STSG removes subscripts.

One way to use this function would be to use the Viterbi algorithm or Labelled Recall algorithms with the PCFG to find the best tree, and then use the function to map each node's label to the original name. But we would presumably get better results following a more principled approach, such as trying to solve the following optimization problem:

$$\arg\max_{T_G} E\left(\sum_{\langle j,X,k\rangle \in T_C} \begin{cases} 1 & \textit{if } \langle j, \text{MAP\_STSG}(X), k\rangle \in T_G \\ 0 & \textit{otherwise} \end{cases} \right)$$

Here, T_C is a parse tree with symbols in the PCFG, while T_G is a parse tree with symbols in the original STSG grammar. This maximization is exactly parallel to the maximization performed by the Labelled Recall algorithm, except that the symbol names are transformed with MAP_STSG before the maximization is performed. In fact, if we define two other mapping functions, MAP_IDENT (the identity mapping function) and MAP_BRACK (the function that maps everything to the same symbol), then the similarity between the three maximizations becomes more clear.

The MAP_IDENT function

$$\text{MAP\_IDENT}(X) \quad = \quad X \tag{3.7}$$

can be used to define the Labelled Recall maximization,

$$\max_{T_G} E\left(\sum_{\langle j,X,k\rangle \in T_C} \begin{cases} 1 & \textit{if } \langle j, \text{MAP\_IDENT}(X), k\rangle \in T_G \\ 0 & \textit{otherwise} \end{cases}\right)$$

while the MAP_BRACK function

$$\text{MAP\_BRACK}(X) = S$$

can be used to define the Bracketed Recall maximization.

$$\max_{T_G} E\left(\sum_{\langle j,X,k\rangle \in T_C} \begin{cases} 1 & \textit{if } \langle j, \text{MAP\_BRACK}(X), k\rangle \in T_G \\ 0 & \textit{otherwise} \end{cases}\right)$$

If we substitute

$$maxG := \max_X \sum_{Y|X=map(Y)} g(i,Y,j);$$

into the Labelled Recall algorithm of Figure 3.3, we get the General Recall algorithm. *map* is a function that maps from nonterminals in the relevant PCFG to nonterminals relevant for the output. It could be any of MAP_STSG, MAP_IDENT, or MAP_BRACK, among others.

Given this general algorithm, other uses for this technique become apparent. For instance, in the latest version of the Penn Treebank, nonterminals annotated with grammatical function are included. That is, rather than using simple nonterminals such as *NP*, more complex nonterminals, such as *NP-SBJ* and *NP-PRD* are used (to indicate subject and predicate). Depending on the application, one might choose to use the annotated nonterminals in a PCFG, but to return only the simpler, unannotated nonterminals (Collins (1997) uses a version of the grammatically annotated nonterminals internally, parsing with the Labelled Tree (Viterbi) algorithm, and then stripping the annotations.) So, we could define a mapping function MAP_SEM.

$$\begin{aligned} \text{MAP\_SEM}(X) &= X \\ \text{MAP\_SEM}(X\text{-}SY) &= X \end{aligned}$$

We could then use the General Recall algorithm with this function. If we used a simpler scheme, such as parsing using the Viterbi algorithm, and then mapping the resulting parse, problems might occur. For instance, since noun phrases are divided into classes, while verb phrases are not, an inadvertent bias against noun phrases will have been added into the system. Using the General Recall algorithm removes such biases, since all the different kinds of noun-phrases are summed over, and is theoretically better motivated.

An item-based description of the General Recall algorithm is given in Figure 3.13. The General Recall description is the same as the Bracketed Recall description, except that we now add nonterminals to the items, and use the function *map*. In the next chapter, we will show how to use the General Recall algorithm to greatly speed up Data-Oriented Parsing (Bod, 1992).

3.8 N-Ary Branching Parse Trees

Previously, we have been discussing binary branching parse trees. However, in practice, it is often useful to have trees with more than two branches. In this section, we discuss n-ary branching parse trees (denoted by the symbol $\overset{n}{T}$). In general, n-ary branching parse trees can have any number of branches, including zero or one, but the discussion in this chapter will be limited to those trees where every node has at least two branches.

We begin by discussing why, with n-ary branching parse trees, different evaluation metrics should be used from those for binary branching ones. In particular, we discuss Consistent Brackets Recall (Crossing Brackets) and Consistent Brackets Tree (Zero Crossing Brackets) versus Bracketed or Labelled Precision and Recall. We then go on to discuss approximations for Bracketed and Labelled Precision and Recall, the Bracketed Combined

Item form:

$[i, A, j]$	inside semiring
$[i, A, j]^{\dagger}$	inside semiring
$[i, A, j]^{*}$	arctic semiring, or similar

Primary Goal:

$[1, S, n+1]^{*}$

Secondary Goal:

$[1, S, n+1]$

Rules:

$$\frac{R(A \rightarrow w_i)}{[i, A, i+1]} \qquad \text{Unary}$$

$$\frac{R(A \rightarrow BC) \qquad [i, B, k] \qquad [k, C, j]}{[i, A, j]} \qquad \text{Binary}$$

$$\frac{\frac{VZ}{V}_{in}([i, A, j], [1, S, n+1])}{[i, B, j]^{\dagger}} \; map(A) = B \qquad \text{Summation}$$

$$\frac{R(A, w_i, [i, i+1]^{\dagger}, [1, S, n+1])}{[i, i+1]^{*}} \qquad \text{Unary Labels}$$

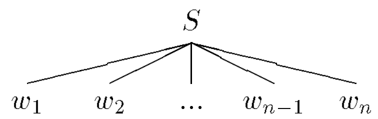
$$\frac{R(A, B, C, [i, j]^{\dagger}) \qquad [i, k]^{*} \qquad [k, j]^{*}}{[i, j]^{*}} \qquad \text{Binary Labels}$$

Figure 3.13: General Recall Description

rate and Labelled Combined rate. Next, we show an algorithm for maximizing these two rates. Finally, we give results using this new algorithm.

3.8.1 N-Ary Branching Evaluation Metrics

If we wish to model n-ary branching trees, rather than just binary branching ones, we could continue to use the same metrics as before. For instance, we could use the Consistent Brackets Recall rate or Consistent Brackets Tree rate. However, both of these have a problem, that simply by returning trees like



i.e., trees that have only one node and all terminals as children of that node, a 100% score can be achieved. The problem here is that there is no reward for returning correct constituents, only a penalty for returning incorrect ones. The Labelled Recall rate has the opposite problem: there is no penalty for including too many nodes, e.g. returning a binary branching tree.

What is needed is some criterion that combines both the penalty and the reward. The traditional solution has been to use a pair of criteria, the Bracketed Recall rate and the Bracketed Precision rate, or more common recently, the Labelled Recall rate and the Labelled Precision rate. The Bracketed Recall rate gives the reward, yielding higher scores for answers with more correct constituents, while the Bracketed Precision rate gives the penalty, giving lower scores when there are more incorrect constituents. These Precision Rates are defined as follows, and the Recall rates are included for comparison:

- *Labelled Precision Rate* $= L/N_G$.
- *Bracketed Precision Rate* $= B/N_G$.
- *Labelled Recall Rate* $= L/N_C$.
- *Bracketed Recall Rate* $= B/N_C$.

Ideally, we would now develop an algorithm that maximizes some weighted sum of a Recall rate and a Precision rate. However, it is relatively time-consuming to maximize Precision, because the denominator is N_G, the number of nodes in the guessed tree. Thus, the relative penalty for making a mistake differs, depending on how many constituents total are returned, slowing dynamic programming algorithms. On the other hand, we can create metrics closely related to the Precision metrics, which we will call Mistakes and define as follows:

- *Labelled Mistakes Rate* $= (N_G - L)/N_C$.
- *Bracketed Mistakes Rate* $= (N_G - B)/N_C$.

N_G is the number of guessed constituents, and L is the number of correctly labelled constituents, so $N_G - L$ is the number of constituents that are not correct, i.e. mistakes. We normalize by dividing through by N_C, the number of constituents in the guessed parse. While this factor is less intuitive than N_G, it will remain constant across parse trees, making the cost of each mistake independent of the size of the guessed tree.

Now, we can define combinations of Mistakes and Recall, using a weighting factor λ:

- *Labelled Combined Rate* $= L/N_C - \lambda(N_G - L)/N_C$
- *Bracketed Combined Rate* $= B/N_C - \lambda(N_G - B)/N_C$

The positive term, L/N_C, provides a reward for correct constituents, and the negative term $\lambda(N_G - L)/N_C$ provides a penalty for incorrect constituents.

3.8.2 Combined Rate Maximization

We can now state the Labelled Combined Rate Maximization problem, as

$$\arg\max_{\overset{n}{T}_G} E(L/N_C - \lambda(N_G - L)/N_C)$$

Using manipulations similar to those used previously, we can rewrite this as

$$\arg\max_{\overset{n}{T}_G} \sum_{\langle i,X,j\rangle \in \overset{n}{T}_G} g(i,X,j) - \lambda(1 - g(i,X,j))) \tag{3.8}$$

The preceding expression says that we want to find that parse tree which maximizes our score, where our score is one point for each correct constituent and $-\lambda$ points for each incorrect one.

The preceding expression does not address an interesting question. In general, if we are inducing n-ary branching trees, it will be because we are interested in grammars with other than two expressions on the right hand sides of their productions. While grammars with zero or one elements on their right hand side are interesting, they are in general significantly harder to parse, so we will only be concerned in this chapter with grammars with at least two nonterminals on the right side of productions.[4]

Grammars with at least two elements on the right can easily be converted to grammars with exactly two elements on the right, using a trick best known from Earley's algorithm, in which new nonterminals are created by dotting. A rule such as

$$A \;\rightarrow\; B\;C\;D\;E \quad (p)$$

is converted into three rules of the form

$$\begin{array}{rcll} A & \rightarrow & B\;\; B{\bullet}CDE & (p) \\ B{\bullet}CDE & \rightarrow & C\;\; BC{\bullet}DE & (1) \\ BC{\bullet}DE & \rightarrow & D\;\; E & (1) \end{array}$$

This works fine in that it produces a grammar which produces the same strings with the same probabilities, and which is amenable to simple chart parsing. However, the parse trees produced are binary branching.

Let us define an operator, $nodot(X)$ that is true if and only if the label does not have a

[4] Furthermore, even though using the techniques of Chapter 2 we can parse grammars that have loops from unary or epsilon productions, it becomes more complicated to define precision and recall appropriately for these grammars. For instance, given our definitions, recall scores above 100% are possible by returning trees of the form

$$\begin{array}{c} S \\ | \\ S \\ | \\ S \\ | \\ S \end{array}$$

because this could lead to L or B exceeding N_C, since the repeated S constituents were all counted as correct. In fact, when we examined scoring code used by Collins (1996), we found this problem; while Collins (personal communication) says that his parser could not return trees of this form, it illustrates the problems that begin to surface in scoring trees with unary branches.

dot in it, and then modify Expression 3.8 to be

$$\arg\max_{\overset{n}{T}_G} \sum_{\langle i,X,j\rangle \in \overset{n}{T}_G | nodot(X)} g(i,X,j) - \lambda(1 - g(i,X,j)) \tag{3.9}$$

The modified expression does not count the value of any dotted nodes. There is one more complication: for any given constituent, even those without a dot, it may contribute a negative score, even though its children contribute positive scores. We thus want to be able to omit any constituent which contributes negatively, while keeping the children. The final expression we maximize, which returns a binary branching tree, is

$$\arg\max_{T_G} \sum_{\langle i,X,j\rangle \in T_G | nodot(X)} \max\left(0, g(i,X,j) - \lambda(1 - g(i,X,j))\right) \tag{3.10}$$

We take the binary branching tree which maximizes this expression, and remove all dotted nodes, and all nodes which contribute a negative score, while leaving their children in the tree. The resulting tree is the optimal n-ary branching tree. Because expression 3.10 returns a binary branching tree, we can use our previous CKY-style parsing algorithms to efficiently perform the maximization, and remove the dummy and negative nodes as a post-processing step.

A similar maximization can be used for the bracketed case. We first define $g(i,j)$ as

$$g(i,j) = \sum_{X | nodot(X)} g(i,X,j)$$

Now we can give the expression for Bracketed Combined maximization:

$$\arg\max_{T_G} \sum_{\langle i,X,j\rangle \in T_G} \max\left(0, g(i,j) - \lambda(1 - g(i,j))\right)$$

Using these expressions, we could modify the Labelled Recall algorithm to maximize either the Labelled Combined rate or the Bracketed Combined rate, or more generally, we could modify the General Recall algorithm. If we substitute

$$bestG := \max_X \sum_{Y | X = map(Y)} g\langle i,Y,j\rangle;$$

$$maxG := \max\left(0, bestG - \lambda \times (1 - bestG)\right);$$

into the Labelled Recall algorithm of Figure 3.3, we get the general algorithm for n-ary branching parse trees, which we call the General Combined algorithm.

In Figure 3.14 we give an item-based description for the Labelled Combined algorithm, very similar to the Labelled Recall description. This algorithm uses the function *combined* to give the score of constituents:

$$combined(i, A, j) = \begin{cases} \max\left(0, \begin{array}{l} \frac{VZ}{V}{}_{in}([i, A, j], [1, S, n+1]) \\ -\lambda(1 - \frac{VZ}{V}{}_{in}([i, A, j], [1, S, n+1])) \end{array}\right) & \textit{if nodot}(X) \\ 0 & \textit{otherwise} \end{cases}$$

Rather than just using the unary labels rules, the description also contains unary omit and binary omit rules, which are triggered for items whose expected contribution is 0. These items then use rules with a $\bullet$ on the left hand side in the derivation, which can be used to reconstruct an appropriate n-ary branching tree.

3.8.3 N-Ary Branching Experiments

We performed a simple experiment using the Penn Treebank, version II, sections 2-21 for training, section 23 for test, extracting a grammar in the same way as in Section 3.5.2. We varied the precision-recall tradeoff over a range of 65 values.

Figure 3.15 gives the results. As can be seen, the Labelled Combined algorithm not only produced a smooth tradeoff between Labelled Precision and Labelled Recall, it also worked better than the Labelled Tree (Viterbi) algorithm on both measures simultaneously. The algorithm asymptotes vertically at about 70% Labelled Recall. This asymptote corresponds to such a strong weight on recall that the resulting tree is binary branching. The horizontal asymptote is at 94% Labelled Precision, corresponding to a single constituent containing the entire sentence. The reason the asymptote does not approach 100% is that we measured Labelled Precision; because of our scoring mechanism, not all top nodes in the Penn Treebank were labelled the same, leading to occasional mistakes in the top node label.[5]

[5] In the Penn Treebank version II, the top bracket is always unlabelled and unary branching. We collapsed unary branches before scoring, since this algorithm could not produce unary branches. This lead to a top node that was usually an *S* but could be something else, such as *S-INV* for questions.

Item form:

$[i, A, j]$ inside semiring

$[i, A, j]^*$ arctic semiring, or similar

Primary Goal:

$[1, S, n+1]^*$

Secondary Goal:

$[1, S, n+1]$

Rules:

$$\frac{R(A \rightarrow w_i)}{[i, A, i+1]} \qquad \text{Unary}$$

$$\frac{R(A \rightarrow BC) \qquad [i, B, k] \qquad [k, C, j]}{[i, A, j]} \qquad \text{Binary}$$

$$\frac{R(A, w_i, \mathit{combined}(i, A, i+1))}{[i, A, i+1]^*} \; \mathit{combined}(i, A, i+1) > 0 \qquad \text{Unary Labels}$$

$$\frac{R(A, B, C, \mathit{combined}(i, A, j)) \qquad [i, B, k]^* \qquad [k, C, j]^*}{[i, A, j]^*} \; \mathit{combined}(i, A, j) > 0 \qquad \text{Binary Labels}$$

$$\frac{R(\bullet, w_i, \mathit{combined}(i, A, i+1))}{[i, A, i+1]^*} \; \mathit{combined}(i, A, i+1) = 0 \qquad \text{Unary Omit}$$

$$\frac{R(\bullet, B, C, \mathit{combined}(i, A, j)) \qquad [i, B, k]^* \qquad [k, C, j]^*}{[i, A, j]^*} \; \mathit{combined}(i, A, j) = 0 \qquad \text{Binary Omit}$$

Figure 3.14: N-ary Labelled Recall Description

Algorithm	Time
Viterbi algorithm	420
Inside algorithm	302
Outside algorithm	603
Find 1 Combined tree	3
Find 65 Combined trees	10

Table 3.3: Algorithm Timings in Seconds

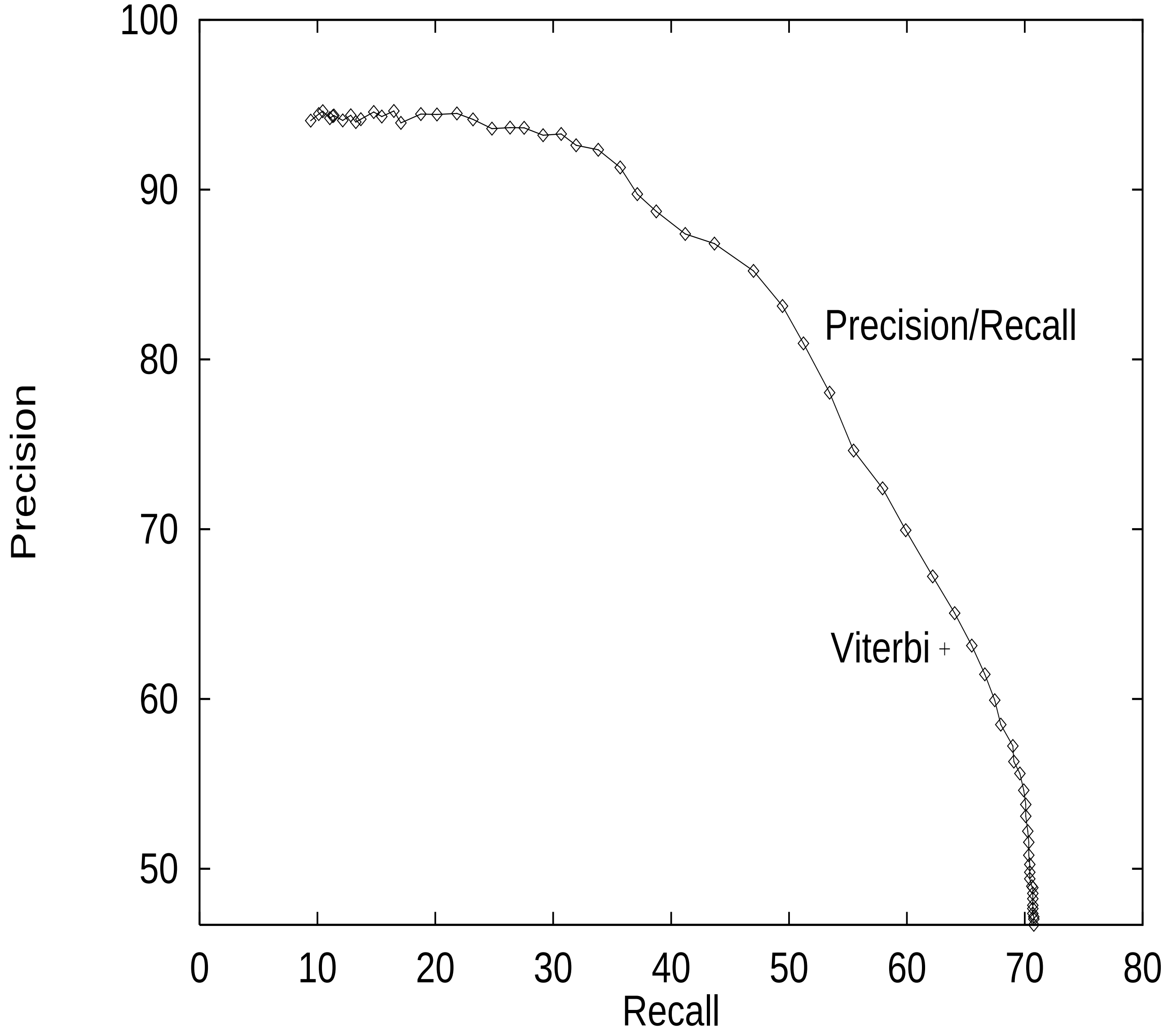


Figure 3.15: Labelled Combined Algorithm vs. Labelled Tree Algorithm

	Bracketed	Labelled
Tree	(NP-Complete)	Labelled Tree
Recall	Bracketed Recall	Labelled Recall

Table 3.4: Metrics and corresponding algorithms

Table 3.3 gives the timings of the various algorithms in seconds. The Viterbi algorithm and inside algorithm take approximately the same amount of time, while the outside algorithm takes about twice as long. Once the inside and outside probabilities are computed, the Labelled Combined trees can be computed very quickly. A portion of the work in computing combined trees, computing the g values, needs to be done only once per sentence, which is why the time to find 65 trees is not 65 times the time of finding 1 tree. It is very significant that almost all of the work to compute a combined tree needs only to be done once: this means that if we are going to compute one tree, we might as well compute the entire ROC (precision-recall) curve.

The fact that we can compute a precision-recall curve makes it much easier to compare parsing algorithms. If there are two parsing algorithms, and one gets a better score on Labelled Precision, and the other gets a better score on Labelled Recall, we cannot determine which is better. However, if for one or both algorithms, an ROC curve exists, then we can determine which is the better algorithm, unless of course the curves cross. But even in the crossing case, we have learned the useful fact that one algorithm is better for some things, and the other algorithm is better for other things.

3.9 Conclusions

Matching parsing algorithms to evaluation criteria is a powerful technique that can be used to achieve better performance than standard algorithms. In particular, the Labelled Recall algorithm has better performance than the Labelled Tree algorithm on the Consistent Brackets Recall, Labelled Recall, and Bracketed Recall rates. Similarly, the Bracketed Recall algorithm has better performance than the Labelled Tree algorithm on Consistent Brackets and Bracketed Recall rates. Thus, these algorithms improve performance not only on the measures that they were designed for, but also on related criteria. For n-ary branching trees, we have shown that the Labelled Combined algorithm can lead to improvements on both precision and recall, and can allow us to trade precision and recall off against each other, or even to quickly produce the curve showing this tradeoff.

Of course, there are limitations to the approach. For instance, the problem of maximizing the Bracketed Tree rate (equivalent to Zero Crossing Brackets rate in the case of binary branching data) is NP-Complete: not all criteria can be optimized directly.

In the next chapter, we will see that these techniques can also in some cases speed parsing. In particular, we will introduce Data-Oriented Parsing (Bod, 1992), and show that while we cannot efficiently maximize the Labelled Tree rate, we can use the General Recall algorithm to maximize the Labelled Recall rate in time $O(n^3)$, leading to significant speedups.

Appendix

3–A Proof of Crossing Brackets Theorem

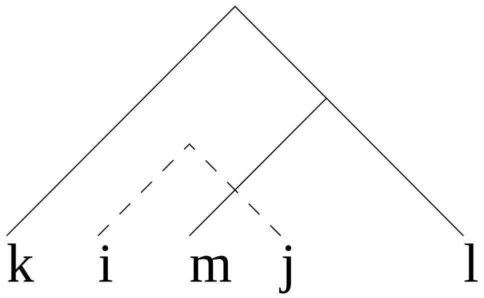


Figure 3.16: Crossing trees

In this appendix, we prove that if the correct parse tree is binary branching, then Consistent Brackets and Bracketed Match are identical. Pereira and Schabes (1992) make reference to an equivalent observation, but without proof. The proof is broken into two parts. In the first part, we show that any match does not cross. In the second part, we show that any constituent $\langle i, j \rangle$ that does not have a Bracketed Match does cross.

To show that any element which does match does not cross, we note that by a simple induction, the left descendants of any constituent can never cross the right descendants of that constituent. Now, assume that an element which did match, also crossed. Find the lowest common ancestor of the match and the crossing element. One of them must be a descendant of the left child, and the other must be a descendant of the right child. But then, by the lemma, they would not cross, and we have a contradiction, so our assumption must be wrong.

To show that any element $\langle i, j \rangle$ that does not match must have a crossing element, we find the smallest element $\langle k, l \rangle$ containing $\langle i, j \rangle$ (that is, $k \leq i$ and $l \geq j$). Now, since $\langle i, j \rangle$ doesn't match $\langle k, l \rangle$, one of these inequalities must be strict (that is, $k < i$ or $l > j$). Assume without loss of generality that $l > j$. Since we assumed that T_C is binary branching, $\langle k, l \rangle$ has two children. Let $\langle m, l \rangle$ represent the right child. This configuration is illustrated in Figure 3.16. We know that $m > i$, since otherwise $\langle m, l \rangle$ would be a constituent smaller than $\langle k, l \rangle$ containing $\langle i, j \rangle$. Similarly, we know that $m < j$, since otherwise $\langle k, m \rangle$ would be a constituent smaller than $\langle k, l \rangle$. Thus, we have $i < m < j < l$, meeting the definition for a cross.

Appendix

3–B Glossary

B Number of correctly bracketed constituents; see Section 3.2.

Bracketed Mistakes rate $(N_G - B)/N_C$. Approximation to unlabelled precision. See Section 3.8.1.

Bracketed Combined rate $B/N_C - \lambda(N_G - B)/N_C$. The weighted difference of Bracketed Recall and Bracketed Mistakes. See Section 3.8.1.

Bracketed Precision rate B/N_G. A score which penalizes incorrect guesses. See Section 3.8.1.

Bracketed Recall algorithm Algorithm maximizing Bracketed Recall rate. See Section 3.4.

Bracketed Recall rate B/N_C. Closely related to Consistent Brackets rate. See Section 3.2.

Bracketed match $\langle i, X, j\rangle$ bracketed matches $\langle i, Y, j\rangle$. See Section 3.2.

Bracketed Tree rate $\begin{cases} 1 & \mathit{if}\ B = N_C \\ 0 & \mathit{otherwise} \end{cases}$ See Section 3.2.

C Number of constituents that do not cross a correct constituent; see Section 3.2.

Consistent Brackets Recall rate C/N_G. Often called Crossing Brackets rate. When the parses are binary branching, the same as the Bracketed Recall rate.

Consistent Brackets match $\langle i, X, j\rangle$ matches if there is no $\langle k, Y, l\rangle$ crossing it.

Consistent Brackets Tree rate $\begin{cases} 1 & \mathit{if}\ C = N_G \\ 0 & \mathit{otherwise} \end{cases}$ Closely related to Bracketed Tree rate. When the parses are binary branching, the two metrics are the same. Also called the Zero Crossing Brackets rate. See Section 3.2.

Crossing Brackets rate Conventional name for Consistent Brackets rate.

E Expected value function.

$\mathit{inside}(i, X, j)$ Inside value.

Exact Match rate Conventional name for Labelled Tree rate.

$outside(i, X, j)$ Outside value.

$g(i, X, j)$ Normalized inside-outside value.

L Number of correct constituents; see Section 3.2.

Labelled Mistakes rate $(N_G - L)/N_C$. An approximation to Labelled Precision. See Section 3.8.1.

Labelled Combined rate $L/N_C - \lambda(N_G - L)/N_C$. The weighted difference of Labelled Recall and Labelled Mistakes. See Section 3.8.1.

Labelled Precision rate L/N_G. A score which penalizes incorrect guesses. See Section 3.8.1.

Labelled Recall algorithm Algorithm for maximizing Labelled Recall rate. See Section 3.3.

Labelled Recall rate L/N_C. See Section 3.2.

Labelled match $\langle i, X, j \rangle$ occurs in both correct and guessed parse trees.

Labelled Tree algorithm Algorithm for maximizing Labelled Tree rate. Also called Viterbi algorithm.

Labelled Tree rate $\begin{cases} 1 & \textit{if } L = N_C \\ 0 & \textit{otherwise} \end{cases}$ This metric is also called the Viterbi criterion or the Exact Match rate.

N_C Number of constituents in tree in treebank.

N_G Number of constituents in tree output by parser.

T_C Correct parse tree – tree in treebank.

T_G Guessed parse tree – tree output by parser.

Viterbi algorithm Well-known CKY style algorithm for maximizing Labelled Tree rate.

w_i Word i of input sentence.

Zero Crossing Brackets rate Conventional name for Consistent Brackets Tree rate.

Chapter 4

Data-Oriented Parsing

In this chapter we describe techniques for parsing the Data-Oriented Parsing (DOP) model 500 times faster than with previous parsers. This work (Goodman, 1996a) represents the first replication of the DOP model, and calls into question the source of the previously reported extraordinary performance levels. The results in this chapter rely primarily on two techniques: an efficient conversion of the DOP model to a PCFG, and the General Recall algorithm of the preceding chapter.

4.1 Introduction

The Data-Oriented Parsing (DOP) model has an interesting and controversial history. It was introduced by Remko Scha (1990) and was then studied by Rens Bod. Bod (1993c; 1992) was not able to find an efficient exact algorithm for parsing using the model; however he did discover and implement Monte Carlo approximations. He tested these algorithms on a cleaned up version of the ATIS corpus (Hemphill *et al.*, 1990), in which inconsistencies in the data had been removed by hand. Bod achieved some very exciting results, reportedly getting 96% of his test set exactly correct, a huge improvement over previous results. For instance, Bod (1993b) compares these results to Schabes (1993), in which, for short sentences, 30% of the sentences have no crossing brackets (a much easier measure than exact match). Thus, Bod achieves an extraordinary 8-fold error rate reduction.

Other researchers attempted to duplicate these results, but because of a lack of details of the parsing algorithm in his publications, were unable to confirm the results (Magerman, Lafferty, personal communication). Even Bod's thesis (Bod, 1995b) does not contain enough information to replicate his results.

Parsing using the DOP model is especially difficult. The model can be summarized as a special kind of Stochastic Tree Substitution Grammar (STSG): given a bracketed, labelled training corpus, let *every* subtree of that corpus be an elementary tree, with a probability proportional to the number of occurrences of that subtree in the training corpus. Unfortunately, the number of trees is in general exponential in the size of the training corpus trees, producing an unwieldy grammar.

In this chapter, we introduce a reduction of the DOP model to an exactly equivalent Probabilistic Context-Free Grammar (PCFG) that is linear in the number of nodes in the training data. Next, we show that the General Recall algorithm (introduced in Section 3.7), which uses the inside-outside probabilities, can be used to efficiently parse the DOP model in time $O(Tn^3)$, where T is training data size. This polynomial run time is especially significant given that Sima'an (1996a) showed that computing the most probable parse of an STSG is NP-Complete. We also give a random sampling algorithm equivalent to Bod's that runs in time $O(Gn^2)$ rather than Bod's $O(Gn^3)$ per sample, where G is the grammar size. We use the reduction and the two parsing algorithms to parse held out test data, comparing these results to a replication of Pereira and Schabes (1992) on the same data. These results are disappointing: both the Monte Carlo parser and the General Recall parser applied to the DOP model perform about the same as the Pereira and Schabes method. We present an analysis of the runtime of our algorithm and Bod's. Finally, we analyze Bod's data, showing that some of the difference between our performance and his is due to a fortuitous choice of test data.

This work was the first published replication of the full DOP model, i.e. using a parser that sums over derivations. It also contains algorithms implementing the model with significantly fewer resources than previously needed. Furthermore, for the first time, the DOP model is compared to a competing model on the same data.

4.2 Previous Research

The DOP model itself is extremely simple and can be described as follows: for every sentence in a parsed training corpus, extract every subtree. In general, the number of subtrees will

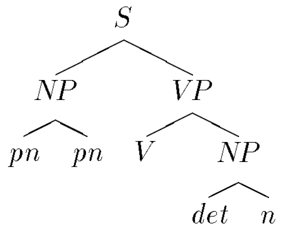


Figure 4.1: Training corpus tree for DOP example

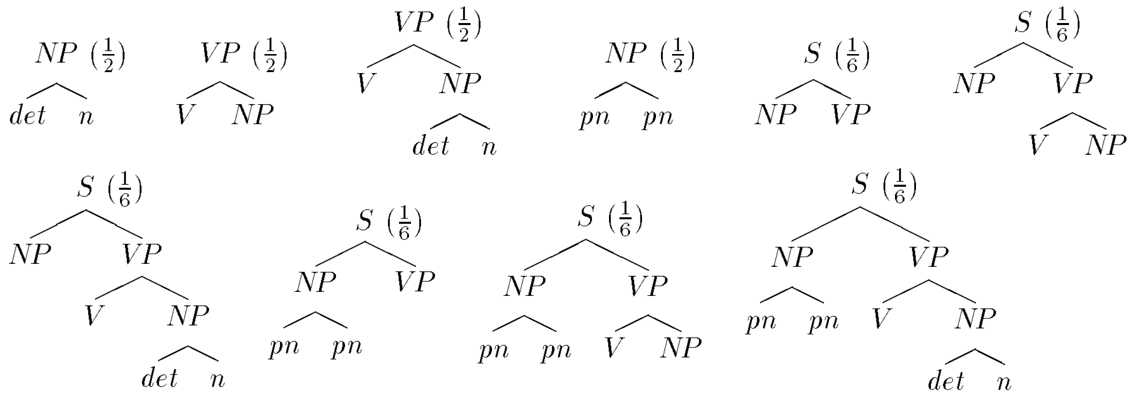
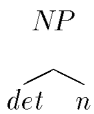


Figure 4.2: Sample STSG Produced from DOP Model

be very large, typically exponential in sentence length. Now, use these trees to form a Stochastic Tree Substitution Grammar (STSG).[1] Each tree is assigned a number of counts, one count for each time it occurred as a subtree of a tree in the training corpus. Each tree is then assigned a probability by dividing its number of counts by the total number of counts of trees with the same root nonterminal.

Given the tree of Figure 4.1, we can use the DOP model to convert it into the STSG of Figure 4.2. The numbers in parentheses represent the probabilities. To give one example, the tree



is assigned probability 0.5 because the tree occurs once in the training corpus, and there are two subtrees in the training corpus that are rooted in NP, so $\frac{1}{2} = 0.5$. The resulting STSG can be used for parsing.

In theory, the DOP model has several advantages over other models. Unlike a PCFG, the

[1]STSGs were described in Section 3.7.

use of trees allows capturing large contexts, making the model more sensitive. Since every subtree is included, even trivial ones corresponding to rules in a PCFG, novel sentences with unseen contexts may still be parsed.

Because every subtree is included, the number of subtrees is huge; therefore Bod randomly samples 5% of the subtrees, throwing away the rest. This 95% reduction in grammar size significantly speeds up parsing.

As we discussed in Section 3.7, there are three ways to parse a STSG and thus to parse DOP. The two ways Bod considered were the most probable derivation, and the most probable parse (best according to the Labelled Tree criterion). The most probable derivation and the most probable parse may differ when there are several derivations of a given parse, as previously illustrated in Figure 3.11. The third way to parse a STSG is to use the General Recall algorithm to find the best Labelled Recall parse.

Bod (1993c) shows how to approximate the most probable parse using a Monte Carlo algorithm. The algorithm randomly samples possible derivations, then finds the tree with the most sampled derivations. Bod shows that the most probable parse yields better performance than the most probable derivation on the exact match criterion.

Sima'an (1996b) implemented a version of the DOP model, which parses efficiently by limiting the number of trees used and by using an efficient most probable derivation model. His experiments differed from ours and Bod's in many ways, including his use of a different version of the ATIS corpus; the use of word strings, rather than part of speech strings; and the fact that he did not parse sentences containing unknown words, effectively throwing out the most difficult sentences. Furthermore, Sima'an limited the number of substitution sites for his trees, effectively using a subset of the DOP model. Sima'an (1996a) shows that computing the most probable parse of a STSG is NP-Complete.

4.3 Reduction of DOP to PCFG

Bod's reduction to a STSG is extremely expensive, even when throwing away 95% of the grammar. However, it is possible to find an equivalent PCFG that contains at most eight PCFG rules for each node in the training data; thus it is $O(n)$. Because this reduction is so much smaller, we do not discard any of the grammar when using it. The PCFG is equivalent in two senses: first it generates the same strings with the same probabilities;

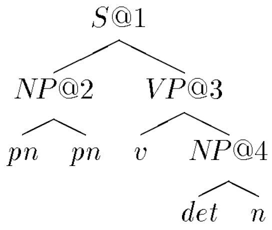


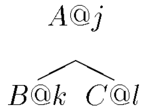
Figure 4.3: Example tree with addresses

second, using an isomorphism defined below, it generates the same trees with the same probabilities, although one must sum over several PCFG trees for each STSG tree.

To show this reduction and equivalence, we must first define some terminology. We assign every node in every tree a unique number, which we will call its address. Let $A@k$ denote the node at address k, where A is the non-terminal labeling that node. Figure 4.3 shows the example tree augmented with addresses. We will need to create one new non-terminal for each node in the training data. We will call this non-terminal A_k. We will call non-terminals of this form "interior" non-terminals, and the original non-terminals in the parse trees "exterior."

Let a_j represent the number of nontrivial subtrees headed by the node $A@j$. Let a represent the number of nontrivial subtrees headed by nodes with non-terminal A, that is $a = \sum_j a_j$.

Consider a node $A@j$ of the form:



How many nontrivial subtrees does it have? Consider first the possibilities on the left branch. There are b_k non-trivial subtrees headed by $B@k$, and there is also the trivial case where the left node is simply B. Thus there are $b_k + 1$ different possibilities on the left branch. Similarly, for the right branch there are $c_l + 1$ possibilities. We can create a subtree by choosing any possible left subtree and any possible right subtree. Thus, there are $a_j = (b_k + 1)(c_l + 1)$ possible subtrees headed by $A@j$. In our example tree of Figure 4.3, both noun phrases have exactly one subtree: $np_4 = np_2 = 1$; the verb phrase has 2 subtrees: $vp_3 = 2$; and the sentence has 6: $s_1 = 6$. These numbers correspond to the number of subtrees in Figure 4.2.

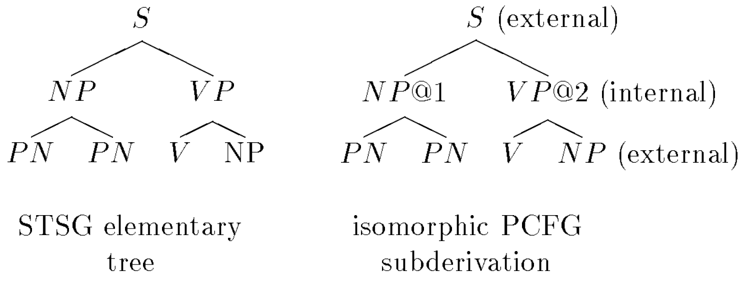
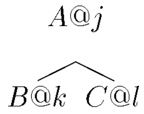


Figure 4.4: STSG elementary tree isomorphic to a PCFG subderivation

We will call a PCFG subderivation isomorphic to a STSG elementary tree if the subderivation begins with an external non-terminal, uses internal non-terminals for intermediate steps, and ends with external non-terminals. Figure 4.4 gives an example of an STSG elementary tree taken from Figure 4.2, and an isomorphic PCFG subderivation.

We will give a simple small PCFG with the following surprising property: for every subtree in the training corpus headed by A, the grammar will generate an isomorphic subderivation with probability $1/a$. In other words, rather than using the large, explicit STSG, we can use this small PCFG that generates isomorphic derivations, with identical probabilities.

The construction is as follows. For a node such as



we will generate the following eight PCFG rules, where the number in parentheses following a rule is its probability.

$$\begin{array}{llll} A_j \rightarrow BC & (1/a_j) & A \rightarrow BC & (1/a) \\ A_j \rightarrow B_kC & (b_k/a_j) & A \rightarrow B_kC & (b_k/a) \\ A_j \rightarrow BC_l & (c_l/a_j) & A \rightarrow BC_l & (c_l/a) \\ A_j \rightarrow B_kC_l & (b_kc_l/a_j) & A \rightarrow B_kC_l & (b_kc_l/a) \end{array} \tag{4.1}$$

Theorem 4.1

Subderivations headed by A with external non-terminals at the roots and leaves and internal non-terminals elsewhere have probability $1/a$. Subderivations headed by A_j with external non-terminals only at the leaves and internal non-terminals elsewhere, have probability $1/a_j$.

Proof The proof is by induction on the depth of the trees. For trees of depth 1, there are two cases:

$$\begin{array}{c} A \\ \diagup \quad \diagdown \\ B \qquad C \end{array} \qquad\qquad \begin{array}{c} A@j \\ \diagup \quad \diagdown \\ B \qquad C \end{array}$$

Trivially, these trees have the required probabilities.

Now, assume that the theorem is true for trees of depth n or less. We show that it holds for trees of depth $n+1$. There are eight cases, one for each of the eight rules. We show two of them. Let $\overset{B@k}{\vdots}$ represent a tree of at most depth n with external leaves, headed by $B@k$, and with internal intermediate non-terminals. Then, for trees such as

$$\begin{array}{c} A@j \\ \diagup \quad \diagdown \\ B@k \quad C@l \\ \vdots \qquad \vdots \end{array}$$

the probability of the tree is $\frac{1}{b_k}\frac{1}{c_l}\frac{b_k c_l}{a_j} = \frac{1}{a_j}$. Similarly, for another case, trees headed by

$$\begin{array}{c} A \\ \diagup \quad \diagdown \\ B@k \quad C \end{array}$$

the probability of the tree is $\frac{1}{b_k}\frac{b_k}{a} = \frac{1}{a}$. The other six cases follow trivially with similar reasoning. $\diamond$

We call a PCFG derivation isomorphic to a STSG derivation if for every substitution in the STSG there is a corresponding subderivation in the PCFG. Figure 4.5 contains an example of isomorphic derivations, using two subtrees in the STSG and four productions in the PCFG.

We call a PCFG tree isomorphic to a STSG tree if they are identical when internal non-terminals are changed to external non-terminals.

Theorem 4.2
This construction produces PCFG trees isomorphic to the STSG trees with equal probability.

Proof If every subtree in the training corpus occurred exactly once, the proof would be trivial. For every STSG subderivation, there would be an isomorphic PCFG subderivation, with equal probability. Thus for every STSG derivation, there would be an isomorphic

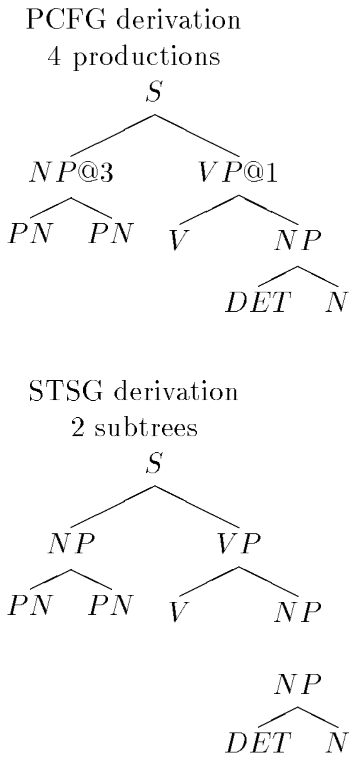


Figure 4.5: Example of Isomorphic Derivation

PCFG derivation, with equal probability. Thus every STSG tree would be produced by the PCFG with equal probability.

However, it is extremely likely that some subtrees, especially trivial ones like

$$\overbrace{NP \quad VP}^{S}$$

will occur repeatedly.

If the STSG formalism were modified slightly, so that trees could occur multiple times, then our relationship could be made one-to-one. Consider a modified form of the DOP model, in which the counts of subtrees which occurred multiple times in the training corpus were not merged: both identical trees would be added to the grammar. Each of these trees will have a lower probability than if their counts were merged. This would change the probabilities of the derivations; however the probabilities of parse trees would not change, since there would be correspondingly more derivations for each tree. Now, the desired one-to-one relationship holds: for every derivation in the new STSG there is an isomorphic derivation in the PCFG with equal probability. Thus, summing over all derivations of a tree in the STSG yields the same probability as summing over all the isomorphic derivations in the PCFG. Thus, every STSG tree would be produced by the PCFG with equal probability.

It follows trivially from this that no extra trees are produced by the PCFG. Since the total probability of the trees produced by the STSG is 1, and the PCFG produces these trees with the same probability, no probability is "left over" for any other trees. $\diamond$

4.4 Parsing Algorithms

As we discussed in Chapter 3, there are several different evaluation metrics one could use for finding the best parse. The three most interesting are the most probable derivation (which can be found using the Viterbi algorithm); the most probable parse, which can be found by random sampling; and the Labelled Recall parse, which can be found using the General Recall algorithm of Section 3.7.

Bod (1993a; 1995b) shows that the most probable derivation does not perform as well as the most probable parse for the DOP model, getting 65% exact match for the most probable derivation, versus 96% exact match for the most probable parse. This performance difference is not surprising, since each parse tree can be derived by many different derivations; the most probable parse criterion takes all possible derivations into account. Similarly, the Labelled Recall parse is also derived from the sum of many different derivations. Furthermore, although the Labelled Recall parse should not do as well on the exact match criterion, it should perform even better on the Labelled Recall rate and related criteria such as the Crossing Brackets rate. In the preceding chapter, we performed a detailed comparison between the most likely parse (the Labelled Tree parse) and the Labelled Recall parse for PCFGs; we showed that the two have very similar performance on a broad range of measures, with at most a 10% difference in error rate (i.e., a change from 10% error rate to 9% error rate.) We therefore think that it is reasonable to use the General Recall algorithm (which for STSGs can compute the Labelled Recall parse) to parse the DOP model, especially since our comparisons will be on the Crossing Brackets rate, where we expect from both theoretical and empirical considerations that an algorithm maximizing the Labelled Recall rate will outperform one maximizing the exact match (Labelled Tree) rate. Bod (1995a) complains that if we use the General Recall algorithm, we are not really parsing the DOP model, since our parser does not return a most probable parse. As we will show in the results section, our parser performs at least as well as a parser that does return the most probable parse, so this objection is immaterial.

Although the General Recall algorithm was described in detail in the previous chapter, we review it here. First, for each potential constituent, where a constituent is a non-terminal, a start position, and an end position, the algorithm uses the inside-outside values to find the probability that that constituent is in the correct parse. After that, the algorithm uses dynamic programming to put the most likely constituents together to form an output parse tree; the output parse tree maximizes the expected Labelled Recall rate.

Recall from Section 3.3.1 that the run time of the General Recall algorithm is dominated by the time to compute the inside and outside probabilities. For a grammar with r rules, this is $O(rn^3)$. Now, since there are at most eight rules for each node in the training data, if the training data is of size T, the overall run time is $O(Tn^3)$.

4.4.1 Sampling Algorithms

Bod (1995b, p. 56) gives the simple algorithm of Figure 4.6 for finding the most probable parse (mpp) (i.e. the parse that maximizes the expected Exact Match rate). Essentially, the

```
repeat until the standard error and the mpp error are smaller than a threshold
    sample a random derivation from the derivation forest
    store the parse generated by the sampled derivation
    mpp := parse with maximal frequency
    calculate the standard error of the mpp and the mpp error
```

Figure 4.6: Monte Carlo parsing algorithm

for *length* := 1 **to** n
    for *start* := 1 **to** $n - length + 1$
        for each node $X \in chart[start, start + length]$
            select at random a subderivation of X;
            eliminate the other subderivations;

Figure 4.7: Bod's $O(Gn^3)$ sampling algorithm

Function *fastsample*(i, X, j)
    if $i = j + 1$
        return *leaf*(X);
    else
        select at random a subderivation of X: i, Y, k and k, Z, j;
        return *tree*$(X, \mathit{fastsample}(i, Y, k), \mathit{fastsample}(k, Z, j))$;

Figure 4.8: Faster $O(Gn^2)$ sampling algorithm

Criteria	Min	Max	Range	Mean	StdDev
Cross Brack DOP	86.53%	96.06%	9.53%	90.15%	2.65%
Cross Brack P&S	86.99%	94.41%	7.42%	90.18%	2.59%
Cross Brack DOP−P&S	-3.79%	2.87%	6.66%	-0.03%	2.34%
Zero Cross Brack DOP	60.23%	75.86%	15.63%	66.11%	5.56%
Zero Cross Brack P&S	54.02%	78.16%	24.14%	63.94%	7.34%
Zero Cross Brack DOP−P&S	-5.68%	11.36%	17.05%	2.17%	5.57%

Table 4.1: DOP Labelled Recall versus Pereira and Schabes on Minimally Edited ATIS

Criteria	Min	Max	Range	Mean	StdDev
Cross Brack DOP	95.63%	98.62%	2.99%	97.16%	0.93%
Cross Brack P&S	94.08%	97.87%	3.79%	96.11%	1.14%
Cross Brack DOP−P&S	-0.16%	3.03%	3.19%	1.05%	1.04%
Zero Cross Brack DOP	78.67%	90.67%	12.00%	86.13%	3.99%
Zero Cross Brack P&S	70.67%	88.00%	17.33%	79.20%	5.97%
Zero Cross Brack DOP−P&S	-1.33%	20.00%	21.33%	6.93%	5.65%
Exact Match DOP	58.67%	68.00%	9.33%	63.33%	3.22%

Table 4.2: DOP Labelled Recall versus Pereira and Schabes on Bod's Data

algorithm is to randomly sample parses from the derivation forest, and pick the maximal frequency parse. In practice, rather than compute standard errors, Bod simply ran the outer loop 100 times. The algorithm Bod used for sampling a random derivation is given in Figure 4.7; Bod analyzes the run time of the algorithm for computing one sample as $O(Gn^3)$, although by using tables, it can be efficiently approximated in time $O(Gn^2)$ (Bod, personal communication).

We used a different, but mathematically equivalent sampling algorithm. Rather than a bottom-up algorithm, we used a top-down algorithm, as shown in Figure 4.8. The run time for our sampling algorithm, if naively implemented (as we did), is at worst $O(Gn^2)$, although using the same trick that Bod used, it can be implemented in time $O(n)$.

4.5 Experimental Results and Discussion

We are grateful to Bod for supplying us with data edited for his experiments (Bod, 1995c; Bod, 1995b; Bod, 1993c), although it appears not to have been exactly the data he used. We have been unable to obtain the exact same data he used, and since we cannot get it, we

	Labelled Recall Parse	Most Probable Parse	Pereira and Schabes	Significant
Cross Brack	90.1	90.0	90.2	
0 Cross Brack	66.1	65.9	63.9	
Exact Match	40.0	39.2		

Table 4.3: Three way comparison on minimally edited ATIS data

	Labelled Recall Parse	Most Probable Parse	Pereira and Schabes	Significant
Cross Brack	97.2	97.1	96.1	$\surd$
0 Cross Brack	86.1	86.1	79.2	$\surd$
Exact Match	63.3	63.1		

Table 4.4: Three way comparison on ATIS data edited by Bod

use the data Bod gave us.[2]

The original ATIS data from the Penn Treebank, version 0.5, is very noisy; it is difficult to even automatically read this data, due to inconsistencies between files. Researchers are thus left with the difficult decision as to how to clean up the data. For this chapter, we conducted two sets of experiments: one using a minimally cleaned up set of data, the same as described in Section 3.5, making our results comparable to previous results; the other using the ATIS data prepared by Bod, which contained much more significant revisions.

Ten data sets were constructed by randomly splitting minimally edited ATIS sentences into a 700 sentence training set, and an 88 sentence test set, then discarding sentences of length > 30. For each of the ten sets, the Labelled Recall parse, the sampling algorithm given in Figure 4.8 (equivalent to but faster than Bod's), and the grammar induction experiment of Pereira and Schabes (1992) were run. All sentences output by the parser were made binary branching using the Continued transformation, as described in Section 4.7,

[2]The data Bod gave us contained no epsilon productions (traces), while Bod's (1995a) data apparently did contain epsilons as explicit part-of-speech tags in the data. We note that this is an unconventional way to handle epsilon productions, since real data would typically not contain traces annotated in this way, although it might be possible to train a tagger to produce them. We have asked Bod for the correct data, but we have never received it. We note that soon after pointing out to us that the data he had given us was incorrect (since it did not contain epsilons) Bod mailed another researcher, John Maxwell, data without epsilons. Strangely, the data Maxwell received is different from the data we received. In particular, the data Maxwell received is a subset of the data we received, with some lines repeated to reach the same total number of lines.

since otherwise the crossing brackets measures are meaningless (Magerman, 1994). A few sentences were not parsable; these were assigned right branching period high structure, a good heuristic (Brill, 1993). Crossing brackets, zero crossing brackets, and the paired differences between Labelled Recall and Pereira and Schabes are presented in Table 4.1. The results are disappointing. In absolute value, the results are significantly below the 96% exact match reported by Bod. In relative value, they are also disappointing: while the DOP results are slightly higher on average than the Pereira and Schabes results, the differences are small.

We also ran experiments using Bod's data, 75 sentence test sets, and no limit on sentence length. However, while Bod provided us with data, he did not provide us with a split into test and training data; as before, we used ten random splits. The DOP results, while better, are still disappointing, as shown in Table 4.2. They continue to be noticeably worse than those reported by Bod, and again comparable to the Pereira and Schabes algorithm. Even on Bod's data, the 86% DOP achieves on the zero crossing brackets criterion is not close to the 96% Bod reported on the much harder exact match criterion. It is not clear what exactly accounts for these differences. It is also noteworthy that the results are much better on Bod's data than on the minimally edited data: crossing brackets rates of 96% and 97% on Bod's data versus 90% on minimally edited data. Thus it appears that part of Bod's extraordinary performance can be explained by the fact that his data is much cleaner than the data used by other researchers.

DOP does do slightly better on most measures. We performed a statistical analysis using a t-test on the paired differences between DOP and Pereira and Schabes performance on each run. On the minimally edited ATIS data, the differences were statistically insignificant, while on Bod's data the differences were statistically significant beyond the 98'th percentile. Our technique for finding statistical significance is more strenuous than most: we assume that since all test sentences were parsed with the same training data, all results of a single run are correlated. Thus we compare paired differences of entire runs, rather than of sentences or constituents. This makes it harder to achieve statistical significance.

Notice also the minimum and maximum columns of the "DOP−P&S" lines, constructed by finding for each of the paired runs the difference between the DOP and the Pereira and Schabes algorithms. Notice that the minimum is usually negative, and the maximum is usually positive, meaning that on some tests DOP did worse than Pereira and Schabes and on some it did better. It is important to run multiple tests, especially with small test sets like these, in order to avoid misleading results.

Tables 4.3 and 4.4 show a three-way comparison between all the algorithms; the sampling algorithm and the Labelled Recall algorithm perform almost identically. In the next section, we will show that the sampling algorithm's performance probably does not scale well to longer sentences.

4.6 Timing Analysis

In this section, we examine the empirical runtime of the General Recall algorithm, and analyze the runtime of Bod's Monte Carlo algorithm. We also note that Bod's algorithm will probably be particularly inefficient on longer sentences.

It takes about 6 seconds per sentence to run our algorithm on an HP 9000/715, versus 3.5 hours to run Bod's algorithm on a Sparc 2 (Bod, 1995c). Factoring in that the HP is roughly four times faster than the Sparc, the new algorithm is about 500 times faster. Of course, some of this difference may be due to differences in implementation, so this estimate is approximate.

Furthermore, we believe Bod's analysis of his parsing algorithm is flawed. Letting G represent grammar size, and ϵ represent maximum estimation error, Bod correctly analyzes his runtime as $O(Gn^3\epsilon^{-2})$. However, Bod then neglects analysis of this ϵ^{-2} term, assuming that it is constant. Thus he concludes that his algorithm runs in polynomial time. However, for his algorithm to have some reasonable chance of finding the most probable parse, the number of times he must sample his data is, as a conservative estimate, inversely proportional to the conditional probability of that parse. For instance, if the maximum probability parse had probability 1/50, then he would need to sample at least 50 times to be reasonably sure of finding that parse.

Now, we note that the conditional probability of the most probable parse tree will in general decline exponentially with sentence length. We assume that the number of ambiguities in a sentence will increase linearly with sentence length; if a five word sentence has on average one ambiguity, then a ten word sentence will have two, etc. A linear increase in ambiguity will lead to an exponential decrease in probability of the most probable parse.

Since the probability of the most probable parse decreases exponentially in sentence length, the number of random samples needed to find this most probable parse increases exponentially in sentence length. Thus, when using the Monte Carlo algorithm, one is left with the uncomfortable choice of exponentially decreasing the probability of finding the most probable parse, or exponentially increasing the runtime.

We admit that this argument is somewhat informal. Still, the Monte Carlo algorithm has never been tested on sentences longer than those in the ATIS corpus; there is good reason to believe the algorithm will not work as well on longer sentences. We note that our algorithm has true runtime $O(Tn^3)$, as shown previously.

4.7 Analysis of Bod's Data

In the DOP model, a sentence cannot be given an exactly correct parse unless all productions in the correct parse occur in the training set. Thus, we can get an upper bound on performance by examining the test corpus and finding which parse trees could not be generated using only productions in the training corpus. As mentioned in Section 4.5, the data Bod provided us with may not have been the data he used for his experiments; furthermore, the data was not divided into test and training. Nevertheless, we analyze this data to find an upper bound on average case performance.

In our paper on DOP (Goodman, 1996a), we performed an analysis of Bod's data based on the following lines from his thesis (Bod, 1995c, p. 64):

> It may be relevant to mention that the parse *coverage* was 99%. This means that for 99% of the test strings the perceived [test corpus] parse was in the derivation forest generated by the system.

Using this 99% figure, we were able to achieve a strong bound on the likelihood of achieving Bod's results. However, Bod later informed us (personal communication) that

> The 99% coverage refers to the percentage of sentences for which a parse was found. I did not check whether the "appriopriate" [*sic*] parse was among the found parses. I just assumed that this would have been the case, but probably it wasn't.

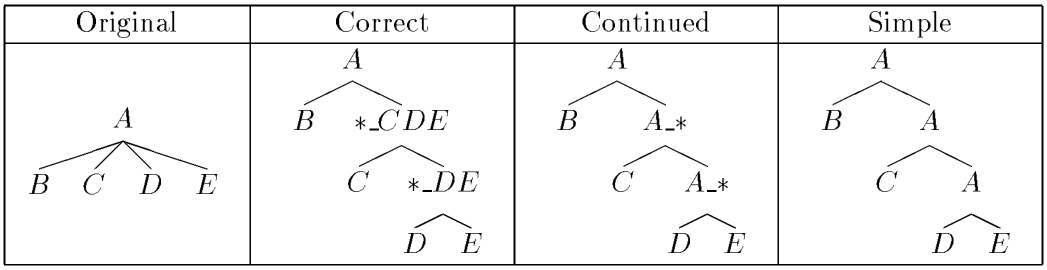


Table 4.5: Transformations from N-ary to Binary Branching Structures

	Correct		Continued		Simple	
no unary	0.78	0.0000195	0.88	0.0202670	0.90	0.0620323
unary	0.80	0.0000744	0.90	0.0577880	0.92	0.1568100

Table 4.6: Probabilities of test data with ungeneratable sentences

We can still use the fact that Bod got a 96% exact match rate to aid us, although this leads to a weaker upper bound than that in the original paper.

Bod randomly split his corpus into test and training. From the 96% exact match rate of his parser, we conclude that only three of his 75 test sentences had a correct parse that could not be generated from the training data. This small number turns out to be very surprising. An analysis of Bod's data shows that at least some of the difference in performance between his results and ours must be due to a fortuitous choice of test data, or to the data he used being even easier than the data he sent us (which was significantly easier than the original ATIS data). Bod did examine versions of DOP that smoothed, allowing productions that did not occur in the training set; however his exact match rate and his reference to coverage are both with respect to a version that does no smoothing.

In order to perform our analysis, we must determine certain details of Bod's parser that affect the probability of having most sentences correctly parsable. When using a chart parser, as Bod did, three problematic cases must be handled: ϵ productions, unary productions, and n-ary ($n > 2$) productions. The first two kinds of productions can be handled with a probabilistic chart parser, but large and difficult matrix manipulations are required (Stolcke, 1993); these manipulations would be especially difficult given the size of Bod's grammar. In the data Bod gave us there were no epsilon productions; in other data, Bod (personal communication) treated epsilons the same as other part of speech tags, a strange strategy. We also assume that Bod made the same choice we did and eliminated unary productions, given the difficulty of correctly parsing them. Bod himself does not know which technique he used for n-ary productions, since the chart parser he used was written by a third party (Bod, personal communication).

The n-ary productions can be parsed in a straightforward manner, by converting them to binary branching form; however, there are at least three different ways to convert them, as illustrated in Table 4.5. In method "Correct", the n-ary branching productions are converted in such a way that no overgeneration is introduced. A set of special non-terminals is added, one for each partial right hand side. In method "Continued", a single new non-terminal is introduced for each original non-terminal. Because these non-terminals occur in multiple contexts, some overgeneration is introduced. However, this overgeneration is constrained, so that elements that tend to occur only at the beginning, middle, or end of the right hand side of a production cannot occur somewhere else. If the "Simple" method is used, then no new non-terminals are introduced; using this method, it is not possible to recover the n-ary branching structure from the resulting parse tree, and significant overgeneration occurs.

Table 4.6 shows the undergeneration probabilities for each of these possible techniques for handling unary productions and n-ary productions.[3] The first column gives the probability that a sentence contains a production found only in that sentence, and the second column contains the probability that a random set of 75 test sentences would contain at most three such sentences:[4]

$$\frac{p^{72} \times (1-p)^3 \times 75!}{72!3!} + \frac{p^{73} \times (1-p)^2 \times 75!}{73!2!} + \frac{p^{74} \times (1-p)^1 \times 75!}{74!1!} + \frac{p^{75} \times (1-p)^0 \times 75!}{75!0!}$$

The table is arranged from least generous to most generous: in the upper left hand corner is a technique Bod might reasonably have used; in that case, the probability of getting the test set he described is less than one in 50,000. In the lower right corner we give Bod the absolute maximum benefit of the doubt: we assume he used a parser capable

[3] A perl script for analyzing Bod's data is available by anonymous FTP from ftp://ftp.das.harvard.edu/pub/goodman/analyze.perl

[4] Actually, this is a slight overestimate for a few reasons, including the fact that the 75 sentences are drawn without replacement. Also, consider a sentence with a production that occurs only in one other sentence in the corpus; there is some probability that both sentences will end up in the test data, causing both to be ungeneratable.

of parsing unary branching productions, that he used a very overgenerating grammar, and that he used a loose definition of "Exact Match." Even in this case, there is only about a 15% chance of getting the test set Bod describes.

4.8 Conclusion

We have given efficient techniques for parsing the DOP model. These results are significant since the DOP model has perhaps the best reported parsing accuracy; previously the full DOP model had not been replicated due to the difficulty and computational complexity of the existing algorithms. We have also shown that previous results were partially due to heavy cleaning of the data, which reduced the difficulty of the task, and partially due to an unlikely choice of test data – or to data even easier than that which Bod gave us.

Of course, this research raises as many questions as it answers. Were previous results due only to the choice of test data, or are differences in implementation partly responsible? In that case, there is significant future work required to understand which differences account for Bod's exceptional performance. This will be complicated by the fact that sufficient details of Bod's implementation are not available. However, based on the fact that further extraordinary DOP results have not been reported since the conference version of this work was published, it appears that problems in our implementation are not the source of the discrepancy.

This research also shows the importance of testing on more than one small test set, as well as the importance of not making cross-corpus comparisons; if a new corpus is required, then previous algorithms should be duplicated for comparison.

The speedups we achieved were critical to the success of this chapter. Running even one experiment without the 500 times speedup we achieved would have been difficult, never mind running the ten experiments that allowed us to compute accurate average performance and to compute statistical significance. These speedups were made possible by the combination of our efficient equivalent grammar and our use of the General Recall algorithm, which depends on the inside-outside probabilities.

Chapter 5

Thresholding

In this chapter, we show how to efficiently threshold Probabilistic Context-Free Grammars and Probabilistic Feature Grammars, using three new thresholding algorithms: beam thresholding with the prior; global thresholding; and multiple pass thresholding (Goodman, 1997). Each of these algorithms approximates the inside-outside probabilities. We also give an algorithm that uses the inside probabilities to efficiently optimize the settings of all of the parameters simultaneously.

5.1 Introduction

In this chapter, we examine thresholding techniques for statistical parsers. While there exist theoretically efficient ($O(n^3G)$) algorithms for parsing Probabilistic Context-Free Grammars (PCFGs), n^3G can still be fairly large in practice. Sentence lengths of 30 words ($n^3 = 27,000$) are common, and large grammars are increasingly frequent. For instance, Charniak (1996) used a 10,000 rule grammar built by simply reading rules off a tree bank. The grammar size of more recent, lexicalized grammars, such as Probabilistic Feature Grammars (PFGs), described in the next chapter, is effectively much larger. The product of n^3 and G can quickly become very large; thus, practical parsing algorithms usually make use of pruning techniques, such as beam thresholding, for increased speed.

We introduce two novel thresholding techniques, global thresholding and multiple-pass parsing, and one significant variation on traditional beam thresholding; each of these three techniques uses a different approximation to the inside-outside probabilities to improve

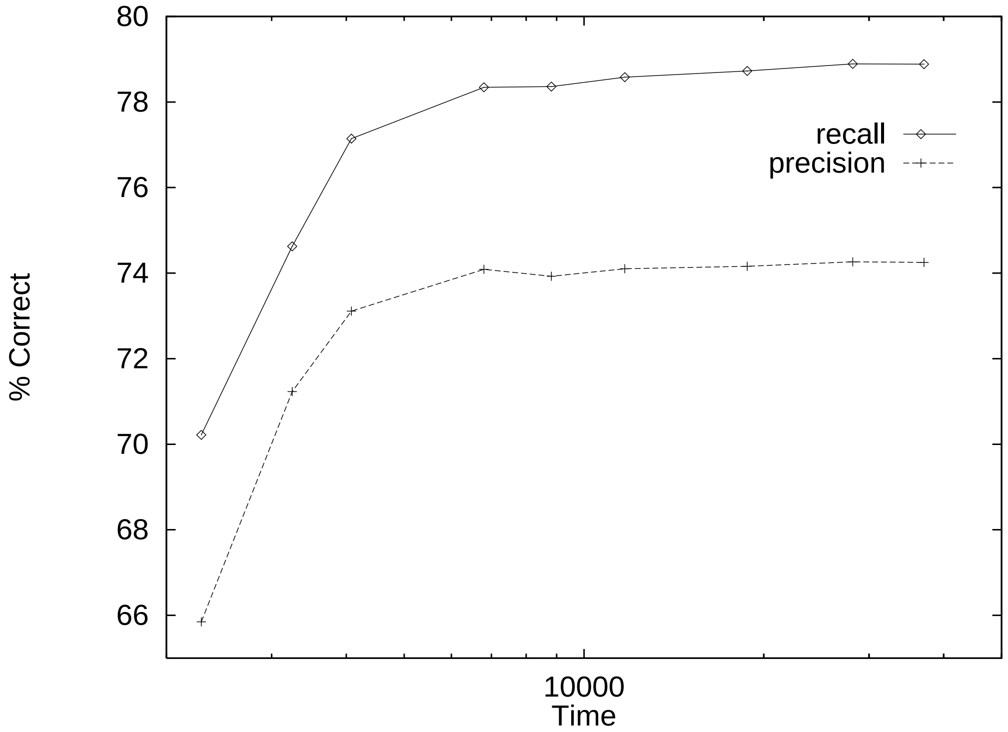


Figure 5.1: Precision and Recall versus Time in Beam Thresholding

thresholding. We examine the value of these techniques when used separately, and when combined. In order to examine the combined techniques, we also introduce an algorithm for optimizing the settings of multiple thresholds, using the inside probabilities. When all three thresholding methods are used together, they yield very significant speedups over traditional beam thresholding alone, while achieving the same level of performance.

We apply our techniques to CKY chart parsing, one of the most commonly used parsing methods in natural language processing, as described in Section 1.2.1. Recall that in a CKY chart parser, a two-dimensional matrix of cells, the chart, is filled in. Each cell in the chart corresponds to a span of the sentence, and each cell of the chart contains the nonterminals that could generate that span. The parser fills in a cell in the chart by examining the nonterminals in lower, shorter cells, and combining these nonterminals according to the rules of the grammar. The more nonterminals there are in the shorter cells, the more combinations of nonterminals the parser must consider.

In some grammars, such as PCFGs and PFGs, probabilities are associated with the grammar rules. This introduces problems, since in many grammars, almost any combination of nonterminals is possible, perhaps with some low probability. The large number of possibilities can greatly slow parsing. On the other hand, the probabilities also introduce new opportunities. For instance, if in a particular cell in the chart there is some nonterminal that generates the span with high probability, and another that generates that span with low probability, then we can remove the less likely nonterminal from the cell. The less likely nonterminal will probably not be part of either the correct parse or the tree returned by the parser, so removing it will do little harm. This technique is called *beam thresholding*.

If we use a loose beam threshold, removing only those nonterminals that are much less probable than the best nonterminal in a cell, our parser will run only slightly faster than with no thresholding, while performance measures such as precision and recall will remain virtually unchanged. On the other hand, if we use a tight threshold, removing nonterminals that are almost as probable as the best nonterminal in a cell, then we can get a considerable speedup, but at a considerable cost. Figure 5.1 shows the tradeoff between accuracy and time, using a beam threshold that ranged from .2 to .0002.

When we beam threshold, we remove less likely nonterminals from the chart. There are many ways to measure the likelihood of a nonterminal. The ideal measure would be the normalized inside-outside probability, which would give the probability that the nonterminal was correct, given the whole sentence. However, we cannot compute the outside probability of a nonterminal until we are finished computing all of the inside probabilities, so this technique cannot be used in practice.

We can, though, approximate the inside-outside probability, in several different ways. In this chapter, we will consider three different kinds of thresholding, using three different approximations to the inside-outside probability. In traditional beam search, only the inside probability is used, the probability of the nonterminal generating the terminals of the cell's span. We have found that a minor variation, introduced in Section 5.2, in which we also consider the average outside probability of the nonterminal (which is proportional to its prior probability of being part of the correct parse) can lead to nearly an order of magnitude improvement.

The problem with beam search is that it only compares nonterminals to other nonterminals in the same cell. Consider the case in which a particular cell contains only bad nonterminals, all of roughly equal probability. We cannot threshold out these nodes, because even though they are all bad, none is much worse than the best. Thus, what we want is a thresholding technique that uses some global information for thresholding, rather than just using information in a single cell. The second kind of thresholding we consider is a novel technique, *global thresholding*, described in Section 5.3. Global thresholding uses an approximation to the outside probability that uses all nonterminals not covered by the constituent, allowing inside-outside probabilities of nonterminals covering different spans to be compared.

The last technique we consider, *multiple-pass parsing*, is introduced in Section 5.4. The basic idea is that we can use inside-outside probabilities from parsing with one grammar as approximations to inside-outside probabilities in another. We run two passes with two different grammars. The first grammar is fast and simple. We compute the inside-outside probabilities using this first pass grammar, and use these probabilities to avoid considering unlikely constituents in the second pass grammar. The second pass is more complicated and slower, but also more accurate. Because we have already eliminated many low probability nodes using the inside-outside probabilities from the first pass, the second pass can run much faster, and, despite the fact that we have to run two passes, the added savings in the second pass can easily outweigh the cost of the first one.

Experimental comparisons of these techniques show that they lead to considerable speedups over traditional thresholding, when used separately. We also wished to combine the thresholding techniques; this is relatively difficult, since searching for the optimal thresholding parameters in a multi-dimensional space is potentially very time consuming. Attempting to optimize performance measures such as precision and recall using gradient descent is not feasible, because these measures are too noisy. However, we found that the inside probability was monotonic enough to optimize, and designed a variant on a gradient descent search algorithm to find the optimal parameters. Using all three thresholding methods together, and the parameter search algorithm, we achieved our best results, running an estimated 30 times faster than traditional beam search, at the same performance level.

5.2 Beam Thresholding

The first, and simplest, technique we will examine is beam thresholding. While this technique is used as part of many search algorithms, beam thresholding with PCFGs is most

for each start s
    for each rule $A \to w_s$
        $chart[s, A, s+1] := P(A \to w_s)$;
for each length l, shortest to longest
    for each start s
        for each split length t
            for each B s.t. $chart[s, B, s+t] > 0$
                for each C s.t. $chart[s+t, C, s+l] > 0$
                    for each rule $A \to BC \in R$
                        $chart[s, A, s+l] := chart[s, A, s+l] +$
                            $P(A \to BC) \times chart[s, B, s+t] \times chart[s+t, C, s+l]$);
        $best := \max_A chart[s, A, s+l]$;
        for each A
            if $chart[s, A, s+l] < T_B \times best$
                $chart[s, A, s+l] := 0$;
return $chart[1, S, n+1]$

Figure 5.2: Inside Parser with Beam Thresholding

similar to beam thresholding as used in speech recognition. Beam thresholding is often used in statistical parsers, such as that of Collins (1996).

Consider a nonterminal X in a cell covering the span of terminals $w_j...w_{k-1}$. We will refer to this as *node* $\langle j, X, k\rangle$, since it corresponds to a potential node in the final parse tree. In beam thresholding, we compare nodes $\langle j, X, k\rangle$ and $\langle j, Y, k\rangle$ covering the same span. If one node is much more likely than the other, then it is unlikely that the less probable node will be part of the correct parse, and we can remove it from the chart, saving time later.

There is some ambiguity about what it means for a node $\langle j, X, k\rangle$ to be more likely than some other node. According to folk wisdom, the best way to measure the likelihood of a node $\langle j, X, k\rangle$ is to use the inside probability, $inside(j, X, k) = P(X \stackrel{*}{\Rightarrow} w_j...w_{k-1})$. Figure 5.2 shows a PCFG parser for computing inside probabilities that uses this traditional beam thresholding. This parser is just a conventional PCFG parser with two changes. The most important change is that after parsing a given span, we find the most probable nonterminal in that span. Then, given a thresholding factor $T_B \leq 1$, we find all nonterminals less probable than the best by a factor of T_B, and set their probabilities to 0. The other change from the way we have specified PCFG parsers elsewhere is that we loop over child symbols B, C with non-zero probabilities, and then find rules consistent with these children. Elsewhere, we looped over all rules; but doing that here would mean that beam thresholding would not lead to any reduction in the number of rules examined.

Recall that the outside probability of a node $\langle j, X, k\rangle$ is the probability of that node given the surrounding terminals of the sentence, i.e. $outside(j, X, k) = P(S \stackrel{*}{\Rightarrow} w_1...w_{j-1} X w_k...w_n)$. Ideally, we would multiply the inside probability by the outside probability, and normalize, computing

$$\frac{inside(j, X, k) \times outside(j, X, k)}{inside(1, S, n+1)} = P(S \stackrel{*}{\Rightarrow} w_1...w_{j-1} X w_k...w_n \stackrel{*}{\Rightarrow} w_1...w_n | S \stackrel{*}{\Rightarrow} w_1...w_n)$$

This expression would give us the overall probability that the node is part of the correct parse, which would be ideal for thresholding. However, there is no good way to quickly compute the outside probability of a node during bottom-up chart parsing (although it can be efficiently computed afterwards). One simple approximation to the outside probability of a node $\langle j, X, k\rangle$ is just the average outside probability of the nonterminal X across the language:

$$\sum_{j,k \geq j, n \geq k, w_1...w_n} outside(i, X, j) \times P(S \stackrel{*}{\Rightarrow} w_1...w_n) =$$
$$\sum_{j,k \geq j, n \geq k, w_1...w_n} P(S \stackrel{*}{\Rightarrow} w_1...w_{j-1} X w_k...w_n | S \stackrel{*}{\Rightarrow} w_1...w_n) \times P(S \stackrel{*}{\Rightarrow} w_1...w_n)$$

We will show that the average outside probability of a nonterminal X is proportional to the prior probability of X, where by prior probability we mean the probability that a random nonterminal of a random parse tree will be X. Formally, letting $C(D, X)$ denote the number of occurrences of nonterminal X in a derivation D, we can write the prior probability as

$$P(X) = \frac{\sum_{D \text{ a derivation}} P(D) \times C(D, X)}{\sum_Y \sum_{D \text{ a derivation}} P(D) \times C(D, Y)} \tag{5.1}$$

Now,

$$\sum_{j,k \geq j, n \geq k, w_1...w_n} P(S \stackrel{*}{\Rightarrow} w_1...w_{j-1} X w_k...w_n | S \stackrel{*}{\Rightarrow} w_1...w_n) \times P(S \stackrel{*}{\Rightarrow} w_1...w_n) =$$
$$\sum_{j,k \geq j, n \geq k, w_1...w_n} P(S \stackrel{*}{\Rightarrow} w_1...w_{j-1} X w_k...w_n) \times P(X \stackrel{*}{\Rightarrow} w_j...w_{j-1}) =$$
$$\sum_{D \text{ a derivation}} P(D) \times C(D, X)$$

Item form:

$[i, A, j]$	inside or Viterbi
$[i, j]$	inside or Viterbi

Goal:

$[1, S, n+1]$

Rules:

$$\frac{R(A \to w_i)}{[i, A, i+1]} \qquad \text{Unary}$$

$$\frac{R(A \to BC) \qquad [i, B, k] \qquad [k, C, j]}{[i, A, j]} \quad \begin{array}{l} R(B) \times [i, B, k] \geq T_B \times [i, k] \wedge \\ R(C) \times [k, C, j] \geq T_B \times [k, j] \end{array} \qquad \text{Binary}$$

$$\frac{R(A) \qquad [i, A, j]}{[i, j]} \qquad \text{Thresholding}$$

Figure 5.3: Beam thresholding item-based description

Thus, the average outside probability of a nonterminal X is the same as the prior probability of X, except for a factor equal to the normalization term of Expression 5.1:

$$\frac{1}{\sum_Y \sum_{D \text{ a derivation}} P(D) \times C(D, Y)}$$

Since it is easier to compute the prior probability than the average outside probability, and since all of our values will have the same normalization factor, we use the prior probability rather than the average outside probability.

Our final thresholding measure then is $P(X) \times \mathit{inside}(j, X, k)$; the algorithm of Figure 5.2 is modified to read:

$\mathit{best} := \max_A \mathit{chart}[s, A, s+l] \times P(A);$
for each A
    if $\mathit{chart}[s, A, s+l] \times P(A) < T_B \times \mathit{best}$
        $\mathit{chart}[s, A, s+l] := 0;$

We can also give an item-based description for the CKY algorithm with beam thresholding with the prior, as shown in Figure 5.3. The item-based descriptions in this chapter can be skipped for those not familiar with semiring parsing, as described in Chapter 2; the

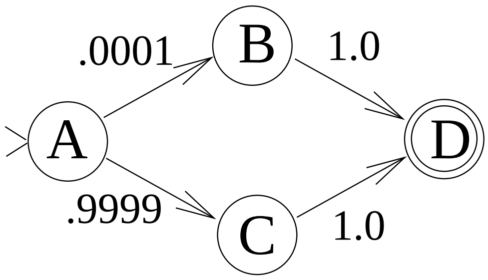


Figure 5.4: Example Hidden Markov Model

descriptions in this chapter use minor extensions to Chapter 2 described in Section 3.3.3. The thresholding algorithm is the same as the usual CKY algorithm, except that there is an added item form, $[i, j]$ containing the probability of the most probable nonterminal in the span, and an added side condition on the binary rule, which ensures that both children are sufficiently probable. For the item-based description, where we indicate rule values with the function R, we have used the notation $R(X)$ to indicate the prior probability of nonterminal X.

In Section 5.7.4, we will show experiments comparing inside-probability beam thresholding to beam thresholding using the inside probability times the prior. Using the prior can lead to a speedup of up to a factor of 10, at the same performance level.

To the best of our knowledge, using the prior probability in beam thresholding is new, although not particularly insightful on our part. Collins (personal communication) independently observed the usefulness of this modification, and Caraballo and Charniak (1996) used a related technique in a best-first parser. We think that the main reason this technique was not used sooner is that beam thresholding for PCFGs is derived from beam thresholding in speech recognition using Hidden Markov Models (HMMs), and in HMMs, this extra factor is almost never needed.

Consider the simple HMM of Figure 5.4. We have not annotated any output symbols: all states output the same symbol. After a single input symbol, we are in state B with probability 0.0001 and in state C with probability 0.9999. If we are using thresholding, we should threshold out state B. After one more symbol, we arrive in the final state, D, and we are done. Now, consider the same process backwards. HMMs can be run backwards, starting from the final state, and the last time, and moving towards the start state and beginning time. The total probability of any string will be the same computed backwards as it was forwards. However, notice what happens to the thresholding: state B and state C are both equally likely, with value 1, when we move backwards, and no thresholding occurs. When moving forwards, as long as every state in an HMM has some path to a final state – and in essentially all speech recognition applications, this is the case – every state in an HMM has probability 1 of eventually, perhaps after transitioning through many other states, reaching the final state. When processed backwards, the probability of reaching the start state can be much less than one, and in order to perform optimal thresholding, it is necessary to factor in the prior probability of reaching each state from the start.

In speech recognition, where beam thresholding was developed, processing is usually done forwards, and this extra factor is not needed.[1] In contrast, in parsing, the processing is usually bottom up, corresponding to a backwards processing from end states (terminals) to the start state (the start symbol S). It is because of this bottom-up, backwards processing that we need the extra factor that indicates the probability of getting from the start symbol to the nonterminal in question, which is proportional to the prior probability. As we noted, this can be very different for different nonterminals.

5.3 Global Thresholding

As mentioned earlier, the problem with beam thresholding is that it can only threshold out the worst nodes of a cell. It cannot threshold out an entire cell, even if there are no good nodes in it. To remedy this problem, we introduce a novel thresholding technique, global thresholding, that uses an approximation to the outside probability which takes into account all terminals not covered by the span under consideration. This allows nonterminals in different cells to be compared to each other.

The key insight of global thresholding is due to Rayner and Carter (1996). Rayner and Carter noticed that a particular node cannot be part of the correct parse if there are no nodes in adjacent cells. In fact, it must be part of a sequence of nodes stretching from the start of the string to the end. In a probabilistic framework where almost every node will have some (possibly very small) probability, we can rephrase this requirement as being that

[1] In the cases where HMM processing is done backwards, typically the forward probabilities are available, and techniques more sophisticated than beam thresholding can be used.

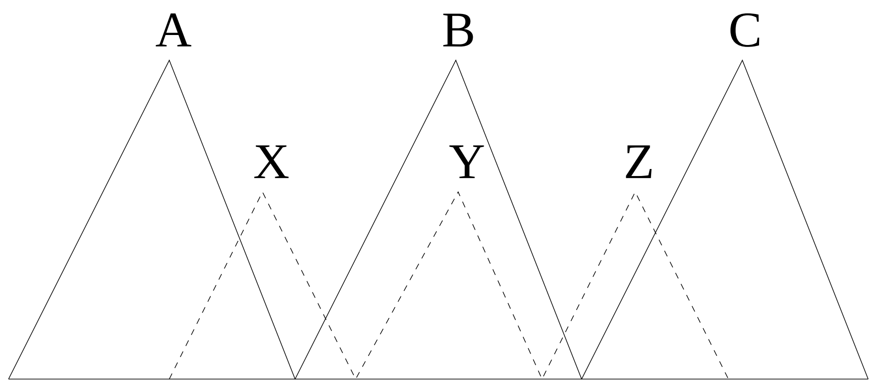


Figure 5.5: Global Thresholding Motivation

the node must be part of a reasonably probable sequence.

Figure 5.5 shows an example of this insight. Nodes A, B, and C will not be thresholded out, because each is part of a sequence from the beginning to the end of the chart. On the other hand, nodes X, Y, and Z will be thresholded out, because none is part of such a sequence.

Rayner and Carter used this insight for a hierarchical, non-recursive grammar, and only used their technique to prune after the first level of the grammar. They computed a score for each sequence as the minimum of the scores of each node in the sequence, and computed a score for each node in the sequence as the minimum of three scores: one based on statistics about nodes to the left, one based on nodes to the right, and one based on unigram statistics.

We wanted to extend the work of Rayner and Carter to general PCFGs, including those that were recursive. Our approach therefore differs from theirs in many ways. Rayner and Carter ignore the inside probabilities of nodes. While this approach may work after processing only the first level of a grammar, when the inside probabilities will be relatively homogeneous, it could cause problems after other levels, when the inside probability of a node will give important information about its usefulness. On the other hand, because long nodes will tend to have low inside probabilities, taking the minimum of all scores strongly favors sequences of short nodes. Furthermore, their algorithm requires time $O(n^3)$ to run just once. This runtime is acceptable if the algorithm is run only after the first level, but running it more often would lead to an overall run time of $O(n^4)$. Finally, we hoped to find an algorithm that was somewhat less heuristic in nature.

Our global thresholding technique thresholds out node $\langle j, X, k \rangle$ if the ratio between the most probable sequence of nodes including node $\langle j, X, k\rangle$ and the overall most probable sequence of nodes is less than some threshold, T_G. Formally, denoting sequences of nodes by L, we threshold node $\langle j, X, k\rangle$ if

$$T_G \max_L P(L) > \max_{L|\langle j,X,k\rangle \in L} P(L) \tag{5.2}$$

Now, the hard part is determining $P(L)$, the probability of a node sequence. There is no way to do this efficiently as part of the intermediate computation of a bottom-up chart parser. Thus, we will approximate $P(L)$ as follows:

$$P(L) = \prod_i P(L_i | L_1 ... L_{i-1}) \approx \prod_i P(L_i)$$

That is, we assume independence between the elements of a sequence. The probability of node $L_i = \langle j, X, k\rangle$ is just its prior probability times its inside probability, as before.

Another way to look at global thresholding is as an approximation to the un-normalized inside-outside probability. In particular,

$$\begin{aligned} & P(S \overset{*}{\Rightarrow} w_1...w_{j-1} X w_k ... w_n \overset{*}{\Rightarrow} w_1 ... w_n) \approx \\ \sum_{l,m,A_1...A_l,B_1...B_m} & P(S \overset{*}{\Rightarrow} A_1...A_l X B_1 ... B_m \overset{*}{\Rightarrow} w_1...w_{j-1} X w_k ... w_n \overset{*}{\Rightarrow} w_1 ... w_n) = \\ & \sum_{L|\langle j,X,k\rangle \in L} P(L) \end{aligned} \tag{5.3}$$

Unlike Expression 5.2, Equation 5.3 uses a summation rather than a maximum; in practice we haven't found a performance difference using either form. This approximation is not a very good one, since it will sum most derivations repeatedly. For instance, if we have

$$S \Rightarrow AX \Rightarrow A_1 A_2 X \overset{*}{\Rightarrow} w_1 ... w_n$$

then we will sum both

$$P(S \overset{*}{\Rightarrow} AX \overset{*}{\Rightarrow} w_1...w_{j-1} X \overset{*}{\Rightarrow} w_1 ... w_n)$$

and

$$P(S \overset{*}{\Rightarrow} A_1 A_2 X \overset{*}{\Rightarrow} w_1...w_{j-1} X \overset{*}{\Rightarrow} w_1 ... w_n)$$

float $f[1..n{+}1] := \{1, 0, 0, ..., 0\}$;
for $end := 2$ to $n + 1$
    $f[end] := \max_{start<end,X} f[start] \times inside(start, X, end) \times P(X)$;

float $b[1..n{+}1] := \{0, ..., 0, 0, 1\}$;
for $start := n$ downto 1
    $b[start] := \max_{end>start,X} inside(start, X, end) \times P(X) \times b[end]$;

$bestProb := f[n{+}1]$;
for each node $\langle start, X, end\rangle$
    $total := f[start] \times inside(start, X, end) \times P(X) \times b[end]$;
    $active[start, X, end] := \begin{cases} TRUE & \text{if } total > bestProb \times T_G \\ FALSE & otherwise \end{cases}$

Figure 5.6: Global Thresholding Algorithm

However, we speculate that each node $\langle j, X, k\rangle$ is affected more or less equally by this approximation, and the effects cancel out. Next, we make the same independence assumption as before, that the probability of a sequence of nonterminals is equal to the product of the prior probabilities of the nonterminals:

$$P(S \stackrel{*}{\Rightarrow} A_1...A_l X B_1...B_m) \approx P(A_1) \times \cdots \times P(A_l) \times P(X) \times P(B_1) \times \cdots \times P(B_m)$$

Using this expression we can approximate the inside-outside probabilities of any node $\langle j, X, k\rangle$, and, compare it to the best inside-outside probability of the sentence.

The most important difference between global thresholding and beam thresholding is that global thresholding is global: any node in the chart can help prune out any other node. In stark contrast, beam thresholding only compares nodes to other nodes covering the same span. Beam thresholding typically allows tighter thresholds since there are fewer approximations, but does not benefit from global information.

5.3.1 Global Thresholding Algorithm

Global thresholding is performed in a bottom-up chart parser immediately after each length is completed. It thus runs n times during the course of parsing a sentence of length n.

We use the simple dynamic programming algorithm in Figure 5.6. There are $O(n^2)$ nodes in the chart, and each node is examined exactly three times, so the run time of this algorithm is $O(n^2)$. The first section of the algorithm works forwards, computing, for each i, $f[i]$, which contains the score of the best sequence covering terminals $w_1...w_{i-1}$. Thus $f[n+1]$ contains the score of the best sequence covering the whole sentence, $\max_L P(L)$. The algorithm works analogously to the Viterbi algorithm for HMMs. The second section is analogous, but works backwards, computing $b[i]$, which contains the score of the best sequence covering terminals $w_i...w_n$.

Once we have computed the preceding arrays, computing $\max_{L|\langle j,X,k\rangle \in L} P(L)$ is straightforward. We simply want the score of the best sequence covering the nodes to the left of j, $f[j]$, times the score of the node itself, times the score of the best sequence of nodes from k to the end, which is just $b[k]$. Using this expression, we can threshold each node quickly.

Since this algorithm is run n times during the course of parsing, and requires time $O(n^2)$ each time it runs, the algorithm requires time $O(n^3)$ overall. Experiments will show that the time it saves easily outweighs the time it uses.

In Figure 5.7 we give an item-based description for a global thresholding parser. The algorithm is, again, very similar to the CKY algorithm. We have unary and binary rules, as usual. However, there is now a side condition on the binary rules: both the left and the right child must be part of a reasonably likely sequence.

There are two item types. The first, $[i, A, j]$ has the usual meaning. The second item type, $[i]_j$, can be deduced if there is a sequence of items $[1, A, m], [m, B, n], ..., [o, C, i-1]$ where each item has length at most j, where the length of an item $[i, A, k]$ is $k-i$: the number of words it covers.

The pseudocode of Figure 5.6 contains code only for thresholding, which would be run after each length was processed in the main parser. The item-based description of Figure 5.7 gives a description for the complete parser. In the procedural version, the value of $f[i]$ when it is computed after length j would equal the forward value of $[i]_j$ in the item-based description, and the value of $b[i]$ would equal the reverse value of $[i]_j$.

There are two rules for deducing items of type $[i]_j$: initialization and extension. Initialization simply states that there is a zero length sequence starting at word 1. Extension states that if there is a sequence of items of length at most k covering words 1 through $i-1$, and there is an item covering words i through $j-1$, of length at most k, then there is a sequence of items of length at most k covering words 1 through $j-1$. The trickiest rule is

Item form:

$[i, A, j]$	inside or Viterbi
$[i]_j$	inside or Viterbi

Primary Goal:

$[1, S, n+1]$

Secondary Goals:

$[n+1]_j$

Rules:

$$\frac{R(A \rightarrow w_i)}{[i, A, i+1]} \qquad \text{Unary}$$

$$\frac{R(A \rightarrow BC) \quad [i, B, k] \quad [k, C, j]}{[i, A, j]} \; \begin{array}{l} \frac{VZ}{V}{}_{in}([i, B, k], [n+1]_{j-i-1}) \geq T_G \wedge \\ \frac{VZ}{V}{}_{in}([k, C, j], [n+1]_{j-i-1}) \geq T_G \end{array} \qquad \text{Binary}$$

$$\frac{}{[1]_k} \qquad \text{Initialization}$$

$$\frac{[i]_k [i, A, j]}{[j]_k} \; k \leq j - i \qquad \text{Extension}$$

Figure 5.7: Global thresholding item-based description

the binary rule, which has a fairly complicated side condition. It checks for both $[i, B, k]$ and $[k, C, j]$ that there is a reasonably likely sequence covering the sentence, using items of length at most $j - i$ and including the item.

We should note that an earlier version of global thresholding using a standard pseudo-code specification contained a subtle bug:[2] the reverse values for items $[i]_j$ were incorrectly computed. The item-based description of Figure 5.7 makes it clearer what is going on than the procedural definition of Figure 5.6 can, and makes a bug of this form much less likely.

5.4 Multiple-Pass Parsing

In this section, we discuss a novel thresholding technique, multiple-pass parsing. We show that multiple-pass parsing techniques can yield large speedups. Multiple-pass parsing is a variation on a new technique in speech recognition, multiple-pass speech recognition (Zavaliagkos *et al.*, 1994), which we introduce first.

5.4.1 Multiple-Pass Speech Recognition

The basic idea behind multiple-pass speech recognition is that we can use the normalized forward-backward probabilities of one HMM as approximations to the normalized forward-backward probabilities of another HMM. In an idealized multiple-pass speech recognizer, we first run a simple, fast first pass HMM, computing the forward and backward probabilities. We then use these probabilities as approximations to the probabilities in the corresponding states of a slower, more accurate second pass HMM. We don't need to examine states in this second pass HMM that correspond to low inside-outside probability states in the first pass. The extra time of running two passes is more than made up for by the time saved in the second pass.

The mathematics of multiple-pass recognition is fairly simple. In the first simple pass, we record the forward probabilities, $\mathit{forward}(t, i)$, and backward probabilities, $\mathit{backward}(t, i)$, of each state i at each time t. Now, $\frac{\mathit{forward}(t,i) \times \mathit{backward}(t,i)}{\mathit{forward}(T,\mathit{final})}$ gives the overall probability of being in state i at time t given the acoustics. Our second pass will use an HMM whose states are analogous to the first pass HMM's states. If a first pass state at some time is

[2]Thanks to Michael Collins for catching it.

unlikely, then the analogous second pass state is probably also unlikely, so we can threshold it out.

There are a few complications to multiple-pass recognition. First, storing all the forward and backward probabilities can be expensive. Second, the second pass is more complicated than the first, typically meaning that it has more states. So the mapping between states in the first pass and states in the second pass may be non-trivial. To solve both these problems, only states at word transitions are saved. That is, from pass to pass, only information about where words are likely to start and end is used for thresholding.

5.4.2 Multiple-Pass Parsing

We can use an analogous algorithm for multiple-pass parsing. In this case, we will use the normalized inside-outside probabilities of one grammar as approximations to the normalized inside-outside probabilities of another grammar. We first compute the normalized inside-outside probabilities of a simple, fast first pass grammar. Next, we run a slower, more accurate second pass grammar, ignoring constituents whose corresponding first pass inside-outside probabilities are too low.

Of course, for our second pass to be more accurate, it will probably be more complicated, typically containing an increased number of nonterminals and productions. Thus, we create a mapping function from each first pass nonterminal to a set of second pass nonterminals, and threshold out those second pass nonterminals that map from low-scoring first pass nonterminals. We call this mapping function the *elaborations function*.[3]

There are many possible examples of first and second pass combinations. For instance, the first pass could use regular nonterminals, such as NP and VP and the second pass could use nonterminals augmented with head-word information. The elaborations function then appends the possible head words to the first pass nonterminals to get the second pass ones.

Even though the correspondence between forward/backward and inside/outside probabilities is very close, there are important differences between speech-recognition HMMs and natural-language processing PCFGs. In particular, we have found that in some cases it is more important to threshold productions than nonterminals. That is, rather than just

[3] In this chapter, we will assume that each second pass nonterminal is an elaboration of at most one first pass nonterminal in each cell. The grammars used here have this property. If this assumption is violated, multiple-pass parsing is still possible, but some of the algorithms need to be changed.

```
for length := 2 to n
  for start := 1 to n − length + 1
    for leftLength := 1 to length − 1
      LeftPrev := PrevChart[leftLength][start];
      for each LeftNodePrev ∈ LeftPrev
        for each non-thresholded production instance Prod from
              LeftNodePrev of size length
          for each elaboration L of Prod_Left
            for each elaboration R of Prod_Right
              for each elaboration P of Prod_Parent
                    such that P → L R
                add P to Chart[length][start];
```

Figure 5.8: Second Pass Parsing Algorithm

noticing that a particular nonterminal *VP* spanning the words "killed the rabbit" is very likely, we also note that the production $VP \rightarrow V\ NP$ (and the relevant spans) is likely.

Both the first and second pass parsing algorithms are simple variations on CKY parsing. In the first pass, we now keep track of each *production instance* associated with a node, i.e. $\langle i, X, j\rangle \rightarrow \langle i, Y, k\rangle\ \langle k, Z, j\rangle$, computing the inside and outside probabilities of each. We remove all constituents and production instances from the first pass whose normalized inside-outside probability is too small. The second pass requires more changes. Let us denote the elaborations of nonterminal X by $X_1 ... X_x$. In the second pass, for each production of the form $\langle i, X, j\rangle \rightarrow \langle i, Y, k\rangle\ \langle k, Z, j\rangle$ in the first pass that was not thresholded out by multi-pass, beam, or global thresholding, we consider every elaborated production instance, that is, all those of the form $\langle i, X_p, j\rangle \rightarrow \langle i, Y_q, k\rangle\ \langle k, Z_r, j\rangle$, for appropriate values of p, q, r. This algorithm is given in Figure 5.8, which uses a current pass matrix *Chart* to keep track of nonterminals in the current pass, and a previous pass matrix, *PrevChart* to keep track of nonterminals in the previous pass. We use one additional optimization, keeping track of the elaborations of each nonterminal in each cell in *PrevChart* that are in the corresponding cell of *Chart*.

We tried multiple-pass thresholding in two different ways. In the first technique we tried, production-instance thresholding, we remove from consideration in the second pass the elaborations of all production instances whose combined inside-outside probability falls below a threshold. In the second technique, node thresholding, we remove from considera-

Item form:

$[i, A, j]_x$

Primary Goal:

$[1, S, n{+}1]_p$

Secondary Goals:

$[1, S, n{+}1]_x \qquad x < p$

Rules:

$$\frac{R_x(A \to w_i)}{[i, A, i{+}1]_x} \quad \begin{array}{l} x = 1 \vee \\ \frac{VZ}{V}{}_{in}([i, map_x(A), i{+}1]_{x-1}, [1, S, n{+}1]_{x-1}) \geq T_x \end{array} \qquad \text{Unary}$$

$$\frac{R_x(A \to BC) \quad [i, B, k]_x \quad [k, C, j]_x}{[i, A, j]_x} \quad \begin{array}{l} x = 1 \vee \\ \frac{VZ}{V}{}_{in}([i, map_x(A), j]_{x-1}, [1, S, n{+}1]_{x-1}) \geq T_x \end{array} \qquad \text{Binary}$$

Figure 5.9: Multiple-Pass Parsing Description

tion the elaborations of all nodes whose inside-outside probability falls below a threshold. In our pilot experiments, we found that in some cases one technique works slightly better, and in some cases the other does. We therefore ran our experiments using both thresholds together.

The item-based description format of Chapter 2 allows us to describe multiple pass parsing very succinctly; Figure 5.9 is such a description. This description only does node thresholding; production thresholding could be implemented in a similar way. The description is identical to the CKY item-based description with the following changes. First, every item is annotated with a subscript indicating the pass of that item. We have also annotated the rule value function with a subscript, indicating the pass for that rule. Finally, each deduction rule contains an additional side condition of the form

$$x = 1 \vee \frac{VZ}{V}{}_{in}([i, map_x(A), j]_{x-1}, [1, S, n{+}1]_{x-1}) \geq T_x$$

indicating that the deduction rule should trigger only if we are either on the first pass, or the equivalent item from the previous pass (derived using the map_x function) was within the threshold, T_x. Notice that the first $p - 1$ passes use the inside semiring, but that the final pass can use any semiring.

One nice feature of multiple-pass parsing is that under special circumstances, it is an *admissible* search technique, meaning that we are guaranteed to find the best solution with it. In particular, if we parse using no thresholding, and our grammars have the property that for every non-zero probability parse in the second pass, there is an analogous non-zero probability parse in the first pass, then multiple-pass search is admissible. Under these circumstances, no non-zero probability parse will be thresholded out, but many zero probability parses may be removed from consideration. While we will almost always wish to parse using thresholds, it is nice to know that multiple-pass parsing can be seen as an approximation to an admissible technique, where the degree of approximation is controlled by the thresholding parameter.

5.5 Multiple Parameter Optimization

The use of any one of these techniques does not exclude the use of the others. There is no reason that we cannot use beam thresholding, global thresholding, and multiple-pass parsing all at the same time. In general, it would not make sense to use a technique such as multiple-pass parsing without other thresholding techniques; our first pass would be overwhelmingly slow without some sort of thresholding.

There are, however, some practical considerations. To optimize a single threshold, we could simply sweep our parameters over a one dimensional range, and pick the best speed versus performance tradeoff. In combining multiple techniques, we need to find optimal combinations of thresholding parameters. Rather than having to examine ten values in a single dimensional space, we might have to examine one hundred combinations in a two dimensional space. Later, we show experiments with up to six thresholds. Since we do not have time to parse with one million parameter combinations, we need a better search algorithm.

Ideally, we would simply run some form of gradient descent algorithm, optimizing a weighted sum of performance and time. There are two problems with this approach. First is that most measures of performance are too noisy. It is important when doing gradient descent that the performance measure be smooth and monotonic enough that the numerical derivative can be accurately measured. If the performance measure is noisy, then we must run a large number of sentences in order to accurately measure the derivative. However, if

Metric	decrease	same	increase
Inside	7	65	1625
Viterbi	6	1302	389
Cross Bracket	132	1332	233
Zero Cross Bracket	18	1616	63
Precision	132	1280	285
Recall	126	1331	240

Table 5.1: Monotonicity of various metrics

the number of sentences is too large, then the algorithm will be too slow. Thus, it is important to have a smooth performance measure. We will show that the inside probability is much smoother and more monotonic than conventional performance measures. Because it is so smooth, we can use a relatively small number of sentences when determining derivatives, allowing the optimization algorithm to run in a reasonable amount of time.

The second problem with using a simple gradient descent algorithm is that it does not give us much control over the solution we arrive at. Ideally, we would like to be able to pick a performance level (in terms of either entropy or precision and recall) and find the best set of thresholds for achieving that performance level as quickly as possible. If an absolute performance level is our goal, then a normal gradient descent technique will not work, since we cannot use such a technique to optimize one function of a set of variables (time as a function of thresholds) while holding another one constant (performance). We could use gradient descent to minimize a weighted sum of time and performance, but we would not know at the beginning what performance level we would have at the end. If our goal is to have the best performance we can while running in real time, or to achieve a minimum acceptable performance level with as little time as necessary, then a simple gradient descent function would not work as well as the algorithm we will give.

We will show that the inside probability is a good performance measure to optimize. We need a metric of performance that will be sensitive to changes in threshold values. In particular, our ideal metric would be strictly increasing as our thresholds loosened, so that every loosening of threshold values would produce a measurable increase in performance. The closer we get to this ideal, the fewer sentences we need to test during parameter optimization.

We tried an experiment in which we ran beam thresholding with a tight threshold, and

while not *Thresholds* $\in$ *ThresholdsSet*
  add *Thresholds* to *ThresholdsSet*;
  $(BaseE_T, BaseTime)$:= ParseAll(*Thresholds*);
  for each *Threshold* $\in$ *Thresholds*
    if $BaseE_T > TargetE_T$
      tighten *Threshold*;
      $(NewE_T, NewTime)$:= ParseAll(*Thresholds*);
      $Ratio := (BaseTime - NewTime)$ /
          $(BaseE_T - NewE_T)$;
    else
      loosen *Threshold*;
      $(NewE_T, NewTime)$:= ParseAll(*Thresholds*);
      $Ratio := (BaseE_T - NewE_T)$ /
          $(BaseTime - NewTime)$;
  change *Threshold* with best *Ratio*;

Figure 5.10: Gradient Descent Multiple Threshold Search

then a loose threshold, on all sentences of section 0 of length at most 40. For this experiment only, we discarded those sentences that could not be parsed with the specified setting of the threshold, rather than retrying with looser thresholds. For any given sentence, we would generally expect that it would do better on most measures with a loose threshold than with a tight one, but of course this will not always be the case. For instance, there is a very good chance that the number of crossing brackets or the precision and recall for any particular sentence will not change at all when we move from a tight to a loose threshold. There is even some chance, for any particular sentence that the number of crossing brackets, or the precision or the recall, will even get worse. We calculated, for each measure, how many sentences fell into each of the three categories: increased score, same score, or decreased score. Table 5.1 gives the results. As can be seen, the inside score was by far the most nearly strictly increasing metric. Furthermore, as we will show in Figure 5.14, this metric also correlates well with precision and recall, although with less noise. Therefore, we should use the inside probability as our metric of performance; however inside probabilities can become very close to zero, so instead we measure entropy, the negative logarithm of the inside probability.

We implemented a variation on a steepest descent search technique. We denote the entropy of the sentence after thresholding by E_T. Our search engine is given a target

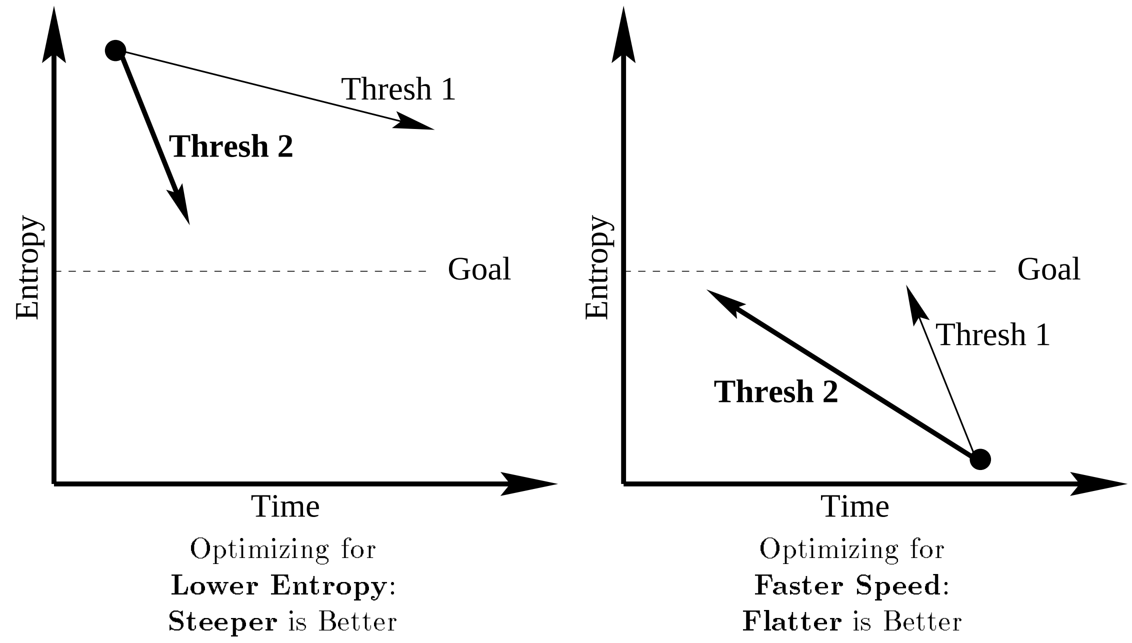


Figure 5.11: Optimizing for Lower Entropy versus Optimizing for Faster Speed

performance level E_T to search for, and then tries to find the best combination of parameters that works at approximately this level of performance. At each point, it finds the threshold to change that gives the most "bang for the buck." It then changes this parameter in the correct direction to move towards E_T (and possibly overshoot it). A simplified version of the algorithm is given in Figure 5.10.

Figure 5.11 shows graphically how the algorithm works. There are two cases. In the first case, if we are currently above the goal entropy, then we loosen our thresholds, leading to slower speed[4] and lower entropy. We then wish to get as much entropy reduction as possible per time increase; that is, we want the steepest slope possible. On the other hand, if we are trying to increase our entropy, we want as much time decrease as possible per entropy increase; that is, we want the flattest slope possible. Because of this difference, we need to compute different ratios depending on which side of the goal we are on.

There are several subtleties when thresholds are set very tightly. When we fail to parse a sentence because the thresholds are too tight, we retry the parse with lower thresholds. This can lead to conditions that are the opposite of what we expect; for instance, loosening

[4]For this algorithm (although not for most experiments), our measurement of time was the total number of productions searched, rather than cpu time; we wanted the greater accuracy of measuring productions.

thresholds may lead to faster parsing, because we don't need to parse the sentence, fail, and then retry with looser thresholds. The full algorithm contains additional checks that our thresholding change had the effect we expected (either increased time for decreased entropy or vice versa). If we get either a change in the wrong direction, or a change that makes everything worse, then we retry with the reverse change, hoping that that will have the intended effect. If we get a change that makes both time and entropy better, then we make that change regardless of the ratio.

Also, we need to do checks that the denominator when computing $Ratio$ is not too small. If it is very small, then our estimate may be unreliable, and we do not consider changing this parameter. Finally, the actual algorithm we used also contained a simple "annealing schedule", in which we slowly decreased the factor by which we changed thresholds. That is, we actually run the algorithm multiple times to termination, first changing thresholds by a factor of 16. After a loop is reached at this factor, we lower the factor to 4, then 2, then 1.414, then 1.15.

We note that this algorithm is fairly task independent. It can be used for almost any statistical parsing formalism that uses thresholds, or even for speech recognition.

5.6 Comparison to Previous Work

Beam thresholding is a common approach. While we do not know of other systems that have used exactly our techniques, our techniques are certainly similar to those of others. For instance, Collins (1996) uses a form of beam thresholding that differs from ours only in that it does not use the prior probability of nonterminals as a factor, and Caraballo and Charniak (1996) use a version with the prior, but with other factors as well.

Much of the previous related work on thresholding is in the similar area of priority functions for agenda-based parsers. These parsers try to do "best first" parsing, with some function akin to a thresholding function determining what is best. The best comparison of these functions is due to Caraballo and Charniak (1996; 1997), who tried various prioritization methods. Several of their techniques are similar to our beam thresholding technique, and one of their techniques, not yet published (Caraballo and Charniak, 1997), would probably work better.

The only technique that Caraballo and Charniak (1996) give that took into account the scores of other nodes in the priority function, the "prefix model," required $O(n^5)$ time to compute, compared to our $O(n^3)$ system. On the other hand, all nodes in the agenda parser were compared to all other nodes, so in some sense all the priority functions were global.

We note that agenda-based PCFG parsers in general require more than $O(n^3)$ run time, because, when better derivations are discovered, they may be forced to propagate improvements to productions that they have previously considered. For instance, if an agenda-based system first computes the probability for a production $S \rightarrow NP\ VP$, and then later computes some better probability for the NP, it must update the probability for the S as well. This could propagate through much of the chart. To remedy this, Caraballo *et al.* only propagated probabilities that caused a large enough change (Caraballo and Charniak, 1997). Also, the question of when an agenda-based system should stop is a little discussed issue, and difficult since there is no obvious stopping criterion. Because of these issues, we chose not to implement an agenda-based system for comparison.

As mentioned earlier, Rayner and Carter (1996) describe a system that is the inspiration for global thresholding. Because of the limitation of their system to non-recursive grammars, and the other differences discussed in Section 5.3, global thresholding represents a significant improvement.

Collins (1996) uses two thresholding techniques. The first of these is essentially beam thresholding without a prior. In the second technique, there is a constant probability threshold. Any nodes with a probability below this threshold are pruned. If the parse fails, parsing is restarted with the constant lowered. We attempted to duplicate this technique, but achieved only negligible performance improvements. Collins (personal communication) reports a 38% speedup when this technique is combined with loose beam thresholding, compared to loose beam thresholding alone. Perhaps our lack of success is due to differences between our grammars, which are fairly different formalisms. When Collins began using a formalism somewhat closer to ours, he needed to change his beam thresholding to take into account the prior, so this hypothesis is not unlikely. Hwa (personal communication) using a model similar to PCFGs, Stochastic Lexicalized Tree Insertion Grammars, also was not able to obtain a speedup using this technique.

There is previous work in the speech recognition community on automatically optimizing some parameters (Schwartz *et al.*, 1992). However, this previous work differed significantly

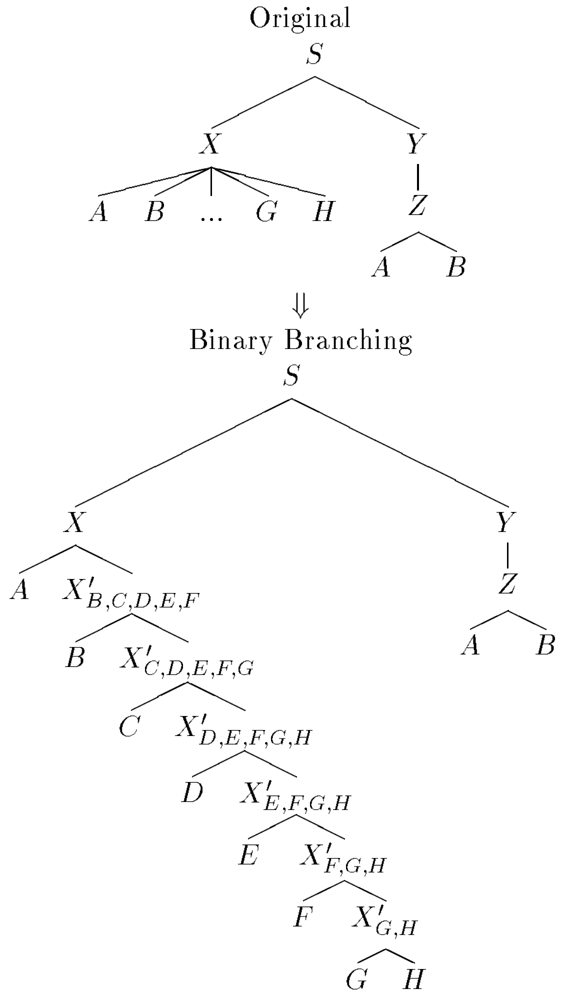


Figure 5.12: Converting to Binary Branching

from ours both in the techniques used, and in the parameters optimized. In particular, previous work focused on optimizing weights for various components, such as the language model component. In contrast, we optimize thresholding parameters. Previous techniques could not be used for or easily adapted to thresholding parameters.

5.7 Experiments

5.7.1 Data

All experiments were trained on sections 2-18 of the Penn Treebank, version II. A few were tested, where noted, on the first 200 sentences of section 00 of length at most 40 words. In one experiment, we used the first 15 of length at most 40, and in the remainder of our experiments, we used those sentences in the first 1001 of length at most 40. Our parameter optimization algorithm always used the first 31 sentences of length at most 40 words from section 19. We ran some experiments on more sentences, but there were three sentences in this larger test set that could not be parsed with beam thresholding, even with loose settings of the threshold; we therefore chose to report the smaller test set, since it is difficult to compare techniques that did not parse exactly the same sentences.

5.7.2 The Grammar

We needed several grammars for our experiments so that we could test the multiple-pass parsing algorithm. The grammar rules, and their associated probabilities, were determined by reading them off of the training section of the treebank, in a manner very similar to that used by Charniak (1996). The main grammar we chose was essentially of the following form:[5]

$$\begin{array}{rcl} X & \Rightarrow & A\,X'_{B,C,D,E,F} \\ X'_{A,B,C,D,E} & \Rightarrow & A\,X'_{B,C,D,E,F} \\ X & \Rightarrow & A \\ X & \Rightarrow & A\,B \end{array}$$

That is, productions were all unary or binary branching. There were never more than five subscripted symbols for any nonterminal, although there could be fewer than five if there were fewer than five symbols remaining on the right hand side. Thus, our grammar was a kind of 6-gram model on symbols in the grammar.

Figure 5.12 shows an example of how we converted trees to the form of our grammar. We refer to this grammar as the *6-gram grammar*. The terminals of the grammar were the part-of-speech symbols in the treebank. Any experiments that do not mention which grammar we used were run with the 6-gram grammar.

For a simple grammar, we wanted something that would be very fast. The fastest grammar we can think of we call the *terminal* grammar, because it has one nonterminal for each terminal symbol in the alphabet. The nonterminal symbol indicates the first

[5] In Chapter 6, we describe Probabilistic Feature Grammars. This grammar can be more simply described as a PFG with six features: the continuation feature, child1, child2,..., child5. No smoothing was done, and some dependencies that could have been captured, such as the dependence between the left child's child1 feature and the parent's child2 feature, were ignored, in order to capture only the dependencies described here. Unary branches were handled as described in Section 6.3.2.

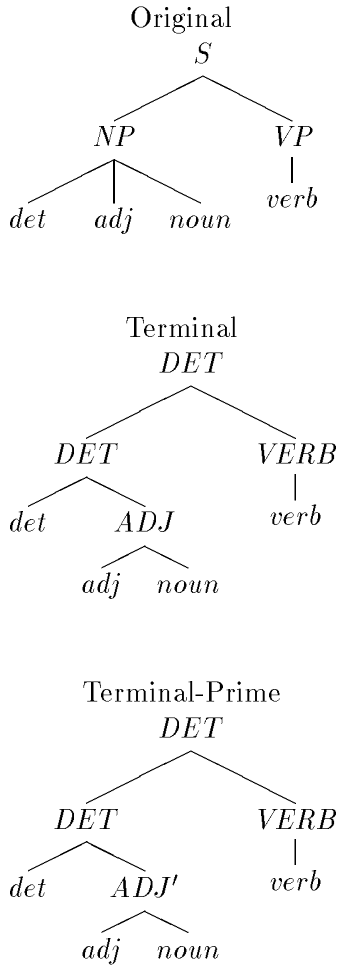


Figure 5.13: Converting to Terminal and Terminal-Prime Grammars

terminal in its span. The parses are unary and binary branching in the same way that the 6-gram grammar parses are. Figure 5.13 shows how to convert a parse tree to the terminal grammar. Since there is only one nonterminal possible for each cell of the chart, parsing is quick for this grammar. For technical and practical reasons, we actually wanted a marginally more complicated grammar, which included the "prime" symbol of the 6-gram grammar, indicating that a cell is part of the same constituent as its parent.[6] Therefore, we doubled the size of the grammar so that there would be both primed and non-primed versions of each terminal; we call this the *terminal-prime* grammar, and also show how to convert to it in Figure 5.13. This grammar is the one we actually used as the first pass in our multiple-pass parsing experiments.[7]

5.7.3 What we measured

The goal of a good thresholding algorithm is to trade off correctness for increased speed. We must thus measure both correctness and speed, and there are some subtleties to measuring each.

The traditional way of measuring correctness is with metrics such as precision and recall, which were described in Chapter 3.8.1. There are two problems with these measures. First, they are two numbers, neither useful without the other. Second, they are subject to considerable noise. In pilot experiments, we found that as we changed our thresholding values monotonically, precision and recall changed non-monotonically (see Figure 5.14). We attribute this to the fact that we must choose a single parse from our parse forest, and, as we tighten a thresholding parameter, we may threshold out either good or bad parses. Furthermore, rather than just changing precision or recall by a small amount, a single thresholded item may completely change the shape of the resulting tree. Thus, precision and recall are only smooth with very large sets of test data. However, because of the large number of experiments we wished to run, using a large set of test data was not feasible. Thus, we looked for a surrogate measure, and decided to use the total inside probability

[6] Our parser is the PFG parser of Chapter 6. Many of the decisions that parser makes depend on the value of the continuation feature, and so the PFG parser cannot run without that feature. Furthermore, keeping information for each constituent about whether it was an internal, "primed" feature or not seemed like it would provide useful information to the first pass.

[7] This grammar can be more simply described as a PFG with two features: the continuation feature, which corresponds to the prime, and a feature for the first terminal. No smoothing was done.

of all parses, which, with no thresholding, is just the probability of the sentence given the model. If we denote the total inside probability with no thresholding by I and the total inside probability with thresholding by I_T, then $\frac{I_T}{I}$ is the probability that we did not threshold out the correct parse, given the model. Thus, maximizing I_T should maximize correctness. Since probabilities can become very small, we instead minimize entropies, the negative logarithm of the probabilities. Figure 5.14 shows that with a large data set, entropy correlates well with precision and recall, and that with smaller sets, it is much smoother. Entropy is smoother because it is a function of many more variables: in one experiment, there were about 16000 constituents that contributed to precision and recall measurements, versus 151 million productions potentially contributing to entropy. Thus, we choose entropy as our measure of correctness for most experiments. When we did measure precision and recall, we used the metric as defined by Collins (1996).

The fact that entropy changes smoothly and monotonically is critical for the performance of the multiple parameter optimization algorithm. Furthermore, we may have to run quite a few iterations of that algorithm to get convergence, so the fact that entropy is smooth for relatively small numbers of sentences is a large help. Thus, the discovery that entropy (or, equivalently, the log of the inside probability) is a good surrogate for precision and recall is non-trivial. The same kinds of observations could be extended to speech recognition to optimize multiple thresholds there (the typical modern speech system has quite a few thresholds), a topic for future research.

For some sentences, with too tight thresholding, the parser will fail to find any parse at all. We dealt with these cases by restarting the parser with all thresholds lowered by a factor of 5, iterating this loosening until a parse could be found. This restarting is why for some tight thresholds, the parser may be slower than with looser thresholds: the sentence has to be parsed twice, once with tight thresholds, and once with loose ones.

Next, we needed to choose a measure of time. There are two obvious measures: amount of work done by the parser, and elapsed time. If we measure amount of work done by the parser in terms of the number of productions with non-zero probability examined by the parser, we have a fairly implementation-independent, machine-independent measure of speed. On the other hand, because we used many different thresholding algorithms, some with a fair amount of overhead, this measure seems inappropriate. Multiple-pass parsing

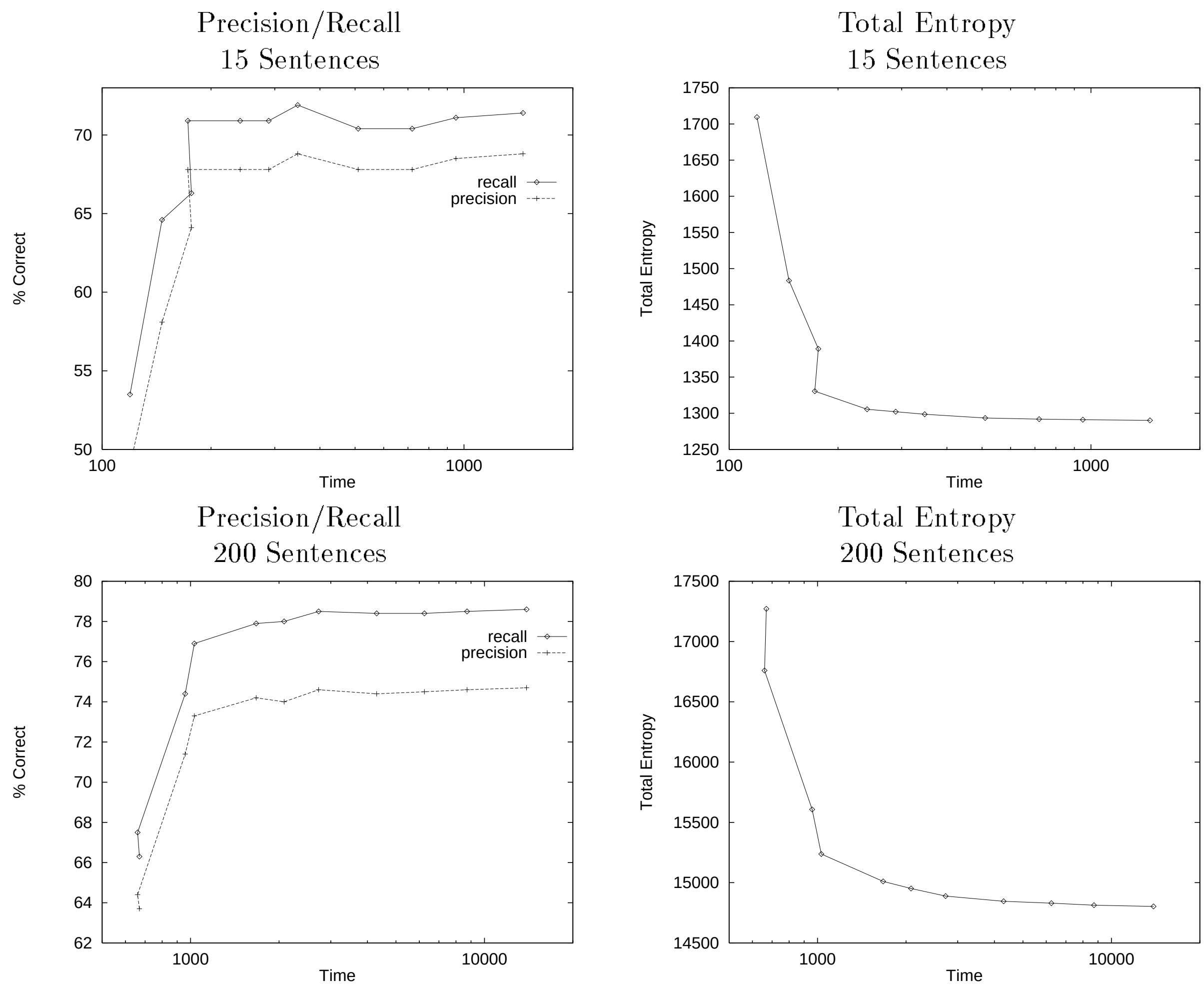


Figure 5.14: Smoothness for Precision and Recall versus Total Inside for Different Test Data Sizes

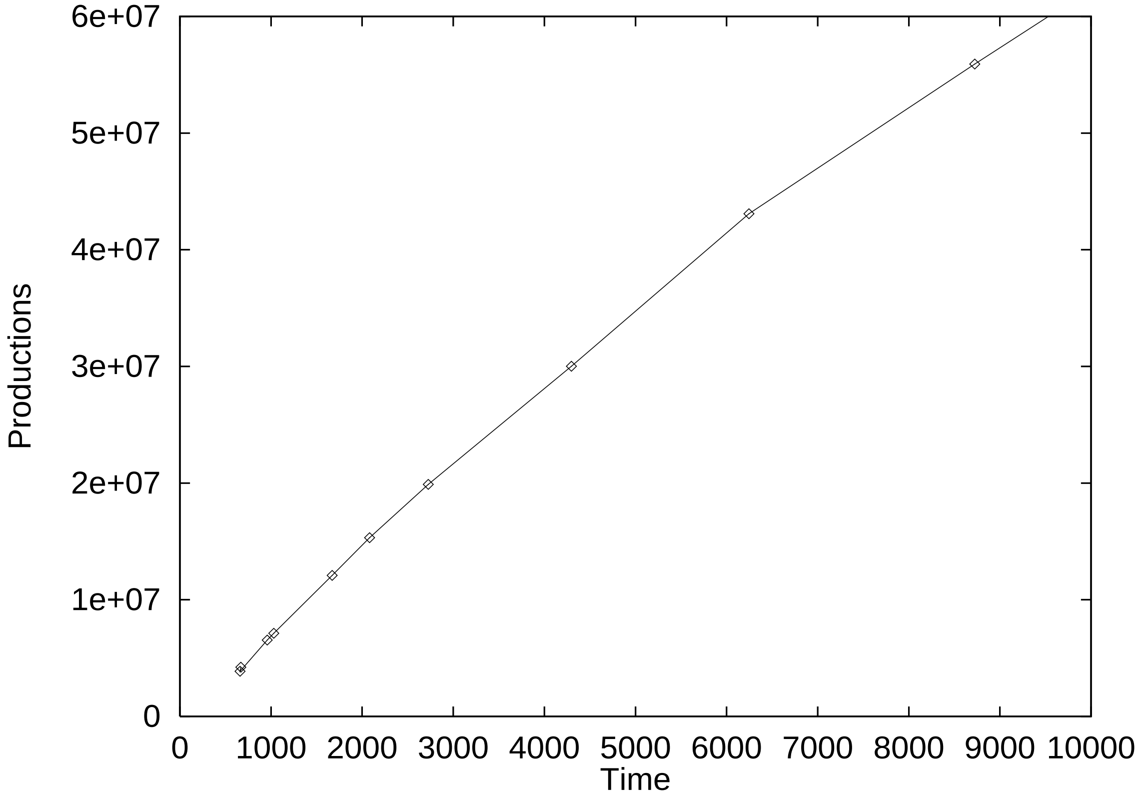
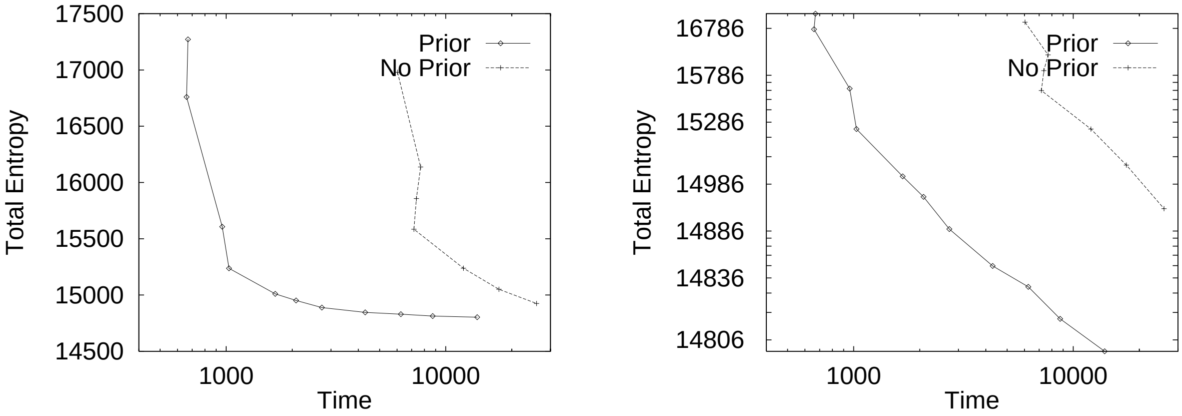


Figure 5.15: Productions versus Time



X axis: $log(time)$
Y axis: $entropy$

X axis: $log(time)$
Y axis: $log(entropy - asymptote)$

Figure 5.16: Beam Thresholding with and without the Prior Probability, Two Different Scales

requires use of the outside algorithm; global thresholding uses its own dynamic programming algorithm; and even beam thresholding has some per-node overhead. Thus, we will give most measurements in terms of elapsed time, not including loading the grammar and other $O(1)$ overhead. We did want to verify that elapsed time was a reasonable measure, so we did a beam thresholding experiment to make sure that elapsed time and number of productions examined were well correlated, using 200 sentences and an exponential sweep of the thresholding parameter. The results, shown in Figure 5.15, clearly indicate that time is a good proxy for productions examined.

5.7.4 Experiments in Beam Thresholding

Our first goal was to show, at least informally, that entropy is a good surrogate for precision and recall. We thus tried two experiments: one with a relatively large test set of 200 sentences, and one with a relatively small test set of 15 sentences. Presumably, the 200 sentence test set should be much less noisy, and fairly indicative of performance. We graphed both precision and recall, and entropy, versus time, as we swept the thresholding parameter over a sequence of values. The results are in Figure 5.14. As can be seen, entropy is significantly smoother than precision and recall for both size test corpora. In Section 5.5, we gave a more rigorous discussion of the monotonicity of entropy versus precision and recall with the same conclusion.

Our second goal was to check that the prior probability is indeed helpful. We ran two experiments, one with the prior and one without. The results, shown in Figure 5.16, indicate that the prior is a critical component. This experiment was run on 200 sentences of test data.

Notice that as the time increases, the data tends to approach an asymptote, as shown in the left hand graph of Figure 5.16. In order to make these small asymptotic changes more clear, we wished to expand the scale towards the asymptote. The right hand graph was plotted with this expanded scale, based on $log(entropy - asymptote)$, a slight variation on a normal log scale. We use this scale in all the remaining entropy graphs. A normal logarithmic scale is used for the time axis. The fact that the time axis is logarithmic is especially useful for determining how much more efficient one algorithm is than another at a given performance level. If one picks a performance level on the vertical axis, then the distance between the two curves at that level represents the ratio between their speeds. There is roughly a factor of 8 to 10 difference between using the prior and not using it at all graphed performance levels, with a slow trend towards smaller differences as the thresholds are loosened. Because of the large difference between using the prior and not using it, all other beam thresholding experiments included the prior.

5.7.5 Experiments in Global Thresholding

We tried an experiment comparing global thresholding to beam thresholding. Figure 5.17 shows the results of this experiment, and later experiments. In the best case, global thresholding works twice as well as beam thresholding, in the sense that to achieve the same level of performance requires only half as much time, although smaller improvements were more typical.

We have found that, in general, global thresholding works better on simpler grammars. In the complicated grammars of Chapter 6 there were systematic, strong correlations between nodes, which violated the independence approximation used in global thresholding. This prevented us from using global thresholding with these grammars. In the future, we may modify global thresholding to model some of these correlations.

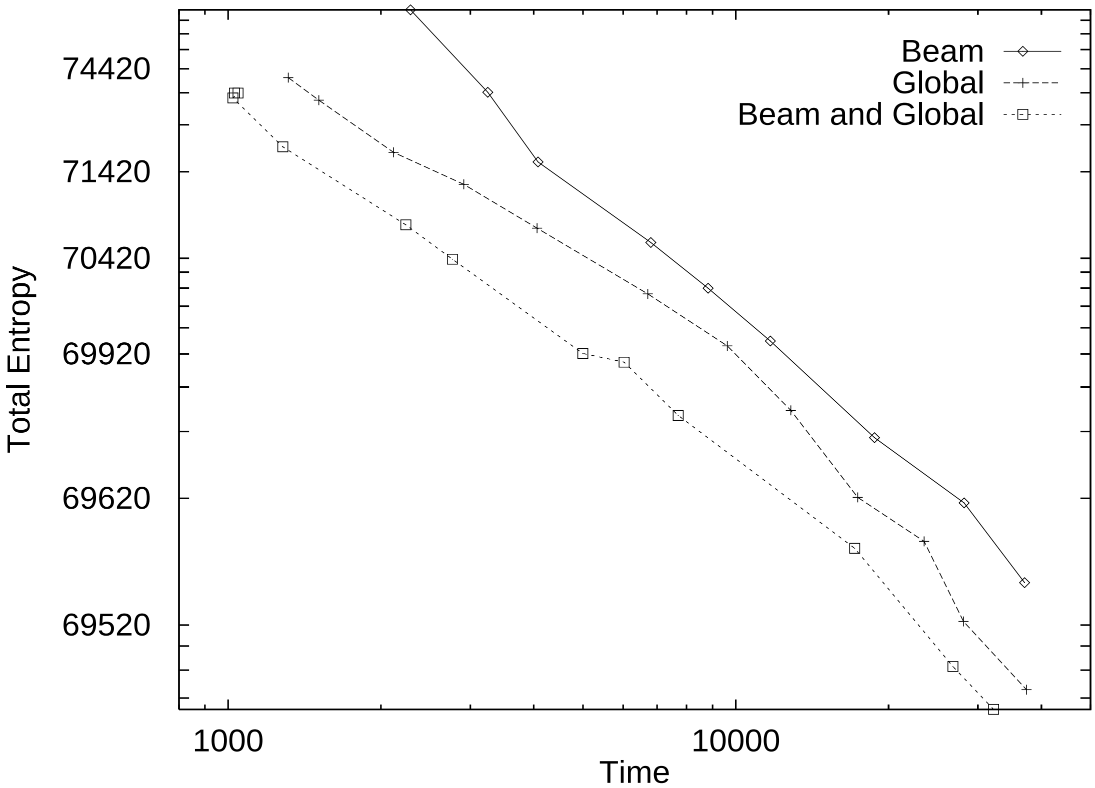


Figure 5.17: Combining Beam and Global Search

5.7.6 Experiments combining Global Thresholding and Beam Thresholding

While global thresholding works better than beam thresholding in general, each has its own strengths. Global thresholding can threshold across cells, but because of the approximations used, the thresholds must generally be looser. Beam thresholding can only threshold within a cell, but can do so fairly tightly. Combining the two offers the potential to get the advantages of both. We ran a series of experiments using the thresholding optimization algorithm of Section 5.5. Figure 5.17 gives the results. The combination of beam and global thresholding together is clearly better than either alone, in some cases running 40% faster than global thresholding alone, while achieving the same performance level. The combination generally runs twice as fast as beam thresholding alone, although up to a factor of three.

5.7.7 Experiments in Multiple-Pass Parsing

Multiple-pass parsing improves even further on our experiments combining beam and global thresholding. In addition to multiple-pass parsing, we used both beam and global thresholding for both the first and second pass in these experiments. The first pass grammar was the very simple terminal-prime grammar, and the second pass grammar was the usual 6-gram grammar.

Our goal throughout this chapter has been to maximize precision and recall, as quickly as possible. In general, however, we have measured entropy rather than precision and recall, because it correlates well with those measures, but is much smoother. While this correlation has held for all of the previous thresholding algorithms we have tried, it turns out not to hold in some cases for multiple-pass parsing. In particular, in the experiments conducted here, our first and second pass grammars were very different from each other. For a given parse to be returned, it must be in the intersection of both grammars, and reasonably likely according to both. Since the first and second pass grammars capture different information, parses that are likely according to both are especially good. The entropy of a sentence measures its likelihood according to the second pass, but ignores the fact that the returned parse must also be likely according to the first pass. Thus, in these experiments, entropy does not correlate nearly as well with precision and recall. We therefore give precision

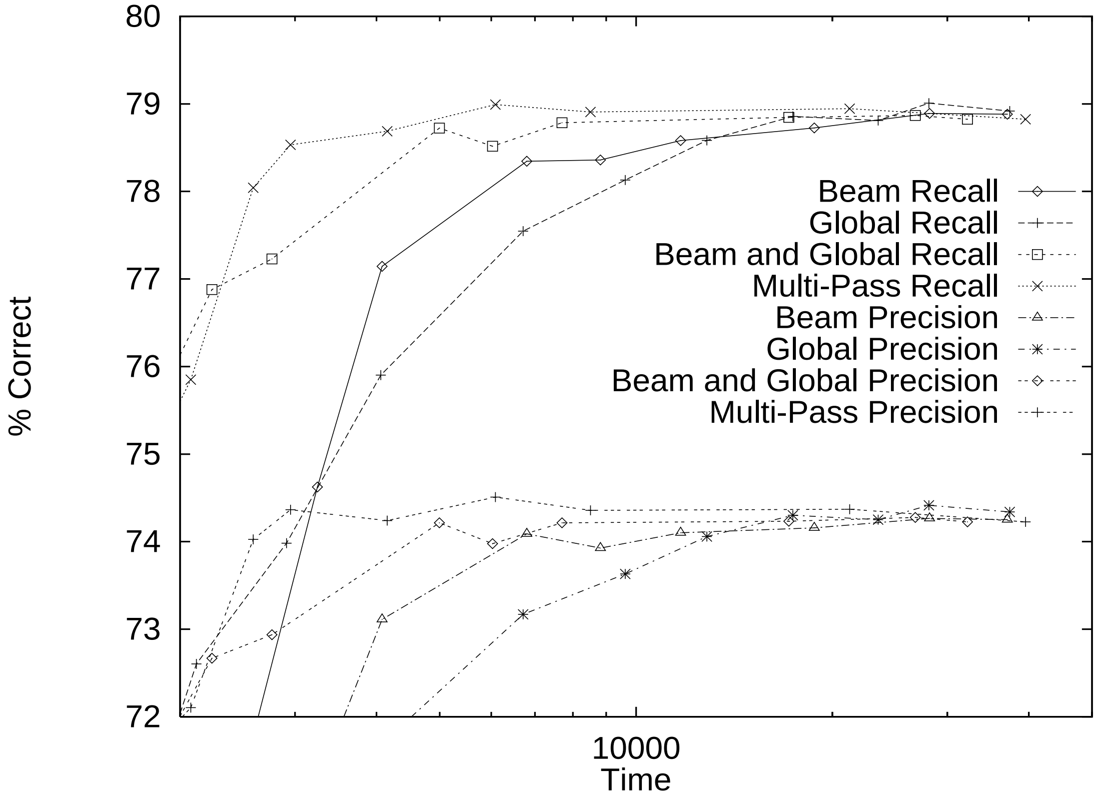


Figure 5.18: Multiple Pass Parsing vs. Beam and Global vs. Beam

and recall results in this section. We still optimized our thresholding parameters using the same 31 sentence held out corpus, and minimizing entropy versus number of productions, as before.

We should note that when we used a first pass grammar that captured a strict subset of the information in the second pass grammar, we have found that entropy is a very good measure of performance. As in our earlier experiments, it tends to be well correlated with precision and recall but less subject to noise. It is only because of the grammar mismatch that we have changed the evaluation.

Figure 5.18 shows precision and recall curves for single pass versus multiple pass experiments. As in the entropy curves, we can determine the performance ratio by looking across horizontally. For instance, the multi-pass recognizer achieves a 74% recall level using 2500 seconds, while the best single pass algorithm requires about 4500 seconds to reach that level. Due to the noise resulting from precision and recall measurements, it is hard to exactly quantify the advantage from multiple pass parsing, but it is generally about 50%.

5.8 Future Work and Conclusion

5.8.1 Future Work

In this chapter, we only considered applying multiple-pass and global thresholding techniques to parsing probabilistic context-free grammars. However, just about any probabilistic grammar formalism for which inside and outside probabilities can be computed can benefit from these techniques. For instance, Probabilistic Link Grammars (Lafferty *et al.*, 1992) could benefit from our algorithms. We have however had trouble using global thresholding with grammars that strongly violated the independence assumptions of global thresholding.

One especially interesting possibility is to apply multiple-pass techniques to formalisms that require greater than $O(n^3)$ parsing time, such as Stochastic Bracketing Transduction Grammar (SBTG) (Wu, 1996) and Stochastic Tree Adjoining Grammars (STAG) (Resnik, 1992; Schabes, 1992). SBTG is a context-free-like formalism designed for translation from one language to another; it uses a four dimensional chart to index spans in both the source and target language simultaneously. It would be interesting to try speeding up an SBTG parser by running an $O(n^3)$ first pass on the source language alone, and using this to prune parsing of the full SBTG.

The STAG formalism is a mildly context-sensitive formalism, requiring $O(n^6)$ time to parse. Most STAG productions in practical grammars are actually context-free. The traditional way to speed up STAG parsing is to use the context-free subset of an STAG to form a Stochastic Tree Insertion Grammar (STIG) (Schabes and Waters, 1994), an $O(n^3)$ formalism, but this method has problems, because the STIG undergenerates since it is missing some elementary trees. A different approach would be to use multiple-pass parsing. We could first find a context-free covering grammar for the STAG, and use this as a first pass, and then use the full STAG for the second pass.

There is also future work that could be done on further improvements to thresholding algorithms. One potential improvement that should be tried is modifying beam thresholding with the prior, or global thresholding, so that each compares only nonterminals which are similar in some way. Both of these algorithms make certain approximations. The more similar two nonterminals are, the smaller the relative error from these approximations will be. Thus, comparing only similar nonterminals will allow tighter thresholding. Obviously, these modified algorithms should be combined with the original algorithms, using the multiple parameter search technique.

5.8.2 Conclusions

The grammars described here are fairly simple, presented for purposes of explication, and to keep our experiments simple enough to replicate easily. In Chapter 6, we use significantly more complicated grammars, Probabilistic Feature Grammars (PFGs). For some PFGs, the improvements from multiple-pass parsing are even more dramatic: single pass experiments are simply too slow to run at all.

We have also found the automatic thresholding parameter optimization algorithm to be very useful. Before writing the parameter optimization algorithm, we had developed a complicated PFG grammar and the multiple-pass parsing technique and ran a series of experiments using hand optimized parameters. We thereafter ran the optimization algorithm and reran the experiments, achieving a factor of two speedup with no performance loss. While we had not spent a great deal of time hand optimizing these parameters, we are very encouraged by the optimization algorithm's practical utility.

This chapter introduces four new techniques: beam thresholding with priors, global thresholding, multiple-pass parsing, and automatic search for the parameters of combined algorithms. Beam thresholding with priors can lead to almost an order of magnitude improvement over beam thresholding without priors. Global thresholding can be up to two times as efficient as the new beam thresholding technique, although the typical improvement is closer to 50%. When global thresholding and beam thresholding are combined, they are usually two to three times as fast as beam thresholding alone. Multiple-pass parsing can lead to up to an additional 50% improvement with the grammars in this chapter. We expect the parameter optimization algorithm to be broadly useful.

Chapter 6

Probabilistic Feature Grammars

This chapter introduces Probabilistic Feature Grammars (PFGs), a relatively simple and elegant formalism that is the first state-of-the-art formalism for which the inside and outside probabilities can be computed (Goodman, 1997). Because we can compute the inside and outside probabilities, we can use the efficient thresholding algorithms of Chapter 5 when parsing PFGs.

6.1 Introduction

Recently, many researchers have worked on statistical parsing techniques which try to capture additional context beyond that of simple probabilistic context-free grammars (PCFGs), including work by Magerman (1995), Charniak (1996; 1997), Collins (1996; 1997), Black *et al.* (1992b), Eisele (1994) and Brew (1995). Each researcher has tried to capture the hierarchical nature of language, as typified by context-free grammars, and to then augment this with additional context sensitivity based on various *features* of the input. However, none of these works combines the most important benefits of all the others, and most lack a certain elegance. We have therefore tried to synthesize these works into a new formalism, *probabilistic feature grammar* (PFG). PFGs have several important properties. First, PFGs can condition on features beyond the nonterminal of each node, including features such as the head word or grammatical number of a constituent. Also, PFGs can be parsed using efficient polynomial-time dynamic programming algorithms, and learned quickly from a treebank. Finally, unlike most other formalisms, PFGs are potentially useful for language

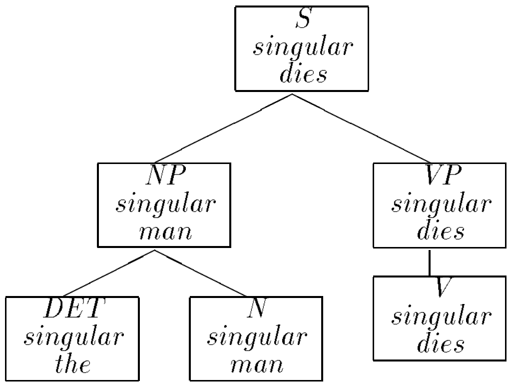


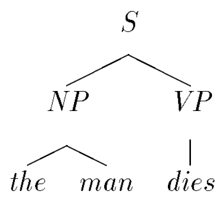
Figure 6.1: Example tree with features

modeling or as one part of an integrated statistical system (e.g. Miller *et al.*, 1996) or for use with algorithms requiring outside probabilities. Empirical results are encouraging: our best parser is comparable to those of Magerman (1995) and Collins (1996) when run on the same data. When we run using part-of-speech (POS) tags alone as input, we perform significantly better than comparable parsers.

6.2 Motivation

PFG can be regarded in several different ways: as a way to make history-based grammars (Magerman, 1995) more context-free, and thus amenable to dynamic programming; as a way to generalize the work of Black *et al.* (1992a); as a way to turn Collins' parser (Collins, 1996) into a generative probabilistic language model; or as an extension of language-modeling techniques to stochastic grammars. The resulting formalism is relatively simple and elegant. In Section 6.4, we will compare PFGs to each of the systems from which it derives, and show how it integrates their best properties.

Consider the following simple parse tree for the sentence "The man dies":



While this tree captures the simple fact that sentences are composed of noun phrases and verb phrases, it fails to capture other important restrictions. For instance, the *NP* and *VP* must have the same number, both singular, or both plural. Also, a man is far more likely to die than spaghetti, and this constrains the head words of the corresponding phrases. This additional information can be captured in a parse tree that has been augmented with features, such as the category, number, and head word of each constituent, as is traditionally done in many feature-based formalisms, such as HPSG, LFG, and others. Figure 6.1 shows a parse tree that has been augmented with these features.

While a normal PCFG has productions such as

$$S \rightarrow NP\ VP$$

we will write these augmented productions as, for instance,

$$(S, \textit{singular}, \textit{dies}) \rightarrow (NP, \textit{singular}, \textit{man})(VP, \textit{singular}, \textit{dies})$$

In a traditional probabilistic context-free grammar, we could augment the first tree with probabilities in a simple fashion. We estimate the probability of $S \rightarrow NP\ VP$ using a tree bank to determine $\frac{C(S \rightarrow NP\ VP)}{C(S)}$, the number of occurrences of $S \rightarrow NP\ VP$ divided by the number of occurrences of S. For a reasonably large treebank, probabilities estimated in this way would be reliable enough to be useful (Charniak, 1996). On the other hand, it is not unlikely that we would never have seen any counts at all of

$$\frac{C((S, \textit{singular}, \textit{dies}) \rightarrow (NP, \textit{singular}, \textit{man})(VP, \textit{singular}, \textit{dies}))}{C((S, \textit{singular}, \textit{dies}))}$$

which is the estimated probability of the corresponding production in our grammar augmented with features.

The introduction of features for number and head word has created a data sparsity problem. Fortunately, the data-sparsity problem is well known in the language-modeling community, and we can use their techniques, n-gram models and smoothing, to help us. Consider the probability of a five word sentence, $w_1 ... w_5$:

$$P(w_1 w_2 w_3 w_4 w_5) = P(w_1) \times P(w_2|w_1) \times P(w_3|w_1 w_2) \times P(w_4|w_1 w_2 w_3) \times P(w_5|w_1 w_2 w_3 w_4)$$

While exactly computing $P(w_5|w_1 w_2 w_3 w_4)$ is difficult, a good approximation is $P(w_5|w_1 w_2 w_3 w_4) \approx$

$P(w_5|w_3w_4)$.

Let $C(w_3w_4w_5)$ represent the number of occurrences of the sequence $w_3w_4w_5$ in a corpus. We can then empirically approximate $P(w_5|w_3w_4)$ with $\frac{C(w_3w_4w_5)}{C(w_4w_5)}$. However, this approximation alone is not enough; there may still be many three word combinations that do not occur in the corpus, but that should not be assigned zero probabilities. So we *smooth* this approximation, for instance by using

$$P(w_5|w_3w_4) \approx \lambda_1 \frac{C(w_3w_4w_5)}{C(w_3w_4)} \ + \ (1-\lambda_1)\left(\lambda_2 \frac{C(w_4w_5)}{C(w_4)} \ + \ (1-\lambda_2)\frac{C(w_5)}{\sum_w C(w)}\right)$$

where λ_1, λ_2 are numbers between 0 and 1 that determine the amount of smoothing.

Now, we can use these same approximations in PFGs. Let us assume that our PFG is binary branching and has g features, numbered $1...g$; we will call the parent features a_i, the left child features b_i, and the right child features c_i. In our earlier example, a_1 represented the parent nonterminal category; a_2 represented the parent number (singular or plural); a_3 represented the parent head word; b_1 represented the left child category; etc. We can write a PFG production as $(a_1, a_2, ..., a_g) \to (b_1, b_2, ..., b_g)(c_1, c_2, ..., c_g)$. If we think of the set of features for a constituent A as being the random variables $A_1, ..., A_g$, then the probability of a production is the conditional probability

$$P(B_1 = b_1, ..., B_g = b_g, C_1 = c_1, ..., C_g = c_g | A_1 = a_1, ..., A_g = a_g)$$

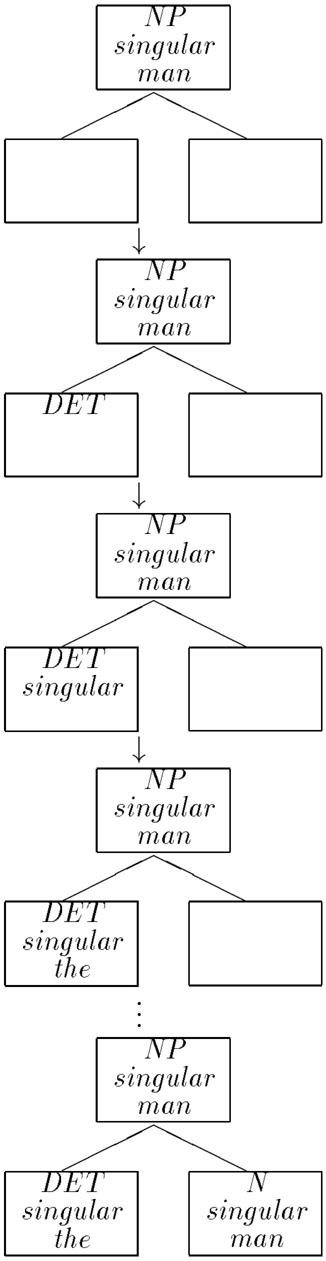
We write a_i as shorthand for $A_i = a_i$, and a_1^k to represent $A_1 = a_1, ..., A_k = a_k$. We can then write this conditional probability as

$$P(b_1^g, c_1^g | a_1^g)$$

This joint probability can be factored as the product of a set of conditional probabilities in many ways. One simple way is to arbitrarily order the features as $b_1, ..., b_g, c_1, ..., c_g$. We then condition each feature on the parent features and all features earlier in the sequence.

$$P(b_1^g, c_1^g | a_1^g) = P(b_1|a_1^g) \times P(b_2|a_1^g, b_1^1) \times P(b_3|a_1^g, b_1^2) \times \cdots \times P(c_g|a_1^g, b_1^g, c_1^{g-1})$$

We can now approximate the various terms in the factorization by making independence



$$P(B_1{=}DET|A_1 = NP,\ A_2{=}singular,\ A_3{=}man) \approx P(B_1{=}DET|A_1 = NP,\ A_2{=}singular)$$

$$P(B_2{=}singular|A_1 = NP,\ A_2{=}singular,\ A_3{=}man,\ B_1{=}DET) \approx P(B_2{=}singular|A_2{=}singular,\ B_1{=}DET)$$

$$P(B_3{=}the|A_1 = NP,\ A_2{=}singular, \ldots,\ B_2{=}singular) \approx P(B_3{=}the|B_1 = DET,\ A_3{=}man)$$

$$P(C_3{=}man|A_1 = NP,\ A_2{=}singular, \ldots,\ C_2{=}singular) \approx P(C_3{=}man|A_3 = man,\ A_1{=}NP)$$

Figure 6.2: Producing *the man*, one feature at a time

assumptions. For instance, returning to the concrete example above, consider feature c_1, the right child nonterminal or terminal category. The following approximation should work fairly well in practice:

$$P(c_1|a_1^g b_1^g) \approx P(c_1|a_1, b_1)$$

That is, the category of the right child is well determined by the category of the parent and the category of the left child. Just as n-gram models approximate conditional lexical probabilities by assuming independence of words that are sufficiently distant, here we approximate conditional feature probabilities by assuming independence of features that are sufficiently unrelated. Furthermore, we can use the same kinds of backing-off techniques that are used in smoothing traditional language models to allow us to condition on relatively large contexts. In practice, a grammarian determines the order of the features in the factorization, and then for each feature, which features it depends on and the optimal order of backoff, possibly using experiments on development test data for feedback. It might be possible to determine the factorization order, the best independence assumptions, and the optimal order of backoff automatically, a subject of future research.

Intuitively, in a PFG, features are produced one at a time. This order corresponds to the order of the factorization. The probability of a feature being produced depends on a subset of the features in a local context of that feature. Figure 6.2 shows an example of this feature-at-a-time generation for the noun phrase "the man." In this example, the grammarian picked the simple feature ordering $b_1, b_2, b_3, c_1, c_2, c_3$. To the right of the figure, the independence assumptions made by the grammarian are shown.

6.3 Formalism

In a PCFG, the important concepts are the terminals and nonterminals, the productions involving these, and the corresponding probabilities. In a PFG, a vector of features corresponds to the terminals and nonterminals. PCFG productions correspond to PFG *events* of the form $(a_1, ..., a_g) \rightarrow (b_1, ..., b_g)(c_1, ..., c_g)$, and our PFG rule probabilities correspond to products of conditional probabilities, one for each feature that needs to be generated.

6.3.1 Events and EventProbs

There are two kinds of PFG events of immediate interest. The first is a *binary event*, in which a feature set a_1^g (the parent features) generates features $b_1^g c_1^g$ (the child features). Figure 6.2 is an example of a single such event. Binary events generate the whole tree except the start node, which is generated by a *start event*.

The probability of an event is given by an *EventProb*, which generates each new feature in turn, assigning it a conditional probability given all the known features. For instance, in a binary event, the EventProb assigns probabilities to each of the child features, given the parent features and any child features that have already been generated.

Formally, an EventProb $\mathcal{E}$ is a 3-tuple $\langle \mathcal{K}, \overline{N}, \overline{F} \rangle$, where $\mathcal{K}$ is the set of conditioning features (the Known features), $\overline{N} = N_1, N_2, ..., N_n$ is an ordered list of conditioned features (the New features), and $\overline{F} = f_1, f_2, ..., f_n$ is a parallel list of functions. Each function $f_i(n_i, k_1, ..., k_k, n_1, n_2, ..., n_{i-1})$ returns $P(N_i = n_i | K_1 = k_1, ...K_k = k_k, N_1 = n_1, N_2 = n_2, ..., N_{i-1} = n_{i-1})$, the probability that feature $N_i = n_i$ given all the known features and all the lower indexed new features.

For a binary event, we may have $\mathcal{E}_B = \langle \{a_1, a_2, ..., a_g\}, \langle b_1, ..., b_g, c_1, ..., c_g \rangle, \overline{F}_B \rangle$; that is, the child features are conditioned on the parent features and earlier child features. For a start event we have $\mathcal{E}_S = \langle \{\}, \langle a_1, a_2, ..., a_g \rangle, \overline{F}_S \rangle$; i.e. the parent features are conditioned only on each other in sequence.

6.3.2 Terminal Function, Binary PFG, Alternating PFG

We need one last element: a function T from a set of g features to $\langle T, N \rangle$ which tells us whether a part of an event is terminal or nonterminal: the *terminal function*. A Binary PFG is then a quadruple $\langle g, \mathcal{E}_B, \mathcal{E}_S, T \rangle$: a number of features, a binary EventProb, a start EventProb, and a terminal function.

Of course, using binary events allows us to model n-ary branching grammars for any fixed n: we simply add additional features for terminals to be generated in the future, as well as a feature for whether or not this intermediate node is a "dummy" node (the continuation feature). We demonstrate how to do this in detail in Section 6.6.1.

On the other hand, it does not allow us to handle unary branching productions. In general, probabilistic grammars that allow an unbounded number of unary branches are very difficult to deal with (Stolcke, 1993). There are a number of ways we could have handled unary branches. The one we chose was to enforce an alternation between unary and binary branches, marking most unary branches as "dummies" with the continuation feature, and removing them before printing the output of the parser.

To handle these unary branches, we add one more EventProb, $\mathcal{E}_U$. Thus, an *Alternating PFG* is a quintuple of $\{g, \mathcal{E}_B, \mathcal{E}_S, \mathcal{E}_U, T\}$.

It is important that we allow only a limited number of unary branches. As we discussed in Chapter 2, unlimited unary branches lead to infinite sums. For conventional PCFG-style grammar formalisms, these infinite sums can be computed using matrix inversion, which is still fairly time-consuming. For a formalism such as ours, or a similar formalism, the effective number of nonterminals needed in the matrix inversion is potentially huge, making such computations impractical. Thus, instead we simply limit the number of unary branches, meaning that the sum is finite, for computing both the inside and the outside values. Two competing formalisms, those of Collins (1997) and Charniak (1997) allow unlimited unary branches, but because of this, can only compute Viterbi probabilities, not inside and outside probabilities.

6.4 Comparison to Previous Work

PFG bears much in common with previous work, but in each case has at least some advantages over previous formalisms.

Some other models (Charniak, 1996; Brew, 1995; Collins, 1996; Black *et al.*, 1992b) use probability approximations that do not sum to 1, meaning that they should not be used either for language modeling, e.g. in a speech recognition system, or as part of an integrated model such as that of Miller *et al.* (1996). Some models (Magerman, 1995; Collins, 1996) assign probabilities to parse trees conditioned on the strings, so that an unlikely sentence with a single parse might get probability 1, making these systems unusable for language modeling. PFGs use joint probabilities, so can be used both for language modeling and as part of an integrated model.

Furthermore, unlike all but one of the comparable systems (Black *et al.*, 1992a), PFGs can compute outside probabilities, which are useful for grammar induction, as well as for the parsing algorithms of Chapter 3, and the thresholding algorithms of Chapter 5.

Bigram Lexical Dependency Parsing. Collins (1996) introduced a parser with extremely good performance. From this parser, we take many of the particular conditioning features that we will use in PFGs. As noted, this model cannot be used for language modeling. There are also some inelegancies in the need for a separate model for Base-NPs, and the treatment of punctuation as inherently different from words. The model also contains a non-statistical rule about the placement of commas. Finally, Collins' model uses memory proportional to the sum of the squares of each training sentence's length. PFGs in general use memory that is only linear.

Generative Lexicalized Parsing. Collins (1997) worked independently from us to construct a model that is very similar to ours. In particular, Collins wished to adapt his previous parser (Collins, 1996) to a generative model. In this he succeeded. However, while we present a fairly simple and elegant formalism, which captures all information as features, Collins uses a variety of different techniques. First, he uses variables, which are analogous to our features. Next, both our models need a way to determine when to stop generating child nodes; like everything else, we encode this in a feature, but Collins creates a special STOP nonterminal. For some information, Collins modifies the names of nonterminals, rather than encoding the information as additional features. Finally, all information in our model is generated top-down. In Collins' model, most information is generated top-down, but distance information is propagated bottom-up. Thus, while PFGs encode all information as top-down features, Collins' model uses several different techniques. This lack of homogeneity fails to show the underlying structure of the model, and the ways it could be expanded. While Collins' model could not be encoded exactly as a PFG, a PFG that was extremely similar could be created.

Furthermore, our model of generation is very general. While our implementation captures head words through the particular choice of features, Collins' model explicitly generates first the head phrase, then the right children, and finally the left children. Thus, our model can be used to capture a wider variety of grammatical theories, simply by changing the choice of features.

Simple PCFGs. Charniak (1996) showed that a simple PCFG formalism in which the rules are simply "read off" of a treebank can perform very competitively. Furthermore, he showed that a simple modification, in which productions at the right side of the sentence have their probability boosted to encourage right branching structures, can improve performance even further. PFGs are a superset of PCFGs, so we can easily model the basic PCFG grammar used by Charniak, although the boosting cannot be exactly duplicated. However, we can use more principled techniques, such as a feature that captures whether a particular constituent is at the end of the sentence, and a feature for the length of the constituent. Charniak's boosting strategy means that the scores of constituents are no longer probabilities, meaning that they cannot be used with the inside-outside algorithm. Furthermore, the PFG feature-based technique is not extra-grammatical, meaning that no additional machinery needs to be added for parsing or grammar induction.

PCFG with Word Statistics. Charniak (1997) uses a grammar formalism which is in many ways similar to the PFG model, with several minor differences, and one important one. The main difference is that while we binarize trees, and encode rules as features about which nonterminal should be generated next, Charniak explicitly uses rules, in the style of traditional PCFG parsing, in combination with other features. This difference is discussed in more detail in Section 6.6.1.

Stochastic HPSG. Brew (1995) introduced a stochastic version of HPSG. In his formalism, in some cases even if two features have been constrained to the same value by unification, the probabilities of their productions are assumed independent. The resulting probability distribution is then normalized so that probabilities sum to one. This leads to problems with grammar induction pointed out by Abney (1996). Our formalism, in contrast, explicitly models dependencies to the extent possible given data sparsity constraints.

IBM Language Modeling Group. Researchers in the IBM Language Modeling Group developed a series of successively more complicated models to integrate statistics with features.

The first model (Black *et al.*, 1993; Black *et al.*, 1992b) essentially tries to convert a unification grammar to a PCFG, by instantiating the values of the features. Because of data sparsity, however, not all features can be instantiated. Instead, they create a grammar where many features have been instantiated, and many have not; they call these partially instantiated features sets *mnemonics*. They then create a PCFG using the mnemonics as terminals and nonterminals. Features instantiated in a particular mnemonic are generated probabilistically, while the rest are generated through unification. Because no smoothing is done, and because features are grouped, data sparsity limits the number of features that can be generated probabilistically, whereas because we generate features one at a time and smooth, we are far less limited in the number of features we can use. Their technique of generating some features probabilistically, and the rest by unification, is somewhat inelegant; also, for the probabilities to sum to one, it requires an additional step of normalization, which they appear not to have implemented.

In their next model (Black *et al.*, 1992a), which strongly influenced our model, five attributes are associated with each nonterminal: a syntactic category, a semantic category, a rule, and two lexical heads. The rules in this grammar are the same as the mnemonic rules used in the previous work, developed by a grammarian. These five attributes are generated one at a time, with backoff smoothing, conditioned on the parent attributes and earlier attributes. Our generation model is essentially the same as this. Notice that in this model, unlike ours, there are two kinds of features: those features captured in the mnemonics, and the five categories; the categories and mnemonic features are modeled very differently. Also, notice that a great deal of work is required by a grammarian, to develop the rules and mnemonics.

The third model (Magerman, 1994), extends the second model to capture more dependencies, and to remove the use of a grammarian. Each decision in this model can in principal depend on any previous decision and on any word in the sentence. Because of these potentially unbounded dependencies, there is no dynamic programming algorithm: without pruning, the time complexity of the model is exponential. One motivation for PFG was to capture similar information to this third model, while allowing dynamic programming. This third model uses a more complicated probability model: all probabilities are determined using decision trees; it is an area for future research to determine whether we can improve our performance by using decision trees.

Probabilistic LR Parsing with Unification Grammars. Briscoe and Carroll describe a formalism (Briscoe and Carroll, 1993; Carroll and Briscoe, 1992) similar in many ways to the first IBM model. In particular, a context-free covering grammar of a unification grammar is constructed. Some features are captured by the covering grammar, while others are modeled only through unifications. Only simple plus-one-style smoothing is done, so data sparsity is still significant. The most important difference between the work of

for each length l, shortest to longest
  for each start s
    for each split length t
      for each b_1^g s.t. $chart[s, s+t, b_1^g] \neq 0$
        for each c_1^g s.t. $chart[s+t, s+l, c_1^g] \neq 0$
          for each a_1 consistent with $b_1^g c_1^g$
            $\vdots$
            for each a_g consistent with $b_1^g c_1^g a_1^{g-1}$
              $chart[s, s+l, a_1^g] += \mathcal{E}_B(a_1^g \to b_1^g c_1^g)$
return $\sum_{a_1^g} \mathcal{E}_S(a_1^g) \times chart[1, n+1, a_1^g])$

Figure 6.3: PFG Inside Algorithm

Briscoe and Carroll (1993) and that of Black *et al.* (1993) is that Briscoe *et al.* associate probabilities with the (augmented) transition matrix of an LR Parse table; this gives them more context sensitivity than Black *et al.* However, the basic problems of the two approaches are the same: data sparsity; difficulty normalizing probabilities; and lack of elegance due to the union of two very different approaches.

6.5 Parsing

The parsing algorithm we use is a simple variation on probabilistic versions of the CKY algorithm for PCFGs, using feature vectors instead of nonterminals (Baker, 1979; Lari and Young, 1990). The parser computes inside probabilities (the sum of probabilities of all parses, i.e. the probability of the sentence) and Viterbi probabilities (the probability of the best parse), and, optionally, outside probabilities. In Figure 6.3 we give the inside algorithm for PFGs. Notice that the algorithm requires time $O(n^3)$ in sentence length, but is potentially exponential in the number of features, since there is one loop for each parent feature, a_1 through a_g.

When parsing a PCFG, it is a simple matter to find for every right and left child what the possible parents are. On the other hand, for a PFG, there are some subtleties. We must loop over every possible value for each feature. At first, this sounds overwhelming, since it requires guessing a huge number of feature sets, leading to a run time exponential in the number of features. In practice, most values of most features will have zero probabilities, and we can avoid considering these; only a small number of values will be consistent with previously determined features. For instance, features such as the length of a constituent take a single value per cell. Many other features take on very few values, given the children. For example, we arrange our parse trees so that the head word of each constituent is dominated by one of its two children. This means that we need consider only two values for this feature for each pair of children. The single most time consuming feature is the Name feature, which corresponds to the terminals and non-terminals of a PCFG. For efficiency, we keep a list of the parent/left-child/right-child name triples that have non-zero probabilities, allowing us to hypothesize only the possible values for this feature given the children. Careful choice of features helps keep parse times reasonable.

6.5.1 Pruning

We use two pruning methods to speed parsing. The first is beam thresholding with the prior, as described in Section 5.2. Within each cell in the parse chart, we multiply each entry's inside probability by the prior probability of the parent features of that entry, using a special EventProb, $\mathcal{E}_P$. We then remove those entries whose combined probability is too much lower than the best entry of the cell.

The other technique we use is multiple-pass parsing, described in Section 5.4. Recall that in multiple-pass parsing, we use a simple, fast grammar for the first pass, which approximates the later pass. We then remove any events whose combined inside-outside product is too low: essentially those events that are unlikely given the complete sentence. The technique is particularly natural for PFGs, since for the first pass, we can simply use a grammar with a superset of the features from the previous pass. The features we used in our first pass were Name, Continuation, and two new features especially suitable for a fast first pass, the length of the constituent and the terminal symbol following the constituent. Since these two features are uniquely determined by the chart cell of the constituent, they work particularly well in a first pass, since they provide useful information without increasing the number of elements in the chart. However, when used in our second pass, these features did not help performance, presumably because they captured information similar to that captured by other features. Multiple-pass techniques have dramatically sped up PFG parsing.

6.6 Experimental Results

The PFG formalism is an extremely general one that has the capability to model a wide variety of phenomena, and there are very many possible sets of features that could be used in a given implementation. We will, on an example set of features, show that the formalism can be used to achieve a high level of accuracy.

6.6.1 Features

In this section, we will describe the actual features used by our parser. The most interesting and most complicated features are those used to encode the rules of the grammar, the child features. We will first show how to encode a PCFG as a PFG. The PCFG has some maximum length right hand side, say five symbols. We would then create a PFG with six features. The first feature would be N, the nonterminal symbol. Since PFGs are binarized, while PCFGs are n-ary branching, we will need a feature that allows us to recover the n-ary branching structure from a binary branching tree. This feature, which we call the continuation feature, C, will be 0 for constituents that are the top of a rule, and 1 for constituents that are used internally. The next 5 features describe the future children of the nonterminal. We will use the symbol $\star$ to denote an empty child. To encode a PCFG rule such as $A \to BCDEF$, we would assign the following probabilities to the features sets:

$$\begin{array}{rcl}
\mathcal{E}_B((A,0,B,C,D,E,F)\to(B,0,X_1,...,X_5)\,(A,1,C,D,E,F,\star)) & = & P(B\to X_1...X_5)\\
\mathcal{E}_B((A,1,C,D,E,F,\star)\to(C,0,X_1,...,X_5)(A,1,D,E,F,\star,\star)) & = & P(C\to X_1...X_5)\\
\mathcal{E}_B((A,1,D,E,F,\star,\star)\to(D,0,X_1,...,X_5)(A,1,E,F,\star,\star,\star)) & = & P(D\to X_1...X_5)\\
\mathcal{E}_B((A,1,E,F,\star,\star,\star)\to(E,0,X_1,...,X_5)\qquad(F,0,Y_1,...,Y_5)) & = & P(E\to X_1...X_5)\times\\
 & & P(F\to Y_1...Y_5)
\end{array}$$

It should be clear that we can define probabilities of individual features in such as way as to get the desired probabilities for the events. We also need to define a distribution over the start symbols:

$$\begin{array}{rl}
\mathcal{E}_S((S,0,A,B,C,D,E)\to(A,0,X_1,...,X_5)(S,1,B,C,D,E,\star)) = & P(S\to ABCDE)\times\\
 & P(A\to X_1...X_5)
\end{array}$$

A quick inspection will show that this assignment of probabilities to events leads to the same probabilities being assigned to analogous trees in the PFG as in the original PCFG.

Now, imagine if rather than using the raw probability estimates from the PCFG, we were to smooth the probabilities in the same way we smooth all of our feature probabilities. This will lead to a kind of smoothed 5-gram model on right hand sides. Presumably, there is some assignment of smoothing parameters that will lead to performance which is at least as good, if not better, than unsmoothed probabilities. Furthermore, what if we had many other features in our PFG, such as head words and distance features. With more features, we might have too much data sparsity to make it really worthwhile to keep all five of the children as features. Instead, we could keep, say, the first two children as features. After each child is generated as a left nonterminal, we could generate the next child (the third, fourth, fifth, etc.) as the Child2 feature of the right child. Our new grammar probabilities might look like:

$$\begin{array}{rcl}
\mathcal{E}_B((A,0,B,C)\rightarrow(B,0,X_1,X_2)\ (A,1,C,D) & = & P(B \rightarrow X_1X_2...) \\
\mathcal{E}_B((A,1,C,D)\rightarrow(C,0,X_1,X_2)\,(A,1,D,E)) & = & P(C \rightarrow X_1X_2...) \\
\mathcal{E}_B((A,1,D,E)\rightarrow(D,0,X_1,X_2)\,(A,1,E,F)) & = & P(D \rightarrow X_1X_2...) \\
\mathcal{E}_B((A,1,E,F)\rightarrow(E,0,X_1,X_2)\,(F,0,Y_1,Y_2) & = & P(E \rightarrow X_1X_2...) \times P(F \rightarrow Y_1Y_2...)
\end{array}$$

where $P(B \rightarrow X_1X_2...) = \sum_\alpha P(B \rightarrow X_1X_2\alpha)$. This assignment will produce a good approximation to the original PCFG, using a kind of trigram model on right hand sides, with fewer parameters and fewer features than the exact PFG. Fewer child features will be very helpful when we add other features to the model. We will show in Section 6.6.4 that with the set of features we use, 2 children is optimal.

We can now describe the features actually used in our experiments. We used two PFGs, one that used the head word feature, and one otherwise identical grammar with no word based features, only POS tags. The grammars had the features shown in Table 6.1. A sample parse tree with these features is given in Figure 6.4.

Recall that in Section 6.5 we mentioned that in order to get good performance, we made our parse trees binary branching in such a way that every constituent dominates its head word. To achieve this, rather than generating children strictly left to right, we first generate all children to the left of the head word, left to right, then we generate all children to the right of the head word, right to left, and finally we generate the child containing the head word. It is because of this that the root node of the example tree in Figure 6.4 has *NP* as its first child, and *VP* as its second, rather than the other way around.

The feature D^L is a 3-tuple, indicating the number of punctuation characters, verbs, and words to the left of a constituent's head word. To avoid data sparsity, we do not count

N	**Name** Corresponds to the terminals and nonterminals of a PCFG.
C	**Continuation** Tells whether we are generating modifiers to the right or the left, and whether it is time to generate the head node.
1	**Child1** Name of first child to be generated.
2	**Child2** Name of second child to be generated. In combination with Child1, this allows us to simulate a second order Markov process on nonterminal sequences.
H	**Head name** Name of the head category.
P	**Head pos** Part of speech of head word.
W	**Head word** Actual head word. Not used in POS only model.
$\mathbf{D}^L$	Δ **left** Count of punctuation, verbs, words to left of head.
$\mathbf{D}^R$	Δ **right** Counts to right of head.
$\mathbf{D}^B$	Δ **between** Counts between parent's and child's heads.

Table 6.1: Features Used in Experiments

higher than 2 punctuation characters, 1 verb, or 4 words. So, a value of $\mathrm{D}^L = 014$ indicates that a constituent has no punctuation, at least one verb, and 4 or more words to the left of its head word. D^R is a similar feature for the right side. Finally, D^B gives the numbers between the constituent's head word, and the head word of its other child. Words, verbs, and punctuation are counted as being to the left of themselves: that is, a terminal verb has one verb and one word on its left.

Notice that the continuation of the lower *NP* in Figure 6.4 is R1, indicating that it inherits its child from the right, with the 1 indicating that it is a "dummy" node to be left out of the final tree.

6.6.2 Experimental Details

The probability functions we used were similar to those of Section 6.2, but with three important differences. The first is that in some cases, we can compute a probability exactly. For instance, if we know that the head word of a parent is "man" and that the parent got its head word from its right child, then we know that with probability 1, the head word of the right child is "man." In cases where we can compute a probability exactly, we do

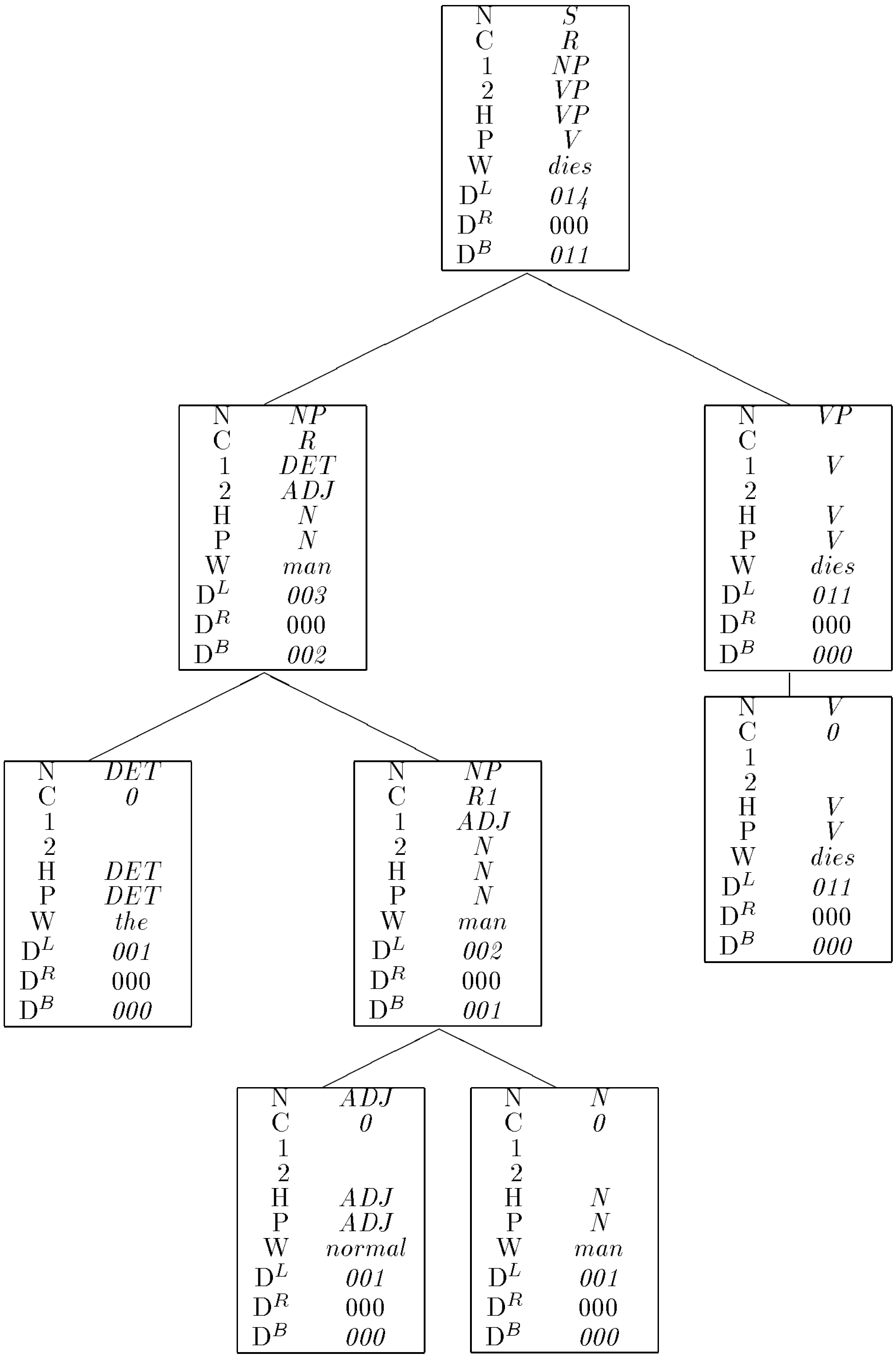


Figure 6.4: Example tree with features: The normal man dies

Model	Labeled Recall	Labeled Precision	Crossing Brackets	0 Crossing Brackets	≤ 2 Crossing Brackets
PFG Words	84.8%	85.3%	1.21	57.6%	81.4%
PFG POS only	81.0%	82.2%	1.47	49.8%	77.7%
Collins 97 best	88.1%	88.6%	0.91	66.5%	86.9%
Collins 96 best	85.8%	86.3%	1.14	59.9%	83.6%
Collins 96 POS only	76.1%	76.6%	2.26		
Magerman	84.6%	84.9%	1.26	56.6%	81.4%

Table 6.2: PFG experimental results

so. Second, we smoothed slightly differently. In particular, when smoothing a probability estimate of the form

$$p(a|bc) \approx \lambda \frac{C(abc)}{C(bc)} + (1-\lambda)p(a|b)$$

we set $\lambda = \frac{C(bc)}{k+C(bc)}$, using a separate k for each probability distribution. Finally, we did additional smoothing for words, adding counts for the unknown word.

The actual tables that show for each feature the order of backoff for that feature are given in Appendix 6–A. In this section, we simply discuss a single example, the order of backoff for the 2_R feature, the category of the second child of the right feature set. The most relevant features are first: $N_R, C_R, H_R, 1_R, N_P, C_P, N_L, C_L, P_R, W_R$. We back off from the head word feature first, because, although relevant, this feature creates significant data sparsity. Notice that in general, features from the same feature set, in this case the right features, are kept longest; parent features are next most relevant; and sibling features are considered least relevant. We leave out entirely features that are unlikely to be relevant and that would cause data sparsity, such as W_L, the head word of the left sibling.

6.6.3 Results

We used the same machine-labeled data as Collins (1996; 1997): TreeBank II sections 2-21 for training, section 23 for test, section 00 for development, using all sentences of 40 words or less.[1] We also used the same scoring method (replicating even a minor bug for the sake of comparison; see the footnote on page 156.).

[1] We are grateful to Michael Collins and Adwait Ratnaparkhi for supplying us with the part-of-speech tags.

Name	Features	Label Recall	Label Prec	Cross Brack	0 Cross Brack	$\leq$ 2 Cross Brack	Time
Base	HNCWPD^{B}D^{L}D^{R}12	86.4%	87.2%	1.06	60.8%	84.2%	40619
NoW	HNC PD^{B}D^{L}D^{R}12	82.7%	84.1%	1.36	52.2%	79.3%	39142
NoP	HNCW D^{B}D^{L}D^{R}12	85.8%	86.5%	1.18	57.5%	81.3%	46226
NoD^B	HNCWP D^{L}D^{R}12	86.0%	84.9%	1.24	56.6%	81.9%	49834
NoD	HNCWP 12	85.9%	85.9%	1.23	59.4%	81.0%	45375
No2	HNCWPD^{B}D^{L}D^{R}1	85.2%	86.5%	1.17	58.7%	81.9%	47977
No12	HNCWPD^{B}D^{L}D^R	76.6%	79.3%	1.65	45.8%	74.9%	37912
BaseH	NCWPD^{B}D^{L}D^{R}12	86.7%	87.7%	1.01	61.6%	85.1%	52785
NoWH	NC PD^{B}D^{L}D^{R}12	82.7%	84.0%	1.38	51.9%	79.5%	40080
NoPH	NCW D^{B}D^{L}D^{R}12	86.1%	87.2%	1.08	59.5%	83.1%	38502
NoD^{B}H	NCWP D^{L}D^{R}12	86.2%	85.2%	1.19	57.2%	82.5%	41415
NoDH	NCWP 12	86.4%	86.4%	1.17	59.9%	81.8%	39387
No2H	NCWPD^{B}D^{L}D^{R}1	82.4%	86.7%	1.11	56.8%	82.8%	37854
No12H	NCWPD^{B}D^{L}D^R	65.1%	80.0%	1.69	41.5%	73.5%	36790
NoNames	CWPD^{B}D^{L}D^R			2.41	41.9%	64.0%	55862
NoHPlus3	NCWPD^{B}D^{L}D^{R}123	86.9%	87.3%	1.07	61.7%	84.0%	41676

Table 6.3: Contribution of individual features

Table 6.2 gives the results. Our results are the best we know of from POS tags alone, and, with the head word feature, fall between the results of Collins and Magerman, as given by Collins (1997). To take one measure as an example, we got 1.47 crossing brackets per sentence with POS tags alone, versus Collins' results of 2.26 in similar conditions. With the head word feature, we got 1.21 crossing brackets, which is between Collins' .91 and Magerman's 1.26. The results are similar for other measures of performance.

6.6.4 Contribution of Individual Features

In this subsection we analyze the contribution of features, by running experiments using the full set of features minus some individual feature. The difference in performance between the full set and the full set minus some individual feature gives us an estimate of the feature's contribution. Note that the contribution of a feature is relative to the other features in the full set. For instance, if we had features for both head word and a morphologically stemmed head word, their individual contributions as measured in this manner would be nearly negligible, because both are so highly correlated. The same effect will hold to lesser degrees for other features, depending on how correlated they are. Some features are not meaningful without other features. For instance, the child2 feature is not particularly meaningful without the child1 feature. Thus, we cannot simply remove just child1 to compute its contribution; instead we first remove child2, compute its contribution, and then remove child1, as well.

When performing these experiments, we used the same dependencies and same order of backoff for all experiments. This probably causes us to overestimate the value of some features, since when we delete a feature, we could add additional dependencies to other features that depended on it. We kept the thresholding parameters constant (as optimized by the parameter search algorithm of Chapter 5), but adjusted the parameters of the backoff algorithm.

In the original work on PFGs, we did not perform this feature contribution experiment, and thus included the head name (H) feature in all of our original experiments. When we performed these experiments, we discovered that the head name feature had a negative contribution. We thus removed the head name feature, and repeated the feature contribution experiment without this feature. Both sets of results are given here. In order to minimize the number of runs on the final test data, all of these experiments were run on the development test section (section 00) of the data.

Table 6.3 shows the results of 16 experiments removing various features. Our first test was the baseline. We then removed the head word feature, leading to a fairly significant but not overwhelming drop in performance (6.8% combined precision and recall). Our results without the head word are perhaps the best reported. The other features – head pos, distance between, all distance features, and second child – all lead to modest drops in performance (1.3%, 2.7%, 1.8%, 1.9%, respectively, on combined precision and recall). On the other hand, when we removed both child1 and child2, we got a 17.7% drop. When we removed the head name feature, we achieved a 0.8% improvement. We repeated these experiments without the head name feature. Without the head name feature, the contributions of the other features are very similar: the head word feature contributed 7.7%. Other features – head pos, distance between, all distance features, and second child – contributed 1.1%, 3.0%, 1.6%, 5.3%. Without child1 and child2, performance dropped 29.3%. This is presumably because there were now almost no name features remaining, except for name itself. When the name feature was also removed, performance dropped even further (to about 2.41 crossing brackets per sentence, versus 1.01 for the no head name baseline – there are no labels here, so we cannot compute labelled precision or labelled recall for this case.) Finally, we tried adding in the child3 feature, to see if we had examined enough child features. Performance dropped negligibly: 0.2%.

These results are significant regarding child1 and child2. While the other features we used are very common, the child1 and child2 features are much less used. Both Magerman (1995) and Ratnaparkhi (1997) make use of essentially these features, but in a history-based formalism. One generative formalism which comes close to using these features is that of Charniak (1997), which uses a feature for the complete rule. This captures the child1, child2, and other child features. However, it does so in a crude way, with two drawbacks. First, it does not allow any smoothing; and second, it probably captures too many child features. As we have shown, the contribution of child features plateaus at about two children; with three children there is a slight negative impact, and more children probably lead to larger negative contributions. On the other hand, Collins (1996; 1997) does not use these features at all, although he does use a feature for the head name. (Comparing No12 to No12H, in Table 6.3, we see that without Child1 and Child2, the head name makes a significant positive contribution.) Extrapolating from these results, integrating the Child1 and Child2 features (and perhaps removing the head name feature) in a state-of-the-art model such as that of Collins (1997) would probably lead to improvements.

The last column of Table 6.3, the Time column, is especially interesting. Notice that the runtimes are fairly constrained, ranging from a low of about 36,000 seconds to a high of about 55,000 seconds. Furthermore, the longest runtime comes with the worst performance, and the fewest features. Better models often allow better thresholding and faster performance. Features such as child1 and child2 that we might expect to significantly slow down parsing, because of the large number of features sets they allow, in some cases (Base to No2) actually speed performance. Of course, to fully substantiate these claims would require a more detailed exploration into the tradeoff between speed and performance for each set of features, which would be beyond the scope of this chapter.

We reran our experiment on section 23, the final test data, this time without the head name feature. The results are given here, with the original results with the head name repeated for comparison. The results are disappointing, leading to only a slight improvement. The smaller than expected improvement might be attributable to random variation, either in the development test data, or in the final test data, or to some systematic difference between the two sets. Analyzing the two sets directly would invalidate any further experiments on the final test data, so we cannot determine which of these is the case.

Model	Labeled Recall	Labeled Precision	Crossing Brackets	0 Crossing Brackets	$\leq$ 2 Crossing Brackets
Words, head name	84.8%	85.3%	1.21	57.6%	81.4%
Words, no head name	84.9%	85.3%	1.19	58.0%	82.0%

6.7 Conclusions and Future Work

While the empirical performance of Probabilistic Feature Grammars is very encouraging, we think there is far more potential. First, for grammarians wishing to integrate statistics into more conventional models, the features of PFG are a very useful tool, corresponding to the features of DCG, LFG, HPSG, and similar formalisms. TreeBank II is annotated with many semantic features, currently unused in all but the simplest way by all systems; it should be easy to integrate these features into a PFG.

PFG has other benefits that we would like to explore, including the possibility of its use as a language model, for applications such as speech recognition. Furthermore, the dynamic programming used in the model is amenable to efficient rescoring of lattices output by speech recognizers.

Another benefit of PFG is that both inside and outside probabilities can be computed, making it possible to reestimate PFG parameters. We would like to try experiments using PFGs and the inside/outside algorithm to estimate parameters from unannotated text.

Because the PFG model is so general, it is amenable to many further improvements: we would like to try a wide variety of them. We would like to try more sophisticated smoothing techniques, as well as more sophisticated probability models, such as decision trees. We would like to try a variety of new features, including classes of words and nonterminals, and morphologically stemmed words, and integrating the most useful features from recent work, such as that of Collins (1997). We would also like to perform much more detailed research into the optimal order for backoff than the few pilot experiments used here.

While the generality and elegance of the PFG model make these and many other experiments possible, we are also encouraged by the very good experimental results. Wordless model performance is excellent, and the more recent models with words are comparable to the state of the art.

Appendix

6–A Backoff Tables

Tables 6.4, 6.5, and 6.6 give the order of backoff used in the experiments. For the wordless experiments, the head word feature – W – was simply deleted. The order of backoff is provided here for those who would like to duplicate these results. However, this table was produced mostly from intuition, and with the help of only a few pilot experiments; there is no reason to think that this is a particularly good set of tables.

The tables require a bit of explanation. A basic table entry was already described in Section 6.6. Briefly, a table entry gives the order of backoff of features, with the most relevant features first. In some cases, we wished to stop backoff beyond a certain point. This is indicated by a vertical bar, $|$. (This feature was especially useful in constructing the 6-gram grammar of Chapter 5, since it allowed smoothing to be easily turned off, making those experiments more easily replicated.) In some cases, it was possible to determine the value of a feature from the value of already known features. For instance, from the parent continuation feature, C_P, the parent first child feature, 1_P, and the parent name feature, N_P, we can determine the left child name feature, N_L. Thus, no backoff is necessary; this is indicated by a single vertical bar in the entry for N_L.

C_L	$C_P N_P 2_P 1_P D_P^B D_P^L D_P^R$
C_R	$C_P C_L N_P 2_P 1_P D_P^B D_P^L D_P^R$
N_R	$\mid$
N_L	$\mid$
P_L	$C_P N_L N_P D_P^B P_P W_P$
P_R	$C_P N_R N_P D_P^B P_P W_P$
W_L	$P_L C_P N_L N_P D_P^B P_P W_P$
W_R	$P_R C_P N_R N_P D_P^B P_P W_P$
H_L	$C_L C_P \mid N_L P_L W_L N_P N_R$
H_R	$C_R C_P \mid N_R P_R W_R N_P N_L$
1_L	$N_L C_L H_L N_P C_P N_R C_R P_L W_L$
2_L	$N_L C_L H_L 1_L N_P C_P N_R C_R P_L W_L$
1_R	$N_R C_R H_R N_P C_P N_L C_L P_R W_R$
2_R	$N_R C_R H_R 1_R N_P C_P N_L C_L P_R W_R$
D_L^L	$C_P C_L D_P^L D_P^B N_L N_P P_L$
D_R^R	$C_P C_R D_P^R D_P^B N_R N_P P_R$
D_L^R	$C_P C_L C_R D_P^B N_L N_R P_L P_R H_L H_R$
D_R^L	$C_P C_L D_P^B D_L^R N_R N_P P_R$
D_L^B	$\mid$
D_R^B	$\mid$

Table 6.4: Binary Event Backoff

D_L^L	D_P^L
D_L^R	D_P^R
C_L	$C_P N_P 1_P 2_P D_L^L D_L^R$
N_L	$\mid$
W_L	$\mid$
P_L	$\mid$
H_L	$N_L C_L C_P N_P D_L^L D_L^R$
1_L	$N_L C_L N_P C_P P_L W_L D_L^L D_L^R$
2_L	$N_L C_L 1_L N_P C_P P_L W_L D_L^L D_L^R$
D_L^B	$C_L D_L^L D_L^R N_L P_L H_L 1_L 2_L W_L$

Table 6.5: Unary Event Backoff

Start/Prior Event Backoff	
N_P	$\vert$
C_P	N_P
H_P	$N_P C_P \vert$
1_P	$N_P C_P H_P$
2_P	$N_P C_P 1_P H_P$
P_P	$N_P C_P H_P 1_P 2_P$
W_P	$P_P N_P C_P H_P 1_P 2_P$
D_P^B	$P_P C_P N_P H_P 1_P 2_P W_P$
D_P^L	$P_P D_P^B C_P N_P 1_P 2_P H_P W_P$
D_P^R	$P_P D_P^B D_P^L C_P N_P 1_P 2_P H_P W_P$

Table 6.6: Start/Prior Event Backoff

Chapter 7

Conclusion

All this means, of course, is that I didn't solve the natural language parsing problem. No big surprise there (although the naive graduate student in me is a little disappointed). *David Magerman*

7.1 Summary

The goal of this thesis has been to show new uses for the inside-outside probabilities and to provide useful tools for finding these probabilities. In Chapter 2, we gave tools for finding inside-outside probabilities in various formalisms. We then showed in Chapters 3, 4 and 5 that these probabilities have several uses beyond grammar induction, including more accurate parsing by matching parsing algorithms to metrics; 500 times faster parsing of the DOP model; and 30 times faster parsing of general PCFGs through thresholding. Finally, in Chaper 6 we gave a state-of-the-art parsing formalism that can compute inside and outside probabilities and that makes a good framework for future research.

We began by presenting in Chapter 2 a general framework for parsing algorithms, semiring parsing. Using the item-based descriptions of semiring parsing, it is easy to derive formulas for a wide variety of parsers, and for a wide variety of values, including the Viterbi, inside, and outside probabilities. As we look towards the future of parsing, we see a movement towards new formalisms with an increasingly lexicalized emphasis, and a movement towards parsing algorithms that can take advantage of the efficiencies of lexicalization (Lafferty *et al.*, 1992). Whether or not this occurs, as long as parsing technology does not remain stagnant, the theory of semiring parsing will simplify the development of whatever new parsers are needed. Furthermore, given the expense of developing treebanks, learning algorithms like the inside-outside reestimation algorithm will probably play a key role in the development of practical systems. Thus, the ease with which semiring parsing allows the inside and outside formulas to be derived will be important.

In Chapter 3, we showed that the inside and outside probabilities could be used to improve performance, by matching parsing algorithms to metrics. This basic idea is powerful. While many researchers have worked on different grammar induction techniques and different grammar formalisms, the only commonly used parsing algorithm was the Viterbi algorithm. Thus, many researchers can use the algorithms of this chapter, or appropriate variations, to get improvements

In Chapter 4, we sped up Data-Oriented Parsing by 500 times, and replicated the algorithms for the first time, using the inside-outside probabilities. While DOP had incredible reported results, we showed that these previous results were probably attributable to chance or to a particularly easy corpus. Negative results are always disappointing, but we hope to have at least helped steer the field in a more fruitful direction.

The normalized inside-outside probabilities are ideal for thresholding, giving the probability that any given constituent is correct. The only problem with using the inside-outside probabilities for thresholding is that the outside probability is unknown until long after it is needed. In Chapter 5 we showed that approximations to the inside-outside probabilities can be used to significantly speed up parsing. In particular, we sped up parsing by a factor of about 30 at the same accuracy level as traditional thresholding algorithms. We expect that these techniques will be broadly useful to the statistical parsing community.

Finally, in Chapter 6 we gave a state-of-the-art parsing formalism that could be used to compute inside and outside probabilities. While others (Charniak, 1997; Collins, 1997) independently worked in a very similar direction, we provided several unique contributions, including: an elegant theoretical framework; a useful way of breaking rules into pieces; an analysis of the value of each feature; and excellent wordless results. We think the theoretical structure of PFGs makes a good platform for future parsing research, allowing a variety of researchers to phrase their models in a unified framework.

At the end of a thesis like this, the question of the future direction of statistical parsing naturally arises. Given that statistical parsing has borrowed so much from speech recognition, it is only natural to compare the progress of statistical parsing systems to the progress of speech recognition systems. In particular, over the last ten or fifteen years, the progress of the speech recognition community has been stunning. Statistical parsing has made progress at a similar pace, but for a much shorter period of time. Given that a state-of-the-art speech system requires perhaps 15 person years of work to develop, while a state-of-the-art parsing system requires only one or two, quite a bit of progress remains to be made, if only by integrating everything we know how to do into a single system. If the most useful lessons of this thesis were combined with the most useful lessons of others (Magerman, 1995; Collins, 1997; Ratnaparkhi, 1997; Charniak, 1997, *inter alia*), we assume a much better parser would result. All of the techniques used in this thesis would fit well in such a system.

Recently, there has been work towards using statistical parsing-style methods as the framework for relatively complete understanding systems (Miller *et al.*, 1996; Epstein *et al.*, 1996). As this higher level work progresses, it seems likely that the techniques developed in this thesis will be be applicable. Our work on maximizing the right criteria and on thresholding algorithms for search could both probably be applied in higher level domains. Our work on PFGs could even serve as the framework for such new approaches, with features used to store or encode the semantic representation.

The lessons learned in this thesis are clearly applicable to future work in statistical NLP, but they are also useful more broadly within computer science. The general lessons include:

- Maximize the true criterion, or as close as you can get.
- Maximize pieces correct, using the overall probability of each piece; this can be easier, and sometimes more effective, than maximizing the whole.
- Guide search using the best approximations to future and past you can find, including local, global, and multiple-pass techniques. It may be possible to get better results by combining all of these techniques together.

Our work in Chapter 2 presented a general framework, which we used for statistical parsing. As pointed out by Tendeau (1997a), this kind of framework can be used to encode almost any dynamic programming algorithm, so the techniques of semiring parsing can actually be applied widely.

In summary, we have shown how to use the inside-outside probabilities and approximations to them to improve accuracy and speed parsing, and we have provided the tools to find these probabilities. These techniques will be useful both in future work on statistical Natural Language Processing, and more broadly.

References

[Abney1996] Steve Abney. 1996. Stochastic attribute-value grammars. Available as cmp-lg/9610003, October.

[Baker1979] J.K. Baker. 1979. Trainable grammars for speech recognition. In *Proceedings of the Spring Conference of the Acoustical Society of America*, pages 547–550, Boston, MA, June.

[Baum and Eagon1967] L.E. Baum and J.A. Eagon. 1967. An inequality with application to statistical estimation for probabilistic functions of Markov processes and to a model for ecology. *Bulletin of the American Mathematicians Society*, 73:360–363.

[Baum1972] L.E. Baum. 1972. An inequality and associated maximization technique in statistical estimation of probabilistic functions of a Markov process. *Inequalities*, 3:1–8.

[Berstel and Reutenauer1988] Jean Berstel and Christophe Reutenauer. 1988. *Rational Series and Their Languages*. Number 12 in EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Germany.

[Billot and Lang1989] Sylvie Billot and Bernard Lang. 1989. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th Annual Meeting of the ACL*, pages 143–151, Vancouver.

[Black *et al.*1992a] Ezra Black, Frederick Jelinek, John Lafferty, David M. Magerman, Robert Mercer, and Salim Roukos. 1992a. Towards history-based grammars: Using richer models for probabilistic parsing. In *Proceedings of the February 1992 DARPA Speech and Natural Language Workshop*.

[Black *et al.*1992b] Ezra Black, John Lafferty, and Salim Roukos. 1992b. Development and evaluation of a broad-coverage probabilistic grammar of English-language computer manuals. In *Proceedings of the 30th Annual Meeting of the ACL*, pages 185–192.

[Black *et al.*1993] Ezra Black, George Garside, and Geoffrey Leech. 1993. *Statistically-Driven Computer Grammars of English: the IBM/Lancaster Approach*, volume 8 of *Language and Computers: Studies in Practical Linguistics*. Rodopi, Amsterdam.

[Bod1992] Rens Bod. 1992. Mathematical properties of the data-oriented parsing model. Paper presented at the *Third Meeting on Mathematics of Language (MOL3)*, Austin Texas.

[Bod1993a] Rens Bod. 1993a. Data-oriented parsing as a general framework for stochastic language processing. In K. Sikkel and A. Nijholt, editors, *Parsing Natural Language*. Twente, The Netherlands.

[Bod1993b] Rens Bod. 1993b. Monte Carlo parsing. In *Proceedings Third International Workshop on Parsing Technologies*, Tilburg/Durbury.

[Bod1993c] Rens Bod. 1993c. Using an annotated corpus as a stochastic grammar. In *Proceedings of the Sixth Conference of the European Chapter of the ACL*, pages 37–44.

[Bod1995a] Rens Bod. 1995a. Efficient algorithms for parsing the DOP model? A reply to Joshua Goodman. Available from `http://xxx.lanl.gov/abs/cmp-lg/9605031`, May.

[Bod1995b] Rens Bod. 1995b. *Enriching Linguistics with Statistics: Performance Models of Natural Language*. University of Amsterdam ILLC Dissertation Series 1995-14. Academische Pers, Amsterdam.

[Bod1995c] Rens Bod. 1995c. The problem of computing the most probable tree in data-oriented parsing and stochastic tree grammars. In *Proceedings of the Seventh Conference of the European Chapter of the ACL*, Dublin, Ireland, March.

[Brew1995] Chris Brew. 1995. Stochastic HPSG. In *Proceedings of the Seventh Conference of the European Chapter of the ACL*, pages 83–89, Dublin, Ireland, March.

[Brill1993] Eric Brill. 1993. *A Corpus-Based Approach to Language Learning*. Ph.D. thesis, University of Pennsylvania.

[Briscoe and Carroll1993] Ted Briscoe and John Carroll. 1993. Generalized probabilistic LR parsing of natural language (corpora) with unification-based grammars. *Computational Linguistics*, 19:25–59.

[Caraballo and Charniak1996] Sharon Caraballo and Eugene Charniak. 1996. Figures of merit for best-first probabilistic chart parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 127–132, Philadelphia, May.

[Caraballo and Charniak1997] Sharon Caraballo and Eugene Charniak. 1997. New figures of merit for best first probabilistic chart parsing. In submission. Available from `http://www.cs.brown.edu/people/sc/NewFiguresofMerit.ps.Z` .

[Carroll and Briscoe1992] John Carroll and Ted Briscoe. 1992. Probabilistic normalisation and unpacking of packed parse forests for unification-based grammars. In *Proceedings of the AAAI Fall Symposium on Probabilistic Approaches to Natural Language*, pages 33–38, Cambridge, MA.

[Charniak1996] Eugene Charniak. 1996. Tree–bank grammars. Technical Report CS-96-02, Department of Computer Science, Brown University. Available from `ftp://ftp.cs.brown.edu/pub/techreports/96/cs96-02.ps.Z` .

[Charniak1997] Eugene Charniak. 1997. Statistical parsing with a context-free grammar and word statistics. In *Proceedings of the AAAI*, pages 598–603, Providence, RI. AAAI Press/MIT Press.

[Chi and Geman1998] Zhiyi Chi and Stuart Geman. 1998. Estimation of probabilistic context-free grammars. *Computational Linguistics*. To appear.

[Chomsky and Schützenberger1963] N. Chomsky and M. P. Schützenberger. 1963. The algebraic theory of context-free languages. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*. North-Holland.

[Collins1996] Michael Collins. 1996. A new statistical parser based on bigram lexical dependencies. In *Proceedings of the 34th Annual Meeting of the ACL*, pages 184–191, Santa Cruz, CA, June.

[Collins1997] Michael Collins. 1997. Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the ACL*, pages 16–23, Madrid, Spain.

[Cormen *et al.*1990] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. 1990. *Introduction to Algorithms.* MIT Press, Cambridge, MA.

[Earley1970] Jay Earley. 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13:94–102.

[Eisele1994] Andreas Eisele. 1994. Towards probabilistic extensions of constraint-based grammars. DYANA-2 Deliverable R1.2.B, September. Available from `ftp://ftp.ims.uni-stuttgart.de/pub/papers/DYANA2/R1.2.B`.

[Epstein *et al.*1996] M. Epstein, K. Papineni, S. Roukos, T. Ward, and S. Della Pietra. 1996. Statistical natural language understanding using hidden clumpings. In *ICASSP*, volume 1, pages 176–9, New York. IEEE.

[Goodman1996a] Joshua Goodman. 1996a. Efficient algorithms for parsing the DOP model. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 143–152, May.

[Goodman1996b] Joshua Goodman. 1996b. Parsing algorithms and metrics. In *Proceedings of the 34th Annual Meeting of the ACL*, pages 177–183, Santa Cruz, CA, June.

[Goodman1997] Joshua Goodman. 1997. Global thresholding and multiple-pass parsing. In *Proceedings of the Second Conference on Empirical Methods in Natural Language Processing*, pages 11–25.

[Goodman1998a] Joshua Goodman. 1998a. *Parsing Inside-Out.* Ph.D. thesis, Harvard University. In preparation.

[Goodman1998b] Joshua Goodman. 1998b. Semiring parsing. In preparation.

[Graham *et al.*1980] Susan L. Graham, Michael A. Harrison, and Walter L. Ruzzo. 1980. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2(3):415–462, July.

[Hemphill *et al.*1990] Charles T. Hemphill, John J. Godfrey, and George R. Doddington. 1990. The ATIS spoken language systems pilot corpus. In *DARPA Speech and Natural Language Workshop*, Hidden Valley, Pennsylvania, June. Morgan Kaufmann.

[Hopcroft and Ullman1979] John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts.

[Jelinek and Lafferty1991] F. Jelinek and J. D. Lafferty. 1991. Computation of the probability of initial substring generation by stochastic context-free grammars. *Computational Linguistics*, pages 315–323.

[Kasami1965] J. Kasami. 1965. An efficient recognition and syntax analysis algorithm for context-free languages. Technical report, University of Hawaii.

[Kuich and Salomaa1986] Werner Kuich and Arto Salomaa. 1986. *Semirings, Automata, Languages.* Number 5 in EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Germany.

[Kuich1997] Werner Kuich. 1997. Semirings and formal power series: Their relevance to formal languages and automata. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, pages 609–677. Springer-Verlag, Berlin.

[Lafferty *et al.*1992] John Lafferty, Daniel Sleator, and Davy Temperley. 1992. Grammatical trigrams: A probabilistic model of link grammar. In *Proceedings of the 1992 AAAI Fall Symposium on Probabilistic Approaches to Natural Language*, October.

[Lari and Young1990] K. Lari and S.J. Young. 1990. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, 4:35–56.

[Lari and Young1991] K. Lari and S.J. Young. 1991. Applications of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, 5:237–257.

[Leermakers1989] R. Leermakers. 1989. How to cover a grammar. In *Proceedings of the 27th Annual Meeting of the ACL*, pages 135–142, Vancouver.

[Magerman and Weir1992] D.M. Magerman and C. Weir. 1992. Efficiency, robustness, and accuracy in picky chart parsing. In *Proceedings of the Association for Computational Linguistics*.

[Magerman1994] David Magerman. 1994. *Natural Language Parsing as Statistical Pattern Recognition*. Ph.D. thesis, Stanford University University, February.

[Magerman1995] David Magerman. 1995. Statistical decision–models for parsing. In *Proceedings of the 33rd Annual Meeting of the ACL*, pages 276–283, Cambridge, MA.

[Miller *et al.*1996] Scott Miller, David Stallard, Robert Bobrow, and Richard Schwartz. 1996. A fully statistical approach to natural language interfaces. In *Proceedings of the 34th Annual Meeting of the ACL*, pages 55–61, Santa Cruz, CA, June.

[Mohri1997] Mehryar Mohri. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311.

[Nijholt1980] Anton Nijholt. 1980. *Contex-Free Grammars: Covers, Normal Forms, and Parsing*. Number 93 in Lecture Notes in Computer Science. Springer Verlag, Berlin, Germany.

[Pereira and Schabes1992] Fernando Pereira and Yves Schabes. 1992. Inside-Outside reestimation from partially bracketed corpora. In *Proceedings of the 30th Annual Meeting of the ACL*, pages 128–135, Newark, Delaware.

[Pereira and Shieber1987] Fernando Pereira and Stuart Shieber. 1987. *Prolog and Natural Language Analysis*. Number 10 in Lecture Notes. Center for the Study of Language and Information, Stanford, California.

[Pereira and Warren1983] Fernando Pereira and David Warren. 1983. Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the ACL*, pages 137–44, Cambridge, Massachusetts.

[Rabiner1989] L.R. Rabiner. 1989. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2), February.

[Ratnaparkhi1997] Adwait Ratnaparkhi. 1997. A linear observed time statistical parser based on maximum entropy models. In *Proceedings of the Second Conference on Empirical Methods in Natural Language Processing*, pages 1–10.

[Rayner and Carter1996] Manny Rayner and David Carter. 1996. Fast parsing using pruning and grammar specialization. In *Proceedings of the 34th Annual Meeting of the ACL*, pages 223–230, Santa Cruz, CA, June.

[Resnik1992] P. Resnik. 1992. Probabilistic tree-adjoining grammar as a framework for statistical natural language processing. In *Proceedings of the 14th International Conference on Computational Linguistics*, Nantes, France, August.

[Salomaa and Soittola1978] Arto Salomaa and Matti Soittola. 1978. *Automata-Theoretic Aspects of Formal Power Series.* Springer-Verlag, Berlin, Germany.

[Scha1990] R. Scha. 1990. Language theory and language technology; competence and performance. In Q.A.M. de Kort and G.L.J. Leerdam, editors, *Computertoepassingen in de Neerlandistiek.* Landelijke Vereniging van Neerlandici (LVVN-jaarboek), Almere. In Dutch.

[Schabes and Waters1994] Y. Schabes and R. Waters. 1994. Tree insertion grammar: A cubic-time parsable formalism that lexicalizes context-free grammar without changing the tree produced. Technical Report TR-94-13, Mitsubishi Electric Research Laboratories.

[Schabes *et al.*1993] Yves Schabes, Michal Roth, and Randy Osborne. 1993. Parsing the Wall Street Journal with the Inside-Outside algorithm. In *Proceedings of the Sixth Conference of the European Chapter of the ACL*, pages 341–347.

[Schabes1992] Yves Schabes. 1992. Stochastic lexicalized tree-adjoining grammars. In *Proceedings of the 14th International Conference on Computational Linguistics*, pages 426–432, Nantes, France, August.

[Schwartz *et al.*1992] Richard Schwartz, Steve Austin, Francis Kubala, John Makhoul, Long Nguyen, Paul Placeway, and George Zavaliagkos. 1992. New uses for the n-best sentence hypothesis within the Byblos speech recognition system. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume I, pages 1–4, San Francisco, California.

[Shieber *et al.*1993] Stuart Shieber, Yves Schabes, and Fernando Pereira. 1993. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 12.

[Sikkel1993] Klaas Sikkel. 1993. *Parsing Schemata.*

[Sima'an1996a] Khalil Sima'an. 1996a. Computational complexity of probabilistic disambiguation by means of tree grammars. In *Proceedings Coling-96.*

[Sima'an1996b] Khalil Sima'an. 1996b. Efficient disambiguation by means of stochastic tree substitution grammars. In R. Mitkov and N. Nicolov, editors, *Recent Advances in NLP 1995*, volume 136 of *Current Issues in Linguistic Theory.* John Benjamins, Amsterdam.

[Stolcke1993] Andreas Stolcke. 1993. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. Technical Report TR-93-065, International Computer Science Institute, Berkeley, CA.

[Teitelbaum1973] Ray Teitelbaum. 1973. Context-free error analysis by evaluation of algebraic power series. In *Proc. Fifth Annual ACM Symposium on Theory of Computing*, pages 196–199, Austin, Texas.

[Tendeau1997a] Frédéric Tendeau. 1997a. Computing abstract decorations of parse forests using dynamic programming and algebraic power series. *Theoretical Computer Science*. To appear.

[Tendeau1997b] Frédéric Tendeau. 1997b. An Earley algorithm for generic attribute augmented grammars and applications. In *Proceedings of the International Workshop on Parsing Technologies 1997*, pages 199–209.

[Viterbi1967] Andrew J. Viterbi. 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, IT-13:260–267.

[Wu1996] Dekai Wu. 1996. A polynomial-time algorithm for statistical machine translation. In *Proceedings of the 34th Annual Meeting of the ACL*, pages 152–158, Santa Cruz, CA, June.

[Younger1967] D. H. Younger. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10:189–208.

[Zavaliagkos *et al.*1994] G. Zavaliagkos, T. Anastasakos, G. Chou, C. Lapre, F. Kubala, J. Makhoul, L. Nguyen, R. Schwartz, and Y. Zhao. 1994. Improved search, acoustic and language modeling in the BBN Byblos large vocabulary CSR system. In *Proceedings of the ARPA Workshop on Spoken Language Technology*, pages 81–88, Plainsboro, New Jersey.