MEMOIZED PARSING WITH DERIVATIVES

by

Tobin Yehle

A Senior Honors Thesis Submitted to the Faculty of
The University of Utah
In Partial Fulfillment of the Requirements for the

Honors Degree in Bachelor of Science

in

Computer Science

Approved:

Vivek Srikumar, PhD Thesis Faculty Supervisor	Ross Whitaker, PhD Director, School of Computing
Erin Parker, PhD Honors Faculty Advisor	Sylvia D. Torti, PhD Dean, Honors College

April 2016

Copyright © 2016
All Rights Reserved

Abstract

Due to the computational complexity of parsing, constituent parsing is not preferred for tasks involving large corpora. However, the high similarity between sentences in natural language suggests that it is not necessary to pay the full computational price for each new sentence. In this work we present a new parser for phrase structure grammars that takes advantage of this fact by caching partial parses to reuse on later matching sentences. The algorithm we present is the first probabilistic extension of the family of derivative parsers that repeatedly apply the Brzozowski derivative to find a parse tree. We show that the new algorithm is easily adaptable to natural language parsing – we introduce a folded variant of the parser that keeps the size of lexical grammars small, thus allowing for efficient implementations of complex parsing models.

Contents

1 Introduction 1

2 Background and Related Work 3
2.1 Formal Languages and Computation . . . 3
2.1.1 Regular Languages . . . 4
2.1.2 Derivatives of Regular Languages . . . 4
2.1.3 Context Free Languages . . . 6
2.1.4 Parsing Context Free Grammars (CFGs) . . . 7
2.2 Parsing Natural Language . . . 8
2.2.1 Parsing Algorithms . . . 8
2.2.2 Lexical Parsing . . . 10
2.2.3 Collins Model . . . 10

3 Derivative Parsing with Probabilities 11
3.1 Derivatives of CFGs . . . 11
3.2 Constructing a Parse Tree . . . 14
3.3 Probabilities of Derived Productions . . . 17
3.4 Nullable Paths as Dijkstra's Algorithm . . . 17
3.5 Probabilistic Parsing with Derivatives . . . 22

4 Adapting to Lexicalized Parsing 24
4.1 The Naïve Approach . . . 24
4.2 Folding the Grammar . . . 25
4.2.1 Folded Dijkstra's Algorithm . . . 26
4.2.2 Changes in Representation . . . 27

5 Replicating the Collins Model 29
5.1 The Collins Parsing Model . . . 29
5.2 Encoding Features in Lexical Information . . . 31
5.3 An Efficient Encoding . . . 31
5.3.1 The Grammar . . . 32
5.3.2 Fixing the Parse Tree . . . 33
5.3.3 Results . . . 36

6 Conclusions 37

A Folded Example 41

Chapter 1

Introduction

Constituent parsing of natural language is notorious for its computational complexity [Jurafsky and Martin, 2009], and this makes it an unattractive choice for processing large volumes of text. The most popular alternatives today are either to not parse at all, or use efficient algorithms for dependency parsing. We believe there is no reason parsing needs to be such a time consuming problem given the availability memory and the high self similarity of natural language.

In this work we forego traditional parsing algorithms in favor of the Derivative Parser with Probabilities (DERP-P). We show that, not only does this algorithm find the most probable parse, it allows partial computation to be cached. The execution of the algorithm is naturally broken into self contained steps for each new word of the sentence. This allows partial parses to be saved, and then reused on future sentences with matching prefixes at no additional cost. Figure 1.1 shows the log frequency versus the log rank of sentence prefixes. This graph shows that many sentences have similar prefixes, so many sentences will be cache hits.

DERP-P is the first extension of derivative based parsers [Brzozowski, 1964] to probabilistic phrase structure grammars. In this work we demonstrate the expressive power of the algorithm by giving an implementation of the Collins parsing model Collins [1999] backed

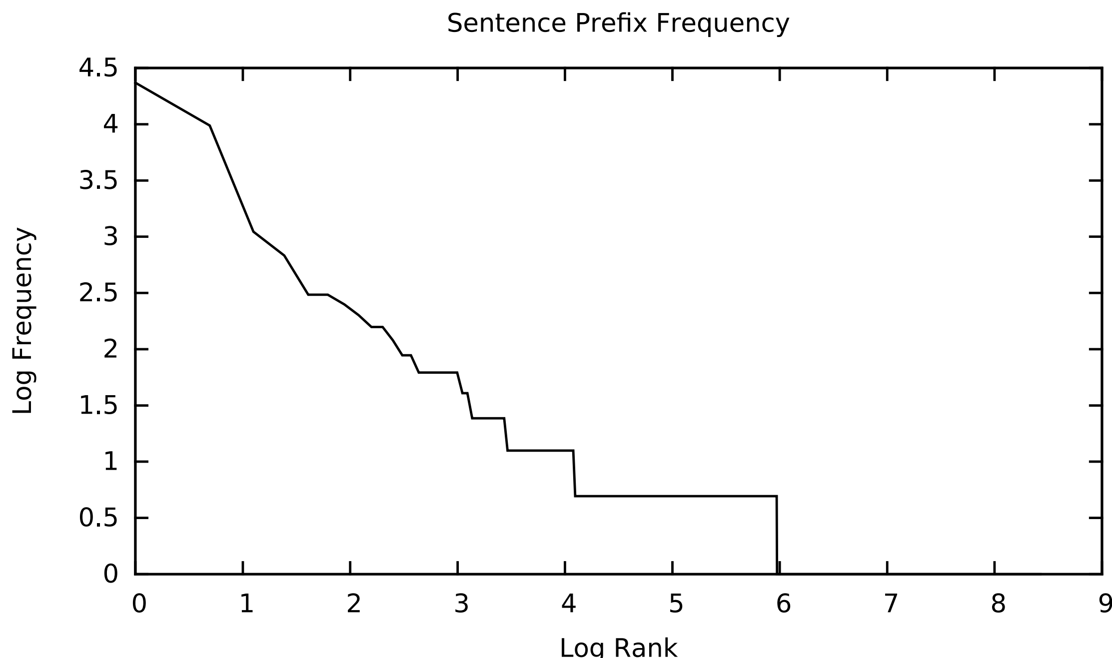


Figure 1.1: The log frequency of a sentence prefix versus the log of its rank. This was generated by looking at all prefixes of length less than 10 on a small set of news articles.

by DERP-P instead of the popularly used Cocke Younger Kasami algorithm. We show that the natural ability to cache partial parses makes DERP-P a good choice when faced with parsing large data sets.

The setting of the original Derivative Parser [Might et al., 2011] is programming languages, which have a much smaller grammar than the lexicalized probabilistic grammars used in natural language. We show how the lexical information can be folded up in order to keep the implementation fast in this new setting.

To summarize, the contributions of this research are

1. We introduce the first probabilistic extension to the derivative parser (DERP-P) that is geared towards parsing natural language.
2. DERP-P is guaranteed to find the maximum probability parse tree.
3. DERP-P can be cached without changing its results, allowing us to parse web-scale corpora.

Chapter 2

Background and Related Work

From a formal standpoint, a language can be viewed as a set of strings over some **alphabet**, where every string in the set is a sequence of letters in the alphabet. There are many ways to define such a set, and the different definitions can describe different classes of language. The most common question to ask about a language is if it contains a particular string. When viewing natural language formally it is common to ask what the syntactic structure of a sentence is. This structure relates the formal definition of the language to a particular sentence and often provides information that is key to understanding natural language. In this chapter we will discuss formal languages and parsing in many contexts.

2.1 Formal Languages and Computation

This section describes the theoretical formulation of language, and algorithms for determining if a string is contained in a language. It closes with a short discussion of parsing context free languages such as the ones encountered in programming languages.

2.1.1 Regular Languages

Languages are defined as sets of strings. Defining small languages by enumerating all strings in the language is only possible if the language contains a finite number of strings. Consider the language containing all strings with zero or more `a`s. Enumerating all strings in this language is not possible, as the set would be $\{\epsilon$, `a, aa, aaa, aaaa, ...` $\}$ and so on. The most simple class of language capable of representing an infinite number of strings is regular language.

Formally the empty language $\emptyset$, the language containing the empty string$\{\epsilon\}$, and all languages containing a single string with a single character are regular. All languages that are unions, concatenations, or repetitions (Kleene Star) of these basic languages are also regular [see Hopcroft et al., 2001].

Regular expressions are a notation for regular language that very closely matches this definition. Repetition is denoted with a $\star$, union with `|`, and concatenation by adjacency. An example of a regular language is: the language of all strings of `a`s with a `b` somewhere in the middle, which would be notated as `a*ba*`. The classic question to ask about a language is containment. For example, the sentence `aaaba` is in the language `a*ba*`, but the sentence `aaa` is not.

2.1.2 Derivatives of Regular Languages

The derivative of a language was introduced as a tool for answering the classic string containment question for languages. The derivative is an operation on languages defined by Brzozowski [1964]. The derivative of a language is taken with respect to a single character in the alphabet of the language. The operation itself is very simple. Suppose we want to take the derivative of a language with respect to a character c. For every string in the original language, cut off the first letter. If the removed letter is c, then the rest of the string is in the new language. We use the notation $D_c(L)$ to denote the derivative of a language L

with respect to a character c. In set builder notation, derivative of a language L is

$$D_c(L) = \{w : cw \in L\}$$

where c is the letter the derivative is taken with respect to and cw is a string in L. The derivative of the language $\{\texttt{foo, bar, baz}\}$ with respect to the character `b` would be $\{\texttt{ar, az}\}$. The string `foo` did not begin with a `b`, so it was removed. The strings `bar` and `baz` both began with the letter `b`, so that letter was removed, and the word that remained was added to the new language.

The derivative of languages provides a natural way to check if a string $s = c_0c_1 \ldots c_{n-1}c_n$ is in a language L. Asking if s is in L is equivalent to asking if ϵ is in $D_{c_n}(D_{c_{n-1}}(\ldots D_{c_1}(D_{c_0}(L))))$. Knowing if ϵ is in a language is much easier than knowing if an arbitrary string is in a language.

Unfortunately the definition of the derivative shown above is for sets of strings. Brzozowski [1964] also defines the derivative for regular expressions. The formal definition is

- $D_c(\emptyset) = \emptyset$
- $D_c(\epsilon) = \emptyset$
- $D_c(c') = \epsilon$ if $c = c'$ $\emptyset$ otherwise
- $D_c(A \cup B) = D_c(A) \cup D_c(B)$
- $D_c(A \bullet B) = (D_c(A) \bullet B) \cup (\delta(A) \bullet D_c(B))$
- $D_c(A*) = D_c(A) \bullet A*$

where $\delta(A)$ is the nullability function $A \cap \{\epsilon\}$. A regular expression is nullable if the language it describes contains the empty string. The result of the function δ is either the empty set, or the set containing only the empty string ϵ.

The language we used before, `a*ba*`, is defined with a regular expression. The definition of the derivative on regular expressions allows us to check if a string is in the language. If we wanted to know if the string `"aaaba"` is in the language we could equivalently ask if ϵ is in the language $D_a(D_b(D_a(D_a(D_a(\texttt{a}\star\texttt{ba}\star)))))$. Evaluating those derivatives yields the language `a*`, and ϵ is indeed in `a*`, so `"aaaba"` is in the original language. Taking derivatives with respect to the second example string, `"aaa"`, yields `a*ba*`, which does not contain the empty string, so `"aaa"` is not in the original language.

The derivative is a powerful tool for recognizing if a string is in a language that can be described by a regular expression, but unfortunately natural language is too complicated to be described by regular expressions.

2.1.3 Context Free Languages

The dominant formalism for representing the syntax of natural language (and programming languages) is the context free model [Jurafsky and Martin, 2009]. In practice, the language we speak is not context free [Shieber, 1985], but algorithms for parsing a richer model of language are not feasible. The context free model is sufficient in most circumstances, so it is the model of choice. Context free languages are often defined via a CFG. This is the specification of language used by popular algorithms in compilers.

Formally a CFG is a set of terminal symbols A, a set of non-terminal symbols, N, a set of rules R, and a start non-terminal symbol $n_0 \in N$. Each rule shows how to replace a single non-terminal symbol with a (possibly empty) list of other symbols, for example $\mathbf{S} \rightarrow a\ \mathbf{S}\ b$. Every string in the language can be found by applying rules from the grammar to a string of symbols a number of times, starting with a sequence containing only the start symbol [Hopcroft et al., 2001].

Notation When writing grammar symbols, bold upper-case letters will always be non-terminal symbols, and lower-case italicized letters will be terminal symbols. For example

S is a non-terminal symbol, and a is a terminal symbol. It is also important to note the difference between the derivative operator, written as $D_c(\mathbf{A})$, and a derived non-terminal symbol, written $\mathrm{D}_c[\mathbf{A}]$. The first denotes a function application, and the second is a regular non-terminal symbol in a grammar.

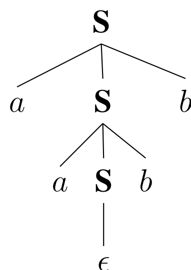
A slight modification of CFGs to include probabilities is popular in Natural Language Processing (NLP), and is known as a Probabilistic Context Free Grammar (PCFG). In these grammars every rule also has a probability which defines how likely that rule is in relation to the other rules in the grammar. These probabilities are used to resolve syntactic ambiguity.

2.1.4 Parsing CFGs

Every sentence in a language defined with a CFG can be produced by a number of expansions using the rules in the grammar. A simple example of a context free language is the language of a number of `'a'`s followed by the same number of `'b'`s. A representation of this language as a CFG is

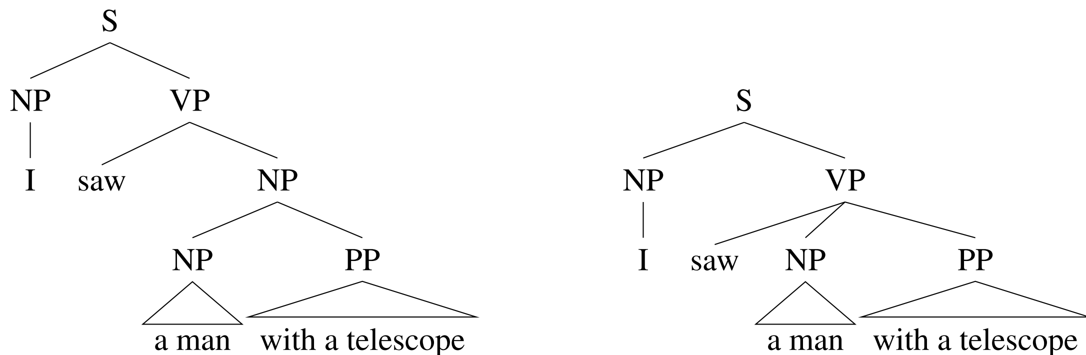
$$\mathbf{S} \rightarrow a\ \mathbf{S}\ b$$
$$\mathbf{S} \rightarrow \epsilon$$

This grammar asserts that any string in the language can be built by starting with an **S** and replacing that **S** with either a **S** b or the empty string ϵ, until there is no longer an **S**. For example the string `"aabb"` can be found as follows. $\mathbf{S} \rightarrow a\ \mathbf{S}\ b \rightarrow aa\ \mathbf{S}\ bb \rightarrow aabb$. An equivalent way of showing how this string was found from the grammar is with a parse tree. The parse tree for the string `aabb` would be



2.2 Parsing Natural Language

In natural language, there is meaning in the structure of the parse tree of a sentence. Consider the sentence *I saw a man with a telescope*. The parse tree of that sentence tells you if you saw a man using a telescope, or if you used a telescope to see a man.



This is why parsing is so important for understanding the meaning of an English sentence.

When parsing a programming language the definition of the language is one of the inputs to the algorithm. This language definition is designed by the creator of the language, but for natural language there is no standard grammar, instead a grammar is learned from a large set of example parse trees. The rules and probabilities of the grammar is inferred from these trees. In English, parsing has been driven by the Penn Treebank data-set [Marcus et al., 1993].

2.2.1 Parsing Algorithms

Of the algorithms available for parsing the most common in NLP is the Cocke Younger Kasami algorithm (CYK) because it is bottom up and can handle any CFG. Another algorithm that is sometimes used is the Earley algorithm [Earley, 1970]. The parser designed by Might et al. [2011] was intended for programming languages, but it is also capable of handling arbitrary CFGs, so it can be used to parse natural language.

CYK The CYK parsing algorithm is the most popular choice for NLP [Jurafsky and Martin, 2009]. It is bottom up, allowing capture of partial parses even when a full parse fails. The bottom up technique also allows the parser to cleanly handle the extra lexical information attached to the internal grammar symbols common to performant parsers. It is used by both the Collins [1999], and the Charniak [1997] parsing implementations. One disadvantage to the traditional CYK algorithm is it must have a grammar in Chomsky Normal Form to parse. This does not add any computational complexity, just a more complicated implementation. The complexity of the algorithm is $\Theta(n^3)$ where n is the number of words in the input sentence.

Earley The Earley parser is a left-to-right parser that proceeds in a top down fashion. Unlike the CYK parser, the Earley parser cannot give a partial parse tree for a failed parse, and some of its major advantages do not apply in the NLP context. It has linear complexity on certain classes of grammars common in programming languages, but if the input grammar falls outside this class, the algorithm still works, with a worst case complexity of $O(n^3)$. While the CYK algorithm only worked on grammars in Chomsky Normal Form, the Earley algorithm will accept any grammar.

Derivative Parser The parser presented by Might et al. [2011] works on arbitrary CFGs like the Earley parser, but instead of explicitly traversing the rules in the grammar, it uses the Brzozowski derivative to transform the grammar until there are no more tokens left in the input. It then reconstructs a parse from the derivation of the empty string from the start symbol of the final grammar. Unlike the Earley parser and CYK, the derivative parser has never been applied to natural language. Like the Earley parser, the derivative parser is a left-to-right, top down algorithm with a worst case of $O(n^3)$ [Adams and Hollenbeck, 2016].

2.2.2 Lexical Parsing

Lexical parsing allows a natural language parser to detect the difference between phrases like *house fly* and *birds fly*. They do this by associating a word and a part of speech tag with every internal symbol (e.g. noun phrases **NP**, verb phrases **VP**, etc.) in the grammar [Charniak, 1997]. This potentially makes the grammar very large, but in practice the whole expanded grammar does not need to be considered. This optimization is especially obvious in the context of CYK. The inclusion of lexical information pushed the accuracy of state-of-the-art parsers at the time [Charniak, 1997, 2000; Collins, 1999]. A full discussion of lexical parsing is given in Chapter 4.

2.2.3 Collins Model

One of the most influential models for parsing natural language is the [Collins, 1999] model. The heart of the Collins parsing model is the types of productions it considers. Every rule expands to a head non-terminal and any number of right and left non-terminals.

$$\mathbf{P} \quad \rightarrow \quad \mathbf{L}_1 \ldots \mathbf{L}_n \ \mathbf{H} \ \mathbf{R}_1 \ldots \mathbf{R}_m$$

This system means that every possible grammar rule has a defined probability, resulting in a problematic infinite grammar. Implementations of the Collins model employ a system of pruning to ensure they find a parse tree in reasonable time [Bikel, 2004]. A full discussion of the Collins parsing model is given in Chapter 5.

Chapter 3

Derivative Parsing with Probabilities

The parser introduced by Might et al. [2011] works with a richer set of operators than a parser for natural language grammars. First we describe a variant of the original algorithm for this simpler context, and then we extend that algorithm to probabilistic parsing.

3.1 Derivatives of CFGs

In this section we describe a formally identical system as Might et al. [2011]. The differences are due to the fact that the procedure for producing a derived grammar with respect to some symbol is simpler in the NLP context. When constructing a parser for a programming language it is often convenient to have operators like the Kleene star and reduction. Since the grammars of NLP are not built by hand and do not need to be understandable by humans, we can ignore all constructs except for concatenation and union. This does not change the set of representable languages, and greatly simplifies the construction of a grammar. Concatenation is represented as usual. Adjacent symbols in productions are concatenated. We encode union by denoting several rules for the same non-terminal, for example the grammar may contain both of the following rules: $\mathbf{S} \rightarrow \mathbf{NP}\ \mathbf{VP}$ and $\mathbf{S} \rightarrow \mathbf{VP}$, indicating that a sentence could be a noun phrase followed by a verb phrase, or just a standalone verb phrase.

In this simpler model taking the derivative of a single production in a grammar only involves the rule for concatenation. Taking a derivative of a concatenation was the most complicated of the cases for regular languages, and it is the most complicated for context free languages as well. In order to do it correctly it is necessary to know if symbols are nullable (i.e. can derive the empty string). The rule for a single concatenation is

$$D_c(\mathbf{A}\,\mathbf{B}) = \mathrm{D}_c[\mathbf{A}]\,\mathbf{B} \cup \delta(\mathbf{A})\,\mathrm{D}_c[\mathbf{B}]$$

Recall that δ is the nullability operator, and is either equal to the empty set or the set containing just the empty string ϵ. The rule for concatenation cannot be directly applied to a production in a grammar. In order to do that we first need to define the derivative for single symbols in the grammar

- $D_c(c) = \epsilon$
- $D_c(c') = \emptyset$ for $c \neq c'$
- $D_c(\mathbf{A}) = \mathrm{D}_c[\mathbf{A}]$

Derivatives of a Production Now the rule for the right hand side of a production can be defined. In regular expressions the derivative of a concatenation produced a union. In the context of CFGs this means the derivative can potentially result in more than one production if the first symbol is nullable. $D_c(\mathbf{P} \rightarrow \mathbf{A}\,\mathbf{B})$ will always include $\mathrm{D}_c[\mathbf{P}] \rightarrow \mathrm{D}_c[\mathbf{A}]\,\mathbf{B}$, but if $\mathbf{A}$ is nullable, then it also includes $\mathrm{D}_c[\mathbf{P}] \rightarrow \mathrm{D}_c[\mathbf{B}]$.

This can be generalized to productions with any number of symbols on the right hand side as follows. Consider a single rule from a CFG

$$\mathbf{P} \quad \rightarrow \quad \mathbf{N}_0 \ldots \mathbf{N}_k\, S\, \alpha$$

where all $\mathbf{N}_i$ are nullable, S is some non-nullable symbol (a terminal or a non-terminal), and α is any sequence of symbols. Taking the derivative of that production with respect to

some character c only requires the rule for concatenation, and will produce the following set of new rules

$$\begin{array}{rcl} \mathrm{D}_c[\mathbf{P}] & \rightarrow & \mathrm{D}_c[\mathbf{N}_0]\,\mathbf{N}_1 \ldots \mathbf{N}_k\, S\, \alpha \\ \mathrm{D}_c[\mathbf{P}] & \rightarrow & \mathrm{D}_c[\mathbf{N}_1]\,\mathbf{N}_2 \ldots \mathbf{N}_k\, S\, \alpha \\ & \vdots & \\ \mathrm{D}_c[\mathbf{P}] & \rightarrow & \mathrm{D}_c[\mathbf{N}_k]\, S\, \alpha \\ \mathrm{D}_c[\mathbf{P}] & \rightarrow & D_c(S)\, \alpha \end{array}$$

Every possible nullable prefix, including the empty prefix, is potentially skipped, resulting in $k + 2$ derived productions. Algorithm 1 describes how to find the derivative of a single rule in a grammar, but which rules need to be derived?

Algorithm 1 Derivative of a Production

Require: a token c, and a production $\mathbf{P} \rightarrow e_0 \ldots e_n$
1: result $\leftarrow$ empty list
2: add $\mathrm{D}_c[\mathbf{P}] \rightarrow D_c(e_0)\, e_1 \ldots e_n$ to result
3: $i \leftarrow 0$
4: **while** $i \leq n$ and NULLABLE(e_i) **do**
5: add $\mathrm{D}_c[\mathbf{P}] \rightarrow D_c(e_i)\, e_{i+1} \ldots e_n$ to result
6: $i \leftarrow i + 1$
7: **end while**

Derivatives of CFGs To find all the rules needed for the derived grammar we start by finding the production of the start non-terminal. We now need rules for all non-terminals produced by these first rules in the new grammar. We just repeat the procedure for all rules for some non-terminal until there are non-terminals left. If we need rules for a non-terminal that is in the original grammar, then we can just copy those rules into the new grammar. In the example above this would happen for all the non-terminals in α.

Once we have all the rules for the new grammar some of them can be removed. Consider the case where a non-terminal **A** has a single production

$$\mathbf{A} \rightarrow \epsilon$$

Taking the derivative of this production yields no new productions, so $\mathrm{D}_c[\mathbf{A}]$ can never be expanded because there are no productions with $\mathrm{D}_c[\mathbf{A}]$ on the left-hand side. Since $\mathrm{D}_c[\mathbf{A}]$ can not be expanded, there are no parse trees with an **A** in them, so every rule that has a $\mathrm{D}_c[\mathbf{A}]$ in its expansion can also be removed from the grammar. Removing these rules may result in other symbols with no valid expansions. Finding the fixed point of this removal process ensures that all rules can be safely expanded, and the new grammar is complete. This process is shown formally in Algorithm 2. Note that this algorithm only removes productions that cannot be included in a parse. The process by which we chose what rules to derive ensured we did not include unnecessary non-terminals in the grammar.

Algorithm 2 Removing Unneeded Productions

Require: productions

```
 1: changed ← true
 2: while changed do
 3:     changed ← false
 4:     names ← PARENT(productions)
 5:     for p ∈ productions do
 6:         if EXPANSION(p) ⊈ names then
 7:             changed ← true
 8:             REMOVE(p, productions)
 9:         end if
10:     end for
11: end while
```

3.2 Constructing a Parse Tree

Here we present a different, but equivalent formalism to Might et al. [2011] for reconstructing the parse tree from a derived grammar. Each time a production is derived, a reference to its parent production, and any nullable trees are stored along with it. All examples in this section come from the grammar defined in Figure 3.1. Consider taking the derivative of the production

$$\mathrm{D}_{Birds}[\mathbf{S}] \rightarrow \mathrm{D}_{Birds}[\mathbf{NP}]\ \mathbf{VP}$$

with respect to *fly*, then one of the resulting productions might be

$$\mathrm{D}_{Birds,fly}[\mathbf{S}] \rightarrow \mathrm{D}_{fly}[\mathbf{VP}]$$

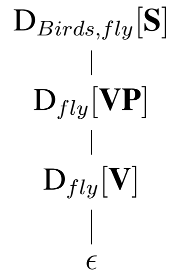
It would have a reference to the nullable path

$$\mathrm{D}_{Birds}[\mathbf{NP}] \rightarrow \mathrm{D}_{Birds}[\mathbf{N}] \rightarrow \epsilon$$

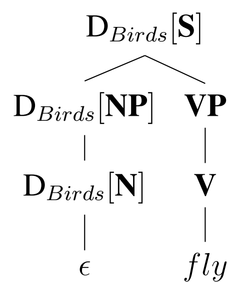
and a back-pointer to the production that produced it. By the definition of the derivative, the production $\mathrm{D}_{Birds,fly}[\mathbf{S}] \rightarrow \mathrm{D}_{fly}[\mathbf{VP}]$ is created because $\mathrm{D}_{Birds}[\mathbf{NP}]$ is nullable. In order to correctly reconstruct a parse tree, the path by which $\mathrm{D}_{Birds}[\mathbf{NP}]$ was nullable must be remembered along with the back-pointer. Storing these two references with each derived production allows a parse tree to be reconstructed from a nullable path in a derived grammar.

The algorithm to build the final parse tree from a nullable path is simple. Given a tree of productions (the nullable path), replace each production with the production stored in its back-pointer until the root of the tree is not a derived production. To replace a derived production with its back-pointer when a nullable path was used to create the production, the nullable path must be inserted into the resulting tree. Consider the *Birds fly* example above. Simply replacing $\mathrm{D}_{Birds,fly}[\mathbf{S}] \rightarrow \mathrm{D}_{fly}[\mathbf{VP}]$ with $\mathrm{D}_{Birds}[\mathbf{S}] \rightarrow \mathrm{D}_{Birds}[\mathbf{NP}]$ **VP** will not produce a tree that matches all the symbols in the expansion to child productions. The nullable path for $\mathrm{D}_{Birds}[\mathbf{NP}]$ needs to be added as the first child of the tree. This makes all of the symbols in the tree match up. The whole process is shown generally in Algorithm 3.

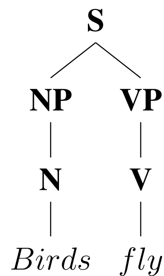
As an example consider the sentence *Birds fly* parsed with the grammar shown in Figure 3.1. The initial nullable path from the start symbol of the grammar would be



Stepping back once would require the insertion of the nullable path for the **NP** dominating *Birds*, and would produce the tree



Finally the full parse tree would be constructed by stepping back a second time



$$
\begin{array}{lcl}
\mathbf{S} & \rightarrow & \mathbf{NP}\,\mathbf{VP} \\
\mathbf{NP} & \rightarrow & \mathbf{N} \\
\mathbf{VP} & \rightarrow & \mathbf{V} \\
\mathbf{N} & \rightarrow & Birds \\
\mathbf{V} & \rightarrow & fly
\end{array}
$$

(a) The original grammar

$$
\begin{array}{lcl}
\mathrm{D}_{Birds,fly}[\mathbf{S}] & \rightarrow & \mathrm{D}_{fly}[\mathbf{VP}] \\
\mathrm{D}_{fly}[\mathbf{VP}] & \rightarrow & \mathrm{D}_{fly}[\mathbf{V}] \\
\mathrm{D}_{fly}[\mathbf{V}] & \rightarrow & \epsilon
\end{array}
$$

(b) The fully derived grammar

Figure 3.1: An example grammar for the sentence *Birds fly*.

Algorithm 3 Building a Parse Tree

Require: root
1: **function** STEPBACK(subtree)
2:     **if** subtree is a derived production **then**
3:         children ← subtree.nullPaths
4:         **for all** child in subtree.children **do**
5:             add STEPBACK(child) to children
6:         **end for**
7:         **return** TREE(subtree.parent, children)
8:     **else**
9:         **return** subtree
10:     **end if**
11: **end function**
12: **while** root is a derived production **do**
13:     root ← STEPBACK(root)
14: **end while**

3.3 Probabilities of Derived Productions

The probability of a derived production is the probability of its parent, times the probability of any nullable paths used in its creation. If the probability of $\text{D}_{Birds}[\textbf{S}] \rightarrow \text{D}_{Birds}[\textbf{NP}]$ **VP** was 0.5, and the probability of the path $\text{D}_{Birds}[\textbf{NP}] \rightarrow \text{D}_{Birds}[\textbf{N}] \rightarrow \epsilon$ was 0.1, then the probability of the derived production $\text{D}_{Birds,fly}[\textbf{S}] \rightarrow \text{D}_{fly}[\textbf{VP}]$ would be $0.5 \cdot 0.1 = 0.05$.

Since the final tree is built by replacing productions with their back-pointers, and all nullable paths, the probability of the tree does not change when replacing productions with their back-pointers. The probability of the nullable path in the derived grammar must be equal to the probability of the fully expanded parse tree. This is much simpler than how probabilities are handled by the stochastic Earley parser of Stolcke [1994].

3.4 Nullable Paths as Dijkstra's Algorithm

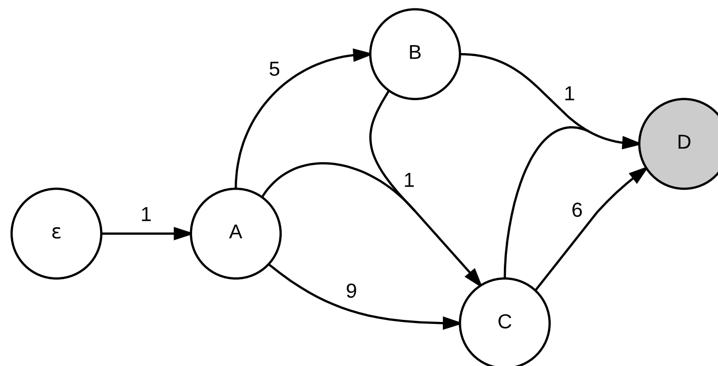
Final piece of the DERP-P is the nullable path algorithm. We do not want any nullable path as in Might et al. [2011], but the *most probable* path, according to the PCFG. Recall that a PCFG associates a probability with every rule. The probability of parse tree for such a

grammar is the product of the probabilities of all the rules present in the parse tree.

Finding a nullable path through a PCFG is equivalent to an execution of Dijkstra's algorithm on a hypergraph. A PCFG can be equivalently encoded into a graph where the nodes are the symbols in the grammar, and the edges are the productions. This encoding makes the application of Dijkstra's algorithm more clear without changing the structure of the grammar. Here we choose to make the arrows on the edges point the opposite direction they do in the productions of the grammar because we will want to find a route from a terminal to a symbol. Considering paths flowing from terminals to the root of a parse tree is the bottom up view point, instead of the top down method grammar productions use to show how to build strings in the language. This encoding is a type of hypergraph where every edge has a list of origin nodes, and a single destination node. Consider the following PCFG

$$\begin{array}{lcl} \mathbf{D} & \rightarrow & \mathbf{C}_{[7]} \mid \mathbf{B}\ \mathbf{C}_{[1]} \\ \mathbf{C} & \rightarrow & \mathbf{A}_{[8]} \mid \mathbf{A}\ \mathbf{B}_{[1]} \\ \mathbf{B} & \rightarrow & \mathbf{A}_{[5]} \\ \mathbf{A} & \rightarrow & \epsilon_{[1]} \end{array}$$

The numbers in brackets at the end of every production represent a score[1]. This grammar can be equivalently expressed by this hypergraph.



[1]Here we consider a more general case of scores that get added. The probabilities of productions must be multiplied, so the weights will actually be the log probabilities. This also ensures very small probability paths do not suffer from floating point error.

Every symbol in the grammar becomes a node in the hypergraph, and every production becomes a hyperedge. The scores of the productions are the weights of the hyperedges. The graph contains all the information of the original PCFG; it is just a different way of representing the grammar.

For the purposes of finding a path that can later become a parse tree we need to define a path as a tree of nodes. In order to traverse an edge you must be at *all* of the origin nodes. This means that the distance to a node accessed by a hyperedge is the weight of that edge plus the sum of the weights of the origin nodes. This differs from the paths presented by Gallo et al. [1993] which only require *one* of the origin nodes to be occupied.

Dijkstra's algorithm traditionally only works on graphs with all non-negative edge weights. Since the weights of the edges in our hypergraphs are log probabilities they will all be in the range $(-\infty, 0]$, and we want to find the *longest* path, not the shortest one. Multiplying all the weights by -1 will not only satisfy the non-negative constraint of Dijkstra's algorithm, it will also make the path with the lowest weight be the path with the greatest probability. This change allows us to use the shortest path algorithm to find the highest probability path through the grammar.

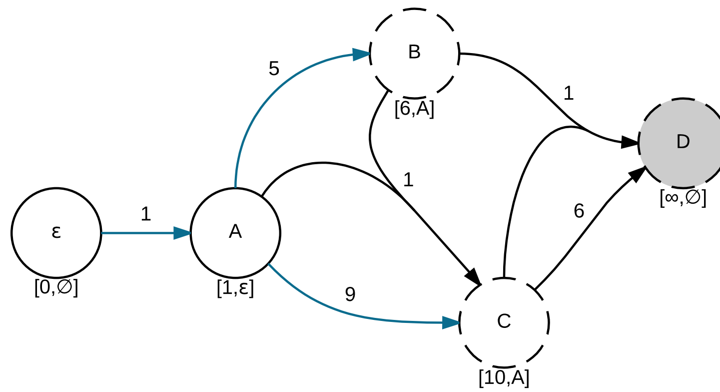
The correctness of the shortest path algorithm relies on the distance to each node being either correct or an overestimate. The fact that the distance to a vertex across a hyperedge is the sum of the distances to every origin vertex means that weight of the destination vertex if the path includes that edge will always be greater than or equal to the distance to any one of the origin vertices. This means that while at least one of the origin vertices of an edge is not marked as completed, there is no need to consider that edge when updating the distance estimates to adjacent nodes. The update step of the shortest path algorithm then only needs to consider edges where all the other origin vertices are marked done. The shortest path algorithm is stated formally in Algorithm 4. When the algorithm terminates, all the information needed to construct shortest paths through the graph is stored in the `parent` array. Using this information, a tree of productions representing a path through

the hypergraph can be produced. The process by which such a tree can be built is shown in Algorithm 5. It recursively builds a path by constructing a tree based on the contents of the `parent` array.

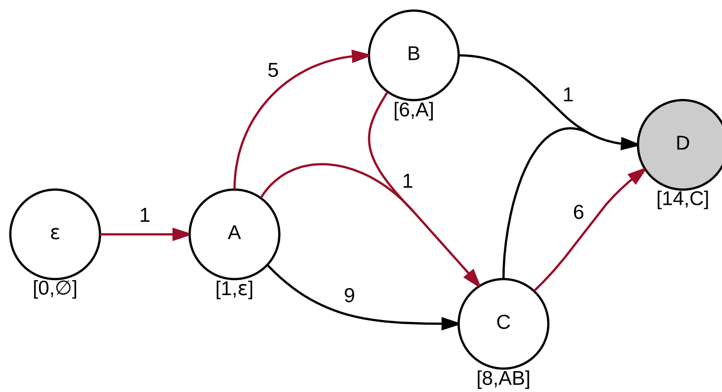
Algorithm 4 Dijkstra's algorithm on a hypergraph

Require: graph, a set of origin nodes, a destination node
Ensure: parent gives shortest paths, distance gives shortest total weight
1: **function** TRAVERSE(edge)
2: os $\leftarrow$ edge.origins
3: d $\leftarrow$ edge.destination
4: distance $\leftarrow$ WEIGHT(edge) $+ \sum_{\text{os}}$ distance
5: **if** distance[d] $>$ distance **then**
6: distance[d] $\leftarrow$ distance
7: parent[d] $\leftarrow$ os
8: **end if**
9: **end function**
10: **for all** node in nodes **do**
11: distance[node] $\leftarrow \infty$
12: parent[node] $\leftarrow \emptyset$
13: **end for**
14: **for all** node in origins **do**
15: distance[node] $\leftarrow 0$
16: **end for**
17: fringe $\leftarrow$ origins
18: done $\leftarrow \emptyset$
19: **for all** edge in edges s.t. ORIGIN(e) is empty **do**
20: TRAVERSE(edge)
21: ADD(edge.destination, fringe)
22: **end for**
23: **while** fringe is not empty **do**
24: best $\leftarrow$ node in fringe with min distance
25: ADD(best, done)
26: **for all** edge from best s.t. origins in done **do**
27: TRAVERSE(edge)
28: ADD(edge.destination, fringe)
29: **end for**
30: **end while**

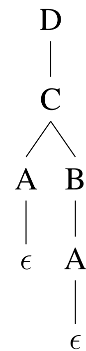
An example of the execution of the shortest path algorithm on the PCFG given above is shown in Figure 3.2.



(a) The state of the graph after ϵ and A have been processed, but B, C, and D have not.



(b) The state of the graph after the algorithm is completed.



(c) The shortest path from ϵ to D.

Figure 3.2: An example execution of Dijkstra's algorithm on the example PCFG of Section 3.4. The solid nodes have been marked done, while the dashed nodes have not. The current parent and path cost are shown beneath every vertex.

Algorithm 5 Building a Null Tree

Require: an array of cumulative distances for each node, an array of parent backpointers for each node

```
1: function BUILDTREE(root)
2:     if parent[root] is ∅ then
3:         return root
4:     else
5:         children ← []
6:         for all child in parent[root] do
7:             add BUILDTREE(child) to children
8:         end for
9:         return TREE(root, children)
10:    end if
11: end function
```

3.5 Probabilistic Parsing with Derivatives

Section 3.1 showed how to take a derivative of a CFG, assuming the existence of an algorithm for checking the nullability of a grammar symbol. Taking the derivative with respect to every word in an input sentence will give a grammar that contains the empty string if and only if that sentence was in the language. Section 3.2 showed how including back pointers in the derived productions allowed us to transform a path to null in the final grammar back into a parse tree containing only symbols in the input grammar. Section 3.4 showed how the most likely path to null can be found by executing Dijkstra's algorithm on the PCFG.

These pieces give us the tools needed to define the derivative parser for PCFG (DERP-P). The paths to null effectively enumerate all possible parse trees, and choose the argmax. The algorithm has the same worst case bound as CYK [Adams and Hollenbeck, 2016], and does not need the grammar to be transformed to Chomsky Normal Form. It parses left-to-right in a very similar way as the Earley parser, but it has clear intermediate states (the grammar) that can be saved for later reuse on a similar sentence.

DERP-P algorithm differs from the Earley algorithm and CYK principally in its use of intermediate states. For every word in the input sentence, the algorithm produces a new grammar that captures incremental structure of the parse. An intermediate grammar must

be able to parse the remainder of any sentence because information about the remainder of the sentence is not available. For this reason the parser has to do more work than a CYK parser because it must keep track of partially formed structures, even when parsing the last word in a sentence. While the parser does have to do more work, the complexity of the algorithm is the same, and the intermediate grammars can be cached. A cached grammar can be reused on a sentence starting with the same sequence of words in the future. The final grammar created by the parsing the sentence *Birds fly* reused to parse the sentence *Birds fly south*, or *Birds fly daily*. Figure 1.1 suggests that such reuse will happen often. This is a classic example of the time-memory trade off that has not been explored for parsing natural language.

Chapter 4

Adapting to Lexicalized Parsing

The algorithm described above works in general on PCFGs. When parsing natural language it is very common to parse a **lexicalized grammar** (introduced in section 4.1) instead of a grammar containing only the bare non-terminal symbols. The lexicalized grammar is much larger than the original one, and it is not practical or necessary to consider every lexical symbol in its own right.

4.1 The Naïve Approach

Lexicalizing a grammar takes every non-terminal symbol and associates a word and a part of speech tag with it. Process of lexicalization for a simple rule **S** $\rightarrow$ **NP VP** would produce

$$\begin{array}{rcl} \mathbf{S}_{\langle eats,VBZ\rangle} & \rightarrow & \mathbf{NP}_{\langle he,PRP\rangle}\,\mathbf{VP}_{\langle eats,VBZ\rangle} \\ \mathbf{S}_{\langle fly,VBP\rangle} & \rightarrow & \mathbf{NP}_{\langle birds,NNS\rangle}\,\mathbf{VP}_{\langle fly,VBP\rangle} \\ & \vdots & \end{array}$$

where VBZ, PRP, etc. are parts-of-speech tags from the Penn Treebank data-set [Marcus et al., 1993].

It is possible to consider each of pair of internal symbol and lexical information a new internal symbol, and parse on that grammar. Unfortunately this leads to catastrophic

grammar sizes, which is a factor in the running time of a parser. Not considering function tags, the Penn Treebank has 26 internal symbols, and 36 part of speech tags [Santorini, 1990]. If we assume a small dictionary of only 10,000 words, there would be $26 \cdot 36 \cdot 10000 = 9.36 \times 10^6$ internal symbols in a lexical grammar. This is clearly too many to allow for efficient parsing because the size of the grammar dominates parsing time [Klein and Manning, 2001]. A different approach is required.

4.2 Folding the Grammar

The solution to the problem of the size of the grammar is to do parsing on the unlexicalized grammar, and only using the lexical information when it is necessary to compute a score. This is the same approach taken by Collins [1999], but here it is manifested as what we call a **fold** of the lexical grammar.

The structure of the folded hypergraph is the same as that of the unlexicalized hypergraph, but every edge includes functions to produce lexical information when required. We change only the definitions of the edges and the weighting function. This small change allows parsing the full lexicalized PCFG. Formally, an edge changes to include a function, `extract`, from lists of lexical information to a single piece of lexical information, `extract` : $[Info] \to Info$. This function gives the lexical information of the destination vertex given the lexical information of the source vertices. This function is extracting the lexical information of the destination vertex from the lexical information of the source vertices. As an example, the edge $\mathbf{S}_{\langle fly,VBP\rangle} \to \mathbf{NP}_{\langle birds,NNS\rangle}\ \mathbf{VP}_{\langle fly,VBP\rangle}$ would be folded (along with many other edges) into the edge **S** $\to$ **NP VP** with the extraction function ensuring `extract`$(\langle birds, NNS\rangle, \langle fly, VBP\rangle) \mapsto \langle fly, VBP\rangle$[1].

This defines a folded grammar, but does not define scores for the folded edges. The weighting function from the original definition of a hypergraph cannot assign the correct

[1]The extraction function in this case is the head percolation function defined by Collins [1999].

scores because it may need to assign a different score based on the lexical information of the source nodes. To fix this issue, the scoring function not only requires an edge to score, but also the lexical information of the source nodes. These changes to the definition of a hypergraph still allow the operations required to parse with derivatives and greatly reduce the size of the grammar. In this new formalism we are allowed to consider only the lexical information of the words and part of speech tags we have seen in the sentence instead of the entire English language.

4.2.1 Folded Dijkstra's Algorithm

In order to produce a parse tree, we must be able to find the shortest path from ϵ to the start non-terminal in a derived grammar. We have seen in Section 3.4 that this is possible on hypergraphs, which allowed PCFG parsing. However, there are some notable differences when operating on a folded graph.

The back pointers become more complicated. Instead of just keeping a reference to the origins of the best edge into every vertex, we also need to keep track of the lexical information for each origin. There are also a lot of unfolded nodes, and we need to keep track of the distance and parents for all of them. This is not practical, but since we know what the initial values of the distance and parent should be, we don't need to store them explicitly. Instead every time we try to get the back pointer information of a vertex we first check if it exists, and if not then we substitute the default values. In other words, we dynamically materialize back-pointer information on an as-needed basis, and only for nodes that actually exist.

This provides the same advantages seen in bottom up parsers, which never store the lexical information for words that do not appear in the sentence. In our case, the only source of lexical information are the words that have already been consumed. The only lexical information a node could possibly have are words that appeared earlier in the sentence. This stops the explosion of grammar symbols that kills the performance of parsers on lexicalized

grammars.

The other major change from Algorithm 4 is how an edge updates the distance estimate to its destination. Since there is a function that determines the weight of the edge based on the lexical information of the source vertices, we need to consider all combinations of lexical information of the source vertices. For every combination there is also the possibility that the lexical information produced by the edge will be different. Inside the function that handles edges there is a loop over the Cartesian product of the lexical information of the source vertices. This ensures that all possible combinations are considered during the update. This algorithm is stated formally in Algorithm 6.

Algorithm 6 Folded Dijkstra Update

```
 1: function TRAVERSE(edge)
 2:     os ← edge.origins
 3:     d ← edge.destination
 4:     sum ← ∑_os distance
 5:     for all info in CARTESIAN(os.info) do
 6:         distance ← WEIGHT(edge, info) + sum
 7:         dInfo ← EXTRACT(edge, info)
 8:         if distance[d, dInfo] > distance then
 9:             distance[d, dInfo] ← distance
10:             parent[d, dInfo] ← os
11:         end if
12:     end for
13: end function
```

An example of the execution of this algorithm on the input sentence *Eat sushi with tuna* is given in Appendix A.

4.2.2 Changes in Representation

This section summarizes all the changes needed to parse a lexicalized PCFG. Recall from the previous chapter that every rule in the grammar needed to have: 1. A parent non-terminal, 2. A list of symbols the parent expanded to, 3. A score. In addition to the previous elements, the derived rules also needed to have the back-pointers required to compute a

parse tree. This back-pointer information is: 1. A pointer to the previous rule, 2. A list of nullable paths used when creating the rule.

There were too many lexical rules to store them this way, so we presented a system in which we could essentially produce lexical rules on demand. These folded rules are represented by: 1. A parent non-terminal, 2. A list of symbols the parent expands to, 3. An extraction function $[Info] \rightarrow Info$, 4. A scoring function $[Info] \rightarrow \mathbb{R}$. In addition to the previous elements, the derived rules need more complex back pointers to ensure the maximum probability parse tree is found. The back-pointer information is: 1. A pointer to the previous rule, 2. A map from lexical information to a corresponding nullable path and score $Info \rightarrow (Path, \mathbb{R})$. This information can be used to construct the possible lexical rules on demand. Only the lexical information seen in the previous words in the sentence is saved, greatly reducing the memory required to do lexical parsing.

Chapter 5

Replicating the Collins Model

The algorithm presented in the previous chapter allows arbitrary lexical information to be added to the grammar symbols, and requires two functions, `weight(edge, [info])`, and `extract(edge, [info])`. In order to capture the complexity of the Collins model (described in Section 5.1 below), the added lexical information cannot simply be the head word and part of speech tag for that symbol. There is still no way of writing down the grammar implied by Collins' model. Every possible production has a score, so the grammar would be infinitely large.

5.1 The Collins Parsing Model

Unlike the standard PCFG model, which associates a single probability with each rule, Collins' models for parsing introduce additional independence assumptions that allow the learned grammar to generalize better to unseen grammatical constructions. The heart of the Collins [1999] parsing model is the types of productions it considers. Every rule expands to a head non-terminal and any number of right and left non-terminals. Every terminal is directly produced by its part of speech tag.

$$\begin{aligned} \mathbf{P} &\rightarrow \mathbf{L}_1 \ldots \mathbf{L}_n\ \mathbf{H}\ \mathbf{R}_1 \ldots \mathbf{R}_m \\ \mathbf{T} &\rightarrow word \end{aligned}$$

This system means that every possible grammar rule has a defined probability, resulting in a problematic infinite grammar. Implementations of the Collins model cannot search over the entire infinite space of possible parse trees, and employ a system of pruning to ensure they find a parse tree in reasonable time [Bikel, 2004].

Here we only consider only Collins' Model 1, which includes some specialized lexical information as well as the conditional probabilities of head, left, and right, non-terminals. The base Collins model assumes the probability of a rule is equivalent to the probability of producing the head non-terminal

$$\mathbb{P}(\mathbf{H}|\mathbf{P}, h)$$

times the probability of producing the left non-terminals

$$\prod_{i=1}^{n+1} \mathbb{P}(\mathbf{L}_i l_i|\mathbf{P}, \mathbf{H}, h)$$

times the probability of producing the right non-terminals

$$\prod_{i=1}^{m+1} \mathbb{P}(\mathbf{R}_i r_i|\mathbf{P}, \mathbf{H}, h)$$

where h is the lexical information of the head child, and l_i and r_i are the lexical information of the ith left and right children respectively.

Collins' Model 1 also includes a distance function that encodes whether the tag being generated is adjacent to the head symbol, and if there is a verb dominated by the generated symbols. This extra information allows the model to learn right branching structures and modifiers on the previous verb, which both increase the accuracy on the Penn treebank [Collins, 1999]. The addition of the distance function changes the left and right probabilities to

$$\mathbb{P}(\mathbf{L}_i l_i|\mathbf{P}, \mathbf{H}, h, distance_l(i-1))$$

The inclusion of the distance and other features increase the performance of the Collins

parser, but for simplicity, we omit them in this work. Such features could be easily added by changing the definitions given below.

5.2 Encoding Features in Lexical Information

The CYK based implementation of the Collins model relied on the fact that CYK is a bottom up parsing algorithm to build rules as it needed them. In this way, the algorithm did not need to consider the entire infinite set of rules available. For a derivative parser to work we need to be able to write down the entire grammar so that we can take a derivative. The strategy for handling lexical information described in the previous chapter provides a solution to this problem.

There is no restriction on the type of the lexical information. Instead of a word tag pair, we encode all the information the Collins model needs to calculate probabilities into the lexical information of a rule. We are only ever interested in the score of a nullable rule. As long as the lexical information of every rule deriving ϵ is known, scores can be computed.

Now, instead of lexical information, every non-terminal symbol can have arbitrary information associated with it. To reflect its more general nature we will refer to this information as the non-terminal's tag, instead of its lexical information.

5.3 An Efficient Encoding

The solution to this problem of the infinite grammar is to not use a grammar where constituents produce other constituents. Since every constituent has a probability of producing every other constituent those symbols can also be included in the tags on the grammar symbols. Instead we will have only two grammar symbols, **U** and **F**, for unfinished and finished constituents. When every word in the input sentence has been parsed, the path from ϵ to **F** will be a rule for a complete sentence if an **S** is the constituent in its tag. Searching for such a node is trivial compared to the cost of parsing, and now there are only two non-terminal

Production	score	extract
$\mathbf{U} \to \mathbf{F}_{\langle a \rangle}\, \mathbf{U}_{\langle b \rangle}$	$\mathbb{P}(a, \text{Left}\vert b)$	b
$\mathbf{U} \to \mathbf{U}_{\langle a \rangle}\, \mathbf{F}_{\langle b \rangle}$	$\mathbb{P}(b, \text{Right}\vert a)$	a
$\mathbf{U}_P \to \mathbf{F}_{\langle a \rangle}$	$\mathbb{P}(a\vert P, a)$	(P, a)
$\mathbf{F} \to \mathbf{U}_{\langle a \rangle}$	$\mathbb{P}(\text{STOP}\vert a)$	a
$\mathbf{F} \to *$	1	generated

Table 5.1: The rules for the finished/unfinished realization of Collins' Model 1

symbols to consider when taking a derivative. Each of those symbols will have many tags associated with them at one time, but the total number of items to keep track of is the same as in Collins original model.

5.3.1 The Grammar

The full grammar for the finished/unfinished realization of the Collins model encodes the ways constituents can combine to form new constituents in exactly the same way as described in by Collins [1999]. The tags stored on each node are the tuple of (the symbol for the node, the symbol of the head child, the head word, and the head tag). This is the same set of information associated with the productions stored in the CYK chart in the original implementation of Collins' Model 1 [Collins, 1999].

Table 5.1 defines the rules in this realization of Collins' Model 1. Each type of rule is described briefly below. If we think of the rules going bottom up instead of top down, there is an obvious one-to-one mapping to the CYK implementation of Collins' Model 1 [Collins, 1999].

Join Rules The first two rules, $\mathbf{U} \to \mathbf{F}_{\langle a \rangle}\, \mathbf{U}_{\langle b \rangle}$, and $\mathbf{U} \to \mathbf{U}_{\langle a \rangle}\, \mathbf{F}_{\langle b \rangle}$, join a finished production onto an unfinished production on either the left or the right. The scores of these rules are one piece of the product defining the probability of the left or right constituents of a

Collins production.

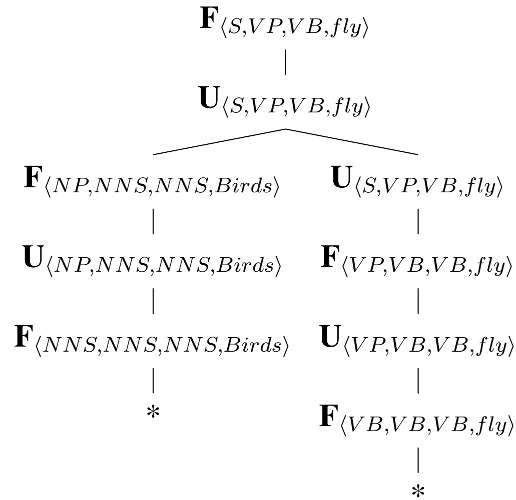
Head Rules There is a distinct rule of the form $\mathbf{U}_P \rightarrow \mathbf{F}_{\langle a \rangle}$ for every internal grammar symbol P. This rule builds a new unfinished constituent that can be joined with other finished constituents to make a new Collins production. The tag for this rule specifies that $\mathbf{F}_{\langle a \rangle}$ is the head of $\mathbf{U}_P$, and P is the constituent. The probability of this rule is the same as Collins' head probability.

Finish Rule The finishing rule $\mathbf{F} \rightarrow \mathbf{U}_{\langle a \rangle}$ adds the STOP probabilities to to an unfinished rule. The tag is exactly the same as $\mathbf{U}_{\langle a \rangle}$.

POS Rule The last rule constructs the initial part of speech tag from a word in the sentence. This rule uses a new grammar symbol, $*$. This symbol represents any token, its derivative with respect to anything is ϵ. The probability of a part-of-speech rule is 1 as in Collins [1999]. The tag associated with the **F** non-terminal is generated based on the token seen in the input. In this work we assume the part of speech tags are known, but you could also use the same trick here as with the internal nodes, duplicating the rule for every part of speech tag that should be generated. The tag extracted for the last rule just contains the part of speech tag **F** represents, and the word that was seen in the input.

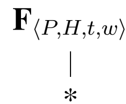
5.3.2 Fixing the Parse Tree

The parse trees resulting from this grammar are not the parse trees of the Penn Treebank. Using the tags $\langle$ parent, head, tag, word $\rangle$, and thre grammar in Table 5.1, the tree for the sentence *Birds fly* would be

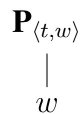


To change this tree into the correct parse tree we perform two bottom up tree transformations. The first replaces all the finished symbols **F** with the constituent they represent. The second transform removes all the unfinished symbols **U**.

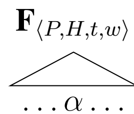
To replace a finished symbol $\mathbf{F}_{\langle P,H,t,w \rangle}$ we replace part-of-speech subtrees



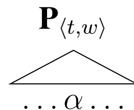
with



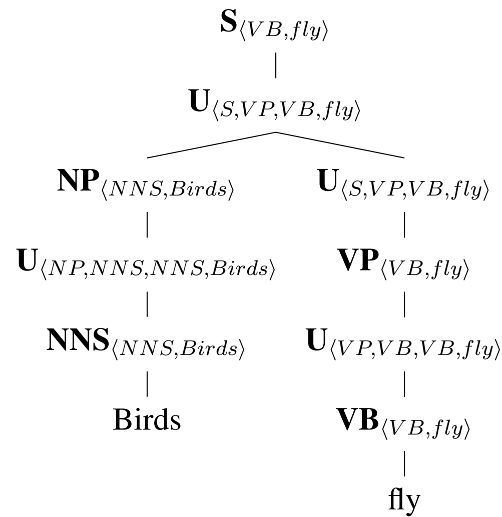
and we replace internal subtrees



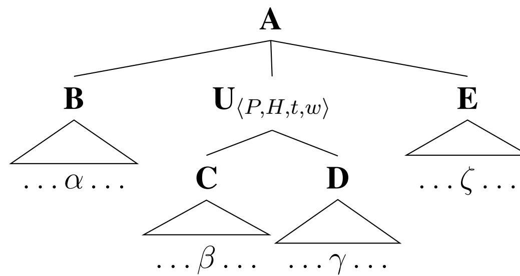
with



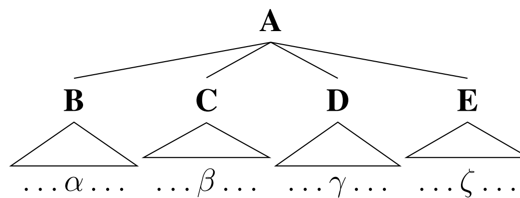
This transformation results in the following tree for *Birds fly*.



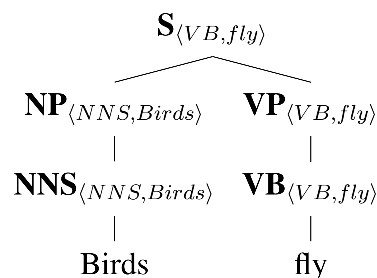
To remove an unfinished symbol $\mathbf{U}_{\langle P,H,t,w \rangle}$ we the **U** node with its children, so the tree



would become



This transform results in the correct parse tree for the sentence *Birds fly*.



5.3.3 Results

This encodes the Collins parsing model into a folded grammar over which the derivative is defined. It does this not by copying the grammar of the Collins model, but by encoding the way the symbols are combined in the CYK implementation of the model. This successfully encodes an infinite grammar into a small number of rules, and allows for many of the same optimizations as the original implementation to make it run in reasonable time.

Chapter 6

Conclusions

We have defined a new parsing algorithm for natural language called DERP-P (the Derivative Parser with Probabilities). This algorithm generalizes derivative parsing to include probabilistic grammars.

The algorithm uses derivatives to produce a new grammar for each word in an input sentence. When all of the words have been seen, a final execution of Dijkstra's algorithm results in the highest probability nullable path through the grammar. This path is then expanded into the most probable parse tree by following back-pointers saved when taking derivatives. This results in a left to right parser that has easily cacheable intermediate states with the same complexity as popular algorithms used in NLP. A further optimization allowed us to fold up a lexical grammar, ensuring efficient execution for complex parsing models like Collins' Model 1.

This new parsing algorithm is useful when parsing large data sets. It allows partial parses to be saved and then reused on a future sentence that shares a prefix with any sentence seen so far. This offers massive potential boosts in the time required to parse a large number of sentences.

The algorithm does do more work than an implementation that is not cache aware. It needs to save any place a future word could hook into the current parse tree, so there are

fewer opportunities to do pruning during parsing. If a data-set is small, then this algorithm could be slower than a traditional parsing algorithm.

Just because a data-set is small does not necessarily mean this algorithm does not have benefits. If we have access to a cache that has been warmed up with many sentences seen before, then loading from a pre-trained cache could also increase performance.

As the size of textual data becomes larger and larger, we need algorithms that can go faster as they see more data. Parsing is such a crucial aspect of most NLP pipelines that speeding up parsing can make a substantial contribution towards the goal of building text understanding systems that function on web-scale data.

Bibliography

Adams, M. D. and Hollenbeck, C. (2016). On the Complexity and Performance of Parsing with Derivatives. *37th annual ACM SIGPLAN conference on Programming Language Design and Implementation*.

Bikel, D. M. (2004). Intricacies of Collins' Parsing Model. *Computational Linguistics*, 30(4):479–511.

Brzozowski, J. a. (1964). Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494.

Charniak, E. (1997). Statistical parsing with a context-free grammar and word statistics. *Proceedings of the 14th National Conference on Artificial Intelligence*, (CS-95-28):598–603.

Charniak, E. (2000). A Maximum-Entropy-Inspired Parser. *1st North American chapter of the Association for Computational Linguistics conference (NAACL' 2000)*, (c):132–139.

Collins, M. (1999). *HEAD-DRIVEN STATISTICAL MODELS FOR NATURAL LANGUAGE PARSING Michael Collins*. PhD thesis.

Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.

Gallo, G., Longo, G., Pallottino, S., and Nguyen, S. (1993). Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42(2-3):177–201.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001). Introduction to Automata Theory, Languages, and Computation, 2Nd Edition. *SIGACT News*, 32(1):60–65.

Jurafsky, D. and Martin, J. H. (2009). Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. *Speech and Language Processing An Introduction to Natural Language Processing Computational Linguistics and Speech Recognition*, 21:0–934.

Klein, D. and Manning, C. D. (2001). Parsing with Treebank Grammars: Empirical Bounds, Theoretical Models, and the Structure of the Penn Treebank. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, ACL '01, pages 338–345, Stroudsburg, PA, USA. Association for Computational Linguistics.

Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics*, 19(2):313–330.

Might, M., Darais, D., and Spiewak, D. (2011). Parsing with Derivatives: A Functional Pearl. pages 189–195.

Santorini, B. (1990). Part-of-speech tagging guidelines for the Penn Treebank Project (3rd revision).

Shieber, S. M. (1985). *Evidence against the context-freeness of natural language*. Springer.

Stolcke, A. (1994). An Efficient Probabilistic Context-Free Parsing Algorithm that Computes Prefix Probabilities. *Computational Linguistics*, 21(2):165–201.

Appendix A

Folded Example

In this appendix we will see an example of parsing using a folded grammar. Consider the sentence

Eat sushi with tuna.

Suppose we want to parse this sentence using the following grammar:

$$
\begin{array}{lcl}
\mathbf{S}_{\langle h \rangle} & \rightarrow & \mathbf{VP}_{\langle h \rangle} \\
\mathbf{VP}_{\langle h \rangle} & \rightarrow & \mathbf{VT}_{\langle h \rangle}\ \mathbf{NP}_{\langle w_1 \rangle} \\
\mathbf{VP}_{\langle h \rangle} & \rightarrow & \mathbf{VT}_{\langle h \rangle}\ \mathbf{NP}_{\langle w_1 \rangle}\ \mathbf{PP}_{\langle w_2 \rangle} \\
\mathbf{NP}_{\langle h \rangle} & \rightarrow & \mathbf{N}_{\langle h \rangle} \\
\mathbf{NP}_{\langle h \rangle} & \rightarrow & \mathbf{N}_{\langle h \rangle}\ \mathbf{PP}_{\langle w_1 \rangle} \\
\mathbf{PP}_{\langle h \rangle} & \rightarrow & \mathbf{P}_{\langle h \rangle}\ \mathbf{NP}_{\langle w_1 \rangle} \\
\mathbf{N}_{\langle sushi \rangle} & \rightarrow & sushi \\
\mathbf{N}_{\langle tuna \rangle} & \rightarrow & tuna \\
\mathbf{N}_{\langle gusto \rangle} & \rightarrow & gusto \\
\mathbf{VT}_{\langle eat \rangle} & \rightarrow & eat \\
\mathbf{P}_{\langle with \rangle} & \rightarrow & with
\end{array}
$$

The extraction function is shown explicitly for each rule, and is equivalent to the Collins head percolation function.

We will not write the part of speech tag of the lexical information to save space. It is assumed that *tuna* always has the POS tag **N** and *eat* always has the tag **VT**. We will use the indices of the words in the sentence when referring to the in a derivative instead of the words themselves to try to keep things from getting too out of hand. For example $\mathrm{D}_{eat}[\mathbf{VT}]$ will be written $\mathrm{D}_0[\mathbf{VT}]$, and $\mathrm{D}_{sushi,with}[\mathbf{NP}]$ will be written $\mathrm{D}_{12}[\mathbf{NP}]$.

Taking the derivative with respect to *eat* yields the following new rules (in addition to all the rules in the original grammar)

$$\begin{aligned}
\mathrm{D}_0[\mathbf{S}_{\langle h\rangle}] &\rightarrow \mathrm{D}_0[\mathbf{VP}_{\langle h\rangle}] \\
\mathrm{D}_0[\mathbf{VP}_{\langle h\rangle}] &\rightarrow \mathrm{D}_0[\mathbf{VT}_{\langle h\rangle}]\,\mathbf{NP}_{\langle w_1\rangle} \\
\mathrm{D}_0[\mathbf{VP}_{\langle h\rangle}] &\rightarrow \mathrm{D}_0[\mathbf{VT}_{\langle h\rangle}]\,\mathbf{NP}_{\langle w_1\rangle}\,\mathbf{PP}_{\langle w_2\rangle} \\
\mathrm{D}_0[\mathbf{VT}_{\langle eat\rangle}] &\rightarrow \epsilon
\end{aligned}$$

Note the omission of rules such as $\mathbf{P}_{\langle with\rangle} \rightarrow with$. The derivative of these rules produced the empty set because the token did not match, so they were removed from the grammar. Deriving with respect to *sushi* yields

$$\begin{aligned}
\mathrm{D}_{01}[\mathbf{S}_{\langle h\rangle}] &\rightarrow \mathrm{D}_{01}[\mathbf{VP}_{\langle h\rangle}] \\
\mathrm{D}_{01}[\mathbf{VP}_{\langle eat\rangle}] &\rightarrow \mathrm{D}_1[\mathbf{NP}_{\langle w_1\rangle}] \\
\mathrm{D}_{01}[\mathbf{VP}_{\langle eat\rangle}] &\rightarrow \mathrm{D}_1[\mathbf{NP}_{\langle w_1\rangle}]\,\mathbf{PP}_{\langle w_2\rangle} \\
\mathrm{D}_1[\mathbf{NP}_{\langle h\rangle}] &\rightarrow \mathrm{D}_1[\mathbf{N}_{\langle h\rangle}] \\
\mathrm{D}_1[\mathbf{NP}_{\langle h\rangle}] &\rightarrow \mathrm{D}_1[\mathbf{N}_{\langle h\rangle}]\,\mathbf{PP}_{\langle w_1\rangle} \\
\mathrm{D}_1[\mathbf{N}_{\langle sushi\rangle}] &\rightarrow \epsilon
\end{aligned}$$

Note the **VP** rules from the original grammar are not longer needed, and the $\mathrm{D}_{01}[\mathbf{VP}]$ rules have had their lexical information forced to eat. Deriving with respect to *with* yields

$$
\begin{array}{lcl}
D_{012}[\mathbf{S}_{\langle h \rangle}] & \rightarrow & D_{012}[\mathbf{VP}_{\langle h \rangle}] \\
D_{012}[\mathbf{VP}_{\langle eat \rangle}] & \rightarrow & D_{12}[\mathbf{NP}_{\langle w_1 \rangle}] \\
D_{012}[\mathbf{VP}_{\langle eat \rangle}] & \rightarrow & D_{12}[\mathbf{NP}_{\langle w_1 \rangle}]\ \mathbf{PP}_{\langle w_2 \rangle} \\
D_{012}[\mathbf{VP}_{\langle eat \rangle}] & \rightarrow & D_{2}[\mathbf{PP}_{\langle w_2 \rangle}] \\
D_{12}[\mathbf{NP}_{\langle sushi \rangle}] & \rightarrow & D_{2}[\mathbf{PP}_{\langle w_1 \rangle}] \\
D_{2}[\mathbf{PP}_{\langle h \rangle}] & \rightarrow & D_{2}[\mathbf{P}_{\langle h \rangle}]\ \mathbf{NP}_{\langle w_1 \rangle} \\
D_{2}[\mathbf{P}_{\langle with \rangle}] & \rightarrow & \epsilon
\end{array}
$$

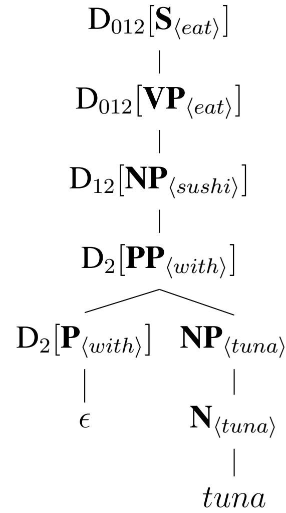
Finally deriving with respect to *tuna* gives the final grammar

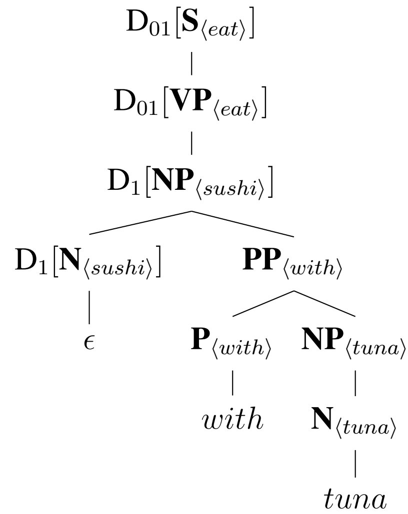
$$
\begin{array}{lcl}
D_{0123}[\mathbf{S}_{\langle h \rangle}] & \rightarrow & D_{0123}[\mathbf{VP}_{\langle h \rangle}] \\
D_{0123}[\mathbf{VP}_{\langle eat \rangle}] & \rightarrow & D_{123}[\mathbf{NP}_{\langle w_1 \rangle}] \\
D_{0123}[\mathbf{VP}_{\langle eat \rangle}] & \rightarrow & D_{123}[\mathbf{NP}_{\langle w_1 \rangle}]\ \mathbf{PP}_{\langle w_2 \rangle} \\
D_{0123}[\mathbf{VP}_{\langle eat \rangle}] & \rightarrow & D_{23}[\mathbf{PP}_{\langle w_2 \rangle}] \\
D_{123}[\mathbf{NP}_{\langle sushi \rangle}] & \rightarrow & D_{23}[\mathbf{PP}_{\langle w_1 \rangle}] \\
D_{23}[\mathbf{PP}_{\langle h \rangle}] & \rightarrow & D_{3}[\mathbf{NP}_{\langle w_1 \rangle}] \\
D_{3}[\mathbf{NP}_{\langle h \rangle}] & \rightarrow & D_{3}[\mathbf{N}_{\langle h \rangle}] \\
D_{3}[\mathbf{NP}_{\langle h \rangle}] & \rightarrow & D_{3}[\mathbf{N}_{\langle h \rangle}]\ \mathbf{PP}_{\langle w_1 \rangle} \\
D_{3}[\mathbf{N}_{\langle tuna \rangle}] & \rightarrow & \epsilon
\end{array}
$$

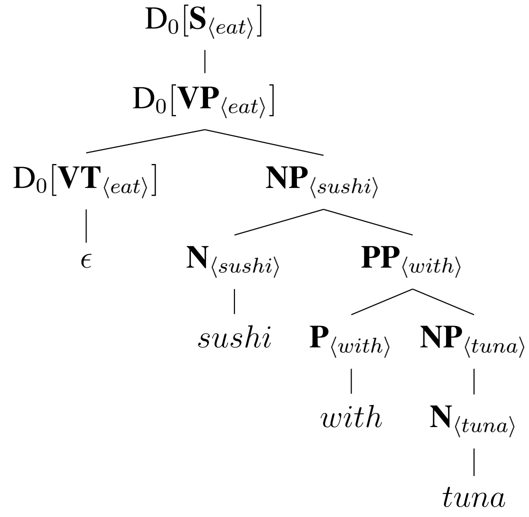
The nullable from ϵ to $D_{0123}[\mathbf{S}]$ that should have the highest score is

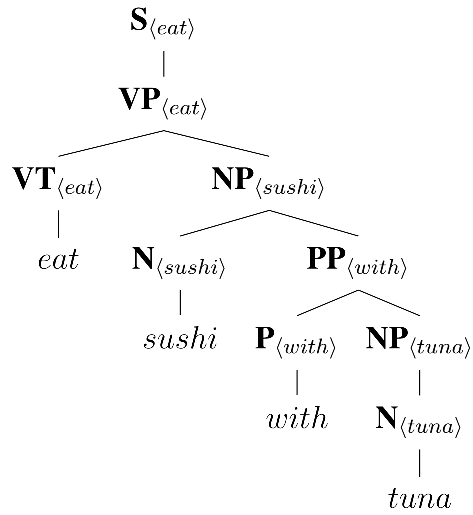
$$
\begin{array}{c}
D_{0123}[\mathbf{S}_{\langle eat \rangle}] \\
| \\
D_{0123}[\mathbf{VP}_{\langle eat \rangle}] \\
| \\
D_{123}[\mathbf{NP}_{\langle sushi \rangle}] \\
| \\
D_{23}[\mathbf{PP}_{\langle with \rangle}] \\
| \\
D_{3}[\mathbf{NP}_{\langle tuna \rangle}] \\
| \\
D_{3}[\mathbf{N}_{\langle tuna \rangle}] \\
| \\
\epsilon
\end{array}
$$

The following trees show the states of the back pointer replacing algorithm. The final tree is the parse tree for the input *Eat sushi with tuna*.






D0[S⟨eat⟩]
D0[VP⟨eat⟩]
D0[VT⟨eat⟩]
NP⟨sushi⟩
ϵ
N⟨sushi⟩
PP⟨with⟩
sushi
P⟨with⟩
NP⟨tuna⟩
with
N⟨tuna⟩
tuna


S⟨eat⟩
VP⟨eat⟩
VT⟨eat⟩
NP⟨sushi⟩
eat
N⟨sushi⟩
PP⟨with⟩
sushi
P⟨with⟩
NP⟨tuna⟩
with
N⟨tuna⟩
tuna

Name of Candidate:	Tobin Yehle
Birth date:	May 16, 1993
Birth place:	Salt Lake, Utah
Address:	2807 E. Sherwood Dr. Salt Lake, UT 84108